Layer 1 Assessment
Story

HALBORN

Layer 1 Assessment - Story

Prepared by: HALBORN

Last Updated 12/12/2024

Date of Engagement by: November 7th, 2024 - December 6th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	1	2	2	0	3

TABLE OF CONTENTS

1. Introduction

`Story` engaged Halborn to conduct a security assessment on their Cosmos project, beginning on November 07th, 2024, and ending on December 06th, 2024. The security assessment was scoped to the smart contracts provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn assigned one full-time security engineer to assessment the security of the merge requests. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the **Golang components** and **Smart Contracts** operate as intended.
- Identify potential security issues with the Cosmos application and Smart Contracts in scope.

In summary, `Halborn` identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Story team`. The main recommendations were the following:

- `Implement compatibility with common features in Geth such as blobs.`
- `Charge fees accordingly to prevent from DoS attacks or trash transactions.`
- `Proper error handling implementations.`
- `Use algorithms applied in CometBFT.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the custom modules. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of structures and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment :

- Research into architecture and purpose.
- Static Analysis of security for scoped repository, and imported functions. (e.g., `staticcheck`, `gosec`, `unconvert`, `codeql`, `ineffassign` and `semgrep`)
- Manual Assessment for discovering security vulnerabilities on the codebase.
- Ensuring the correctness of the codebase.
- Dynamic Analysis of files and modules in scope.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: story

(b) Assessed Commit ID: e6d2d51

(c) Items in scope:

- /evmengine/keeper/abci.go
- /evmengine/keeper/db.go
- /evmengine/keeper/evmmsgs.go
- /evmengine/keeper/genesis.go
- /evmengine/keeper/helpers.go
- /evmengine/keeper/hooks.go
- /evmengine/keeper/keeper.go
- /evmengine/keeper/msg_server.go
- /evmengine/keeper/params.go
- /evmengine/keeper/proposal_server.go
- /evmengine/keeper/ubi.go
- /evmengine/keeper/upgrades.go
- /evmengine/module/depinject.go
- /evmengine/module/module.go
- /evmengine/types/codec.go
- /evmengine/types/cpayload.go
- /evmengine/types/events.go
- /evmengine/types/expected_keepers.go
- /evmengine/types/genesis.go
- /evmengine/types/keys.go
- /evmengine/types/params.go
- /evmengine/types/tx.go
- /evmengine/types/ubi_contract.go
- /evmengine/types/upgrade_contract.go
- /evmstaking/keeper/abci.go
- /evmstaking/keeper/delegator_address.go
- /evmstaking/keeper/deposit.go
- /evmstaking/keeper/genesis.go
- /evmstaking/keeper/grpc_query.go
- /evmstaking/keeper/keeper.go
- /evmstaking/keeper/keys.go
- /evmstaking/keeper/params.go
- /evmstaking/keeper/redelegation.go
- /evmstaking/keeper/reward_queue.go
- /evmstaking/keeper/singularity.go
- /evmstaking/keeper/staking_queue.go
- /evmstaking/keeper/ubi.go
- /evmstaking/keeper/unjail.go
- /evmstaking/keeper/update_commission.go
- /evmstaking/keeper/utils.go
- /evmstaking/keeper/validator.go
- /evmstaking/keeper/withdrawal_queue.go
- /evmstaking/keeper/withdraw.go
- /evmstaking/module/depinject.go

- /evmstaking/module/module.go
- /evmstaking/types/codec.go
- /evmstaking/types/events.go
- /evmstaking/types/expected_keepers.go
- /evmstaking/types/genesis.go
- /evmstaking/types/keys.go
- /evmstaking/types/params.go
- /evmstaking/types/staking_contract.go
- /evmstaking/types/withdraw.go
- /mint/keeper/abci.go
- /mint/keeper/genesis.go
- /mint/keeper/grpc_query.go
- /mint/keeper/keeper.go
- /mint/keeper/params.go
- /mint/module/depinject.go
- /mint/module/module.go
- /mint/types/codec.go
- /mint/types/events.go
- /mint/types/expected_keepers.go
- /mint/types/genesis.go
- /mint/types/keys.go
- /mint/types/params.go

