Veridise

# Auditing Report

## Hardening Blockchain Security with Formal Methods

FOR

# Security Consortium Smart Contracts

Veridise Inc.

Dec. 18, 2024

► **Prepared For:**

Lombard
https://www.lombard.finance/

► **Prepared By:**

Alberto Gonzalez
Jon Stephens
Bryan Tan
Jacob Van Geffen

► **Contact Us:**

contact@veridise.com

► **Version History:**

Dec. 18, 2024 V2
Dec. 17, 2024 V1

# Contents



 

 

# 1 Executive Summary

From Oct. 14, 2024 to Dec. 13, 2024, Lombard engaged Veridise to conduct a security assessment of their Security Consortium Smart Contracts. Specifically, the assessment covered an update to the implementation of the Lombard protocol, which Veridise had reviewed several times before*. Compared to the previous version, the updated version splits up the monolithic codebase for the Consortium validator node into separate, simpler, and more focused components. Veridise conducted the assessment over 18 person-weeks, with 4 security analysts reviewing the project over 9 weeks on the commits of the following projects:

- Ledger: 77dd574
- Approver: 420efcb
- Smart contracts: 109a3f26 - b7d71250

The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Although the engagement involved a review of all three projects, this report focuses solely on the smart contracts. The audit report for the Ledger and Approver can be found on the Veridise website (see the footnote).

**Project Summary.** The security assessment covered three major components of the Lombard Security Consortium: the ledger, the approver, and the smart contracts. Altogether, these components facilitate the following user actions:

- *Deposit BTC*: a user can exchange Bitcoin (BTC) to a custom token named LBTC by sending BTC to a public key derived from Lombard's deposit wallet public key. Lombard services will then call the LBTC smart contract of the user-specified EVM-based chain to mint an equivalent amount of LBTC tokens for the user-specified recipient.
- *Redeem LBTC (Unstake)*: a user can exchange their LBTC to BTC by burning their LBTC tokens on an Ethereum-based chain. Lombard services will then send an equivalent amount of BTC (minus a fee) to the user-specified Bitcoin script pubkey (i.e., address).
- *EVM-to-EVM Bridged Deposit*: a user can use Lombard's bridge smart contract to send their LBTC tokens to another EVM blockchain that supports the Lombard protocol. This is facilitated by off-chain Lombard services as well as third-party bridging protocols.

This report specifically focuses on the smart contract review for the Lombard Security Consortium. The smart contracts consist of a `Consortium` smart contract that records the validator set and public keys, an ERC-20 token contract called `LBTC`, a set of `Bridge` smart contracts, and a swapping contract that allows BTCB tokens (not to be confused with BTC) to be swapped (one-way) to LBTC tokens. The `LBTC` contract implements the EVM-portion of the deposit and redeem/unstake actions, and the `Bridge` contract implements the EVM-to-EVM bridge deposits. The "sending" side of the actions will be handled by off-chain Lombard services, which will request notarization of the actions (i.e., to get authorization) and then call the smart contracts to

* The previous audit reports, if they are publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

 

execute the action. In turn, the "receiving" side of the actions, such as the deposit functionality of `LBTC` and the token receiving side of the `Bridge`, will use the `Consortium` contract to validate that the actions have been signed by the ledger's validator set.

Lombard's bridge contracts contain a custom bridge and integrations with two existing bridges: LayerZero and Chainlink's Cross-Chain Interoperability Protocol (CCIP). The Lombard bridge itself is used to emit bridging events that correspond to an EVM-to-EVM bridged deposit mentioned above. Upon a deposit, an event will be emitted for notarization by the Lombard consortium. These bridge requests may also be processed by existing bridging infrastructure to deliver the bridging payload to the destination endpoint. At the destination, the Lombard bridge will process messages from existing bridges and will validate consortium notarization if necessary before minting bridged funds to users.

Of the two integrations with existing bridges, the CCIP integration is more complex as it directly integrates with Lombard's bridge. This means that the Lombard bridge will submit messages over the CCIP. If a user direcly bridges using the CCIP, it also will eventually invoke the Lombard bridge so that messages can undergo notarization. When a CCIP message is delivered, the Lombard bridge will also perform additional validation such as checking the notarization that is provided to the CCIP as off-chain token data. The LayerZero integration is simpler, as it implements an OTCAdapter smart contract that rate-limits funds sent and received by a given endpoint. It also contains an adapter to override the default credit and debit functionality so that tokens would be minted and burnt directly from the adapter rather than using the default lock/unlock mechanism.

**Code Assessment.** The Lombard developers provided the source code of the Security Consortium Smart Contracts for review. The code appears to be mostly original with some components based on utilities provided by LayerZero and Chainlink's CCIP. The source code contains some documentation in the form of READMEs and documentation comments on functions and storage variables; however, documentation on the high level organization of the code itself was lacking. The source code contained a test suite, which the Veridise security analysts noted contains unit tests that exercise a variety of positive scenarios.

During the security assessment, the Lombard developers made several functional changes to the smart contract code. This is because the smart contract code, particularly those related to the bridge, was still in development during the audit. Consequently, Veridise had to assign two more security analysts towards the end of the audit to cover the updated smart contract code, resulting in delays.

**Summary of Issues Detected.** The security assessment uncovered 30 issues, 6 of which are assessed to be of high or critical severity by the Veridise analysts. More specifically, V-CSC-VUL-001 identifies missing access control that could allow arbitrary LBTC to be created during the bridging process, V-CSC-VUL-002 identifies a reentrancy in the Bridge contract allowing funds to be created during the bridging process and V-CSC-VUL-004 identifies an incorrect interaction with the CCIP contracts that could prevent bridge requests from being processed. The Veridise analysts identified 4 medium-severity issues, including the potential for non-configured bridge destinations to skip essential validation (V-CSC-VUL-008), as well as 5 low-severity issues, 13 warnings, and 2 informational findings.

