OneSig

Security Assessment

May 2nd, 2025 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

LayerZero Labs engaged OtterSec to assess the `onesig` program. This assessment was conducted between February 12th and February 14th, 2025. A follow-up review was performed between April 28th and May 2nd, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We also made recommendations to ensure adherence to coding best practices (OS-OSG-SUG-00) and advised the need for implementing a mechanism to synchronize state across multiple blockchains to ensure consistency in signature generation (OS-OSG-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/OneSig. This audit was performed against 3609952. A follow-up review was performed against d44bc55.

A brief description of the program is as follows:

Name	Description
onesig	It enables pre-authorization of transaction batches on EVM-compatible blockchains, eliminating the need for repeated multisig approvals. By storing approvals in advance, it allows for instant execution of arbitrary calldata, reducing delays and operational overhead.

02 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-OSG-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-OSG-SUG-01	The Onesig contract relies on a shared seed across multiple chains, but ensuring simultaneous updates is challenging. Differences in block timestamps may also result in signature expirations to vary across chains, requiring re-generation.

Code Maturity

OS-OSG-SUG-00

Description

1. The hardcoded `VERSION` in `OneSig` differs from the value utilized in the domain separator. To ensure consistency, it is recommended to utilize the same value in both.
2. It is recommended to add validation in `verifyNSignatures` to disallow a zero `_threshold` from being passed.
3. It would be appropriate to double-hash the leaf in `OneSig::encodeLeaf` to enhance security by reducing the probability of accidental hash collisions.

>_ *OneSig.sol* SOLIDITY

```
/**
* @notice Encodes the transaction leaf for inclusion in the merkle tree.
* @dev Includes the chain-specific TRANSACTION_DOMAIN, the current nonce, and the calls
    ↪  array.
* @param _nonce The nonce of the transaction.
* @param _calls The calls to be made in this transaction.
* @return The keccak256 hash of the encoded leaf.
*/
function encodeLeaf(uint256 _nonce, Call[] calldata _calls) public view returns (bytes32)
    ↪ {
    return keccak256(abi.encode(TRANSACTION_DOMAIN, _nonce, _calls));
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

Fixed in 2d69c32.

Cross-Chain Signature Synchronization OS-OSG-SUG-01

Description

The `seed` must remain consistent across all chains to ensure uniform signature usage. However, chains operate independently, and there is currently no mechanism to simultaneously update the `seed` across all chains when invalidating prior signatures, creating potential synchronization issues. Similarly, differences in `block.timestamp` across chains may require signature regeneration for certain chains due to varying expiry times.

Remediation

Implement a mechanism to synchronize state across multiple blockchains to ensure consistency in signature generation.

Patch

Fixed in 2d69c32.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.