OMNI NETWORK

Omni Portal

Security Assessment Report

Version: 2.1

July, 2024

Contents

Introduction **2**
Disclaimer . . . 2
Document Structure . . . 2
Overview . . . 2

Security Assessment Summary **3**
Scope . . . 3
Approach . . . 3
Coverage Limitations . . . 4
Findings Summary . . . 4

Detailed Findings **5**

Summary of Findings **6**
Insufficient Validation In `AddVotes()` and `VerifyVoteExtension()` . . . 7
Insufficient Validation In `evmengine::pushPayload()` . . . 10
Missing Address Validation In `VerifyVoteExtension()` . . . 12
`ProcessProposal()` Allows Multiples Of Each Transaction Type . . . 14
Old Validator Set Can Sign Newly Supported Chains . . . 15
XMsg Execution Can Be Halted Or Delayed . . . 17
XCalls To Unsupported Chains Can Break New XStreams . . . 19
Gas Buffer Before External Call Is Insufficient . . . 20
Gas Griefing XMsg Execution With Return Bombs . . . 22
Relayers Can Be Griefed With `xsubmit()` Reentrancy . . . 24
System Call Execution Without `_execSys()` . . . 25
Index Out Of Bounds Panic In `GetMultiProof()` . . . 26
Mismatch Between XBlock And XMsg `sourceChainId` . . . 28
Cleartext Storage Of Sensitive Information In Config File . . . 29
Vote Extensions May Occur On Non-Existent Chains . . . 30
Index Out Of Bounds Panic In `PubKeyToBytes64()` . . . 31
Admin Can Fail XMsg Execution By Updating `xmsgMaxGasLimit` . . . 32
Extra Native Token Amount Is Not Refunded . . . 33
Usage of Deprecated Dependency Functions . . . 34
Lack Of `stateJSON` Validation When Loading From File . . . 35
`nil` Pointer Deference Panics In `AddVotes()`, `Add()` and `addOne()` . . . 36
Unaddressed TODO Comments . . . 37
Additional Chain Height And Header Checks Required . . . 38
No Linear Search Data Set Restrictions . . . 39
Lack Of `Validators` Array Size Checks . . . 40
Miscellaneous General Comments . . . 41

A Test Suite **43**

B Vulnerability Severity Classification **44**

Introduction

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

Disclaimer

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

Document Structure

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

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

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

Overview

The `OmniPortal` facilitates cross-chain smart contract calls between the Omni Chain and EVM-compatible Ethereum rollups.

The cross-chain communication is secured by Omni Chain's consensus layer dPoS validators, which monitor for finalised blocks containing XMsg events and package them into XBlocks.

Once these XBlocks have been attested to by a quorum of 2/3rds of the validator set, they are approved. Relayers submit XMsgs from approved XBlocks to `OmniPortals` on destination chains to execute cross-chain messages.

Security Assessment Summary

Scope

The scope of this time-boxed review was strictly limited to files at commit 797bf4b.

Specifically, for the on-chain part of the review, the scope included the following files:

- `protocol/Omniportal.sol`
- `libraries/Quorum.sol`
- `libraries/Secp256k1.sol`
- `libraries/XBlockMerkleProof.sol`
- `libraries/XTypes.sol`
- `protocol/OmniStake.sol`

For the off-chain part of the review, the scope included the following directories:

- `halo/attest/`
- `halo/evmengine/`
- `halo/evmstaking/`
- `halo/valsync/`
- `lib/xchain/`
- `lib/merkle/`
- `lib/cchain/`
- `halo/` (rest of codebase)

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

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

For the Go libraries and modules, the review focused on internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime and use of the Ethereum protocol.

To support this review, the testing team used the following automated testing tools:

For Solidity based smart contracts:

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

For Golang code:

- `golangci-lint`: `https://github.com/golangci/golangci-lint`
- `semgrep-go`: `https://github.com/dgryski/semgrep-go`
- `go-geiger`: `https://github.com/jlauinger/go-geiger`
- native go fuzzing: `https://go.dev/doc/fuzz/`

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 26 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.
- High: 4 issues.
- Medium: 5 issues.
- Low: 7 issues.
- Informational: 6 issues.

Detailed Findings

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

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

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

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

Summary of Findings

ID	Description	Severity	Status
OMP-01	Insufficient Validation In `AddVotes()` and `VerifyVoteExtension()`	Critical	Resolved
OMP-02	Insufficient Validation In `evmengine::pushPayload()`	Critical	Resolved
OMP-03	Missing Address Validation In `VerifyVoteExtension()`	Critical	Resolved
OMP-04	`ProcessProposal()` Allows Multiples Of Each Transaction Type	Critical	Resolved
OMP-05	Old Validator Set Can Sign Newly Supported Chains	High	Resolved
OMP-06	XMsg Execution Can Be Halted Or Delayed	High	Resolved
OMP-07	XCalls To Unsupported Chains Can Break New XStreams	High	Resolved
OMP-08	Gas Buffer Before External Call Is Insufficient	High	Resolved
OMP-09	Gas Griefing XMsg Execution With Return Bombs	Medium	Resolved
OMP-10	Relayers Can Be Griefed With `xsubmit()` Reentrancy	Medium	Resolved
OMP-11	System Call Execution Without `_execSys()`	Medium	Resolved
OMP-12	Index Out Of Bounds Panic In `GetMultiProof()`	Medium	Resolved
OMP-13	Mismatch Between XBlock And XMsg `sourceChainId`	Medium	Resolved
OMP-14	Cleartext Storage Of Sensitive Information In Config File	Low	Resolved
OMP-15	Vote Extensions May Occur On Non-Existent Chains	Low	Resolved
OMP-16	Index Out Of Bounds Panic In `PubKeyToBytes64()`	Low	Resolved
OMP-17	Admin Can Fail XMsg Execution By Updating `xmsgMaxGasLimit`	Low	Resolved
OMP-18	Extra Native Token Amount Is Not Refunded	Low	Closed
OMP-19	Usage of Deprecated Dependency Functions	Low	Resolved
OMP-20	Lack Of `stateJSON` Validation When Loading From File	Low	Resolved
OMP-21	`nil` Pointer Deference Panics In `AddVotes()`, `Add()` and `addOne()`	Informational	Resolved
OMP-22	Unaddressed TODO Comments	Informational	Resolved
OMP-23	Additional Chain Height And Header Checks Required	Informational	Closed
OMP-24	No Linear Search Data Set Restrictions	Informational	Resolved
OMP-25	Lack Of `Validators` Array Size Checks	Informational	Resolved
OMP-26	Miscellaneous General Comments	Informational	Resolved