Among the 30 issues, 27 issues have been acknowledged by Lombard, and 3 issues have been determined to be intended behavior after discussions with Lombard. Of the 27 acknowledged issues, Lombard has fixed 21 issues and does not plan to fix the other 6 acknowledged issues at this time.

**Recommendations.** After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Security Consortium Smart Contracts. More specifically, we would recommend more thorough testing of the Security Consortium Smart Contracts, particularly focusing on the bridge contracts where a majority of the significant vulnerabilities were reported. We also strongly recommend testing flows that do not originate from the Bridge contract as the analysts identified several errors that would only be identified by testing different entry points (V-CSC-VUL-001, V-CSC-VUL-004, V-CSC-VUL-014). We additionally recommend writing negative tests for the infrastructure to ensure undesired behavior is not possible.

**Disclaimer.** We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

# 2 Project Dashboard

**Table 2.1:** Application Summary.

| Name | Version | Type | Platform |
|---|---|---|---|
| Smart Contracts | 109a3f26 - b7d71250 | Solidity | Ethereum |

**Table 2.2:** Engagement Summary.

| Dates | Method | Consultants Engaged | Level of Effort |
|---|---|---|---|
| Oct. 14–Dec. 13, 2024 | Manual & Tools | 4 | 18 person-weeks |

**Table 2.3:** Vulnerability Summary.

| Name | Number | Acknowledged | Fixed |
|---|---|---|---|
| Critical-Severity Issues | 2 | 2 | 2 |
| High-Severity Issues | 4 | 4 | 3 |
| Medium-Severity Issues | 4 | 4 | 4 |
| Low-Severity Issues | 5 | 5 | 3 |
| Warning-Severity Issues | 13 | 11 | 8 |
| Informational-Severity Issues | 2 | 1 | 1 |
| TOTAL | 30 | 27 | 21 |

**Table 2.4:** Category Breakdown.

| Name | Number |
|---|---|
| Data Validation | 13 |
| Logic Error | 6 |
| Maintainability | 3 |
| Access Control | 2 |
| Denial of Service | 2 |
| Reentrancy | 1 |
| Replay Attack | 1 |
| Frontrunning | 1 |
| Usability Issue | 1 |

# 3 Security Assessment Goals and Scope

## 3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Lombard's ledger, approver, and smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Is it possible to replay EVM-to-EVM bridged deposits?
- Do the bridge contracts correctly integrate with third party bridging communication protocols?
- Is sufficient data validation performed in the smart contracts?
- Is it possible to create tokens during a bridging request?
- Is it possible for a single entity to prevent normal bridge operation?
- Can a user steal funds from another user or gain unapproved access to funds?
- Is it possible to replay Consortium proofs or otherwise bypass consortium notarization?
- Is it possible to bypass authorization when depositing funds from Bitcoin?
- Are the smart contracts free from common smart contract vulnerabilities such as reentrancy vulnerabilities?

## 3.2 Security Assessment Methodology & Scope

**Security Assessment Methodology.** To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the Veridise security analysts performed a thorough code review of all components in the scope of the project.

Additionally, the security assessment of the smart contracts was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, the security analysts leveraged Veridise's custom smart contract analysis tool Vanguard. This tool is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

*Scope*. The scope of this security assessment is limited to the following paths of the following projects in the source code provided by the Security Consortium Smart Contracts developers:

- Smart contracts (commit: `109a3f2699bc79dde728c81133bbad23a7f4e9af`)
  - `contracts/**.sol`
  - ...excluding `contracts/bascule/Bascule.sol`, `contracts/bridge/**.sol`, and test/-mock contracts
- Smart contracts (commit: `ebfda9fdbfd710a7c5991aa9876dba8e8af8914a`)
  - `contracts/bridge/**.sol`

 

- `contracts/bascule/BasculeV2.sol`

*Methodology*. Veridise security analysts inspected the provided tests and read the Security Consortium Smart Contracts documentation. They then began a review of the code assisted by both static analyzers and automated testing.

During the security assessment, the Veridise security analysts regularly met with the Security Consortium Smart Contracts developers to ask questions about the code.

## 3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

**Table 3.1:** Severity Breakdown.

| | Somewhat Bad | Bad | Very Bad | Protocol Breaking |
|---|---|---|---|---|
| Not Likely | Info | Warning | Low | Medium |
| Likely | Warning | Low | Medium | High |
| Very Likely | Low | Medium | High | Critical |

The likelihood of a vulnerability is evaluated according to the Table 3.2.

**Table 3.2:** Likelihood Breakdown

| | |
|---|---|
| Not Likely | A small set of users must make a specific mistake |
| Likely | Requires a complex series of steps by almost any user(s)<br>- OR -<br>Requires a small set of users to perform an action |
| Very Likely | Can be easily performed by almost anyone |

The impact of a vulnerability is evaluated according to the Table 3.3:

**Table 3.3:** Impact Breakdown

| | |
|---|---|
| Somewhat Bad | Inconveniences a small number of users and can be fixed by the user |
| Bad | Affects a large number of people and can be fixed by the user<br>- OR -<br>Affects a very small number of people and requires aid to fix |
| Very Bad | Affects a large number of people and requires aid to fix<br>- OR -<br>Disrupts the intended behavior of the protocol for a small group of users through no fault of their own |
| Protocol Breaking | Disrupts the intended behavior of the protocol for a large group of users through no fault of their own |

