Lido

# BLS Library

## Security Assessment Report

*Version: 2.0*

January, 2026

## Contents

# Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Lido components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

## Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

## Document Structure

The first section provides an overview of the functionality of the Lido components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Lido components in scope.

## Overview

Lido is a liquid staking solution for the Ethereum proof-of-stake (PoS) consensus layer. It is built to improve the liquidity, composability and accessibility experience for those participating in consensus layer staking. Users can deposit ETH into a smart contract in order to mint a corresponding amount of stETH in return.

This review focuses on on-chain BLS signature validation for new validators. The BLS signature supplied as part of the predeposit process is verified on-chain to ensure correctness and to provide security against invalid or malicious deposits.

# Security Assessment Summary

## Scope

The review was conducted on the files hosted on the lidofinance/core repository.

The scope of this time-boxed review was strictly limited to files at commit a70087d. This audit focused specifically only on `contracts/common/lib/BLS.sol` and functions imported to `BLS.sol` and supplied by other files.

Retesting was performed at commit 33f2f59 to verify the resolution of previously reported issues.

*Note: third party libraries and dependencies were excluded from the scope of this assessment.*

## Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity antipatterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

## Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

## Findings Summary

The testing team identified a total of 6 issues during this assessment. Categorised by their severity:

- Low: 1 issue.
- Informational: 5 issues.

## Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Lido components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

# Summary of Findings

| ID | Description | Severity | Status |
|---|---|---|---|
| LIDO4-01 | Amount Truncation In Deposit Message Signature Validation | Low | Resolved |
| LIDO4-02 | Error Cases Not Returned To Caller | Informational | Resolved |
| LIDO4-03 | Redundant Memory Write In Gas Optimized Function | Informational | Resolved |
| LIDO4-04 | Undocumented Working Memory Allocation In Assembly | Informational | Resolved |
| LIDO4-05 | Contract Requires Cancun EVM Version For `mcopy` Opcode | Informational | Resolved |
| LIDO4-06 | Lack Of `returndatasize` Validation In Precompile Calls | Informational | Resolved |

| **LIDO4-01** | Amount Truncation In Deposit Message Signature Validation | | |
|---|---|---|---|
| Asset | `contracts/common/lib/BLS.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `depositMessageSigningRoot()` function in `BLS.sol` performs the integer division `amount / 1 gwei` without validating that the input amount is gwei-aligned. This causes sub-gwei precision to be truncated during signing root computation.

As a consequence, different deposit amounts within the same gwei range will produce identical signing roots, meaning a BLS signature created for 1.0 gwei would also validate successfully for 1.5 gwei or 1.9 gwei, or any value in the range [1.0 gwei, 2.0 gwei).

contracts/common/lib/BLS.sol::depositMessageSigningRoot()

```
function depositMessageSigningRoot(
    bytes calldata pubkey,
    uint256 amount,
    bytes32 withdrawalCredentials,
    bytes32 depositDomain
) internal view returns (bytes32 root) {
    root = sha256Pair(
        sha256Pair(
            sha256Pair(pubkeyRoot(pubkey), withdrawalCredentials),
            sha256Pair(
                SSZ.toLittleEndian(amount / 1 gwei),  //@audit Truncation occurs here
                bytes32(0)
            )
        ),
        depositDomain
    );
}
```

The practical concern arises from the public `verifyDepositMessage()` function in `PredepositGuarantee.sol`, which performs no validation of the amount before delegating to the BLS library. Offchain integrators calling this view function can pass non-gwei-aligned amounts and receive misleading validation results, causing them to believe a signature is valid for an amount it was not actually signed for.

contracts/0.8.25/vaults/predeposit_guarantee/PredepositGuarantee.sol::verifyDepositMessage()

```
function verifyDepositMessage(
    IStakingVault.Deposit calldata _deposit,
    BLS12_381.DepositY calldata _depositsY,
    bytes32 _withdrawalCredentials
) public view {
    BLS12_381.verifyDepositMessage(
        _deposit.pubkey,
        _deposit.signature,
        _deposit.amount, // @audit amount not checked in public function
        _depositsY,
        _withdrawalCredentials,
        DEPOSIT_DOMAIN
    );
}
```

The impact is limited to potential confusion in offchain integrations and future development. Direct fund loss is not possible because the Ethereum deposit contract blocks invalid amounts. Accounting discrepancies could arise if offchain systems verify signatures with truncated amounts, and developers unfamiliar with this precision limitation might make incorrect assumptions when extending the codebase.

