Omni Network

Omni Chain Review 2

Security Assessment Report

Version: 1.0

October, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Omni Network smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Omni Network smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Omni Network smart contracts.

Overview

Omni is a chain abstraction protocol that enables developers to create applications that are accessible across multiple rollups. Apps can interact with the `OmniPortal` contract to send cross-chain messages.

This review focused on new features and components added to Omni such as confirmation levels, the L1-Omni bridge, and gas exchange contracts.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the omni repository.

The scope of this time-boxed review was strictly limited to the following files at commit 99cbad6:

- `halo/app/`
- `halo/attest/`
- `halo/comet/`
- `halo/config/`
- `halo/evmslashing/`
- `halo/evmstaking/`
- `halo/evmupgrade/`
- `halo/portal/`
- `halo/registry/`
- `halo/valsync/`
- `lib/xchain/`
- `lib/cchain/`
- `octane/evmengine/`
- `contracts/core/src/`
  - `libraries/`
  - `octane/`
  - `pkg/`
  - `token/`
  - `xchain/`

Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity antipatterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

To support this review, the testing team also utilised the following Solidity automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

For the Golang libraries and modules, the review focused on internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime. Known Golang antipatterns such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks and a multitude of panics including but not limited to `nil` pointer deferences, index out of bounds, calls to `panic()`.

The following Golang automated testing tools were used:

- golangci-lint: `https://golangci-lint.run/`
- vet: `https://pkg.go.dev/cmd/vet`
- errcheck: `https://github.com/kisielk/errcheck`

Output for these automated tools is available upon request.

Coverage Limitations

Due to a time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 14 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 4 issues.
- Low: 5 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Omni Network smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
OM2-01	Nil Dereference In `XBlock()`	High	Open
OM2-02	Lost Bridged Funds When Pausing `ACTION_WITHDRAW`	Medium	Open
OM2-03	Incorrect Data Cost Calculation	Medium	Open
OM2-04	Lack Of Support For Non-EVM Data Pricing	Medium	Open
OM2-05	Unprotected Secret Files	Medium	Open
OM2-06	Bridge Fee Requirement Is Too Strict	Low	Open
OM2-07	Incorrect `OmniGasPump` ETH Quote	Low	Open
OM2-08	Nested `XMsgs` Break `MsgContext`	Low	Open
OM2-09	Missing Validation Of `cchain.SDKValidator` Structure	Low	Open
OM2-10	Validator Addresses Not Verified Against Signature	Low	Open
OM2-11	Missing Storage Gap From `OmniPortalStorage`	Informational	Open
OM2-12	Incorrect `XCall` Condition In `XAppBase`	Informational	Open
OM2-13	Return `nil` Both For Error And `abci.ValidatorUpdate`	Informational	Open
OM2-14	Miscellaneous General Comments	Informational	Open

OM2-01	Nil Dereference In `XBlock()`		
Asset	`lib/cchain/provider/xblock.go`		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A reachable `nil` pointer deference may occur in `Block()` causing a panic.

The function `XBlock()` on line [22] calls the `portalBlock()` function to get a `pbtypes.BlockResponse` at given height / offset. The `portalBlock()` function is defined in `newABCIPortalFunc()`, which calls `queryClient.Block()`, seen in the following code segment.

```
func (c *queryClient) Block(ctx context.Context, in *BlockRequest, opts ...grpc.CallOption) (*BlockResponse, error) {
   out := new(BlockResponse)
   err := c.cc.Invoke(ctx, "/halo.portal.types.Query/Block", in, out, opts...)
   if err != nil {
       return nil, err
   }
   return out, nil
}
```

The definition of `pbtypes.BlockResponse` can be seen in the following code segment.

```
type BlockResponse struct {
    Id            uint64 `protobuf:"varint,1,opt,name=id,proto3" json:"id,omitempty"`
    CreatedHeight uint64 `protobuf:"varint,2,opt,name=created_height,json=createdHeight,proto3" json:"created_height,omitempty"`
    Msgs          []*Msg `protobuf:"bytes,3,rep,name=msgs,proto3" json:"msgs,omitempty"`
}
```

Within the function `XBlock()` on line [43] each value in the array `Msgs []*Msg` array will be deferenced in the call `msg.ShardID()`. If any of these values are `nil`, a `nil` deference panic will occur.

