Zellic

Prepared for
Tyler Tarsi
Omni Network Foundation

Prepared by
Frank Bachman
Ayaz Mammadov
Zellic

July 15, 2024

Omni Network
Application Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Omni Network Foundation from June 24th to July 12th, 2024. During this engagement, Zellic reviewed Omni Network's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any set of operations that can block submissions from being made (taking down the bridge) — for example, by not allowing a certain message to be submitted, and as such all subsequent messages cannot be submitted?
- Are there any ways to bypass the attestation on the bridge, submitting unapproved and unattested messages?
- What powers does a malicious proposer have; can they attest entire blocks on their own or take down the chain?
- What are the possible issues related to a chain reorganization?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- The use of the new experimental Cosmos optimistic execution feature

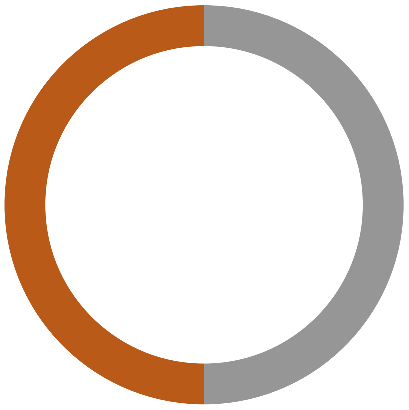
1.4. Results

During our assessment on the scoped Omni Network modules, we discovered two findings. No critical issues were found. One finding was of high impact and the other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Omni Network Foundation's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	0
🟩 Low	0
⬜ Informational	1



2. Introduction

2.1. About Omni Network

Omni Network Foundation contributed the following description of Omni Network:

> Omni combines an EVM execution layer with native cross chain messaging. Both are secured by Omni's consensus layer dPoS validator set.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Omni Network Modules

Type	Golang
Platform	Cosmos

Target	omni
Repository	https://github.com/omni-network/omni ↗
Version	`99b8ed78f7d4f1c80544f121071754b4e20f2db0`

Programs	`halo/*`
	`octane/*`
	`contracts/src/xchain/*`
	`contracts/src/octane/*`
	`contracts/src/libraries/*`
	`lib/xchain/*`
	`lib/cchain/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.6 person-weeks. The assessment was conducted by two consultants over the course of 2.5 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Frank Bachman
Engineer
frank@zellic.io ↗

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 24, 2024	Start of primary review period
June 27, 2024	Kick-off call
July 12, 2024	End of primary review period

3. Detailed Findings

3.1. Chain halt due to unbounded votes by proposer

Target	halo/attest/keeper.go		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Due to a lack of validation, the number of votes that can be included in `MsgAddVote` is limitless. As such, this poses problems as attestations that were registered are deleted after a certain block window. As there is no penalty to double voting and the maximum block size is 100 MB, these unbounded computations in halo's `EndBlocker` could result in several issues.

```
func (k *Keeper) deleteBefore(ctx context.Context, height uint64) error {
    ...
    for iter.Next() {
        ...

        // Delete signatures
        if err := k.sigTable.DeleteBy(ctx,
    SignatureAttIdIndexKey{}.WithAttId(att.GetId())); err != nil {
            return errors.Wrap(err, "delete sigs")
        }

        // Delete attestation
        err = k.attTable.Delete(ctx, att)
        if err != nil {
            return errors.Wrap(err, "delete att")
        }
    }
```

Impact

A malicious proposer could propose a block with up to ~100 MB of votes. This large computation could result in a liveness issue with the processing of votes, which would result in a consensus failure and an eventual chain halt.

Recommendations

Harden the vote verification to ensure that only votes that were in the voting rounds can be proposed. This includes ensuring that double voting cannot happen and that the vote limit cannot be bypassed.

Remediation

This was remediated in commit c4050e17 ↗ by hardening the vote verification. This included preventing double signing, applying the vote extension limit on the votes and also fixed an issue that would allow proposers to submit votes for unsupported ConfLevels allowing proposers to insert invalid votes on chain without risking slashing.

3.2. Possible race condition in valsync init

Target	halo/valsync/keeper.go		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

If a subscriber is initialized as a valid validator, and if a previous round where the validator was not a validator is attested to, that validator will mistakenly think it's not a validator.

```
func (k *Keeper) EndBlock(ctx context.Context) ([]abci.ValidatorUpdate, error)
    {
    ...
    // The subscriber is only added after `InitGenesis`, so ensure we notify it
    of the latest valset.
    if err := k.maybeInitSubscriber(ctx); err != nil {
        return nil, err
    }

    // Check if any unattested set has been attested to (and return its
    updates).
    return k.processAttested(ctx)
    ...
}
```

Impact

If such a situation occurs, a participant in the network which is a validator might mistakenly think it is not a validator and not vote. In the future, this might result in penalties or slashing.

