PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For LayerZero (EVM OneSig)

19 March 2025

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for LayerZero's OneSig contracts on the Ethereum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	LayerZero
URL	https://layerzero.network/
Platform	Ethereum
Language	Solidity
Preliminary Contracts	https://github.com/LayerZero-Labs/OneSig/blob/2d69c329f6cf0f6cd453d1b02b925d340aa05e58
Resolution #1	https://github.com/LayerZero-Labs/OneSig/pull/121/commits/a5b7e86285a6f18d6084aeb433115b341ffc8d0e
Resolution #2	https://github.com/LayerZero-Labs/OneSig/commit/d44bc555099d5599b12bb341136cd088005120d9

1.2 Contracts Assessed

Name	Contract	Live Code Match
MultiSig		
OneSig		
MultiSigHarness		
OneSigHarness		
ExecutorStore		
SelfCallable		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	0	-	-	-
Medium	0	-	-	-
Low	2	1	-	1
Informational	6	1		2
Total	**8**	**2**	**0**	**3**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 MultiSig

ID	Severity	Summary	Status
01	INFO	Signature verification should offer a non-reverting function	ACKNOWLEDGED
02	INFO	Typographical issues	

1.3.2 OneSig

ID	Severity	Summary	Status
03	LOW	Unordered execution might become possible due to reentrancy	✓ RESOLVED
04	LOW	Old signatures could be reactivated if the seed is set to an old value	ACKNOWLEDGED
05	INFO	Missing gas configuration per call	ACKNOWLEDGED
06	INFO	Passing chain id as a constructor argument	✓ RESOLVED
07	INFO	Typographical issues	

1.3.3 MultiSigHarness

No issues found.

1.3.4 OneSigHarness

No issues found.

1.3.5 ExecutorStore

ID	Severity	Summary	Status
08	INFO	Typographical issues	

1.3.6 SelfCallable

No issues found.

2 Findings

2.1 MultiSig

`MultiSig` is an abstract contract that manages a set of signers and a signature threshold. Designed to be inherited by contracts requiring multi-signature verification.

Features:

- Requires signatures to be in strictly ascending order by signer address (to prevent duplicates)
- Each signature must be exactly 65 bytes (r=32, s=32, v=1)
- Uses OpenZeppelin's ECDSA library for signature recovery
- Signatures are concatenated together in a single bytes parameter when they need to be verified

As this is an abstract contract, there are no replay/expiry protection in place — the contract holds only signers in its state.

2.1.1 Privileged Functions

- `setThreshold [MULTISIG]`
- `setSigner [MULTISIG]`

2.1.2 Issues & Recommendations

Issue #01	Signature verification should offer a non-reverting function
Severity	INFORMATIONAL
Description	The signature verification function reverts in certain cases: - If _threshold is 0 - If _signatures.length does not equal _threshold * 65 - If any recovered signer address is less than or equal to the previous signer (unsorted or duplicate) - If any recovered signer is not in the authorized signers set - If any signature recovery fails It may be beneficial to implement a version that returns true/false instead. This would allow implementations to handle invalid signatures more gracefully without needing to wrap the call in a try/catch block.
Recommendation	Consider crafting an additional `view` function that returns true if the signature is valid and false if the signature is not valid.
Resolution	ACKNOWLEDGED

Issue #02	Typographical issues
Severity	INFORMATIONAL
Description	Consider replacing the number 65 throughout the code with a constant variable for increased readability. In general, avoid using magic numbers. — todo comments should be removed. — * Consider updating the verifyNSignatures() function comment from "The number of signatures doesn't match N (each signature is 65 bytes)." to "The number of signatures are below the the threshold" — * Consider replacing `SignatureError` with `InsufficientSignatureLength` on L170 and on L172 with `InsufficientSignatureCount` for better readability during reverts. — * The comment "* @dev Verifies that exactly threshold signatures are present, sorted by ascending signer addresses." is misleading because it must be at least, not exactly.
Recommendation	Consider fixing the typographical issues above.
Resolution	

2.2 OneSig

`OneSig` extends the `MultiSig` contract and uses a Merkle tree of transaction leaves.

It allows batch of transactions to be signed once (off-chain) and then executed on multiple chains as long as the Merkle proof is valid, the threshold of signers is met and that the batch of transactions are intended for the specific chain.

2.2.1 Privileged Functions

- `setThreshold [MULTISIG]`
- `setSigner [MULTISIG]`
- `setSeed [MULTISIG]`

2.2.2 Issues & Recommendations

Issue #03	Unordered execution might become possible due to reentrancy
Severity	LOW SEVERITY
Description	One of the qualities mentioned in the documentation is that the contract must support ordered execution. There is an edge case where one of the calls executed by `executeTransaction()` could call a contract that has the ability to reenter the same function with the next valid merkle root and the rest of the function arguments if they are known beforehand. This would execute part of the calls in the batch out of order and could cause unexpected behaviour.
Recommendation	Cconsider using a reentrancy guard for `executeTransaction()`.
Resolution	RESOLVED

Issue #04	Old signatures could be reactivated if the seed is set to an old value
Severity	LOW SEVERITY

Description	The seed is included in the signatures so that if there is a need to revoke a set of previously unexecuted signatures, the seed can be changed.
	