The likelihood is rated as medium as it requires a malicious response from the `queryClient` to trigger the panic.

Recommendations

Implement a check to for each `msg` in `BlockResponse.Msgs` to ensure the pointer is not `nil` before accessing the members of the struct.

OM2-02	Lost Bridged Funds When Pausing `ACTION_WITHDRAW`		
Asset	`OmniBridgeL1.sol & OmniBridgeNative.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Pausing `ACTION_WITHDRAW` in the Omni bridges can lead to bridged funds becoming permanently lost.

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 by pausing the `ACTION_WITHDRAW` action, the `XMsg` will fail to be executed, causing any bridged tokens to be permanently lost.

In this case, there is no way to recover the bridged tokens, even if `OMNI` is being bridged to the Omni chain as the `OmniBridgeNative.claimable` mapping is not updated when `ACTION_WITHDRAW` is paused.

Recommendations

Ensure that `ACTION_DEPOSIT` is paused on the source chain bridge before pausing `ACTION_WITHDRAW` on the destination chain bridge. Also, ensure that any pending `withdraw()` `XMsgs` are executed before pausing `ACTION_WITHDRAW`.

Alternatively, a similar mechanism to the `OmniGasPump.owed` mapping can be implemented in both bridge contracts to allow users to retry bridging if it fails.

OM2-03	Incorrect Data Cost Calculation		
Asset	`FeeOracleV1.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There are components to the data cost calculation that are incorrect or do not account for gas and asset price volatility, which can lead to a fee that is lower than intended.

The `FeeOracleV1.feeFor()` function uses the size of the transaction input data (also referred to as calldata) to calculate `dataGas`. This assumes that rollups only post transaction input data and does not include other transaction fields such as transaction nonce, gas price, and gas limit.

```
IFeeOracleV1.ChainFeeParams storage dataP = _feeParams[execP.postsTo];

// ...

// @audit dataGasPrice uses the current `dataP.gasPrice` and `dataP.toNativeRate` values
uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;

// 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
// @audit data.length refers to the size of the cross-chain call's calldata
uint256 dataGas = data.length * 16;
```

Rollups such as Optimism and Base post the entire signed transaction serialised with RLP encoding. This means that the size of the posted data for a transaction is larger than just the transaction input data.

Furthermore, the `dataGasPrice` used in data cost calculation does not account for the volatility of the destination chain's gas price, as well as the volatility in the exchange rate of the destination chain's native token relative to the source chain's native token. These values can vary greatly at time of execution of the `XMsg` in `OmniPortal.xsubmit()`, resulting in the user paying less than the intended amount of fees.

Recommendations

To account for the data size of the total RLP encoded signed transaction, add a fixed amount of data to the data cost calculation as overhead.

To account for the volatility in the destination chain's gas price and the exchange rate of the destination chain's native token relative to the source chain's native token, add a premium to `dataGasPrice` that is dependent on the confirmation level of the `XMsg`. For example, an `XMsg` that uses the `latest` confirmation level can be charged a 10% premium, while an `XMsg` that uses the `finalized` confirmation level can be charged a 20% premium since the delay between calling `xcall()` and `xsubmit()` is longer.

OM2-04	Lack Of Support For Non-EVM Data Pricing		
Asset	`FeeOracleV1.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`FeeOracleV1.feeFor()` does not support rollups that use blobs as opposed to EVM calldata to post data, and hence will overprice the fee for data availability.