OMP-01	Insufficient Validation In `AddVotes()` and `VerifyVoteExtension()`		
Asset	`halo/attest/keeper/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The functions `keeper.go::VerifyVoteExtension()` and `proposal_server.go::AddVotes()` lack validation to ensure that each vote's `BlockHeader` is contained within the `AttestationRoot` tree. Additionally, `AddVotes()` does not validate the signature of each vote.

halo/attest/types/tx.go

```
26  func (v *Vote) Verify() error {
        if v == nil {
28          return errors.New("nil attestation")
        }
30
        if err := v.BlockHeader.Verify(); err != nil {
32          return err
        }
34
        if v.Signature == nil {
36          return errors.New("nil signature")
        }
38
        if len(v.AttestationRoot) != len(common.Hash{}) {
40          return errors.New("invalid attestation root length")
        }
42
        if len(v.BlockHeader.Hash) != len(common.Hash{}) {
44          return errors.New("invalid block header hash length")
        }
46
        if len(v.Signature.Signature) != len(xchain.Signature65{}) {
48          return errors.New("invalid signature length")
        }
50
        if len(v.Signature.ValidatorAddress) != len(common.Address{}) {
52          return errors.New("invalid validator address length")
        }
54
        ok, err := k1util.Verify(
56          common.Address(v.Signature.ValidatorAddress),
            common.Hash(v.AttestationRoot), // @audit lacks checks to ensure v.BlockHeader is in this root
58          xchain.Signature65(v.Signature.Signature),
        )
60      if err != nil {
            return err
62      } else if !ok {
            return errors.New("invalid attestation signature")
64      }

66      return nil
    }
```

During a call to `VerifyVoteExtension()` the validation of a vote occurs in the function `Vote::Verify()`. However, no checks exist to ensure `vote.BlockHeader` is a leaf in the attestation root tree. A malicious node could create a vote which has a block header of a different block to the attestation root. So long as the signature is valid, the malicious

vote will pass `Verify()`.

As `AttestationRoot` is not verified to be a hash, it may be set to an arbitrary value. A limitation of ECDSA, is that valid signatures may be forged if an attacker is able to select an arbitrary value as the message hash, i.e. `AttestationRoot`. As an attacker can create random attestations roots they are able to generate valid signatures of other validators without knowledge of the private key.

Similarly, during `ProcessProposal()` the function `AddVotes()` is called for the proposal server. The validation occurs in the function `AggVote::Verify()`, but lacks checks to ensure each `AggVote` has a `BlockHeader` contained in the `AttestationRoot` tree.

Finally, `AggVote::verify()` does not validate the signature is a valid ECDSA. Therefore, an attacker may include malicious votes inside an aggregate vote without providing a valid signature.

halo/attest/types/tx.go

```
112 func (a *AggVote) Verify() error {
        if a == nil {
114         return errors.New("nil aggregate vote")
        }
116
        if err := a.BlockHeader.Verify(); err != nil {
118         return errors.Wrap(err, "block header")
        }
120
        if len(a.AttestationRoot) != len(common.Hash{}) { // @audit should ensure block header exists in this tree root
122         return errors.New("invalid attestation root length")
        }
124
        if len(a.Signatures) == 0 {
126         return errors.New("empty signatures")
        }
128
        for _, sig := range a.Signatures {
130         if err := sig.Verify(); err != nil { // @audit does not ensure `sig.ValidatorAddress` matches `sig.Signature`
                return errors.Wrap(err, "signature")
132         }
        }
134
        return nil
136 }
```

The impact is rated as high, as a malicious validator may create proposals which will be validated and later finalized but contain invalid votes in the `AddVotes()` transaction. Furthermore, the likelihood is rated as high as these attacks may be performed by any validator during `ExtendVote()` or when they are the leader and make a proposal.

Recommendations

The following patches will resolve the issues described above.

Each vote, whether aggregate or not should include a proof that `vote.Blockheader` / `aggVote.Blockheader` is a leaf node in the tree with root `vote.AttestationRoot` / `aggVote.AttestationRoot`. This will prevent creating malicious ECDSA message hashes to forge signatures. Additionally, it will prevent attaching signatures and attestation roots unrelated to the block header.

`SigTuple::Verify()` should take the `AttestationRoot` as input and verify each signature is a valid ECDSA signature for the `AtttestationRoot`.

Resolution

The recommendation has been implemented in PR #1252.

Furthermore, additional validation of votes was added in PR #1432 to prevent duplicate votes being sent or received.

OMP-02	Insufficient Validation In `evmengine::pushPayload()`		
Asset	`/halo/evmengine/keeper/msg_server.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

During the `ProcessProposal()` stage there is insufficient validation of each execution payload. The fields `Withdrawals`, `Timestamp` and `ParentHash` are not sufficiently validated, hence a malicious proposer may exploit the execution client.

halo/evmengine/keeper/msg_server.go

```go
161 func pushPayload(ctx context.Context, engineCl ethclient.EngineClient, msg *types.MsgExecutionPayload,
    ) (engine.ExecutableData, engine.PayloadStatusV1, error) {
163     var payload engine.ExecutableData
        if err := json.Unmarshal(msg.ExecutionPayload, &payload); err != nil {
165         return engine.ExecutableData{}, engine.PayloadStatusV1{}, errors.Wrap(err, "unmarshal payload")
        }
167
        // TODO(corver): Figure out what to use for BeaconBlockRoot.
169     var zeroBeaconBlockRoot common.Hash
        emptyVersionHashes := make([]common.Hash, 0) // Cannot use nil.
171
        // Push it back to the execution client (mark it as possible new head).
173     status, err := engineCl.NewPayloadV3(ctx, payload, emptyVersionHashes, &zeroBeaconBlockRoot) // @audit payload is not
            ↪ sufficiently validated before being sent to the execution client
        if err != nil {
175         return engine.ExecutableData{}, engine.PayloadStatusV1{}, errors.Wrap(err, "new payload")
        }
177
        return payload, status, nil
179 }
```

First, there is no validation that the field `payload.Withdrawals` is empty. Therefore, a malicious proposer could create an execution payload and set `Withdrawals` to include payments to their own address. During execution of the payload the specified funds would be minted to the proposers address.