# 4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

**Table 4.1:** Summary of Discovered Vulnerabilities.

| ID | Description | Severity | Status |
|---|---|---|---|
| V-CSC-VUL-001 | Missing CCIP adapter access control . . . | Critical | Fixed |
| V-CSC-VUL-002 | CCIP adapter reentrancy allows funds to . . . | Critical | Fixed |
| V-CSC-VUL-003 | Bridge payload will be sent to token recipient | High | Fixed |
| V-CSC-VUL-004 | Incorrect return value in Token Pool . . . | High | Fixed |
| V-CSC-VUL-005 | Attacker can DoS CCIP messages that . . . | High | Acknowledged |
| V-CSC-VUL-006 | Transfer-then-call antipattern can cause . . . | High | Fixed |
| V-CSC-VUL-007 | Bridge does not validate toContract . . . | Medium | Fixed |
| V-CSC-VUL-008 | Non configured chains can be used to . . . | Medium | Fixed |
| V-CSC-VUL-009 | Validate payload size on abi.decode | Medium | Fixed |
| V-CSC-VUL-010 | OFTAdapter halt function will not pause . . . | Medium | Fixed |
| V-CSC-VUL-011 | Centralization risk | Low | Acknowledged |
| V-CSC-VUL-012 | Missing validation for fromContract in . . . | Low | Fixed |
| V-CSC-VUL-013 | Missing argument validation in Bridge | Low | Fixed |
| V-CSC-VUL-014 | Chainlink configuration may break . . . | Low | Acknowledged |
| V-CSC-VUL-015 | Unsafe downcast in CLAdapter | Low | Fixed |
| V-CSC-VUL-016 | Treasury address not set in LBTC init | Warning | Fixed |
| V-CSC-VUL-017 | Consortium _checkProof() no-ops on . . . | Warning | Intended Behavior |
| V-CSC-VUL-018 | Missing validation on selector in . . . | Warning | Fixed |
| V-CSC-VUL-019 | Missing validation on the first byte of . . . | Warning | Fixed |
| V-CSC-VUL-020 | Include version in Bridge Payload | Warning | Acknowledged |
| V-CSC-VUL-021 | Remove duplicated and unused function | Warning | Fixed |
| V-CSC-VUL-022 | Missing Validation in CCIP Adapter | Warning | Fixed |
| V-CSC-VUL-023 | OFTAdapter rate limit may be slightly . . . | Warning | Acknowledged |
| V-CSC-VUL-024 | Missing validation in LBTC contract | Warning | Fixed |
| V-CSC-VUL-025 | Fee signatures can be replayed | Warning | Intended Behavior |
| V-CSC-VUL-026 | mintWithFee functions vulnerable to . . . | Warning | Acknowledged |
| V-CSC-VUL-027 | Invalid consortium signatures cause revert | Warning | Fixed |
| V-CSC-VUL-028 | Missing argument validation in . . . | Warning | Fixed |
| V-CSC-VUL-029 | Minor issues in Smart Contracts | Info | Fixed |
| V-CSC-VUL-030 | Different rate limiting behavior | Info | Intended Behavior |

## 4.1 Detailed Description of Issues

### 4.1.1 V-CSC-VUL-001: Missing CCIP adapter access control allows funds to be created

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | ebfda9f |
| **Type** | Access Control | **Status** | Fixed |
| **File(s)** | `bridge/adapters/CLAdapter.sol` | | |
| **Location(s)** | deposit() | | |
| **Confirmed Fix At** | 5e0c436 | | |

The `CLAdapter` contract serves as an intermediary between the `Bridge` and `TokenPool` contracts to ensure compatibility with the CCIP system. This contract contains a `deposit` function that is used by the `Bridge` contract to initiate a token transfer with the CCIP Router via the `ccipSend` call.

The issue arises because the `deposit()` function can be invoked by any address, not just the intended `Bridge` contract. This could allow an attacker to pass a minimal `_amount` (e.g., 1 satoshi in LBTC) while providing a `_payload` with a significantly larger LBTC amount. The `initiateDeposit` function would burn the minimal amount but return the payload (with a big LBTC amount) to the `TokenPool`, which will be used in the withdrawal flow in the destination chain.

**Impact** The lack of access control on the `deposit()` function allows malicious users to create free LBTC tokens on the destination chain. This occurs because the `_amount` burned on the source chain can be set to a value smaller than the amount specified within the `_payload`. Consequently, the amount of LBTC to be minted on the destination chain during the withdrawal flow will exceed the amount burned, leading to the creation of unbacked `LBTC` tokens.

**Recommendation** To mitigate this issue, implement access control on the `deposit()` function to ensure that only the `bridge` contract can invoke it. This can be achieved by adding a modifier that checks the caller's address against the expected `bridge` contract address. Additionally, consider validating that the information within the `payload` is consistent with the other arguments such as `fromAddress`, `_toChain`, `_toAddress` and `_amount`. These validations will prevent possible future changes to the `Bridge` contract which current code makes sure these arguments are consistent with the `payload`.

**Developer Response** The developers fixed the issue by allowing only the `Bridge` contract to call the `deposit` function.

