OpenZeppelin | security

OneSig Audit

March 31, 2025

Table of Contents

Table of Contents 2

Summary 3

Scope 4

System Overview 5

Security Model and Trust Assumptions 5

Privileged Roles 6

Low Severity 7

L-01 verifyNSignatures Strictly Requires Exact Threshold Signatures 7

L-02 Invalid verifyingContract Address Breaks EIP-712 Compliance 7

L-03 EVM Version May Lead to Cross-Chain Deployment Issues 8

Notes & Additional Information 9

N-01 Redundant Version String in DOMAIN_SEPARATOR 9

N-02 Missing Docstrings 9

N-03 Lack of Security Contact 9

N-04 Function Visibility Overly Permissive 10

N-05 Floating Pragma 10

N-06 Lack of Indexed Event Parameters 10

Conclusion 12

Summary

Type	DeFi/Stablecoin
Timeline	From 2025-03-24 To 2025-03-28
Languages	Solidity

Total Issues	9 (0 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	3 (0 resolved)
Notes & Additional Information	6 (0 resolved)

Scope

We audited the Everdawn-Labs/OneSig repository at commit 5ec9d07.

In scope were the following files:

```
packages/onesig-evm/contracts/MultiSig.sol
packages/onesig-evm/contracts/OneSig.sol
```

System Overview

The OneSig is a smart contract system designed to facilitate secure, cross-chain execution of pre-authorized transactions. It enables a group of signers to collectively approve a batch of transactions off-chain, which are then represented on-chain as a Merkle root. Once the batch is signed, any user can execute individual transactions without additional permissions, as long as a valid Merkle proof and sufficient signatures are provided.

Each transaction batch is uniquely identified by a nonce and a deployment-specific identifier, ensuring ordered execution and preventing replay. This design aims to improve scalability and reduce operational complexity for multi-chain deployments, especially in environments like LayerZero where frequent configuration updates across numerous chains are necessary.

Security Model and Trust Assumptions

This audit only covers the on-chain components of the OneSig system. The off-chain infrastructure responsible for generating Merkle trees was not included in the scope. As such, we assume that this off-chain system correctly implements the double-hashing and encoding process defined in the `encodeLeaf` function of the contract.

We also assume that all signers configured in the `OneSig` contract are trustworthy entities, and that the `threshold` value is set responsibly — i.e., not too low relative to the number of signers, in accordance with best practices for multisig systems.

Additionally, both OpenZeppelin and Everdawn are aware that the system allows for signature reuse across different chains due to the use of a static EIP-712 domain. Based on the protocol's documentation and stated design goals, we assume that this cross-chain signature reusability is intentional and not considered a security risk by the project team.

Privileged Roles

The privileged actors in the system are the designated multisig signers. They are responsible for authorizing transaction batches by signing Merkle roots off-chain. Through the contract's internal logic, these signers can also collectively update the signer set and the contract’s seed value, which plays a role in signature validity. All such actions require the same threshold of signatures used for transaction execution.

Low Severity

L-01 `verifyNSignatures` Strictly Requires Exact Threshold Signatures

The `verifyNSignatures` function requires the number of provided signatures to be exactly equal to `threshold`, rejecting any input with more than the required number of signatures. This design limits flexibility and may cause execution failures in practical scenarios where extra valid signatures are included (e.g., by automated signers or off-chain tooling). It also diverges from widely adopted multisig patterns, such as Gnosis Safe, which accept any number of valid signatures greater than or equal to the threshold.

Consider relaxing the check to allow the number of signatures to be greater than or equal to the threshold. This improves compatibility with common multisig practices and reduces the chances of transaction rejections due to harmless over-signing.

L-02 Invalid `verifyingContract` Address Breaks EIP-712 Compliance

The `OneSig` contract defines a fixed `DOMAIN_SEPARATOR` using `verifyingContract = address(0xdEaD)` and `chainId = 1`, to allow reuse of the same signature across different chains. While this approach enables cross-chain transaction execution from a single off-chain signature, it breaks compliance with the EIP-712 standard, which expects `verifyingContract` to be the actual contract address. This deviation undermines the security guarantees of EIP-712 by removing the binding between a signature and a specific contract instance, and could lead to confusion or incompatibility with external tooling.