## Recommendations

Consider adding explicit validation at the BLS library level to prevent confusion and ensure signature verification accurately reflects Ethereum consensus layer requirements. The simplest approach would be to add a gwei alignment check in `depositMessageSigningRoot()`.

Alternatively, validation could be added specifically in the public-facing `PredepositGuarantee.verifyDepositMessage()` function to protect offchain integrators from misuse. At minimum, clear documentation should be added explaining the truncation behaviour and the requirement that callers ensure gwei alignment before invoking these functions.

## Resolution

The issue has been resolved by adding a validation check in the `PredepositGuarantee.verifyDepositMessage()` function to ensure the amount is gwei-aligned before proceeding with signature verification.

Changes can be seen in commit 05e19ec.

| LIDO4-02 | Error Cases Not Returned To Caller |
|---|---|
| Asset | `contracts/common/lib/BLS.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

Two functions, `sha2()` and `modfield()`, do not return error messages if calls to precompile addresses are unsuccessful. Opaque reverts increase friction when debugging reverting calls, especially if encountered by third parties unfamiliar with the contract source code.

The two functions are shown below:

contracts/common/lib/BLS.sol::hashToG2()

```
/// @dev Calls SHA256 precompile with `data_` of length `n_`, returns 32-byte hash
function sha2(data_, n_) -> _h {
    if iszero(and(eq(returndatasize(), 0x20), staticcall(gas(), SHA256, data_, n_, 0x00, 0x20))) {
        revert(calldatasize(), 0x00) // Revert on failure //@audit no error message is returned
    }
    _h := mload(0x00) // Load and return hash result
}

/// @dev Modular reduction using MOD_EXP precompile: computes `base^1 mod p` i.e. `base mod p`.
/// @param s_ Pointer to the MOD_EXP input buffer:
///        [baseLen|expLen|modLen|base(64)|exp(32)=1|mod(64)=p] (total 0x100 bytes).
/// @param b_ Pointer to the 64-byte big-endian `base` value (overwritten with the reduced value).
function modfield(s_, b_) {
    mcopy(add(s_, 0x60), b_, 0x40) // Copy base (64 bytes) into structure
    if iszero(and(eq(returndatasize(), 0x40), staticcall(gas(), MOD_EXP, s_, 0x100, b_, 0x40))) {
        revert(calldatasize(), 0x00) // Revert on failure //@audit no error message is returned
    }
}
```

While this does not pose a security issue, given that all other revert pathways propagate an error message to the end user, the testing team chose to raise this as an informational finding.

## Recommendations

Add custom error messages to indicate the reason for reverting when encountering errors in these functions.

## Resolution

Custom error messages have been added to the revert statements in both `sha2()` and `modfield()` functions.

Changes can be seen in commit 205b50f.

| LIDO4-03 | Redundant Memory Write In Gas Optimized Function |
|---|---|
| Asset | `contracts/common/lib/BLS.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

In the `hashToG2()` function there is a redundant memory write operation where the data written is never subsequently read.

The affected code section is shown below:

contracts/common/lib/BLS.sol::hashToG2()

```
// === DST prime and initial hash ===
let b0 := sha2(s, sub(dstPrime(add(0x02, o), 0), s)) // First SHA2 with DST index 0
mstore(0x20, b0) // Save `b0` for use in XOF loop //@audit b0 is written to memory at 0x20 but never read from later
mstore(s, b0) // Store b0 at start of buffer
mstore(b, sha2(s, sub(dstPrime(add(0x20, s), 1), s))) // Store next hash at `b`
```

While not posing a security risk, given the level of gas optimisation present throughout the contract, the testing team thought this observation was worthwhile raising as an informational finding.

## Recommendations

The additional memory write should be removed to further reduce the gas cost of calling `hashToG2()`.

## Resolution

The issue is resolved by removing the redundant memory write operation.

Changes can be seen in the commit f1a938b.

| LIDO4-04 | Undocumented Working Memory Allocation In Assembly |
|---|---|
| Asset | `contracts/common/lib/BLS.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

The `hashToG2()` function allocates working memory in assembly without documenting the memory layout or updating the free memory pointer, which could confuse auditors reviewing the code for potential memory corruption vulnerabilities.