### 4.1.2 V-CSC-VUL-002: CCIP adapter reentrancy allows funds to be created

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | ebfda9f |
| **Type** | Reentrancy | **Status** | Fixed |
| **File(s)** | `bridge/adapters/CLAdapter.sol` | | |
| **Location(s)** | deposit, initiateDeposit | | |
| **Confirmed Fix At** | 2acc58a | | |

The `CLAdapter` contract is used as an intermediate contract for communication between the Lombard bridge and Chainlink's CCIP bridge. It is responsible for sending CCIP messages from the bridge, enabling communication between the two entity's contracts and burning funds that are sent using a CCIP bridge. To send a CCIP message, the CLAdapter's deposit function (shown below) accepts native currency and will send only enough to cover the fees when sending a CCIP message. Any remaining funds are returned to the user via a low-level call with no gas restriction which gives control to the `fromAddress` and can be used to reenter the contract.

```
function deposit(
    address fromAddress,
    bytes32 _toChain,
    bytes32,
    bytes32 _toAddress,
    uint256 _amount,
    bytes memory _payload
) external payable virtual override {
    ...

    _lastBurnedAmount = _amount;
    _lastPayload = _payload;

    ...

    uint256 fee = IRouterClient(router).getFee(chainSelector, message);

    if (msg.value < fee) {
        revert NotEnoughToPayFee(fee);
    }
    if (msg.value > fee) {
        uint256 refundAm = msg.value - fee;
        (bool success, ) = payable(fromAddress).call{value: refundAm}("");
        if (!success) {
            revert CLRefundFailed(fromAddress, refundAm);
        }
    }

    IERC20(address(lbtc())).approve(router, _amount);
    IRouterClient(router).ccipSend{value: fee}(chainSelector, message);
}
```

**Snippet 4.1:** Definition of the `deposit` function

**Impact** Once control is transferred to the `fromAddress`, they can re-enter the `CLAdapter` contract directly through the `deposit` function or via `ccipSend` as it will eventually invoke the `initateDeposit` function shown below. If they do so, the incorrect number of tokens will be burnt either in the reentrant call or the original call depending on the API used. As an example, consider:

1. Invoke `deposit` with `amount = 1` and overpay the fee.
2. On the low-level call, invoke `ccipSend` with `amount = 100` which will be collected by `ccipSend`.
    a) Once `initiateDeposit` is executed by `ccipSend`, 1 LBTC will be burnt since the `lastBurnAmount = 1`.
    b) Note that the `100 LBTC` will still be sent cross-chain but 99 BTC will remain in `CLAdapter`.
3. The outermost `deposit` execution will complete which will execute the else branch of `initiateDeposit`, and since `amount = 1`, another LBTC will be burnt and leave 98 in the `CLAdapter` contract.

Note that the above can also be launched from the `Bridge` contract without violating the reentrancy guard. With 98 LBTC remaining in the contract, however, `deposit` can be invoked again without depositing any funds to bridge the remaining balance. This effectively creates 98 LBTC.

```
function initiateDeposit(
    uint64 remoteChainSelector,
    bytes calldata receiver,
    uint256 amount
) external returns (uint256 lastBurnedAmount, bytes memory lastPayload) {
    _onlyTokenPool();

    if (_lastPayload.length > 0) {
        // just return if already initiated
        lastBurnedAmount = _lastBurnedAmount;
        lastPayload = _lastPayload;
        _lastPayload = new bytes(0);
        _lastBurnedAmount = 0;
    } else {
        IERC20(address(lbtc())).approve(address(bridge), amount);
        (lastBurnedAmount, lastPayload) = bridge.deposit(
            getChain[remoteChainSelector],
            bytes32(receiver),
            uint64(amount)
        );
    }

    bridge.lbtc().burn(lastBurnedAmount);
}
```

**Snippet 4.2:** Definition of the `initiateDeposit` function which is eventually executed by `ccipSend`

**Recommendation** We would recommend the following:

1. Restrict access to the deposit function as suggested by V-CSC-VUL-001.
2. Prefer `transferFrom` over transfer then call as suggested by V-CSC-VUL-006.
3. Track outbound request payloads in the bridge rather than maintaining the last amount and payload.
4. Rather than pushing funds to users via a low-level call in `deposit` allow users to pull refunds.

**Developer Response** The developers implemented most of the recommendation above. They opted not to change the `_lastBurnedAmount` and `_lastPayload` code as without a reentrancy, they should not be vulnerable to attack.

**Veridise Response** While the reentrancy issue is fixed, we wanted to note that if someone could reenter in the future, a similar attack could be possible (i.e. if the `_lastBurnedAmount` flow from `deposit` to `initiateDeposit` can be interrupted). In future developments it is important to ensure a new reentrancy is not introduced in this part of the code and also to ensure that future CCIP upgrades do not introduce a reentrancy. Also note that if the bridge allows tokens with callbacks to be sent in the future, a similar reentrancy attack may be possible.

### 4.1.3 V-CSC-VUL-003: Bridge payload will be sent to token recipient

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | ebfda9f |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `bridge/adapters/CLAdapter.sol` | | |
| **Location(s)** | _buildCCIPMessage() | | |
| **Confirmed Fix At** | 8bb4e9b | | |

The `CLAdapter` contract is responsible for facilitating token transfers across chains using Chainlink's Cross-Chain Interoperability Protocol (CCIP). The `_buildCCIPMessage()` function constructs a message that is sent to the CCIP router. This message includes a `data` field and a `gasLimit` parameter within `extraArgs`. The current implementation incorrectly populates the `data` field with the bridge `payload` and sets a non-zero `gasLimit` (as shown in the code snippet below), which is unnecessary for only token transfers.

```
function _buildCCIPMessage(
        bytes memory _receiver,
        uint256 _amount,
        bytes memory _payload
    ) private view returns (Client.EVM2AnyMessage memory) {

        // ........

        return
            Client.EVM2AnyMessage({
                receiver: _receiver,
                data: data,
                tokenAmounts: tokenAmounts,
                extraArgs: Client._argsToBytes(
                    Client.EVMExtraArgsV2({
                        gasLimit: getExecutionGasLimit,
                        allowOutOfOrderExecution: true
                    })
                ),
                feeToken: address(0) // let's pay with native tokens
            });
    }
```

**Snippet 4.3:** Code snippet from the `_buildCCIPMessage` function of the `CLAdapter.sol` contract.

The `data` and `gasLimit` fields should only be populated if the CCIP message is expected to trigger a call to `receiver.ccipReceive()`. Since the purpose of this contract is to only handle token transfers without sending a message and invoking additional logic on the receiving end, the `data` field should remain empty. Similarly, the `gasLimit` should be set to zero to avoid incurring extra fees charged by the CCIP protocol.