Additionally, the field `payload.ParentHash` is not validated. Hence, a malicious node could set it to a value other than the current finalized head. If the parent hash points to a block earlier than the last finalized block the behaviour is undefined by the execution client and may result in a re-org of finalised blocks. If the parent hash is set after the current finalised head the result may be a chain that is disconnected as the status will be `Accepted` or `Syncing`.

Recommendations

To resolve the issues add validation to the following fields during `evmengine::pushPayload()`:

- `Withdrawals` should be empty.
- `ParentHash` is the current finalised head of the execution chain.
- `Timestamp` is within a sane window.
- `FeeRecipient` should be the burn address.

- `Random` should be the burn address.

Resolution

Additional validation of execution payloads has been added in PR #1248.

OMP-03	Missing Address Validation In `VerifyVoteExtension()`		
Asset	`halo/attest/keeper/keeper.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `VerifyVoteExtension()` does not ensure that the consensus address matches the validator address. That is `req.ValidatorAddress` matches `req.VoteExtension[i].Signature.ValidatorAddress`.

halo/attest/keeper/keeper.go

```
490 func (k *Keeper) VerifyVoteExtension(ctx sdk.Context, req *abci.RequestVerifyVoteExtension) (
        *abci.ResponseVerifyVoteExtension, error,
492 ) {
        respAccept := &abci.ResponseVerifyVoteExtension{
494         Status: abci.ResponseVerifyVoteExtension_ACCEPT,
        }
496     respReject := &abci.ResponseVerifyVoteExtension{
            Status: abci.ResponseVerifyVoteExtension_REJECT,
498     }

500     // Adding logging attributes to sdk context is a bit tricky
        ctx = ctx.WithContext(log.WithCtx(ctx, log.Hex7("validator", req.ValidatorAddress)))
502
        votes, ok, err := votesFromExtension(req.VoteExtension)
504     if err != nil {
            log.Warn(ctx, "Rejecting invalid vote extension", err)
506         return respReject, nil
        } else if !ok {
508         log.Info(ctx, "Accepting nil vote extension") // This can happen in some edge-cases.
            return respAccept, nil
510     } else if len(votes.Votes) > int(k.voteExtLimit) {
            log.Warn(ctx, "Rejecting vote extension exceeding limit", nil, "count", len(votes.Votes), "limit", k.voteExtLimit)
512         return respReject, nil
        }
514
        for _, vote := range votes.Votes {
516         if err := vote.Verify(); err != nil {
                log.Warn(ctx, "Rejecting invalid vote", err)
518             return respReject, nil
            }
520         if cmp, err := k.windowCompare(ctx, vote.BlockHeader.ChainId, vote.BlockHeader.Height); err != nil {
                return nil, errors.Wrap(err, "windower")
522         } else if cmp != 0 {
                log.Warn(ctx, "Rejecting out-of-window vote", nil, "cmp", cmp)
524             return respReject, nil
            }
526     }

528     return respAccept, nil
    }
```

Thus, a validator could submit votes from other nodes or addresses outside the current validator set.

The impact is rated as high as `PrepareProposal()` does not filter votes from outside the validator set, however `ProcessProposal()` will reject proposals with votes outside the validator set. Therefore, sending a malicious vote extensions with an address outside of the validator set will result in correctly operating nodes preparing invalid proposals, which stalls the chain.

Recommendations

Since `abci.RequestVerifyVoteExtension.ValidatorAddress` is a 20 byte Ethereum style address and `xchain.Vote.Signature.ValidatorAddress` is a 20 byte Bitcoin-style address, they are not comparable directly. Creating a look-up between the two address types for each validator would prevent replaying other node votes.

Resolution

A mitigation has been added in PR #1250. The mitigation ensures that a vote extensions validator address matches the consensus address.

OMP-04	`ProcessProposal()` Allows Multiples Of Each Transaction Type		
Asset	`halo/app/prouter.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Multiple `MsgExecutionPayload` or `MsgAddVotes` transactions may be included in a single proposal.

The function for processing proposals iterates through each transaction without performing validation this transaction type has not yet been executed. The following code snippet is taken from the function `makeProcessProposalHandler()`

halo/app/prouter.go

```
42  for _, rawTX := range req.Txs {
        tx, err := app.txConfig.TxDecoder()(rawTX)
44      if err != nil {
            return handleErr(ctx, errors.Wrap(err, "decode transaction"))
46      }

48      for _, msg := range tx.GetMsgs() {
            handler := router.Handler(msg)
50          if handler == nil {
                return handleErr(ctx, errors.New("msg handler not found",
52                  "msg_type", fmt.Sprintf("%T", msg),
                ))
54          }

56          _, err := handler(ctx, msg)
            if err != nil {
58              return handleErr(ctx, errors.Wrap(err, "execute message"))
            }
60      }
    }
```

Repeating `MsgExecutionPayload` allows multiple execution blocks to be added in a single proposal. If these blocks are set to fork each other i.e. they are at the same height and have the same parent hash. Then an error will occur while processing a finalized proposal, causing the chain to stall. The error occurs during `msg_server.go::AddVotes()` when `ForkchoiceUpdatedV3()` is called on the second execution payload as it re-orgs a finalized block.

The impact is high as it allows a malicious validator to stall the chain or include multiple execution blocks.

Recommendations

To resolve this issue ensure at most one of each transaction type is included in a proposal.

Resolution

The recommendation has been implemented in PR #1245.

OMP-05	Old Validator Set Can Sign Newly Supported Chains	
Asset	protocol/OmniPortal.sol	
Status	**Resolved:** See Resolution	
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When a new source chain is supported, the `attestationRoot` of `XMsgs` from that chain can be signed by an older `valSetId`.

