Zellic

Prepared for

LayerZero Labs

Prepared by
Sunwoo Hwang
Jinheon Lee
Zellic

February 18, 2025

LayerZero OneSig

Smart Contract 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 LayerZero Labs from February 13th to February 17th, 2025. During this engagement, Zellic reviewed LayerZero OneSig'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:

- Is the signature-verification process secure?
- Is the Merkle proof process secure?
- Is the multi-sig process secure?

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

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, gaps in response times as well as a lack of developer-oriented documentation and thorough end-to-end testing impacted the momentum of our auditors.

1.4. Results

During our assessment on the scoped LayerZero OneSig contracts, we discovered one finding, which was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of LayerZero Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About LayerZero OneSig

LayerZero Labs contributed the following description of LayerZero OneSig:

> OneSig is a smart contract solution designed to streamline the signing and execution of arbitrary ‘calldata’ on any EVM compatible blockchains.

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 contracts.

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 contracts 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:

LayerZero OneSig Contracts

Type	Solidity
Platform	EVM-compatible

Target	OneSig
Repository	https://github.com/LayerZero-Labs/OneSig ↗
Version	`aed5db9379fdd35451b6e404e872e1748b663ff5`

Programs	`OneSig.sol`
	`MultiSig.sol`
	`ExecutorStore.sol`
	`lib/SelfCallable.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.9 person-weeks. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Jinheon Lee
Engineer
jinheon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 13, 2025	Kick-off call
February 13, 2025	Start of primary review period
February 17, 2025	End of primary review period
March 9, 2025	Scope changed from `04c6c1ee` ↗ to `41d6d17c` ↗
March 9, 2025	Scope changed from `41d6d17c` ↗ to `3609952c` ↗
March 9, 2025	Start of secondary review period
March 17, 2025	Scope changed from `3609952c` ↗ to `a5b7e862` ↗
March 18, 2025	End of secondary review period
May 7, 2025	Start of third review period
May 7, 2025	Scope changed from `a5b7e862` ↗ to `aed5db93` ↗
May 7, 2025	End of third review period

3. Detailed Findings

3.1. Unexpected behavior from `delegatecall`

Target	OneSig.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The contract provides two options for executing calls: `call` and `delegatecall`. However, allowing the `delegatecall` option can lead to unexpected behavior, as it allows modification of any storage variables in the contract, which cannot be set by the contract's functionality.

```
function _executeCall(Call calldata _call, uint256 _index) internal {
    bool success;
    if (_call.isDelegateCall) {
        (success, ) = _call.to.delegatecall(_call.data);
    } else {
        (success, ) = _call.to.call{ value:  _call.value }(_call.data);
    }

    // Revert if the call fails
    if (!success) revert ExecutionFailed(_index);
}
```

Impact

The nonce is incremented before the call. Although the nonce is only incremented and the contract does not have functionality to modify it directly, using `delegatecall` can potentially set it to its maximum value. This would prevent further transactions from being executed due to nonce-overflow errors.

Recommendations

We recommend removing the `delegatecall` option.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `a3cf8ad2` ↗.

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. Transaction revert

In the current design, when an external call reverts, the entire transaction reverts. This means that if any single transaction in a batch fails, subsequent transactions cannot be executed because the nonce cannot be incremented.

In such cases, the reverted transaction can either be resubmitted or invalidated by changing the seed and increasing the nonce. This mechanism allows the off-chain system to potentially reuse the same nonce for other signed transactions.

The LayerZero Labs team responded with the following:

> This is intentional, we want the calls to be atomic, and if a transaction reverts, it means something went wrong and we need to resubmit signatures, and potentially also invalidate the seed so those previously signed nonces/transactions can't be processed.

4.2. Executor functionality

The executor functionality was implemented during the third review period. This implementation introduces the following key features:

The system only allows executors or signers to call the `executeTransaction` function. The `ExecutorStore` contract keeps track of all executors in an array, and this list can only be changed through the `executeTransaction` function itself. Every time `executeTransaction` is called, it needs valid signatures from signers. If `executorRequired` is set to false, anyone can call `executeTransaction`, but they still need valid signatures from signers.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts 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: MultiSig.sol

Function: `setSigner(address _signer, bool _active)`

The function removes or adds signers. The function can only be called via this contract itself.

Inputs

- `_signer`
    - **Validation**: In the case of the *add* operation, the `_signer` is not allowed to have zero address or have already been included in the signers. In the case of the *remove* operation, the `signer` must have already been registered and the number of signers must fit the minimum threshold or more after being removed.
    - **Impact**: It determines the signers who will sign the signatures of the upcoming requested transaction.
- `_active`
    - **Control**: Fully controllable by users.
    - **Impact**: It determines whether to perform the add or remove operation.

Branches and code coverage (including function calls)

Intended branches

- It removes or adds signers.
    - ☑ Test coverage

Negative behavior

- If users do not call `setSigner` via this contract itself, the function reverts.
    - ☑ Negative test
- If the `_signer` does not satisfy the aforementioned requirements, the function reverts.
    - ☑ Negative test

Function: `setThreshold(uint256 _threshold)`

The function sets a threshold, which indicates the minimum size for signers. The function can only be called via this contract itself.

Inputs

- `_threshold`
    - **Validation**: It must have a value that satisfies the range `0 < value <= the number of registered signers`.
    - **Impact**: To verify Merkle roots, it must have been by/from a certain threshold of signers or more.

Branches and code coverage (including function calls)

Intended branches

- It sets the minimum size for signers to verify Merkle roots.
    - ☑ Test coverage

Negative behavior

- If users do not call `setThreshold` via this contract itself, the function reverts.
    - ☑ Negative test
- It prevents users from executing their transaction through `executeTransaction` if its Merkle root does not satisfy the required number of signers (= `_threshold`).
    - ☑ Negative test

5.2. Module: OneSig.sol

Function: `executeTransaction(Transaction calldata _transaction, bytes32 _merkleRoot, uint256 _expiry, bytes calldata _signatures)`

The function executes a single transaction that has valid signatures. The function can be executed by anyone (permissionlessly).

Inputs

- `_transaction`
    - **Validation**: It must satisfy the Merkle root proof.
    - **Impact**: It is used to call arbitrary contracts with its data.

- `_merkleRoot`
    - **Validation**: It must have the valid signatures of signers.
    - **Impact**: It is used to prove and validate whether `_transaction` is valid.
- `_expiry`
    - **Validation**: It must have a value bigger than or equal to the current `block.timestamp`.
    - **Impact**: If the time the `_transaction` was executed exceeds this expiration, the function reverts during the proof.
- `_signatures`
    - **Validation**: It must have an appropriate value against `_merkleRoot`.
    - **Impact**: It is instrumental in proving `_merkleRoot`.

Branches and code coverage (including function calls)

Intended branches

- If all arguments have valid values, the `_transaction` will be executed.
    - ☑ Test coverage

Negative behavior

- If any of the arguments are not satisfied, the function reverts.
    - ☑ Negative test

Function: `setSeed(bytes32 _seed)`

The function sets a seed for the transactions, which will be requested from users. The function can only be called via this contract itself.

Inputs

- `_seed`
    - **Control**: Fully controllable by the caller.
    - **Impact**: The reset seed will not allow a user to execute transactions that have been signed previously.

Branches and code coverage (including function calls)

Intended branches

- Users who have gotten the signed transaction previously cannot execute their transaction.
    - ☑ Test coverage

Negative behavior

- If users do not call `setSeed` via this contract itself, the function reverts.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet.

During our assessment on the scoped LayerZero OneSig contracts, we discovered one finding, which was of low impact.

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.