Recommendations

Refactor the code to avoid such potential issues.

Remediation

This was remediated in commit 43f0a05c ↗ by changing the validator subscriber update system to send full validator sets instead of validator set updates (deltas), and ensuring that the validator set update is newer than the old one.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The `block.prevdao` is completely biasable by the proposer

As seen in the excerpt below, the block randao is the hash of all the transactions in the last block. While on-chain dApps should not rely on `block.prevrandao`, if any on-chain dApp does rely on `block.prevrandao`, an opportunistic proposer could brute-force a favorable `randao` value to extract value out of on-chain dApps that use it as source of randomness.

```
// startBuild triggers the building of a new execution payload on top of the
    current execution head.
// It returns the EngineAPI response which contains a status and payload ID.
func (k *Keeper) startBuild(ctx context.Context, appHash common.Hash,
    timestamp time.Time) (engine.ForkChoiceResponse, error) {
    ...
    attrs := &engine.PayloadAttributes{
        Timestamp: ts,
        Random: head.Hash(), // We use head block hash as randao.
        SuggestedFeeRecipient: k.feeRecProvider.LocalFeeRecipient(),
        Withdrawals: []*etypes.Withdrawal{}, // Withdrawals not supported yet.
        BeaconRoot: &appHash,
    }
    ...
    return resp, nil
}
```

4.2. Portal calls cannot be refunded if they fail

If portal calls to staking.sol such as `Delegate` or `CreateValidator` fail, the stake/payment is not refundable. Users should be wary to double-check that the correct payments are being made as they could accidentally self-delegate and lose funds.

4.3. Validators cannot unstake

Once validators join the network and bond their stake, there is no way for validators to unstake their stake. The only possibility is to be jailed or to not vote on blocks. This is a feature in development.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: keeper.go

`ExtendVote`

The `ExtendVote` is the functionality responsible for handling how the validator votes and what they submit.

Firstly, the validator retrieves its votes on blocks that it has seen before; this is stored on disk. It then validates these votes by streams. It ensures this by checking various conditions:

- Is the vote's blockheader's `chainId` a valid one (is it supported)?
- Is the vote fresh enough? (Does it pass the `windowCompare` function?)

Following this, the code is mostly logs, and the votes are returned in vote form.

`VerifyVoteExtension`

The `VerifyVoteExtension` is used by all validators to ensure that the votes that were extended are valid, that a malicious proposer is not sending old votes, and that the votes sent to other validators are indeed votes signed by the validator and not random votes.

The conditions it verifies are the following:

- Are the votes valid (the block headers are the correct size, among other checks such as the vote being signed by the validator in the vote)?
- Is the vote signed by the validator who extended it?
- Is the vote's blockheader's `chainId` a valid one (is it supported)?
- Is the vote fresh enough? (Does it pass the `windowCompare` function?)

`BeginBlock`

The `BeginBlock` handler is called at the start of each block. It deletes all attestations and signatures before `BlockHeight - trimLag` (inclusive). It creates an iterator from block 0 up to `BlockHeight - trimLag`. Before deletion, it checks that each attestation does not surpass the latest approved attestation for the given chain. Then it deletes the corresponding signatures and attestation from the `sigTable` and `attTable` in the keeper.

`EndBlock`

The `EndBlock` handler is called at the end of each block. It approves any pending attestations from the `attTable` in the keeper.

Firstly, it checks the `BlockHeight` and returns if it is the first block (as there are no attestations to approve). Then, it creates an iterator with all the pending attestations in `attTable`. For each attestation, it gets the block offset for the latest approved attestation in that chain. It verifies that the attestation is for the next block; otherwise, it is skipped.

After, it retrieves the signatures for each attestation and verifies that it is approved by the active validator set from the previous block. It checks that the total power from the validators who have voted to approve exceeds two thirds of the total validator set power. Any signatures from validators that were not a part of the active validator set from the previous block are deleted.

If the approval fails, it checks if there is already a finalized attestation that overrides the current one. If so, it updates the attestation status to approved. The invalid signatures are then deleted. The attestation is approved, and `attTable` is updated with the approved attestation. The latest approved attestation's block offset is updated in the cache. After the attestations are approved, votes behind the minimum vote window are removed.