The function `xsubmit()` checks that the `valSetId` that has signed the `attestationRoot` is equal to or newer than the last `valSetId` that signed the last `attestationRoot` of an `XSubmission` from the same source chain:

OmniPortal.sol

```
176 uint64 lastValSetId = inXStreamValidatorSetId[xsub.blockHeader.sourceChainId];

178  // check that the validator set is known and has non-zero power
     require(validatorSetTotalPower[valSetId] > 0, "OmniPortal: unknown val set");
180
     // check that the submission's validator set is the same as the last, or the next one
182  require(valSetId >= lastValSetId, "OmniPortal: old val set"); //@audit lastValSetId is zero for new or non-existent chains
```

However, when a new source chain is supported, the `inXStreamValidatorSetId` corresponding to this new chain would be pre-initialised to zero, allowing submissions from older validator sets to still be valid.

This makes every existing `OmniPortal` vulnerable to a long-range validator set attack where an older and exited validator set colludes to fabricate and sign false `attestationRoots` and XMsgs from the newly supported source chain.

Additionally, this issue also persists with chains that are not yet supported or non-existent, since `inXStreamValidatorSetId` is also zero for these chains.

Due to OMP-13, it would be possible for an old validator set to forge messages from any chain. The old validator set creates a `XSubmission` which has a non-existent `sourceChainId`. Therefore, `inXStreamValidatorSetId[xsub.blockHeader.sourceChainId]` is zero. Then the attacker adds a message with a different `sourceChainId`, say the Omni chain ID. This would allow them to make admin messages such as `addValidatorSet()`.

Recommendations

To resolve the issue, prevent submissions from source chains where `inXStreamValidatorSetId` is zero.

Since Solidity pre-initialises variables to zero, the `OmniPortal` contract would need to set the `inXStreamValidatorSetId` for newly supported source chains to the initial `valSetId` in the `initialize()` function.

This could be achieved by adding a function similar to `addValidatorSet()` that updates the `inXStreamValidatorSetId` of a chain. This function can also be called via a broadcasted system call.

Resolution

The issue has been resolved in PR #1133. The `OmniPortal` contract has now a new function `initSourceChain()` that can only be called by the `XRegistry` contract. This function sets the `inXStreamValidatorSetId` of the new `srcChainId` to `inXStreamValidatorSetId[omniChainId]`. and this latter is set to the `valSetId` during the initialization. Added to that, the `xsubmit()` function checks now that `inXStreamValidatorSetId` of the source Chain of the submission is greater than zero. Note that `valSetId` is not checked during initialization and it could be initialized to zero. So, we recommend adding a check to `valSetId` in the `initialize()`

Further updates have been added in PR #1212. The additional updates require the validator set to be within a constant `XSUB_VALSET_CUTOFF` of the most recent validator set. The constant is currently set to 10, thereby restricting long range attacks to the 10 most recent validator sets.

OMP-06	XMsg Execution Can Be Halted Or Delayed		
Asset	protocol/OmniPortal.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The `validatorSetId` is not included in each `XBlock` header and is an arbitrary input parameter in `xsubmit()` inside `XTypes.Submission`:

OmniPortal.sol

```
172 // validator set id for this submission
    uint64 valSetId = xsub.validatorSetId;
```

This means that it is possible for an `XSubmission` to provide a newer `validatorSetId` and still pass the quorum and merkle multi proof checks.

This allows the following attack to be possible:

1. The latest `XBlock` attested by `valSetId = 100` has just been approved (2/3 of signatures provided).
2. An attacker joins the validator set (growing the validator set) - there is now a `valSetId = 101` that is sent as a broadcast `XCall`.
3. The validator update is submitted to the `OmniPortal`.
4. The attacker only submits the first `XMsg` in the `XBlock` from step 1. They submit with `valSetId = 101` and if required, they sign the `attestationRoot` with their own validator offchain and include it in the submission so that the 2/3rd quorum is still met.
5. Due to the `lastValSetId` requirement in `xsubmit()`, the rest of the `XMsgs` in the `XBlock` and any subsequent blocks which have been signed but not processed on-chain cannot be submitted with `valSetId = 100`. They have to be submitted with `valSetId = 101`, and there is may not be enough power in the existing signatures for the 2/3rd quorum for the new `valSetId`.
6. The relayer has to wait until it's received enough signatures to pass the new quorum for `valSetId = 101`. The larger the attacker validator's power is, the longer this will take.

If the validator set has changed significantly in step 2 (>33% of the set's power has changed), then all available signatures in step 6 will not be enough to reach the 2/3rds quorum as the validators common to both sets will contribute to less than 66% of the total power. In this scenario, the `XStream` will not be able to process anymore subsequent `XMsgs` unless the current validator set signs all approved blocks that have been attested to by previous validator sets, which would require out-of-protocol social coordination.

Recommendations

A mitigation to the issue is to include the `validatorSetId` inside the `XBlock` header. This allows identification of which `validatorSetId` signed each `XBlock`.

Use this value in `XTypes.BlockHeader` instead of allowing the relayer to set an arbitrary `validatorSetId` value in their submission.

Note that this solution means that existing validators need to re-attest to pending `XBlocks` if the validator set changes, since the changed `validatorSetId` will also change the `XBlock` merkle root.

Resolution

The development team has fixed this issue in commit d3c9213 by changing the logic of the `xsubmit()` function. In fact, instead of requiring that the validator set id is increasing for each submission in a stream, this `xsubmit()` now requires that each submission uses a validator set id greater than a global minimum.

OMP-07	XCalls To Unsupported Chains Can Break New XStreams	
Asset	protocol/OmniPortal.sol	
Status	**Resolved:** See Resolution	
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `_xcall()` function does not check if `destChainId` is a supported destination chain by calling `isSupportedChain()`. Hence, it is possible to initiate `XMsgs` for unsupported chains and increase the `outXStreamOffset` of those unsupported streams.

If these chains are supported in the future, these `XMsgs` will have to be executed first before any current `XMsgs` can be executed due to the `inXStreamOffset` requirement starting from 1:

OmniPortal.sol

```
248 require(xmsg_.streamOffset == inXStreamOffset[xmsg_.sourceChainId] + 1, "OmniPortal: wrong streamOffset");
```

However, the `OmniPortal` on the newly-supported destination chain will start with a bigger `inXStreamOffset[omniCChainID]` value, as only the latest validator set will be added. Older validator sets which the older XBlocks have been signed by cannot be added due to `inXStreamOffset[omniCChainID]`.

