Omni

EVM Redenom

Security Assessment Report

Version: 2.0

August, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Omni components in scope. The review focused solely on the security aspects of these components, 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 components in scope. 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 components 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*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Omni components in scope.

Overview

This security engagement focused on the EVM redenomination part of the Earhart upgrade for the Omni network. The `evmredenom` module implements a secure token redenomination process for the Nomina rebrand, multiplying all EVM account balances by 75 with the Earhart network upgrade. Cryptographic proofs are used via the ethereum snapsync protocol to verify and process account balance updates in batches.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the omni-network/omni repository.

The scope of this time-boxed review was strictly limited to files at commit 21652a7.

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

Approach

The security assessment covered components written in Golang.

For the Golang components, the manual review focused on identifying issues associated with the business logic implementation of the libraries and modules. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime.

Additionally, the manual review process focused on identifying vulnerabilities related to known Golang anti-patterns and attack vectors, such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks, and various panic scenarios including `nil` pointer dereferences, index out of bounds, and explicit panic calls.

To support the Golang components of the review, the testing team may use the following automated testing tools:

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

Output for these automated tools is available upon request.

Coverage Limitations

Due to the 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 4 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Omni components in scope. 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 components, including 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 components(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
RDNM-01	Timing Between State Root And Infrastructure Upgrades Allows For Value Extraction	Medium	**Closed**
RDNM-02	No Support For Resuming Batch Submissions	Informational	**Closed**
RDNM-03	Small Balance Rounding During Redenomination	Informational	**Closed**
RDNM-04	Miscellaneous General Comments	Informational	**Closed**

RDNM-01	Timing Between State Root And Infrastructure Upgrades Allows For Value Extraction		
Asset	`halo/app/upgrades/earhart/upgrade.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Attackers can exploit the timing window between the state root capture and infrastructure upgrades during the Earhart upgrade to extract value.

The `CreateUpgradeHandler()` function in `halo/app/upgrades/earhart/upgrade.go` captures the execution header using `evmEngine.GetExecutionHeader()`. This header represents the state from the block before the upgrade block. The state root from this header is then used to initialise the redenomination module and will be used to calculate the redenominated balances.

The vulnerability arises if infrastructure components such as bridges, oracles, and centralised exchange deposits are upgraded during the fork block, which is 1 block after the state snapshot. This creates a small timing window to extract value. For example, assuming the bridge is updated during the fork block an attacker can:

1. Have their balance included in the state root by having a balance at the end of the parent block.
2. Bridge their funds to Ethereum at the top of the upgrade block, and have it land before the bridge upgrade transaction such that they receive OMNI tokens on Ethereum.
3. This results in an almost duplicated balance since the attacker received their balance on Ethereum but also received the redenominated amount on the Omni network.

The staking mechanism is not vulnerable to this type of attack since it is upgraded at the start of the upgrade block.

```
func CreateUpgradeHandler(
    mm *module.Manager,
    configurator module.Configurator,
    evmEngine *evmenginekeeper.Keeper,
    redenom *evmredenomkeeper.Keeper,
    submitCfg evmredenomsubmit.Config,
) upgradetypes.UpgradeHandler {
    return func(ctx context.Context, _ upgradetypes.Plan, fromVM module.VersionMap) (module.VersionMap, error) {
        log.Info(ctx, "Running 4_earhart upgrade handler")

        // Initialize redenomination status to current execution head state root.
        header, err := evmEngine.GetExecutionHeader(ctx) //@audit header of parent block
        if err != nil {
            return nil, errors.Wrap(err, "get execution head")
        }

        if err := redenom.InitStatus(ctx, header.Root); err != nil {
            return nil, errors.Wrap(err, "initialize redenomination status")
        }

        if err := maybeSubmitRedenomination(ctx, submitCfg, header.Root); err != nil {
            return nil, errors.Wrap(err, "maybe submit redenomination")
        }

        return mm.RunMigrations(ctx, configurator, fromVM)
    }
}
```

Seeing as a significant amount of funds are at risk, this issue is marked as high impact. The actual likelihood of this issue depends on multiple factors, such as the intended upgrade mechanism for infrastructure components. Additionally, precise timing is required to pull off this attack, as such the likelihood is rated low.