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: story

(b) Assessed Commit ID: e6d2d51

(c) Items in scope:

- /IPTokenStaking.sol
- /PubKeyVerifier.sol
- /UBIPool.sol
- /UpgradeEntrypoint.sol

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: story

(b) Assessed Commit ID: e6d2d51

(c) Items in scope:

- /abci.go
- /app_config.go
- /app.go
- /cmtlog.go

- /keepers/types.go
- /privkey.go
- ./prouter.go
- /sdklog.go
- /start.go
- /upgrades/historical.go
- /upgrades/types.go
- /upgrades/v0_12_1/constants.go
- /upgrades/v0_12_1/upgrades.go
- /upgrades.go

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: story

(b) Assessed Commit ID: e6d2d51

(c) Items in scope:

- /queue.go

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 10a0ba3
- https://github.com/piplabs/story/pull/409/commits/4e0bc3908b808c70b3673652b74f99d1a9256fb8#diff-c14104289072948d084ab7968e23e51935073fe24f4a6169b30c54406a7fcd4f
- 596907d
- https://github.com/piplabs/story/pull/402
- https://github.com/piplabs/story/pull/419

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	2	2	0	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCOMPATIBILITY WITH BLOBS LEAD TO HALT OF BLOCK PROCESSING	CRITICAL	FUTURE RELEASE - 12/11/2024
MISSING CHARGEFEE MODIFIER ALLOWS SPAMMING THE CL	HIGH	SOLVED - 11/23/2024
MISMATCH BETWEEN PROPOSER SELECTION ALGORITHM	HIGH	SOLVED - 12/10/2024
POTENTIAL NON-DETERMINISM ISSUE IN FINALIZEBLOCK	MEDIUM	SOLVED - 11/25/2024
LACK OF PUBKEY INTEGRITY CHECKS	MEDIUM	SOLVED - 12/04/2024
FLOATING PRAGMA	INFORMATIONAL	SOLVED - 12/11/2024
USE OF CUSTOM ERRORS	INFORMATIONAL	ACKNOWLEDGED - 12/09/2024
CONTRACTS PAUSE FEATURE MISSING	INFORMATIONAL	ACKNOWLEDGED - 12/10/2024

7. FINDINGS & TECH DETAILS

7.1 INCOMPATIBILITY WITH BLOBS LEAD TO HALT OF BLOCK PROCESSING

// CRITICAL

Description

The `evmengine` module is responsible for interacting with Geth's APIs (`forkchoiceUpdatedV3` and `getPayloadV3`) to construct optimistic blocks based on transactions occurring in Geth. These APIs, in their version V3, include compatibility with Ethereum blobs introduced by the relevant EIPs (e.g., `EIP-4844`). However, there is a critical issue in the implementation of the `pushPayload` function, which always sets `emptyVersionHashes` to an empty array of hashes when calling Geth's `NewPayloadV3` API.

https://github.com/piplabs/story/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/client/x/evmengine/keeper/proposal_server.go#L33-L53

```
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
        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") // 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
        log.Warn(ctx, "Can't properly verifying proposal: evm syncing", err,
            "payload_height", payload.Number)
    }

    return true, nil // We are done, don't retry.
})
if err != nil {
    return nil, err
}
```

The `pushPayload` function sets `emptyVersionHashes` to an empty array (`make([]common.Hash, 0)`), regardless of whether the payload includes blob transactions. This causes a mismatch between the blob hashes in the payload and the expected versioned hashes during validation in Geth's `ExecutableDataToBlockNoHash`.