5.2. Module: msg_server.go

`MsgAddVotes`

This message is called with all aggregated votes when a block is finalized. The signature verification on the votes are performed in the same way as in `processProposal`. It checks that the signatures of every vote is from active validator sets of previous blocks. It then adds the aggregate votes as pending attestations to the store. The vote is merged if the attestation already exists.

After the attestations are added to the store, it updates the voter state with the local headers and sets the status to "committed".

5.3. Module: proposal_server.go

`ProcessProposal`

The `ProcessProposal` handler uses the attest module to verify all the aggregated votes in a proposed block.

It first verifies that all the votes are valid by checking the signature against the attestation root hash and the validator address. Then it fetches the chain ID from the block header and verifies that it is supported.

Thereafter, there are two constraints:

1. All the votes are from active validators. Note that this is checked against active validators from the previous block since vote extensions are delayed by one block.

2. Verify that the vote block header is within the vote window.

It then updates the voter states with the local headers and sets the status to "proposed".

5.4. Module: keeper.go

`EndBlock`

The `EndBlock` routine for valsync is responsible for keeping validator set updates in sync across the chains such that the endpoints can properly account for new or removed validators when processing attestations.

It is also responsible for signaling to the validators themselves whether they are a validator or not, subscribed listeners will stop/start voting based on the validator-set updates they received.

It does this by initially running the Staking module's `EndBlock` and parsing the updates it returns.

It merges the validator-set updates with the last validator set, resulting in the current validator set. It then adds these to the ORM table `valSetTable`. It also does checks to ensure that one validator does not have too much stake, control consensus, or take down the chain.

It emits a message to the portal. It does this by adding the block and the message to various ORM tables, which will be queried when the validator fetches messages for the Omni consensus chain. Then it initializes any waiting subscribers.

Then, it gets the next unattested validator set and checks if that validator set was attested to. If so, it marks it as attested in the `valSetTable` and updates any subscribed listeners.

5.5. Module: abci.go

`PrepareProposal`

Firstly, a defer is used to ensure that any panics are caught and logged for later inspection. Then, if it is the first block, it's then left empty to account for a quirk. Then if an optimistic payload build was executed, the built Geth payload is retrieved. Otherwise, geth is asked to build a payload waiting `k.BuildDelay` seconds. It then marshals the retrieved data into a `MsgExecutionPayload`.

Then, it retrieves the votes from the last voting round to include in this block. However, there is no binding of votes from the last voting round to the proposed blocks vote's; this related to Finding 3.1. ↗, where a proposer can limitlessly vote to cause a chain halt.

Then, the parent block's important events (XChain calls, evmstaking, evmslashing) are collected and added to the block.

After, these messages are marshaled into a block as TXs, and the block is proposed.

`PostFinalize`

The `PostFinalize` is used and is important as it is called after the finalization of a block. Its purpose is to start an optimistic build if the validator is going to be the next proposer. Once the validator knows it will be the next proposer, it calls the execution layer to start building a block and sets in storage the `payloadID` of the optimize build. Next time `PrepareProposal` is called, this block can be fetched must faster instead of waiting for the block to be built.

5.6. Module: msg_server.go

`ExecutionPayload`

This is the function that is called when a block is finalized and `MsgExcecutionPayload` is delivered.

First, the payload is verified again, similarly to how it is verified in octane's `processProposal`:

- There cannot be any EVM beacon contract withdrawals.
- The fee recipient is `0xDEAD`.
- Geth must accept the payload as a possible new block (`pushPayload`).

After, it is pushed as a possible new block to the execution layer, and using the engine API, it is finalized as the block head.

Then, the last block's agreed-upon events are delivered. This could be creating validators or increasing self-delegation for a validator. However, any failures in event delivery are ignored, and panics are recovered.

The recorded latest execution head is then updated.

5.7. Module: `ProcessProposal`

`ProcessProposal`

The proposed payload is verified against set invariants.

- There cannot be any EVM beacon contract withdrawals.
- The fee recipient is `0xDEAD`.
- Geth must accept the payload as a possible new block (`pushPayload`).

Then, the events in the block are checked to ensure they are not fabricated. This is done by checking the proposed important events against the important events in the last block (done by each validator).

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Omni Network.

During our assessment on the scoped Omni Network modules, we discovered two findings. No critical issues were found. One finding was of high impact and the other finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.