Since the Ethereum Cancun upgrade, most rollups that previously used Ethereum calldata for data availability have switched to Ethereum blobs. Blobs have their own gas fee market, which is separate from the execution layer's gas fee market.

The `FeeOracleV1.feeFor()` function assumes that the destination chain uses EVM calldata to post data, and calculates the cost of data availability based on the execution gas price of the EVM chain that the rollup posts data to.

```
IFeeOracleV1.ChainFeeParams storage dataP = _feeParams[execP.postsTo];

// ...

// @audit dataP.gasPrice represents EVM execution gas price, e.g. ETH L1 gas price
uint256 dataGasPrice = dataP.gasPrice * dataP.toNativeRate / CONVERSION_RATE_DENOM;

// 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
// @audit Calldata costs 16 gas per non-zero byte
uint256 dataGas = data.length * 16;
```

This means that the estimated data cost calculated in `FeeOracleV1.feeFor()` will be substantially higher than the actual cost, since blobs have different gas costs and pricing dynamics compared to calldata that result to lower gas fees for rollup users.

The current fee calculation mechanism is also inflexible in that it does not support destination chains that use alternative data availability services that aren't Ethereum blobs, such as Celestia or EigenDA.

Recommendations

Instead of using another EVM chain's gas price and calldata to estimate the cost of data availability, create a new `struct` that stores the gas price and also gas-per-byte cost for data availability. An example is shown below:

```
struct DataCostParams {
    uint256 gasPrice;
    uint256 gasPerByte;
}
```

Replace `ChainFeeParams.postsTo` with `ChainFeeParams.dataCostId`, such that each destination chain can point to a `DataCostParams` struct that stores the gas price and gas-per-byte cost for a type of data availability service.

For example, multiple rollups like Optimism, Base, and Arbitrum can all use the same `ChainFeeParams.dataCostId` that points to the `DataCostParams` for Ethereum blobs. In this case, the `gasPerByte` for an Ethereum blob is `1` (calculation

shown below).

```
# This calculation assumes that the rollup uses the entire blob
BLOB_GAS_PER_BYTE = BLOB_SIZE_IN_BYTES / GAS_PER_BLOB
                  = (BYTES_PER_FIELD_ELEMENT * FIELD_ELEMENTS_PER_BLOB) / GAS_PER_BLOB
                  = (32 * 4096) / (2**17)
                  = 1
```

Keep in mind that this solution does not account for any data compression that may be applied to reduce the size of the data posted for data availability.

OM2-05	Unprotected Secret Files		
Asset	`halo/app/start.go, halo/app/privkey.go, lib/ethclient/jwt.go`		
Status	Open		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `JWT token`, validator `crypto.PrivKey` and `p2p.NodeKey` are read from unprotected files. In the function `Start()` on line [**107**], `loadPrivVal()` is called, where the validator state and private key are read as described and shown in following code segment.

```
func loadPrivVal(cfg Config) (*privval.FilePV, error) {
    cmtFile := cfg.Comet.PrivValidatorKeyFile()
    cmtExists := exists(cmtFile)

    if !cmtExists {
        return nil, errors.New("cometBFT priv validator key file is required", "comet_file", cmtFile)
    }

    key, err := LoadCometFilePV(cmtFile)
    if err != nil {
        return nil, err
    }

    state, err := loadCometPVState(cfg.Comet.PrivValidatorStateFile())
    if err != nil {
        return nil, err
    }

    // Create a new privval.FilePV with the loaded key and state.
    // This is a workaround for the fact that there is no other way
    // to set FilePVLastSignState filePath field.
    resp := privval.NewFilePV(key, "", cfg.Comet.PrivValidatorStateFile())
    resp.LastSignState.Step = state.Step
    resp.LastSignState.Round = state.Round
    resp.LastSignState.Height = state.Height
    resp.LastSignState.Signature = state.Signature
    resp.LastSignState.SignBytes = state.SignBytes

    return resp, nil
}
```