The `hashToG2()` function reads the free memory pointer to allocate working buffers but does not update it after use:

contracts/common/lib/BLS.sol::hashToG2()

```
// === Begin Main Logic ===

let b := mload(0x40) // Allocate free memory pointer `b`
let s := add(b, 0x100) // Pointer to working buffer after `b`
mstore(add(s, 0x40), message) // Store the message at `s + 0x40`
// ... extensive use of b and s as working memory ...
```

The code allocates working memory at pointer `b` and `s` at `b + 0x100` without updating the free memory pointer at `0x40`. This pattern is safe in this specific context because the function is `internal view`, the working memory is only used within the assembly block, and the returned `G2Point memory result` is properly managed by Solidity's memory allocation system. However, the lack of documentation explaining this memory usage pattern and why not updating the free memory pointer is safe could lead auditors to incorrectly flag this as a potential memory corruption vulnerability.

This issue is rated as informational as it presents no actual security risk.

## Recommendations

Consider adding inline documentation explaining the memory layout and why the free memory pointer is not updated.

## Resolution

Documentation has been added in commit 0d629e4 to explain the working memory allocation pattern in assembly within the `hashToG2()` function and why the free memory pointer is not updated.

| LIDO4-05 | Contract Requires Cancun EVM Version For `mcopy` Opcode |
|---|---|
| Asset | `contracts/common/lib/BLS.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

The `BLS12_381` library uses the `mcopy` opcode which is only available on chains that have implemented the Cancun hard fork or later, limiting deployment compatibility.

The `modfield()` assembly function within `hashToG2()` uses the `mcopy` opcode to copy memory:

contracts/common/lib/BLS.sol::hashToG2()::modfield()

```
function modfield(s_, b_) {
    mcopy(add(s_, 0x60), b_, 0x40) // Copy base (64 bytes) into structure
    if iszero(and(eq(returndatasize(), 0x40), staticcall(gas(), MOD_EXP, s_, 0x100, b_, 0x40))) {
        revert(calldatasize(), 0x00) // Revert on failure
    }
}
```

The `mcopy` opcode was introduced in EIP-5656 as part of the Cancun network upgrade. This opcode provides efficient memory-to-memory copying but is not available on chains that have not yet upgraded to Cancun or later EVM versions. The contract will fail to deploy or execute on pre-Cancun chains, including Ethereum mainnet prior to the Cancun upgrade, Layer 2 networks that have not implemented Cancun, or private networks running older EVM versions.

This issue is rated as informational as Ethereum mainnet has implemented the Cancun upgrade, and the primary deployment target is expected to support this opcode.

## Recommendations

Consider documenting the minimum required EVM version in the contract comments or deployment documentation to ensure operators are aware of the Cancun requirement.

## Resolution

Documentation has been added in commit 0d629e4 to inform users of the Cancun EVM version requirement due to the use of the `mcopy` opcode.

| LIDO4-06 | Lack Of `returndatasize` Validation In Precompile Calls |
|---|---|
| Asset | `contracts/common/lib/BLS.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

The utility functions `sha256Pair()` and `pubkeyRoot()` do not validate the return data size when calling the SHA256 precompile, only checking the success flag.

The `sha256Pair()` function calls the SHA256 precompile without verifying the returned data size:

contracts/common/lib/BLS.sol::sha256Pair()

```
// Call SHA-256 precompile with 64-byte input at memory 0x00.
let success := staticcall(gas(), SHA256, 0x00, 0x40, 0x00, 0x20)
if iszero(success) {
    revert(0, 0)
}

// Load the resulting hash from memory
result := mload(0x00)
```

Similarly, the `pubkeyRoot()` function also only validates the success flag without checking `returndatasize()`.

Defence-in-depth practices suggest validating that precompiles return the expected data size, even though the SHA256 precompile should always return exactly 32 bytes.

This issue is rated as informational as the SHA256 precompile on Ethereum mainnet always returns 32 bytes and the practical security risk is negligible.

## Recommendations

Consider adding `returndatasize()` validation to `sha256Pair()` and `pubkeyRoot()` to ensure the precompile returns exactly 32 bytes as a defence-in-depth measure.

## Resolution

The issue is resolved by adding `returndatasize()` validation to both `sha256Pair()` and `pubkeyRoot()` functions to ensure the SHA256 precompile returns exactly 32 bytes.

Changes can be seen in the commit 33f2f59.

## Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

|  |  | Low | Medium | High |
|---|---|---|---|---|
| **Impact** | High | Medium | High | Critical |
|  | Medium | Low | Medium | High |
|  | Low | Low | Low | Medium |

**Likelihood**

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

## References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

σ'