**Impact** The incorrect construction of the CCIP message results in several undesired consequences:

- The `recipient` of the message will necessarily need to be a smart contract implementing the `ccipReceive` function and be able to process the sent `payload` or the message will not

get executed by the CCIP protocol including the `LBTC` transfer.
- Unnecessary gas fees being paid leading to increased operational costs for users when bridging `LBTC` via the CCIP protocol, as they are charged for gas that is not required for the intended token transfer operation.

**Recommendation** To address this issue, the `_buildCCIPMessage()` function should be modified to ensure that the `data` field is not populated. Additionally, the `gasLimit` parameter within `extraArgs` should be set to zero.

**Developer Response** The developers removed the data field but decided not to set `gasLimit` to zero in the message. They can still do so via the `getExecutionGasLimit` variable.

### 4.1.4 V-CSC-VUL-004: Incorrect return value in Token Pool releaseOrMint

| Severity | High | Commit | ebfda9f |
|---|---|---|---|
| Type | Denial of Service | Status | Fixed |
| File(s) | `bridge/adapters/TokenPool.sol` | | |
| Location(s) | releaseOrMint() | | |
| Confirmed Fix At | 6311d0c | | |

The `releaseOrMint()` function in the `TokenPool` contract is the entry point for the CCIP Offramp contract during token transfers in the destination chain. This function interacts with the `CLAdapter` contract to initiate the `LBTC` withdrawal, either with or without attestation, depending on the `isAttestationEnabled` flag.

The issue arises because the value of `LBTC` minted returned by this function is set to `releaseOrMintIn.amount`, which is incorrect. When the deposit is initiated from the `CCIPRouter` instead of the `Bridge` contract, the `payload.amount` (which considers the deposit fee) and `releaseOrMintIn.amount` (which does not) differ. This discrepancy will cause the CCIP Offramp to revert the transaction, as it validates that the token transfer was carried out successfully based on this returned value.

Reference:

`https://github.com/smartcontractkit/ccip/blob/80dd5d2b4867ee67c348fc1b05669c015bafa14a/contracts/src/v0.8/ccip/offRamp/OffRamp.sol#L693`

**Impact** The incorrect amount returned during the withdrawal process will lead to transaction reversion by the CCIP Offramp contract, disrupting the token transfer process for users. This issue is problematic when deposits are initiated from the `CCIPRouter`, as the `payload.amount` and `releaseOrMintIn.amount` differ due to the inclusion of deposit fees in the former.

**Recommendation** To resolve this issue, ensure that the amount returned by `releaseOrMint()` accurately reflects the amount that will be minted to the user.

**Developer Response** The developers implemented the suggested fix.

### 4.1.5 V-CSC-VUL-005: Attacker can DoS CCIP messages that include LBTC transfers due to missing offchainTokenData validation

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Acknowledged |
| **File(s)** | `bridge/adapters/TokenPool.sol` | | |
| **Location(s)** | releaseOrMint() | | |
| **Confirmed Fix At** | N/A | | |

When transferring `LBTC` across different chains users have the possibility to choose between using the Lombard Bridge or to use the CCIP Router in case they want to also send an arbitrary message to the `recipient` along with the `LBTC` transfer. For example, the CCIP Router can be used to perform an action such as:

- Transfer `X` amount of `LBTC` to `recipient` and also deliver a message to the `recipient` to deposit this `LBTC` as collateral on my behalf.

During the deposit in the source chain, the CCIP OnRamp will call the `lockOrBurn` function in the LBTC `TokenPool` contract which will return either the `sha256` hash of the `payload` or the `payload` itself, depending in the `isAttestationEnabled` flag:

```
if (isAttestationEnabled) {
    destPoolData = abi.encode(sha256(payload));
} else {
    destPoolData = payload;
}

return
   Pool.LockOrBurnOutV1({
     destTokenAddress: getRemoteToken(
        lockOrBurnIn.remoteChainSelector
     ),
     destPoolData: destPoolData
});
```

**Snippet 4.4:** Code snippet from the `lockOrBurn` function in the `TokenPool.sol` contract.

For this issue, we will focus on the case where the `abi.encode(sha256(payload))` is returned back to the CCIP OnRamp contract, that is, when `isAttestationEnabled == True`.

On the destination chain, the CCIP OffRamp contract will call the `releaseOrMint` function, which will start the `LBTC` withdrawal based in the `isAttestationEnabled` flag:

```
if (isAttestationEnabled) {
    adapter.initiateWithdrawal(
        releaseOrMintIn.remoteChainSelector,
        releaseOrMintIn.offchainTokenData
    );
} else {
    adapter.initWithdrawalNoSignatures(
        releaseOrMintIn.remoteChainSelector,
        releaseOrMintIn.sourcePoolData
    );
```

```
11 }
```

**Snippet 4.5:** Code snippet from the `releaseOrMint` function of the `TokenPool.sol` contract.

In order for `adapter.initiateWithdrawal` to mint the LBTC to the user, `releaseOrMintIn.offchainTokenData` has to contain a valid `proof` for the provided `payload` which will be validated using the `authNotary` function in the `Bridge` contract:

```
(bytes memory payload, bytes memory proof) = abi.decode(
    offChainData,
    (bytes, bytes)
);

bridge.authNotary(payload, proof);
```

**Snippet 4.6:** Code snippet from the `initiateWithdrawal` function of the `CLAdapter.sol` contract.

However, the `releaseOrMint` function does not perform any validation to ensure that the `payload` within `offchainTokenData` corresponds to the expected `payload` sent during the deposit phase for the current CCIP message.