To recover from this denial of service, the current validator set would need to sign the past `XBlock` and the relayer submits the `XMsg` with the current validator set, which requires social coordination.

Recommendations

Consider using the `isSupportedChain()` function in `_xcall()` to check if the destination chain is supported to prevent users from initiating `XMsgs` to unsupported destination chains.

Resolution

The development team has fixed this issue in PR #1116 by adding the check `require(isSupportedChain(destChainId), "OmniPortal: unsupported chain")` in the function `_xcall()`.

After reviewing the retesting commit d3c9213, the retesting team has noticed that this check is moved to the fucntion `xcall()` on line [125].

OMP-08	Gas Buffer Before External Call Is Insufficient		
Asset	`protocol/OmniPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Malicious relayers can still intentionally fail XMsgs by setting a gas limit that is just low enough.

Before executing a low-level call, the `_exec()` function uses a small gas buffer of 100 gas to protect against malicious relayers intentionally setting the gas limit just low enough to fail the last XMsg in a submission.

OmniPortal.sol

```
282 // require gasLeft is enough to execute the call. this protects against malicious relayers
    // purposefully setting gasLimit just low enough such that the last xmsg in a submission
284 // fails, despite it's sufficient gasLimit
    //
286 // We add a small buffer to account for the gas usage from here up until the call.
    // TODO: is buffer of 100 correct? Better more than less
288 require(gasLimit + 100 < gasleft(), "OmniPortal: gasLimit too low");
```

However, this buffer does not take into account the 1/64 rule introduced by EIP-150.

Due to the 1/64 rule, the maximum amount of gas that can be forwarded to the external call in `_exec(address,uint64,bytes)` is `gasleft() * 63 / 64`. This makes it possible for a malicious relayer to pass the gas buffer check and intentionally fail the last XMsg by not including enough gas, while still having enough gas after the external call to complete the XSubmission.

For example consider the case where there is 5,000,100 gas left just before a call requiring 5,000,000 gas. The amount that is forwarded to the contract is $5,000,100 * 63/64 = 4,921,973$ about 72,000 gas short.

Recommendations

Due to the 1/64 rule, a constant gas buffer cannot be used.

Instead, consider checking if there is enough `gasleft` taking into account the 1/64 rule, or checking after the call if enough gas was forwarded to the call. This article by Ronan Sandford details the pros and cons of each method.

As an example, OpenZeppelin's `MinimalForwarder` contract implements checks after the call.

Resolution

The recommendation has been implemented in PR #1082. The function `_exec(address,uint64,bytes)` implements the following check after the call:

OmniPortal.sol

```
290 uint256 gasLeftAfter = gasleft();

292 // Esnure relayer sent enough gas for the call
    // See
       ↪  https://github.com/OpenZeppelin/openzeppelin-contracts/blob/bd325d56b4c62c9c5c1aff048c37c6bb18ac0290/contracts/metatx/MinimalForwarder
294 if (gasLeftAfter <= gasLimit / 63) {
    // We could use invalid opcode to consume all gas and bubble-up the effects, since
296             // and emulate an "OutOfGas" exception
                assembly {
298                 invalid()
                }
300         }
```

OMP-09	Gas Griefing XMsg Execution With Return Bombs		
Asset	protocol/OmniPortal.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

It is possible to grief a relayer by making them spend a lot of gas to process return data from an `XCall`, such that it is no longer profitable for them to execute these `XMsgs`.

The `_exec(XTypes.Msg)` function checks if the return data from the `XCall` is too long to avoid spending too much gas emitting the `XReceipt`.

OmniPortal.sol

```
265 // empty error if success is true
    bytes memory error = success ? bytes("") : result;
267
    // if error is too long, return corresponding error code
269 if (error.length > xreceiptMaxErrorBytes) error = XRECEIPT_ERROR_EXCEEDS_MAX_BYTES;

271 emit XReceipt(xmsg_.sourceChainId, xmsg_.streamOffset, gasUsed, msg.sender, success, error);
```

However, this is still not entirely safe from a gas griefing attack with a return bomb, since the return data is still initially loaded into memory and returned in `_exec(address,uint64,bytes)`:

OmniPortal.sol

```
293 (bool success, bytes memory result) = to.call{ gas: gasLimit }(data);

295 gasUsed = gasUsed - gasleft();

297 return (success, result, gasUsed);
```

The following table demonstrates the `XMsg` gas limit required for different sizes of return bombs, as well as the amount of gas the relayer used to execute the `XMsg` and load the return data. A low `XMsg` gas limit and a high relayer gas use indicates that the attack is effective.

Return Bomb Size	XMsg Gas Limit	Relayer Gas Used For Execution & Loading Return Data
0 bytes	21,000 (`xmsgMinGasLimit = 21000`, actual usage is 199 gas)	26,160
10,000 bytes	21,000 (actual usage is 1320)	29,536
100,000 bytes	50,819	93,641
1,590,000 bytes (almost max size with `xmsgMaxGasLimit = 5,000,000`)	4,971,319	10,135,789

For bigger return bomb sizes, the relayer uses double the gas cost than what the attacker submits as their `XMsg` gas limit, indicating that the attack is effective.

Note that due to the different block sizes of rollups, it is possible for the largest return bomb to cause a complete DoS on an XStream if the rollup has a smaller block gas limit (less than 10 million gas). The values above do not take into account the rest of the logic in `xsubmit()` such as quorum and Merkle multi proof verification, which would also use up a dynamic amount of gas depending on the size of the validator set and Merkle multi proof.

Recommendations

To prevent return bomb attacks, consider checking the length of the return data before storing it in memory.

Nomad's ExcessivelySafeCall library provides the `excessivelySafeCall()` function that implements these checks to protect against return bombs.

Resolution

The development team has resolved this issue in PR #1099 by using the function `excessivelySafeCall()` as recommended.

OMP-10	Relayers Can Be Griefed With `xsubmit()` Reentrancy		
Asset	`protocol/OmniPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

An attacker can initiate an `XMsg` that calls a smart contract that reenters into `OmniPortal.xsubmit()` to submit subsequent `XMsgs` in the `XBlock`.

Since `XMsgs` from the same source chain need to be executed in order, this attack will cause the relayer's submission to fail if they have included any `XMsgs` after the attacker's, as the `XMsgs` would have already been executed.