https://github.com/piplabs/story/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/client/x/evmengine/keeper/msg_server.go#L173-L189

```
func pushPayload(ctx context.Context, engineCl ethclient.EngineClient, payload engine.ExecutableDa
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    appHash := common.BytesToHash(sdkCtx.BlockHeader().AppHash)
    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
}
```

This behavior leads to a failure when the `ExecutableDataToBlockNoHash` function is invoked within Geth. Specifically, this function validates that the blob hashes used in the payload match the `versionedHashes` provided. Since `emptyVersionHashes` is always empty, any transaction containing blobs will fail validation, resulting in an error. Consequently, the Cosmos `evmengine` cannot process such transactions, preventing block verification and generation on the chain.

https://github.com/ethereum/go-ethereum/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/beacon/engine/types.go#L245-L251

```
var blobHashes = make([]common.Hash, 0, len(txs))
for _, tx := range txs {
    blobHashes = append(blobHashes, tx.BlobHashes()...)
}
if len(blobHashes) != len(versionedHashes) {
    return nil, fmt.Errorf("invalid number of versionedHashes: %v blobHashes: %v", versionedHashes
```

Proof of Concept

After sending a transaction with a blob attached, the chain halts with the following output:

```
24-11-25 13:40:17.399 INFO 👾 ABCI call: ProcessProposal            height=83 proposer=d269965
24-11-25 13:40:17.400 ERRO Rejecting process proposal               err="execute message: invalid payload, rejecting proposal: payload invalid" validation_err="invalid number of versionedHashes: [] blobHashes: [0x
016d0309f21937f8bf717228adae8e58d8b02db583bb1503f334f0bd9dd637af]" last_valid_hash=nil stacktrace="[errors.go:14 msg_server.go:223 proposal_server.go:41 helpers.go:30 proposal_server.go:33 tx.pb.go:290 msg_service
_router.go:175 tx.pb.go:292 msg_service_router.go:198 prouter.go:74 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:1695]"
24-11-25 13:40:17.400 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=83 round=0 err=<nil>
24-11-25 13:40:18.417 INFO 👾 ABCI call: PrepareProposal            height=83 proposer=d269965
24-11-25 13:40:18.417 INFO Submit new EVM payload                   timestamp="2024-11-25 12:40:14.724996794 +0000 UTC" withdrawals=1 app_hash=0x26008b7110263c7bfa9b228bc4015da79bd31717117ecf4a11ff92f3a5aa973f
24-11-25 13:40:19.044 INFO Proposing new block                      height=83 execution_block_hash=0846f36 evm_events=0
24-11-25 13:40:19.056 INFO 👾 ABCI call: ProcessProposal            height=83 proposer=d269965
24-11-25 13:40:19.057 ERRO Rejecting process proposal               err="execute message: invalid payload, rejecting proposal: payload invalid" validation_err="invalid number of versionedHashes: [] blobHashes: [0x
016d0309f21937f8bf717228adae8e58d8b02db583bb1503f334f0bd9dd637af]" last_valid_hash=nil stacktrace="[errors.go:14 msg_server.go:223 proposal_server.go:41 helpers.go:30 proposal_server.go:33 tx.pb.go:290 msg_service
_router.go:175 tx.pb.go:292 msg_service_router.go:198 prouter.go:74 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:1695]"
24-11-25 13:40:19.058 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=83 round=1 err=<nil>
24-11-25 13:40:20.574 INFO 👾 ABCI call: PrepareProposal            height=83 proposer=d269965
24-11-25 13:40:20.575 INFO Submit new EVM payload                   timestamp="2024-11-25 12:40:14.724996794 +0000 UTC" withdrawals=1 app_hash=0x26008b7110263c7bfa9b228bc4015da79bd31717117ecf4a11ff92f3a5aa973f
24-11-25 13:40:21.225 INFO Proposing new block                      height=83 execution_block_hash=0846f36 evm_events=0
24-11-25 13:40:21.238 INFO 👾 ABCI call: ProcessProposal            height=83 proposer=d269965
24-11-25 13:40:21.239 ERRO Rejecting process proposal               err="execute message: invalid payload, rejecting proposal: payload invalid" validation_err="invalid number of versionedHashes: [] blobHashes: [0x
016d0309f21937f8bf717228adae8e58d8b02db583bb1503f334f0bd9dd637af]" last_valid_hash=nil stacktrace="[errors.go:14 msg_server.go:223 proposal_server.go:41 helpers.go:30 proposal_server.go:33 tx.pb.go:290 msg_service
_router.go:175 tx.pb.go:292 msg_service_router.go:198 prouter.go:74 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:1695]"
```