**Impact** Consider the following scenario:

1. Bob sends a message via the CCIP router to transfer 5 `LBTC` and to deposit them in a lending platform that supports CCIP messages and `LBTC` as collateral.
2. Alice performs a CCIP router transfer of 0.01 `LBTC` (using the same chains as bob).
3. The Lombard Consortium will sign both payloads and send their proofs.
4. Alice will execute her CCIP message but she will use Bob's `payload` and `proof` in the `offchainTokenData`.
    a) Or Alice will execute Bob's CCIP message but she will use her `payload` and `proof` instead.

In this scenario, Alice has managed to use Bob's `payload` and `proof`, which means they cannot be used again. However, for the first case, even though Bob's `recipient` did received the 5 `LBTC`, the tokens were not deposited as collateral as desired.

**Recommendation** To address this issue is recommended to validate that `releaseOrMintIn.sourcePoolData` which is the `sha256` hash value for the `payload` during `lockOrBurn` is equal to the `sha256` hash value of the `payload` contained in the `offchainTokenData` during `releaseOrMint`.

Reference: The CCIP USDC Token Pool which is similar to the `LBTC` pool in the sense that they use an external party to validate the cross chain transfer acknowledge this issue and prevents it by adding a similar validation.

https://github.com/smartcontractkit/ccip/blob/80dd5d2b4867ee67c348fc1b05669c015bafa14a/contracts/src/v0.8/ccip/pools/USDC/USDCTokenPool.sol#L127

**Developer Response** The developers have opted not to fix this issue

### 4.1.6 V-CSC-VUL-006: Transfer-then-call antipattern can cause funds to be misspent

| Severity | High | Commit | ebfda9f |
|---|---|---|---|
| Type | Logic Error | Status | Fixed |
| File(s) | `bridge/adapters/CLAdapter.sol` | | |
| Location(s) | deposit, initiateDeposit | | |
| Confirmed Fix At | c480b44 | | |

The `CLAdapter` contract is used to pass information between Chainlink's CCIP protocol and Lombard's bridge. Doing so requires it to transfer funds between these two entities and to do so and transfer-then-call pattern shown below is used. More specifically, this pattern refers to the case where funds are eagerly transferred to a contract and then a function is invoked to process the transferred funds. We have found that this pattern increases the likelihood of errors as it can allow funds to be inappropriately spent since the contract cannot easily track the receipt of these funds.

```
function _deposit(
    DestinationConfig memory config,
    bytes32 toChain,
    bytes32 toAddress,
    uint64 amount
) internal returns (uint256, bytes memory) {
    ...

    // transfer assets to adapter
    SafeERC20.safeTransferFrom(
        IERC20(address(lbtc())),
        fromAddress,
        address(config.adapter),
        amountWithoutFee
    );
    // let adapter handle the deposit
    config.adapter.deposit{value: msg.value}(
        fromAddress,
        toChain,
        config.bridgeContract,
        toAddress,
        amountWithoutFee,
        payload
    );
    ...
}
```

**Snippet 4.7:** Example of the transfer then call pattern used by the Bridge contract's `_deposit` function

**Impact** The use of the transfer-then-call pattern makes bookkeeping more difficult even in the cases where a contract is not intended to maintain a token balance. Since the contract cannot accurately track the source of funds, the likelihood that funds may be inappropriately spent is increased. Note that we can observe this in the `CLAdapter` contract as this pattern is a

contributing factor to the reentrancy exploit reported in V-CSC-VUL-002. This is because the contract allows intermediate funds to be spent after an incorrect number of tokens are burnt.

**Recommendation** Rather than using the transfer-then-call pattern, use `transferFrom` so that the source and amount of funds are always known.

**Developer Response** The developers have applied the recommendation.

### 4.1.7 V-CSC-VUL-007: Bridge does not validate toContract during withdrawals

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `libs/Actions.sol` | | |
| **Location(s)** | depositBridge() | | |
| **Confirmed Fix At** | aa8c44f | | |

The `withdraw(bytes payload)` function in the `Bridge` contract is used by users to withdraw their `LBTC` by providing a valid `payload`. The `payload` is parsed using the `depositBridge` function from the `Actions.sol` contract, which extracts fields such as `recipient`, `amount`, `toChain` and `toContract` as show in the below code snippet:

```
(
     uint256 fromChain,
     address fromContract,
     uint256 toChain,
     address toContract,
     address recipient,
     uint64 amount,
     bytes32 uniqueActionData
) = abi.decode(
      payload,
        (uint256, address, uint256, address, address, uint64, bytes32)
    );

if (toChain != block.chainid) {
     revert WrongChainId();
}

// ..............
```

**Snippet 4.8:** Code snippet from the `depositBridge` function of the `Actions.sol` contract.

The code then performs several validations to the fields parsed from the `payload` such as validating that `toChain == block.chainId` to ensure they `payload` is not reused across different chains.

However, there is no validation to ensure that the `toContract` field matches the address of the current `Bridge` contract (`toContract == address(this)`). This oversight allows an already used `payload` to be replayed on a different `Bridge` contract within the same chain.

**Impact** Malicious users can exploit this vulnerability to reuse valid `payloads` on other `Bridge` contracts deployed on the same chain.

**Recommendation** To address this issue, add a validation in the `Actions.depositBridge` function or directly in the `withdraw` function to ensure that `toContract == address(this)`. This will bind the `payload` to the specific `Bridge` contract and prevent its reuse across different contracts on the same chain.