This issue has a medium severity as the attacker can grief the relayer, causing them to spend lots of gas for reverting transactions. To do this, they can front-run the relayer with a transaction that posts the submission data to the attacker contract, so that it can be used to call `xsubmit()` inside the `XCall`. Front-running the relayer makes it impossible for them to simulate and detect the revert scenario before submitting their transaction onchain.

Recommendations

Avoid allowing reentrancy in `xsubmit()`. This can be achieved via OpenZeppelin's ReentrancyGuard.

Resolution

The `nonReentrant` modifier has been added to the function `xsubmit()` in PR #1074 as recommended.

OMP-11	System Call Execution Without `_execSys()`		
Asset	`protocol/OmniPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

According to `OmniPortal._execSys()`, a system call invokes the `OmniPortal` itself and is executed using the `_execSys()` function when `xmsg_.to == _VIRTUAL_PORTAL_ADDRESS`.

However, it is possible to initiate a system call without calling the `_execSys()` function, by setting `xmsg_.to` to the address of the `OmniPortal` on the destination chain when initiating the `XMsg` with `xcall()`.

Since `addValidatorSet()` has proper checks to ensure that the `XMsg` is indeed a system call from the `omniCChainID`, this issue is currently not exploitable. However, it is still recommended to address this unintended behaviour, as `XApps` or future code may not correctly implement these checks.

Recommendations

If the system calls are only intended to be called using `_execSys()`, check in `xcall()` using the contract `XRegistry` that the `to` address is not the address of the portal.

If extra checks are desired, the `_exec(XTypes.Msg)` function can implement the following check to skip execution of these types of malicious `XCalls`:

```
if (xmsg_.to == address(this)) return;
```

Note that the check should not cause the function to revert, otherwise it may be possible to DoS the `XStream`.

Resolution

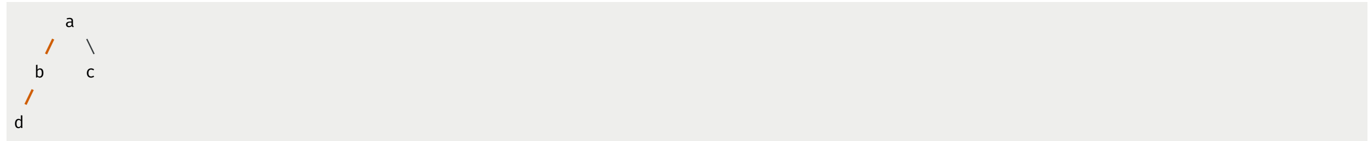
This issues has been addressed in PR #1148. The function `_exec(XTypes.Msg)` now returns from the function without reverting if `xmsg_.to == address(this)`.

OMP-12	Index Out Of Bounds Panic In `GetMultiProof()`		
Asset	`lib/merkle/core.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The Merkle tree's `GetMultiProof()` implementation may result in a runtime index out of bounds panic when a tree contains a node with only one child.

The case where an intermediate node has only a single child occurs if a tree is created with an even number of elements. For example the following tree has 4 nodes `[a,b,c,d]`. The node `d` does not have a sibling and so will panic if it is used as an index.



An example code which triggers a crash is as follows:

```
func panicGetMultiProof() {
    tree := [][32]byte{
        crypto.Keccak256Hash([]byte("leaf1")),
        crypto.Keccak256Hash([]byte("leaf2")),
    }
    treeIndices := []int{1}

    fmt.Println("Tree:", tree)
    fmt.Println("Tree Indices:", treeIndices)
    proof, err := merkle.GetMultiProof(tree, treeIndices...)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }

    fmt.Println("MultiProof:", proof)
}
```

In the example above, the sibling index is calculated as 2, but the size of the tree is 2. As such, an out-of-bound panic is observed.

Note that the function `MakeTree()` will always create a tree with an odd number of nodes. However, the relayer in `CreateSubmissions()` may be passed a malformed tree.

Recommendations

Implement additional checks on line [95] of `lib/merkle/core.go` to ensure calculated `sibilingIndex` is not larger than the tree size and that a tree has an odd number of nodes.

Resolution

This issue has been addressed in PR #1322.

OMP-13	Mismatch Between XBlock And XMsg `sourceChainId`		
Asset	`protocol/OmniPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The function `xsubmit()` does not check if `BlockHeader.sourceChainId == Msg.sourceChainId` for each `XMsg` in a submission.

OmniPortal.sol

```
216 for (uint256 i = 0; i < xsub.msgs.length; i++) {
        _exec(xsub.msgs[i]);
218 }
```

If the Omni consensus chain is compromised or there is a bug with the Halo client that causes `XBlocks` to contain `XMsgs` from a different source chain, then these `XMsgs` can be unintentionally executed.

Recommendations

The contents of `XBlocks` should be verified such that `xsub.blockHeader.sourceChainId` is equal to `xmsg_.sourceChainId` for all `XMsgs` in the `_exec(XTypes.Msg)` function.

Resolution

This issue has been addressed in PR#1133 by checking that `xsub.blockHeader.sourceChainId` is equal to `xmsg_.sourceChainId` in the `xsubmit()` function as follow :

OmniPortal.sol

```
242 require(xsub.msgs[i].sourceChainId == xsub.blockHeader.sourceChainId, "OmniPortal: wrong sourceChainId");
```

After reviewing the retesting commit d3c9213, the testing team has noticed that this check is moved to the function `_exec(XTypes.BlockHeader, XTypes.Msg)` in line [230]

OMP-14	Cleartext Storage Of Sensitive Information In Config File		
Asset	`halo/app/privkey.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

`EigenKeyPassword` is stored with no encryption in the config file.

As per line [33] in `halo/app/privkey.go`, `EigenKeyPassword` is taken from a config file and used directly in `loadEthKeystore()` function call, without prior decryption or decoding.

Anyone with read access to the config file can view the password and use it to decrypt and retrieve a private key.

Recommendations

Consider implementing encryption or, at least, encoding of any sensitive information stored in config files.

Alternatively, store such information in environment variables while ensuring the environment itself is trusted and tightly secured, with restricted access.

Resolution

This issue has been addressed in PR #1147.

OMP-15	Vote Extensions May Occur On Non-Existent Chains		
Asset	`halo/attest/keeper/keeper.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The functions `VerifyVoteExtension()` and `AddVotes()` do not ensure that an incoming vote is for a supported chain.