Similarly, in `Start()` line [**122**], `newEngineClient()` is called which on line [**314**] calls `LoadJWTHexFile()` shown in the following code segment.

```
func LoadJWTHexFile(file string) ([]byte, error) {
    jwtHex, err := os.ReadFile(file)
    if err != nil {
        return nil, errors.Wrap(err, "read jwt file")
    }

    jwtHex = bytes.TrimSpace(jwtHex)
    jwtHex = bytes.TrimPrefix(jwtHex, []byte("0x"))

    jwtBytes, err := hex.DecodeString(string(jwtHex))
    if err != nil {
        return nil, errors.Wrap(err, "decode jwt file")
    }

    return jwtBytes, nil
}
```

Same issue is present when calling `p2p.LoadOrGenNodeKey()` `newCometNode()` line [**217**] in `Start()` on line [**155**]. In case the node was compromised through another vulnerability, this could lead to a serious issue. As such it has been given the impact of medium and likelihood low.

Recommendations

At least restrictive permissions should be enforced on these files. If using docker, there is also an option of using `https://docs.docker.com/compose/use-secrets/` .

OM2-06	Bridge Fee Requirement Is Too Strict		
Asset	`OmniBridgeL1.sol & OmniBridgeNative.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `_bridge()` function in both L1 and native bridges checks for `msg.value == bridgeFee(payor, to, amount)` which is too strict and can cause bridging to fail if the fee changes before the transaction is executed.

`bridgeFee()` points to `FeeOracleV1.feeFor()`, which can return a different value if any of these variables change:

- The gas price on the execution chain
- The gas price on the chain that the destination chain posts data to
- The exchange rate between the native token of the source chain and the native token of the destination chain
- The exchange rate between the native token of the destination chain and the native token of the chain that the destination chain posts data to

If any of these variables changes leading to a different `bridgeFee()`, the `bridge()` function will revert.

Recommendations

Consider changing the condition to `msg.value >= bridgeFee(payor, to, amount)`.

OM2-07	Incorrect `OmniGasPump` ETH Quote		
Asset	`OmniGasPump.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `quote()` function incorrectly adds the scaled up `amtETH` back to `amtETH`.

```
function quote(uint256 amtOMNI) public view returns (uint256) {
    uint256 amtETH = _toEth(amtOMNI);

    // "undo" toll
    // @audit scaled up amtETH is added back to amtETH
    amtETH += (amtETH * TOLL_DENOM / (TOLL_DENOM - toll));

    // "undo" xcall fee
    return amtETH + xfee();
}
```

This leads to an `amtETH` that is higher than intended.

This issue has a low impact because `quote()` is a view function that is not called by any Omni contracts. However, integrators or XApps using this function can end up swapping more ETH than intended through `OmniGasPump`.

Recommendations

Assign the scaled value back to `amtETH` instead of adding it.

```
amtETH = (amtETH * TOLL_DENOM) / (TOLL_DENOM - toll);
```

OM2-08	Nested `XMsgs` Break `MsgContext`		
Asset	`OmniPortal.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A nested `XMsg` can be used to manipulate the current `_xmsg` such that it incorrectly portrays that there is no `MsgContext`.

After `OmniPortal` executes an `XMsg`, it deletes the `_xmsg` state variable.

```
// 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;
```

When nesting an `XMsg`, once the inner `XMsg` has finished executing, the `_xmsg` is deleted instead of being set back to the outer `XMsg`'s `MsgContext`, allowing an attacker to incorrectly portray that there is no `MsgContext`.

Note that the `_exec()` function does not allow user `XCalls` to the portal, however, this can be circumvented by sending the `XMsg` to a contract that calls `OmniPortal.xsubmit()`.

Recommendations

Instead of deleting `_xmsg` after each `XMsg` execution, store the previous `MsgContext` in memory and set `_xmsg` back to it after each `XMsg` execution.