Thus, being unable to process further blocks.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (10.0)

Recommendation

It's recommended the following:

- Modify the `pushPayload` function to correctly populate the `versionedHashes` parameter with the blob hashes associated with the transactions in the payload.
- This requires extracting blob hashes from the payload's transactions and ensuring they are passed to Geth's `NewPayloadV3` API.

Remediation

FUTURE RELEASE: The **Story team** states that this issue can be fixed in future releases as it does not pose an immediate risk for the chain since the EL (Execution Layer) will continue disabling blobs by default.

7.2 MISSING CHARGEFEE MODIFIER ALLOWS SPAMMING THE CL

// HIGH

Description

The `removeOperator`, `redelegate` and `redelegateOnBehalf` **do not have** the `chargeFee` modifier. As none of them requires the user sending funds, it allows a malicious user to spam the CL by sending these events, which requires consumption from the CL.

Code Location

https://github.com/piplabs/story/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/contracts/src/protocol/IPTokenStaking.sol#L162

```
    function removeOperator(
        bytes calldata uncmpPubkey,
        address operator
    ) external verifyUncmpPubkeyWithExpectedAddress(uncmpPubkey, msg.sender)
```

https://github.com/piplabs/story/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/contracts/src/protocol/IPTokenStaking.sol#L423

```
    function redelegate(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpSrcPubkey,
        bytes calldata validatorUncmpDstPubkey,
        uint256 delegationId,
        uint256 amount
    )
        external
        verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey, msg.sender)
        verifyUncmpPubkey(validatorUncmpSrcPubkey)
        verifyUncmpPubkey(validatorUncmpDstPubkey)
```

https://github.com/piplabs/story/blob/e6d2d51550c3eff3f561f8d1b860888ea2bf8060/contracts/src/protocol/IPTokenStaking.sol#L446

```
    function redelegateOnBehalf(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpSrcPubkey,
        bytes calldata validatorUncmpDstPubkey,
        uint256 delegationId,
        uint256 amount
    )
        external
        verifyUncmpPubkey(delegatorUncmpPubkey)
        verifyUncmpPubkey(validatorUncmpSrcPubkey)
        verifyUncmpPubkey(validatorUncmpDstPubkey)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation

It's recommended to add the `chargeFee` modifier to them.

Remediation

SOLVED: The **Story team** fixed this issue as recommended.

Remediation Hash

https://github.com/piplabs/story/commit/10a0ba3c4acf6fc6adbd9c205d1d70ab98457dd7#diff-4c83cc9786fabdfcde64802ab8507b153977c3b31151c8d53b6df730d4b6a0fc

7.3 MISMATCH BETWEEN PROPOSER SELECTION ALGORITHM

// HIGH

Description

The function `isNextProposer` attempts to determine whether the local node is the next proposer for the upcoming block in a CometBFT consensus system. However, there is a fundamental issue with the logic: it incorrectly applies a simple round-robin algorithm to select the next proposer, while CometBFT uses a weighted round-robin algorithm based on validator voting power. This discrepancy can lead to incorrect proposer selection, which may cause the node to behave incorrectly in the consensus process, potentially leading to missed blocks or other consensus failures:

```
// isNextProposer returns true if the local node is the proposer
// for the next block. It also returns the next block height.
//
// Note that the validator set can change, so this is an optimistic check.
func (k *Keeper) isNextProposer(ctx context.Context, currentProposer []byte, currentHeight int64)
    // PostFinalize can be called during block replay (performed in newCometNode),
    // but cmtAPI is set only after newCometNode completes (see app.SetCometAPI), so a nil check i
    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.validatorAddr

    return isNextProposer, nil
}
```

The code uses a modulo operation `(nextIdx := int(idx+1) % len(valset.Validators))` to select the next proposer in a round-robin fashion. However, CometBFT uses a weighted round-robin algorithm that takes into account the voting power of each validator when selecting proposers. This oversight means that validators with higher voting power will not be appropriately prioritized, resulting in an incorrect proposer selection.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation

Modify the proposer selection logic to account for validator voting power in line with CometBFT's weighted round-robin algorithm, as it is explained in CometBFT's documentation.

Remediation

SOLVED: The **Story team** fixed this issue by using the exposed validators functions to check if the local node is the next (predicted) proposer.

Remediation Hash

https://github.com/piplabs/story/pull/409/commits/4e0bc3908b808c70b3673652b74f99d1a9256fb8#diff-c14104289072948d084ab7968e23e51935073fe24f4a6169b30c54406a7fcd4f

References

https://docs.cometbft.com/v0.38/spec/consensus/proposer-selection

7.4 POTENTIAL NON-DETERMINISM ISSUE IN FINALIZEBLOCK

// MEDIUM

Description

There is a possible non-determinism issue in `FinalizeBlock`. That function must be deterministic per the CometBFT specs. However, there is a call path `postFinalize -> isNextProposer -> Validators -> cmtAPI` that can incurr in a non-deterministic scenario:

https://github.com/piplabs/story/blob/e6d2d5155Oc3eff3f561f8d1b860888ea2bf8060/client/app/abci.go#L114

```
// FinalizeBlock calls BeginBlock -> DeliverTx (for all txs) -> EndBlock.
func (l abciWrapper) FinalizeBlock(ctx context.Context, req *abci.RequestFinalizeBlock) (*abci.Res

    ...

    if err := l.postFinalize(sdkCtx); err != nil {
        log.Error(ctx, "PostFinalize callback failed [BUG]", err, "height", req.Height)
        return resp, err
    }

        ...
```

https://github.com/piplabs/story/blob/e6d2d5155Oc3eff3f561f8d1b860888ea2bf8060/client/x/evmengine/keeper/abci.go#L189

```
// PostFinalize is called by our custom ABCI wrapper after a block is finalized.
// It starts an optimistic build if enabled and if we are the next proposer.
//
// This custom ABCI callback is used since we need to trigger optimistic builds
// immediately after FinalizeBlock with the latest app hash
// which isn't available from cosmosSDK otherwise.
func (k *Keeper) PostFinalize(ctx sdk.Context) error {

    ...

    // 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")
    }

        ...
```

https://github.com/piplabs/story/blob/e6d2d5155Oc3eff3f561f8d1b860888ea2bf8060/client/x/evmengine/keeper/keeper.go#L191

```
// isNextProposer returns true if the local node is the proposer
// for the next block. It also returns the next block height.
//
// Note that the validator set can change, so this is an optimistic check.
func (k *Keeper) isNextProposer(ctx context.Context, currentProposer []byte, currentHeight int64)

    ...
```

```
    valset, ok, err := k.cmtAPI.Validators(ctx, currentHeight)
    if err != nil {
        return false, err
    }