Votes and aggregate votes may be sent which contain a `BlockHeader` with an unsupported `ChainId`. As there is a lack of validation these votes will be accepted and stored in the keeper. Unless 2/3rds of the voting power approve these votes they will not be approved and will remain pending until deletion.

Recommendations

To resolve the issue add restrictions such that each vote must be for a supported chain.

Resolution

The issue has been resolved in PR #1217 by adding a portal service which checks the incoming vote chain ID against the `OmniPortal` smart contract.

OMP-16	Index Out Of Bounds Panic In `PubKeyToBytes64()`		
Asset	`lib/k1util/k1util.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `PubKeyToBytes64()` may panic.

lib/k1util/k1util.go

```go
// PubKeyToBytes64 returns the 64 byte uncompressed version of the public key, by removing the prefix (0x04 for uncompressed keys).
func PubKeyToBytes64(pubkey *stdecdsa.PublicKey) []byte {
    return ethcrypto.FromECDSAPub(pubkey)[1:] // @audit panics on `nil`
}
```

If `pubkey` is `nil` then the return value of `ethcrypto.FromECDSAPub()` is a `nil` array, therefore indexing at `[1:]` will cause an index out of bounds panic.

Recommendations

To resolve the issue, handle the case where `ethcrypto.FromECDSAPub(pubkey)` returns an empty list.

Resolution

This issue has been addressed in PR #1321.

OMP-17	Admin Can Fail XMsg Execution By Updating `xmsgMaxGasLimit`		
Asset	`protocol/OmniPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

High `gasLimit` `XMsgs` can inexplicably fail if the admin lowers `xmsgMaxGasLimit` before the `XMsg` is executed.

The `_exec()` function trims the `gasLimit` to `xmsgMaxGasLimit` :

OmniPortal.sol

```
279 // trim gasLimit to max. this requirement is checked in xcall(...), but we trim here to be safe
    if (gasLimit > xmsgMaxGasLimit) gasLimit = xmsgMaxGasLimit;
```

If an `XMsg` with a `gasLimit` that is higher than the new `xmsgMaxGasLimit` is executed after the value is lowered by the admin, the call can fail due to running out of gas.

Recommendations

Consider not trimming the `gasLimit` inside `_exec()` .

Resolution

Trimming the gasLimit to `xmsgMaxGasLimit` has been removed from the function `_exec()` in PR#1126

OMP-18	Extra Native Token Amount Is Not Refunded		
Asset	`protocol/OmniPortal.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When a user initiates a `XCall` using the function `xcall()`, they pay for fees in native token. However, any excess amount paid beyond the required fee is not refunded to the user.

The function `xcall()` checks using the require statement on line [140] that the user pays enough fees. This amount depends on the `destChainId`, the `data` used in the `xcall()` and the `gasLimit`. While a user may accidentally or intentionally pay more than this required amount, any excess payment will not be refunded.

Recommendations

Refund the extra native token amount to the user. Ensure that the refund occurs after the event emission to avoid reentrancy.

Resolution

The development team has decided to not address this issue for now.

OMP-19	Usage of Deprecated Dependency Functions		
Asset	`lib/k1util/k1util.go, halo/evmstaking/evmstaking.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Multiple functions end up calling deprecated functions from `go-ethereum/crypto` `v1.13.15` that could panic under certain conditions, such as points not being on the curve.

Recommendations

Update `go-ethereum` dependency to version `=>1.14.0`.

Resolution

Dependency has been updated in the `main` branch during testing.

OMP-20	Lack Of `stateJSON` Validation When Loading From File		
Asset	`halo/attest/voter/voter.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When loading state from file, there is no validation of loaded `stateJSON` data to ensure it has not been corrupt or malformed.

If any of the `types.Vote` values in the `stateJSON` have any of their parameter values set to `nil`, it may lead to unexpected panics elsewhere in the code.

Recommendations

Implement sanity checks to verify `stateJSON` data loaded from file is not malformed and all of its values, including their parameters, are set correctly and are not `nil`.

Resolution

This issue has been addressed in PR #1320.

OMP-21	`nil` Pointer Deference Panics In `AddVotes()`, `Add()` and `addOne()`
Asset	`halo/attest/keeper/msg_server.go`, `halo/attest/keeper/keeper.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

There is no validation of `Votes` values in `msg` parameter of `MsgAddVotes` type in `Add()` function. This could result in nil dereference panic on line [114] if a `BlockHeader` parameter was `nil`.

There are also no `nil` checks on `agg` parameter values in `addOne()` function, what may lead to `nil` dereference panic on line [145] and lines [150-153].

`addOne()` is called via `Add()` by `AddVotes()` function of `halo/attest/keeper/msg_server.go`, which does not perform validation of `msg` parameter of `MsgAddVotes` type . As a result, it is possible that a `MsgAddVotes` with invalid votes will be processed.

If `Votes` array elements provided as a part of `MsgAddVotes` type have their JSON values set to `null`, or omitted completely, they will be interpreted as `nil` and unexpected panics may occur during dereferencing.

Recommendations

Implement verification of `Votes` values in higher level `AddVotes()` function to ensure they are not set to `nil`, particularly `BlockHeader`.

This would prevent invalid values flowing into `Add()` and `addOne()` functions.

Resolution

This issue has been addressed in PR #1252.

OMP-22	Unaddressed TODO Comments
Asset	`*.go, halo/attest/keeper/keeper.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Number of `//TODO` style comments have been found throughout the codebase. These are marked as known issues and therefore raised as informational, however many of them have security considerations.

Some examples of those identified which have security considerations and are not yet fixed:

- `halo/evmengine/keeper/abci.go` on line [86] - `/ // TODO(corver): Figure out what to use here.` - currently the zero hash is used as the beacon block hash.
- In `halo/evmengine/keeper/abci.go` on line [83] - `// TODO(corver): implement proper randao.` - the randomness passed to the execution client is currently the previous block hash and unsafe for use.

Note, the list above is not exhaustive.

Recommendations

Address all `//TODO` comments throughout the codebase, verify and ensure they have all been addressed where relevant, or clear assumptions and design decisions have been made and documented.

Resolution

This issue has been addressed in PR #1332.