Recommendations

Ensure infrastructure is upgraded at the top of the upgrade block, or the bottom of the parent block, such that no transactions can occur between the capture of state root and the upgrade. In cases where this is not possible, the infrastructure component can also be paused to prevent these attacks.

Resolution

The development team has closed the issue with the following comment:

> *We plan to pause the bridge before the network upgrade, upgrade it to support NOM, then unpause after all balances have been increased.*

RDNM-02	No Support For Resuming Batch Submissions
Asset	`halo/evmredenom/submit/submit.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Do()` function in `submit.go` loops through batches of all the accounts on chain and submits them to a predeployed contract. If an error occurs during the execution of this loop, execution is stopped and that error is returned. However, there is currently no support for resuming the submission of batches from a given point. As such, if an error occurs all batches must be resubmitted from the beginning. Given that many of the batches would already be processed at that point, a large portion of the `submit()` transactions would revert, wasting lots of gas in the process.

```
func Do(
  // ...
) error {
  // ...

  var next common.Hash
  var prevNonce uint64
  eg, ctx := errgroup.WithContext(ctx)
  for i := 0; ; i++ {
    resp, err := cl.AccountRange(ctx, stateRoot, next, batchSize) //@audit next is zero-hash, no support for resuming
    if err != nil {
      return errors.Wrap(err, "get account range")
    } else if len(resp.Accounts) == 0 {
      return errors.New("empty account range response")
    }

    done, err := verifyBatch(stateRoot, next, resp)
    if err != nil {
      return err
    }

    next = incHash(resp.Accounts[len(resp.Accounts)-1].Hash)

    nonceCtx, nonce, err := backend.WithReservedNonce(ctx, from)
    if err != nil {
      return errors.Wrap(err, "get reserved nonce")
    } else if i > 0 && nonce <= prevNonce {
      return errors.New("nonce not incremented", "prev", prevNonce, "got", nonce)
    }

    eg.Go(submitBatch(nonceCtx, from, contract, backend, archive, preimages, resp))

    if !done {
      continue
    }

    if err := eg.Wait(); err != nil {
      return errors.Wrap(err, "wait submit batch")
    }

    log.Info(ctx, "All redenomination account ranges submitted", "total", i+1)

    return nil
  }
}
```

Recommendations

Consider implementing functionality to restart batch submission starting from a given address.

Resolution

The development team has closed the issue with the following comment:

> *Since submissions are idempotent, if anything fails, it can just be redone from the start. This trades a tiny additional gas cost, for simplicity in code. Out of order submissions will simply fail/error/revert when applied in consensus chain, correctly ordered submissions will be processed.*

RDNM-03	Small Balance Rounding During Redenomination
Asset	`halo/evmredenom/keeper/keeper.go, octane/evmengine/keeper/db.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

Accounts with very small balances will not be redenominated due to rounding in the withdrawal process and thus lose 74/75 (~98.6%) of the value of their balance.

During the redenomination process, `InsertWithdrawal()` is used to mint new coins to a users account. However, `InsertWithdrawal()` rounds withdrawal amounts down to the nearest gwei (1e9 wei). As a result, for small balances where the mint amount is less than 1 gwei, no withdrawal request is created.

```
// db.go in InsertWithdrawal()
gwei, dust, err := toGwei(amountWei)
if err != nil {
    return err
}
dustCounter.Add(float64(dust))

if gwei == 0 {
    log.Debug(ctx, "Not creating all-dust withdrawal", "addr", withdrawalAddr, "amount_wei", amountWei)
    return nil // @audit No withdrawal created for dust amounts
}
```

Given that the value loss here is negligible, this issue is marked informational.