OM2-09	Missing Validation Of `cchain.SDKValidator` Structure		
Asset	`lib/cchain/provider.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `cchainSDKValidator.ConsensusPubKey.Value` length is not checked. This could cause a panic in `ConsensusCmtAddr()` on line [**151**] in case the length of this byte slice is anything other than 33.
The call to `pk.Address()`, seen below, panics in case the length of `cosmosk1.PubKey` is not 33.

```
func (v SDKValidator) ConsensusCmtAddr() (cmtcrypto.Address, error) {
    pk := new(cosmosk1.PubKey)
    err := proto.Unmarshal(v.ConsensusPubkey.Value, pk)
    if err != nil {
        return nil, errors.Wrap(err, "unmarshal consensus pubkey")
    }

    return pk.Address(), nil
}
```

The issue has been assigned impact and likelihood low since it is called in `monitorOnce()` on line [**53**] after calling `ConsensusEthAddr()` which errors in the same case

Recommendations

Implement a check for the length before calling `pk.Address()` to prevent the possibility of a vulnerability from occurring in the future.

OM2-10	Validator Addresses Not Verified Against Signature		
Asset	`lib/xchain/abi.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In functions `SubmissionsToBinding()` line [**151**] and `SubmissionsFromBinding()` line [**100**] `sig.ValidatorAddress` and `sig.Signature` are assigned to `SigTuple` and `bindings.ValidatorSigTuple` respectively, without verifying the addresses against the signatures as shown in following code segments.

```
func SubmissionFromBinding(sub bindings.XSubmission, destChainID uint64) Submission {
    sigs := make([]SigTuple, 0, len(sub.Signatures))
    for _, sig := range sub.Signatures {
        sigs = append(sigs, SigTuple{
            ValidatorAddress: sig.ValidatorAddr,
            Signature:        Signature65(sig.Signature),
        })
    }
...
}
```

```
func SubmissionToBinding(sub Submission) bindings.XSubmission {
    // Sort the signatures by validator address to ensure deterministic ordering.
    sort.Slice(sub.Signatures, func(i, j int) bool {
        return sub.Signatures[i].ValidatorAddress.Cmp(sub.Signatures[j].ValidatorAddress) < 0
    })

    sigs := make([]bindings.ValidatorSigTuple, 0, len(sub.Signatures))
    for _, sig := range sub.Signatures {
        sigs = append(sigs, bindings.ValidatorSigTuple{
            ValidatorAddr: sig.ValidatorAddress,
            Signature:     sig.Signature[:],
        })
    }
...
}
```

This issue was given likelihood and impact low, as the signatures are verified on-chain, in `OmniPortal.sol`.

Recommendations

Implement signature verification before converting from bindings to submissions or vice versa.

OM2-11	Missing Storage Gap From `OmniPortalStorage`
Asset	`OmniPortalStorage.sol`
Status	Open
Rating	Informational

Description

The `OmniPortalStorage` contract does not have a storage gap to prevent storage slot collisions that may arise when upgrading `OmniPortal`.

This issue has an informational rating as `OmniPortalStorage` is the last contract in `OmniPortal`'s inheritance chain, and `OmniPortal` does not define any new state variables. However, it is still good practice to leave a storage gap to prevent storage slot collisions that may arise in the future if `OmniPortal` inherits from new contracts.

Recommendations

Add a storage gap at the end of the `OmniPortalStorage` contract.

OM2-12	Incorrect `XCall` Condition In `XAppBase`
Asset	`XAppBase.sol`
Status	Open
Rating	Informational

Description

The `require()` statement in `xcall()` checks for `msg.value` which can be incorrect when any value is sent or reserved for app fees.

The following check in `xcall()` ensures that there is enough native tokens to pay for the `XCall` fee:

```
require(address(this).balance >= fee || msg.value >= fee, "XApp: insufficient funds");
```

Checking `msg.value >= fee` can be incorrect if any value is sent to another address before calling `xcall()`, or if the `XApp` contract takes its own fee in native tokens.