	The seed is used when verifying the Merkle root:
	`// Compute the EIP-712 hash`
	`bytes32 digest = keccak256(`
	`abi.encodePacked(`
	`EIP191_PREFIX_FOR_EIP712,`
	`DOMAIN_SEPARATOR,`
	`keccak256(abi.encode(SIGN_MERKLE_ROOT_TYPE_HASH, seed, _merkleRoot, _expiry))`
	`)`
	`);`
	
	The issue is that old seeds are not stored in a mapping of used seeds. This means that if the seed is reset to a previous value that was initially changed, all signatures that were previously invalidated become valid again.
	
	Executions would be possible only if `block.timestamp` did not pass the expiry value and if the current nonce is the one used to create the leaf with.

Recommendation	If this is a desired behavior, consider documenting it properly; if not, consider keeping a mapping of old seeds and add a check in the `setSeed` that makes sure the new seed was not used before.
Resolution	ACKNOWLEDGED

Issue #05	Missing gas configuration per call
Severity	INFORMATIONAL
Description	`executeTransaction` executes a single transaction (which corresponds to a leaf in the Merkle tree) if valid signatures are provided. `for (uint256 i = 0; i < _transaction.calls.length; i++) { (bool success, ) = _transaction.calls[i].to.call{ value: _transaction.calls[i].value }( _transaction.calls[i].data ); // Revert if the call fails if (!success) revert ExecutionFailed(i); }` As seen above, several configurations can be set per call: - `value`: The amount of native currency that can be sent. - `_transaction.calls[i].data`: The data sent in the call to the target. However, one parameter is missing: the gas forwarded. Some calls may require a gas limit, but currently, there is no way to set this per call. For example, consider the following calls: - Call 1 – No gas requirements. - Call 2 – Requires a maximum of 100k gas. - Call 3 – No gas requirements. This configuration cannot be set, meaning all calls will execute without gas limitations.
Recommendation	Consider if this is a desired feature to be implemented.
Resolution	ACKNOWLEDGED

Issue #06	Passing chain id as a constructor argument
Severity	INFORMATIONAL
Description	Consider using `block.chainId` instead of passing the chain id as a constructor argument in order to be more explicit and to not have situations in which the provided chain id is different from the actual `block.chainId`. Additionally, the `CHAIN_ID` is not used throughout the code so it can be removed.
Recommendation	Consider implementing the suggestions mentioned above.
Resolution	RESOLVED

Issue #07	Typographical issues
Severity	INFORMATIONAL
Description	On line 66, *targetted* should be *targeted*. — The constructor NatSpec is incomplete as the parameters are not ordered and the `_executorRequired` is missing.
Recommendation	Consider fixing the typographical issues above.
Resolution	

2.3 MultiSigHarness

`MultiSigHarness` is a helper contract that provides the missing view functions needed for complete formal verification testing.

2.3.1 Privileged Functions

- `setThreshold [MULTISIG]`
- `setSigner [MULTISIG]`

2.3.2 Issues & Recommendations

No issues found.

2.4 OneSigHarness

`OneSigHarness` is a helper contract that provides the missing view functions needed for complete formal verification testing.

2.4.1 Privileged Functions

- `setThreshold [MULTISIG]`
- `setSigner [MULTISIG]`
- `setSeed [MULTISIG]`

2.4.2 Issues & Recommendations

No issues found.

2.5 ExecutorStore

`ExecutorStore` has setter functions for the management of executor addresses that will be responsible for executing transactions in OneSig if the executor permission is turned on.

2.5.1 Privileged Functions

- `setExecutorRequired [MULTISIG]`
- `setExecutor[MULTISIG]`

2.5.2 Issues & Recommendations

Issue #08	Typographical issues
Severity	INFORMATIONAL
Description	Consider renaming `executorRequired` to `permissionlessExecution` to better describe the functionality since not only executors but signers too can execute transactions. — On L10, "a whether" should be replaced with "whether". — On L30 and L34 “execute” should be “executor”. — Code comment on L54 suggests that if executors are empty, `executorsRequired` will be set to `false` but this condition is not enforced in code. Additionally, there could be a case where executors are empty, `executorsRequired` is `true` and only signers can execute transactions. Consider updating the code comment or enforce that `executorsRequired` is set to `false` in code if executors are indeed empty. — On L124, replace “a executor” with “an executor”. — The constructor NatSpec is missing `_executorRequired`.
Recommendation	Consider fixing the typographical issues above.
Resolution	

2.6 SelfCallable

`SelfCallable` introduces a modifier that restricts access to functions that have it so they can only be called via the contract itself.

2.6.1 Issues & Recommendations

No issues found.


PALADIN
BLOCKCHAIN SECURITY