OMP-23	Additional Chain Height And Header Checks Required
Asset	`lib/xchain/provider/fetch.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

Inside `finalisedInCache()` function on line [143] there are no checks for a scenario where the latest fetched head is larger than the chain's height.

Although, an unlikely condition, it could indicate something has gone fundamentally wrong and, as such, should be handled and managed.

Recommendations

Consider implementing additional checks to cater for an unlikely situation where latest fetched head is larger than the chain's height.

Resolution

The issue has been acknowledged by the development team and closed with the following comment:

> *"We trust the RPC endpoints to return valid data."*

OMP-24	No Linear Search Data Set Restrictions
Asset	`lib/xchain/merkle.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Linear search is utilised to find Merkle tree's leaf indices, but no restrictions on the overall data set size have been implemented.

As Merkle trees grow, so will the time to search through them to find relevant indices, potentially leading to poor performance or creating a DoS (denial-of-service) condition.

Recommendations

Consider implementing checks and restrictions on sizes of data sets being searched through.

Resolution

This issue has been addressed in PR #1319.

OMP-25	Lack Of `Validators` Array Size Checks
Asset	`halo/evmengine/keeper/keeper.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

If validators list is empty, or all validators have power of 0, division by zero panic may occur on line [116] at `halo/evmengine/keeper/keeper.go`.

In `newABCIValsetFunc()` function of `lib/cchain/provider/abci.go`, an array of validators is created based on the `ValidatorSetResponse` type response from a call to `ValidatorSet()`. If no validators existed, or all of them had power of 0, they would not be included in the returned array as per check on line [128] of `halo/valsync/keeper/query.go`. As such, `valSetResponse` would be set with an empty `Validators` array.

Subsequently, call on line [116] at `halo/evmengine/keeper/keeper.go` would result in division by zero panic due to `len(valset.Validators)` being zero:

```
nextIdx := int(idx+1) % len(valset.Validators)
```

Recommendations

Implement checks to ensure `valset.Validators` array is not empty before performing any operations on it, or using its size (which could be zero) in calculations.

Resolution

This issue has been addressed in PR #1318.

OMP-26	Miscellaneous General Comments
Asset	`protocol/*`, `libraries/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **Use Brackets When Performing MulDiv**

   Related Asset(s): `Quorum.sol`

   No brackets are used in the muldiv operation used to calculate if the quorum is reached.

   ```
   return votedPower > totalPower * numerator / denominator;
   ```

   Add brackets to improve the code's readability as shown below:

   ```
   return votedPower > (totalPower * numerator) / denominator;
   ```

2. **`error` Variable Name is Confusing**

   Related Asset(s): `OmniPortal.sol`

   ```
   // empty error if success is true
   bytes memory error = success ? bytes("") : result;
   ```

   The `error` variable name is confusing as the `error` keyword is used to denote custom errors in Solidity after v0.8.4.

   Change the variable name to `errorMsg`.

3. **`_setXMsgDefaultGasLimit` Should Have A Sanity Check**

   Related Asset(s): `OmniPortal.sol`

   The `_setXMsgDefaultGasLimit()` function is missing a sanity check to ensure that the `xmsgDefaultGasLimit` is between the accepted range (between `xmsgMinGasLimit` and `xmsgMaxGasLimit`).

   Add the sanity check.

4. **Other Unaddressed `//TODO` Comments**

   Related Asset(s): `*.go`

   Number of `//TODO` style comments have been found throughout the codebase, some examples identified (note, the list below is not exhaustive):

   - In `halo/attest/keeper/keeper.go` the `verifyAggVotes()` function comment states `Ensure votes represent at least 2/3 of the total voting power. <- This isn't done?`. The function does not in fact do that.
   - In `e2e/app/fund.go` on line [19] - `// Maximum amount to fund in ether. // TODO(corver): Increase this`.

   Address all `//TODO` comments throughout the codebase, verify and ensure they have all been actioned where relevant, or clear assumptions and design decisions have been made and documented.

Recommendations

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

Resolution

The development team has addressed above findings as follows:

1. Addressed in PR #1388.
2. Addressed in PR #1099.
3. Addressed in PR #1180.
4. Addressed in PR #1332.

Appendix A Test Suite

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

```
Ran 1 test for test/OmniStake.t.sol:OmniStakeTest
[PASS] test_deposit() (gas: 53655)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.59ms (342.02µs CPU time)

Ran 25 tests for test/OmniPortal.t.sol:OmniPortalTest
[PASS] test_addValidatorSet_reverts() (gas: 634748)
[PASS] test_collectFees() (gas: 105613)
[PASS] test_feeFor() (gas: 26587)
[PASS] test_isSupportedChain() (gas: 42860)
[PASS] test_isXCall() (gas: 320492)
[PASS] test_pause() (gas: 53418)
[PASS] test_setFeeOracle() (gas: 20635)
[PASS] test_setXMsgDefaultGasLimit() (gas: 20595)
[PASS] test_setXMsgMaxGasLimit() (gas: 20598)
[PASS] test_setXMsgMinGasLimit() (gas: 20661)
[PASS] test_setXReceiptMaxErrorBytes() (gas: 20655)
[PASS] test_setXRegistry() (gas: 20633)
[PASS] test_xcall() (gas: 143540)
[PASS] test_xmsg() (gas: 321454)
[PASS] test_xsubmit_addValidatorSet() (gas: 1487674)
[PASS] test_xsubmit_callCounterIncrement() (gas: 338851)
[PASS] test_xsubmit_gasBombAttack() (gas: 10523144)
[PASS] test_xsubmit_reverts_duplicate_Validator() (gas: 177591)
[PASS] test_xsubmit_reverts_invalidProof() (gas: 207614)
[PASS] test_xsubmit_reverts_invalid_signature() (gas: 190900)
[PASS] test_xsubmit_reverts_noQuorum() (gas: 169539)
[PASS] test_xsubmit_reverts_noXmsgs() (gas: 19708)
[PASS] test_xsubmit_reverts_unknownValSet() (gas: 25467)
[PASS] test_xsubmit_reverts_validators_not_sorted() (gas: 195440)
[PASS] test_xsubmit_reverts_wrong_offset() (gas: 254901)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 8.39s (78.01s CPU time)
```

Appendix B Vulnerability Severity Classification

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

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

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

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].