For example, a cross-L2 ETH bridge may take its own app fee in ETH when bridging from Optimism to Arbitrum. If the user sends enough ETH to cover the `XCall` fee but not enough for the app fee, the `XCall` will still succeed and app fee is not applied.

This issue has an informational rating as it can be mitigated if the `XApp` developer is aware of this behaviour and correctly checks for `msg.value >= xcallFee + appFee`.

Recommendations

Instead of checking for `msg.value >= fee`, allow the `XApp` contract to take an `amtForFee` parameter in `xcall()` and check for `amtForFee >= fee`.

The `XApp` contract can deduct any app fees or transferred value from `msg.value` before calling `xcall()`.

OM2-13	Return `nil` Both For Error And `abci.ValidatorUpdate`
Asset	`halo/valsync/keeper/keeper.go`
Status	Open
Rating	Informational

Description

The `processAttested()` function returns a `nil` value both for an error and for the `abci.ValidatorUpdate` on line [302] and line [316] as can be seen in the following code segment.

```
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.
    }
...
```

This has been given informational severity as there is no available exploit path currently for it, but could lead to easy coding mistakes and a potential vulnerability,

Recommendations

Return custom error when there is no validator updates and handle it specifically when calling the function.

OM2-14	Miscellaneous General Comments
Asset	All contracts
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Gas Optimisations**

   Related Asset(s): Quorum.sol

   The `verify()` function declares `prev` as `memory` even though it is not modified in the function.

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

    if (i > 0) {
        XTypes.SigTuple memory prev = sigs[i - 1];
        // ...
    }
    // ...
}
```

   Change the declaration of `prev` from `memory` to `calldata` to save gas.

2. **Typos In Natspec**

   Related Asset(s): OmniPortalStorage.sol, XTypes.sol, IOmniPortal.sol, OmniPortal.sol

   There are several instances in the codebase with incorrect Natspec comments:

   (a) The comment in `OmniPortalStorage` on line [**80**] reads:

```
/**
* @notice Offset of the last outbound XMsg that was sent to destChainId in shardId
*         Maps destChainId -> shardId -> offset.
*/
mapping(uint64 => mapping(uint64 => uint64)) public inXMsgOffset;
```

   Replace mentions of `destChainId` with `sourceChainId`.

   (b) The comment in `XTypes.sol` on line [**45**] mentions the `BlockHeader` struct field is `sourceChainId`, when it should be `consensusChainId`.