        ...
```

The reason why it can lead to non-determinism is because the call to the CometBFT API is done through an RPC, which is a network connection, which by definition is non-deterministic between nodes (as some can revert due to timeout, network errors or go on flawlessly). That makes it possible that, under the same input, some nodes would go on whilst others would crash, as the errors are not handled in the `FinalizeBlock` context but rather bubbled up to the caller, leading to a chain split and the consensus would be broken.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Ignore any error returned from `PostFinalize` in `FinalizeBlock`.

Remediation

SOLVED: The **Story team** fixed this issue by skipping optimistic builds if any non-determinism error was encountered.

Remediation Hash

https://github.com/piplabs/story/commit/596907db4987212ab59c8d7f5019b86157134dfa

7.5 LACK OF PUBKEY INTEGRITY CHECKS

// MEDIUM

Description

The function `UncmpPubKeyToCmpPubKey` is responsible for converting uncompressed public keys into their compressed format. While it correctly handles the conversion by extracting the `x` and `y` coordinates and determining the prefix based on the parity of `y`, it does not validate whether the provided public key lies on the elliptic curve. This omission can lead to unexpected errors or security vulnerabilities when other modules, such as `evmstaking`, attempt to use these keys.

https://github.com/piplabs/story/blob/e6d2d5155Oc3eff3f561f8d1b86O888ea2bf8O6O/client/x/evmstaking/keeper/keys.go#L14-L35

```
func UncmpPubKeyToCmpPubKey(uncmpPubKey []byte) ([]byte, error) {
    if len(uncmpPubKey) != 65 || uncmpPubKey[0] != 0x04 {
        return nil, errors.New("invalid uncompressed public key length or format")
    }

    // Extract the x and y coordinates
    x := new(big.Int).SetBytes(uncmpPubKey[1:33])
    y := new(big.Int).SetBytes(uncmpPubKey[33:])

    // Determine the prefix based on the parity of y
    prefix := byte(0x02) // Even y
    if y.Bit(0) == 1 {
        prefix = 0x03 // Odd y
    }

    // Construct the compressed public key
    compressedPubKey := make([]byte, 33)
    compressedPubKey[0] = prefix
    copy(compressedPubKey[1:], x.Bytes())

    return compressedPubKey, nil
}
```

As it can be seen above, the function does not verify whether the `x` and `y` coordinates of the uncompressed public key satisfy the elliptic curve equation. Public keys that do not lie on the curve are invalid and can cause cryptographic operations to fail unexpectedly in downstream modules. Since this function is used by other modules, any invalid key processed here could lead to cascading failures or unexpected behavior in those modules.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (5.0)

Recommendation

It is recommended to validate whether the used points are in the described curve in order to discard incorrect keys.

Remediation

SOLVED: The **Story team** solved these issues as it was recommended.

Remediation Hash

https://github.com/piplabs/story/pull/402

7.6 FLOATING PRAGMA

// INFORMATIONAL

Description

Smart contracts in scoped folder use the floating pragma `>=0.8.23`. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a `pragma` version too new which has not been extensively tested.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider locking the pragma version with known bugs for the compiler version by removing the `>=` symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation

SOLVED: The **Story team** fixed this issue by setting `0.8.23` as fixed pragma version.

Remediation Hash

https://github.com/piplabs/story/pull/419

7.7 USE OF CUSTOM ERRORS

// INFORMATIONAL

Description

Failed operations in several contracts are reverted with an accompanying message selected from a set of hard-coded strings.

In the `EVM`, emitting a hard-coded string in an error message costs **~50** more gas than emitting a custom error. Additionally, hard-coded strings increase the gas required to deploy the contract.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It's recommended to use custom errors instead of hardcoded strings.

Remediation

ACKNOWLEDGED: The **Story team** acknowledged this issue by stating the following:

> Acknowledged, deployment costs are not an issue due to those contracts being predeploys. We will consider using custom errors for mainnet.

7.8 CONTRACTS PAUSE FEATURE MISSING

// INFORMATIONAL

Description

It was identified that no high-privileged user can pause any of the scoped contracts. In the event of a security incident, the owner would not be able to stop any plausible malicious actions. Pausing the contract can also lead to more considered decisions.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended including the pause feature in the contracts.

Remediation

ACKNOWLEDGED: The **Story team** acknowledged this issue.