**Developer Response** The developers implemented the suggested fix.

### 4.1.8 V-CSC-VUL-008: Non configured chains can be used to skip payload validations

| Severity | Medium | Commit | ebfda9f |
|---|---|---|---|
| Type | Data Validation | Status | Fixed |
| File(s) | bridge/Bridge.sol | | |
| Location(s) | withdraw() | | |
| Confirmed Fix At | fdf709b | | |

The `withdraw()` function in the `Bridge` contract is responsible for processing withdrawal requests based on a `payload`. To execute a minting of `LBTC`, the function must validate the given `payload`. This validation relies on the `DestinationConfig` of the source chain `fromChain`, which may require validation from either the Chainlink adapter or the consortium. The relevant code snippet from the `withdraw()` function is as follows:

```
DestinationConfig memory destConf = $.destinations[
            bytes32(action.fromChain)
];

bytes32 payloadHash = sha256(payload);
Deposit storage depositData = $.deposits[payloadHash];


if (
   address(destConf.adapter) != address(0) &&
   !depositData.adapterReceived
) {
   revert AdapterNotConfirmed();
}

if (destConf.requireConsortium && !depositData.notarized) {
   revert ConsortiumNotConfirmed();
}
```

**Snippet 4.9:** Code snippet from the `withdraw` function of the `Bridge.sol` contract.

The issue arises because the validity checks for the `payload` are only performed if the `fromChain` configuration includes either an adapter or requires a consortium. This oversight does not account that the default values from a not configured `fromChain` will prevent the `payload` validation. Consequently, if a `fromChain` lacks a `DestinationConfig` configuration, the function fails to validate the `payload`, allowing malicious users to mint free LBTC.

Currently, this vulnerability is mitigated by a rate limit check on the `fromChain` and the `amount` to be minted, which prevents arbitrary `fromChain` usage. However, if a valid `fromChain` with a configured rate limit is removed using the `removeDestination()` function, this exploit becomes feasible, allowing attackers to mint free LBTC up to the current rate limit.

**Impact** The impact of this issue is significant, as it could allow unauthorized minting of LBTC. The exploit is contingent upon the removal of a valid `fromChain` with a configured rate limit, which would bypass the existing rate limit checks.

**Recommendation** To address this issue, it is recommended to implement additional checks in the `withdraw()` function to ensure that a `fromChain` is always configured before processing a withdrawal. Specifically, the function should revert if a `fromChain` is not configured, regardless of the presence of an adapter or consortium requirement. Additionally, when removing a destination using the `removeDestination()` function, the rate limit for that destination should be set to zero to prevent any potential exploits.

**Developer Response** The developers implemented the suggested fix.

### 4.1.9 V-CSC-VUL-009: Validate payload size on abi.decode

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `libs/Actions.sol`, `bridge/adapters/CLAdapter.sol` | | |
| **Location(s)** | depositBtc, depositBridge, validateValSet, feeApproval | | |
| **Confirmed Fix At** | 72b596f | | |

In several locations, the `Bridge` contract receives bytes from potentially untrusted sources. When it receives this data, the protocol uses `abi.decode` to decode the data into the expected format and then performs validation over the returned value as shown below. The `abi.decode` function does not revert if the data has the appropriate format but is too long (e.g. if a struct has additional fields that are not decoded).

```
function depositBridge(
    bytes memory payload
) internal view returns (DepositBridgeAction memory) {
    (
        uint256 fromChain,
        address fromContract,
        uint256 toChain,
        address toContract,
        address recipient,
        uint64 amount,
        bytes32 uniqueActionData
    ) = abi.decode(
            payload,
            (uint256, address, uint256, address, address, uint64, bytes32)
        );

    if (toChain != block.chainid) {
        revert WrongChainId();
    }
    if (recipient == address(0)) {
        revert ZeroAddress();
    }
    if (amount == 0) {
        revert ZeroAmount();
    }

    return
        DepositBridgeAction(
            fromChain,
            fromContract,
            toChain,
            toContract,
            recipient,
            amount,
            uniqueActionData
        );
}
```

**Snippet 4.10:** Snippet from `example()`

**Impact** Without validating the length, it is possible for the same effective payload to have different hashes if a malicious user can append unique bytes. If a user was able to do so, they might be able to process the same order multiple times as the payload bytes themselves are used to compute a hash that is mapped to the state of an order.

**Recommendation** Validate the length of the input bytes whenever possible when decoding bytes.

**Developer Response** The developers have implemented the recommended fix

### 4.1.10 V-CSC-VUL-010: OFTAdapter halt function will not pause operations

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | ebfda9f |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `bridge/oft/LBTCOFTAdapter.sol` | | |
| **Location(s)** | halt | | |
| **Confirmed Fix At** | 9346fe7 | | |

The `LBTCOFTAdapter` contract defines a `halt` function which is intended to stop operation of the bridge endpoint. As the endpoint will lock funds as they are sent and release funds as they are received, the `halt` function attempts to prevent the bridge from operating by burning the endpoint's locked token balance. While this will prevent users from receiving funds via the endpoint in the short-term, it will not prevent operation in the long-term as funds can still be sent. Therefore, if users continue to use the halted endpoint, more funds will be locked and users with locked funds will be able to withdraw them.

```
function halt() external onlyOwner {
    ILBTC(address(innerToken)).burn(innerToken.balanceOf(address(this)));
}
```

**Snippet 4.11:** Definition of the `halt` function

**Impact** As the intention of this function is to completely stop operations of the adapter, this function will not do so in the long-term.