While maintaining the current setup (with `0xdEaD`) works functionally, it sacrifices some security properties of EIP-712 and may lead to compatibility issues with off-chain systems or libraries that expect `verifyingContract` to match the actual contract address.

Consider deploying `OneSig` contracts in a way that maintains signature reusability while preserving EIP-712 compliance. Two possible approaches are:

1. **Deterministic deployments using CREATE2** – Deploying `OneSig` contracts at the same address across all chains would allow `verifyingContract` to be set to

`address(this)`, enabling safe reuse of signatures without breaking the EIP-712 standard. This would require a consistent factory contract or an established deployment process (e.g., using a singleton factory).

2. **Nick's Method** – An alternative is to use Nick's Method to deploy contracts at a fixed address across chains. However, this approach can be more error-prone, especially when managing gas fees and ensuring the successful execution of pre-signed transactions. Additional care must be taken to avoid inconsistencies or failed deployments.

L-03 EVM Version May Lead to Cross-Chain Deployment Issues

The project does not specify `evmVersion` in the Foundry or Hardhat configuration files. As a result, the contracts are compiled using the default EVM version associated with the compiler (likely `cancun` in newer Solidity versions like `^0.8.22`). This may introduce incompatibilities when deploying to blockchains that do not yet support newer opcodes such as `PUSH0` or `MCOPY`, which were introduced in the Shanghai and Cancun upgrades respectively.

Deploying contracts that include unsupported opcodes on older EVM versions can lead to deployment failures or unexpected behavior on chains that haven't yet adopted these upgrades.

Consider explicitly setting `evmVersion` to `"paris"` (the last widely supported EVM version prior to newer opcodes being introduced) in both Foundry and Hardhat configurations. This ensures broader compatibility with a wide range of mainnets and L2s, many of which may not yet support the latest EVM opcodes. In addition, consider verifying the final bytecode produced during compilation to ensure that unsupported opcodes like `PUSH0` and `MCOPY` are not present.

Notes & Additional Information

N-01 Redundant Version String in `DOMAIN_SEPARATOR`

The `OneSig` contract defines the `"0.0.1"` version string manually when constructing the `DOMAIN_SEPARATOR`, despite already declaring it as a constant (`VERSION`). This results in redundancy and may lead to inconsistencies if the version is updated in one place and not the other.

Consider using the existing `VERSION` constant in the construction of the `DOMAIN_SEPARATOR` instead of hardcoding the version string again. This improves maintainability and reduces the risk of version mismatches during future upgrades or audits.

N-02 Missing Docstrings

Within `OneSig.sol`, in the `LEAF_ENCODING_VERSION` state variable, the docstring is missing.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

N-03 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

N-04 Function Visibility Overly Permissive

Throughout the codebase, multiple instances of functions with unnecessarily permissive visibility were identified:

- The `getSigners` function in `MultiSig.sol` with `public` visibility could be limited to `external`.
- The `setSeed` function in `OneSig.sol` with `public` visibility could be limited to `external`.
- The `executeTransaction` function in `OneSig.sol` with `public` visibility could be limited to `external`.

To better convey the intended use of functions and to potentially realize some additional gas savings, consider changing a function's visibility to be only as permissive as required.

N-05 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

`MultiSig.sol` and `OneSig.sol` have the `solidity ^0.8.22` floating pragma directive.

Consider using fixed pragma directives.

N-06 Lack of Indexed Event Parameters

Throughout the codebase, multiple instances of events not having any indexed parameters were identified:

- The `SignerSet` event in `MultiSig.sol`
- The `ThresholdSet` event in `MultiSig.sol`
- The `SeedSet` event in `OneSig.sol`

- The `TransactionExecuted` event in `OneSig.sol`

To improve the ability of off-chain services to search and filter for specific events, consider indexing event parameters.

Conclusion

The `MutiSig` contract requires a threshold of signatures from approved signers before executing a transaction. The `OneSig` contract allows a single signature set to be re-used across multiple chains by using a fixed domain separator and verifying the contract address. The codebase was found to be clean and well-written, with the audit only yielding low-severity issues along with various recommendations for code improvement.