```
* @custom:field sourceChainId      Chain ID of the Omni consensus chain
```

   Replace `sourceChainId` with `consensusChainId`.

   (c) The comments at the following locations indicate that the first byte of `shardId` is the `confLevel`.

   i. `IOmniPortal.sol` on line [**36**]
   ii. `XTypes.sol` on line [**15**]
   iii. `OmniPortal.sol` on line [**135**]

   Correct the comments to indicate that `confLevel` is the last byte of `shardId`.

3. **Missing Input Validation**
   Related Asset(s): Staking.sol, OmniBridgeL1.sol, OmniGasPump.sol, OmniPortal.sol
   There are several instances in the codebase with missing input validation:

   (a) The following functions do not have zero address checks:
      i. `OmniBridgeL1.initialize()`: Does not check that `omni` is not the zero address.
      ii. `OmniGasPump.withdraw()`: Does not check that `to` is not the zero address.
      iii. `OmniGasPump.fillUp()`: Does not check that `recipient` is not the zero address.

   (b) `Staking.delegate()` does not check that `validator` is in the allow list if it is enabled.

   (c) `OmniPortal._setXMsgMinGasLimit()` and `_setXMsgMaxGasLimit()` do not check that `xmsgMinGasLimit <= xmsgMaxGasLimit` when the values are set. If `xmsgMinGasLimit` is accidentally set to a value greater than `xmsgMaxGasLimit` there would be no valid range and `xcall()` will always revert.

   Add the input validation checks to the relevant functions.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `brownie` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/tests-local/Staking.t.sol:StakingTest
[PASS] test_createValidator() (gas: 355171)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.50ms (313.86µs CPU time)

Ran 1 test for test/tests-local/OmniPortal.t.sol:OmniPortalTest
[PASS] test_initial() (gas: 35343)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.81ms (52.11µs CPU time)

Ran 4 tests for test/tests-local/OmniBridge.t.sol:OmniBridgeTest
[PASS] test_bridgeL1() (gas: 1051373)
[PASS] test_bridgeNative() (gas: 1628776)
[PASS] test_claimNative() (gas: 1591098)
[FAIL: Bob's claimable balance is not bridgeAmount: 0 != 1000000000000000000] test_withdraw_LostFundsWhenPaused_Vuln() (gas:
      ↪  1269026)
Suite result: FAILED. 3 passed; 1 failed; 0 skipped; finished in 9.17ms (7.50ms CPU time)

Ran 1 test for test/tests-local/OmniGasPump.t.sol:OmniGasPumpTest
[FAIL: Amount from quote does not match actual amount from fillUp: 15509958328129299 !~= 8163135962173315 (max delta:
      ↪  0.100000000000000%, real delta: 90.000000000000061%); counterexample:
      ↪  calldata=0x5d7d8c2f0000000000000000000000000000000000000000000000000001d005433474f83 args=[8163135962173315 [8.163e15]]]
      ↪  testFuzz_quote_IncorrectAmount_Vuln(uint256) (runs: 0, µ: 0, ~: 0)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 13.30ms (3.39ms CPU time)

Ran 4 test suites in 13.99ms (35.78ms CPU time): 5 tests passed, 2 failed, 0 skipped (7 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

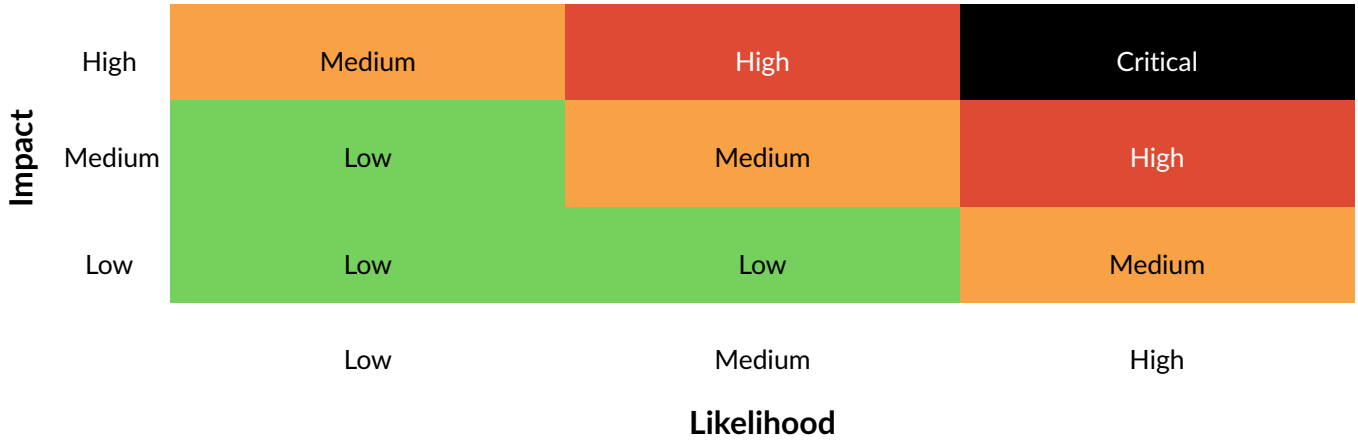

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.