Recommendations

Consider modifying the withdrawal mechanism to handle small withdrawal amounts without losing them to rounding.

Resolution

The development team has closed the issue with the following comment:

> *This rounding to gwei is a limitation of the engineAPI, out of our control. Also, at the current price, 1 gwei NOM is 0.000000000004 USD. We feel this is acceptable.*

RDNM-04	Miscellaneous General Comments
Asset	All files
Status	**Closed:** See Resolution
Rating	Informational

Description

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

1. **Duplicate `incHash` Function Implementation**
   Related Asset(s): halo/evmredenom/keeper/helpers.go, halo/evmredenom/submit/submit.go
   The `incHash()` function is implemented identically in two separate files, creating unnecessary code duplication.

```
// helpers.go line 46
func incHash(h common.Hash) common.Hash {
    return new(uint256.Int).AddUint64(
        new(uint256.Int).SetBytes32(h[:]),
        1,
    ).Bytes32()
}
```

```
// submit.go line 206
func incHash(h common.Hash) common.Hash {
    return new(uint256.Int).AddUint64(
        new(uint256.Int).SetBytes32(h[:]),
        1,
    ).Bytes32()
}
```

   Remove one of the `incHash()` function implementations to eliminate code duplication.

Recommendations

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

Resolution

The development team has closed the issue with the following comment:

> *In this case, we opted for the Go proverb: A little copying is better than a little dependency.*

Appendix A Vulnerability Severity Classification

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

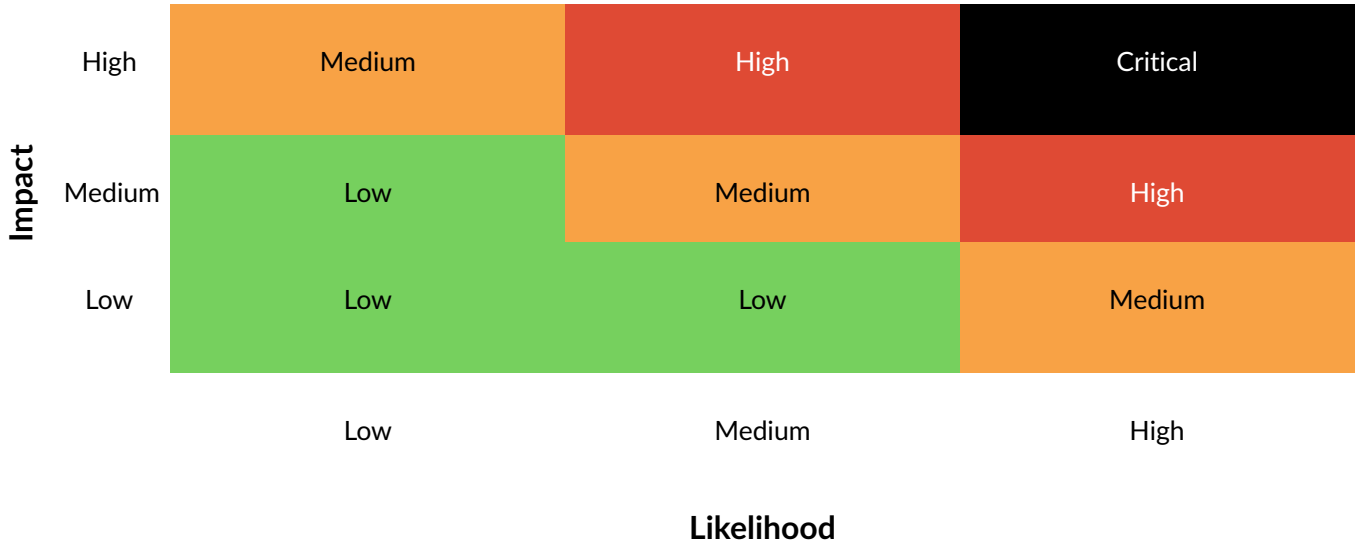

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.