**Recommendation** Consider a different method of halting the adapter. The developers have already indicated that they can enforce the desired long-term behavior by setting all adapter peers to 0.

**Developer Response** The developers renamed this function `empty` so that it would not imply that the bridge should be halted

### 4.1.11 V-CSC-VUL-011: Centralization risk

| Severity | Low | Commit | ebfda9f |
|---|---|---|---|
| Type | Access Control | Status | Acknowledged |
| File(s) | See the Description | | |
| Location(s) | See the Description | | |
| Confirmed Fix At | N/A | | |

Similar to many projects, Lombard's contracts declare an administrator role that is given special permissions. In particular, these administrators are given the following abilities:

- `Bridge`
  - The owner has the ability to add destinations, remove destinations, change bridge commissions, change the adapter address, change the consortium address and change rate limits.
- `CLAdapter`
  - The owner has the ability to set the execution gas limit and the remote chain selector.
- `TokenPool`
  - The owner has the ability to set the router, remote pools, rate limits, allow lists and chain configurations.
- `LBTOFTAdapter` and `LBTCBurnMintOFTAdapter`
  - The owner has the ability to halt the endpoint, configure rate limits and other LayerZero state such as peers.
- `LBTC`
  - The owner has the ability to disable withdrawals and modify state including the token name, token symbol, consortium, maximum mint fee, treasury address, burn commission, dust fee rate, bascule address, pauser address . They also have the ability to add or remove minters and claimers.
  - The pauser has the ability to pause and unpause the contract.
  - Minters have the ability to mint arbitrary tokens and burn tokens from any user.
  - Claimers have the ability to mint tokens on another's behalf with a consortium proof while claiming a fee from the minted tokens
- `Consortium`
  - The owner has the ability to set the initial validator set.
- `BasculeV2`
  - The pauser has the ability to pause and unpause the contract.
  - The default admin has the ability to set the max deposits threshold.
  - The deposit reporter role has the ability to report deposits so they may be immediately approved upon withdrawal.
  - The withdrawal validator has the ability to submit deposits for withdrawal validation (as a deposit may only be withdrawn once).

**Impact** If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious minter could mint a large number of tokens to addresses they control and crash the price of LBTC. They could also burn tokens from other users.

**Recommendation** As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

**Developer Response** Lombard intends to perform all administrative functionality though its established multisig wallet.

### 4.1.12 V-CSC-VUL-012: Missing validation for fromContract in the authNotary function

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `bridge/Bridge.sol` | | |
| **Location(s)** | authNotary() | | |
| **Confirmed Fix At** | 147cfd1 | | |

The `Bridge` contract is responsible for handling cross-chain deposits and withdrawals, utilizing both the Chainlink adapter and the Lombard consortium for notarization. The `authNotary()` function is designed to validate and authorize a payload using a proof from the consortium. However, it lacks a validation step that is present in the `receivePayload()` function. Specifically, the `authNotary()` function does not verify that the `fromContract` field in the payload matches the expected `bridgeContract` for the given source configuration.

**Impact** The absence of this validation step could result in unauthorized withdrawals if a payload is crafted with a valid proof but originates from an unexpected or malicious contract. This vulnerability concerns withdrawals that rely solely on the consortium notarization, as the `receivePayload` function used by the Chainlink adapter does perform the discussed validation.

**Recommendation** To mitigate this issue, the `authNotary()` function should be updated to include a validation step that checks whether the `fromContract` field in the payload matches the `bridgeContract` specified in the source configuration. This will ensure that only payloads originating from the expected contract are processed.

**Developer Response** The developers implemented the suggested fix.

### 4.1.13 V-CSC-VUL-013: Missing argument validation in Bridge

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `bridge/Bridge.sol` | | |
| **Location(s)** | Multiple | | |
| **Confirmed Fix At** | ca8e184 | | |

The `Bridge` contract allows admins to manipulate the contract's configuration. The Veridise security analysts noted that some validation of the input values was missing and could prevent potential mistakes. More specifically:

1. `__Bridge_init`
   a) This function takes as input the LBTC address but does not validate that the input address is non-zero and that it is a contract (i.e. has code). Note that after initialization this address cannot be changed without upgrading the contract.
   b) This function takes as input the treasury address but does not validate that the input address is non-zero. Additionally, if the treasury is intended to be maintained by a smart contract it should be checked that the input address has code. Note that after initialization this address cannot be changed without upgrading the contract.
2. `_changeTreasury`
   a) This function takes as input the treasury address but does not validate that the input address is non-zero. Additionally, if the treasury is intended to be maintained by a smart contract it should be checked that the input address has code. Note that after initialization this address cannot be changed without upgrading the contract.
3. `changeConsortium`
   a) This function takes as input the consortium address but does not validate that the input address is non-zero and that it is a contract (i.e. has code).
4. `setRateLimits`
   a) This function takes as input rate limits to restrict the number of funds that can be sent to and received from another chain. It does not, however, validate that a specific chain is a valid destination. Note that adding rate limits to chains that are not also destinations can result in the following issue: V-CSC-VUL-008
   b) There is no validation that the given limit and window describe a valid limit. For example, if the window is set to 0 or the limit is set to UINT_MAX, then the rate limits will be ineffective which may be undesired.
5. `addDestination`
   a) When adding a destination, the owner can specify an adapter that is used to manage funds sent to and received from the underlying bridge (currently Chainlink CCIP). While it is validated that an adapter address must be non-zero if the consortium is not required, there is no check that the given adapter is a contract (i.e. has code).

**Impact** Configuration errors could prevent the Bridge contract from operating as expected.

**Recommendation** Perform the validation listed above.

**Developer Response** The developers implemented most of the fixes suggested above but opted not to add the EOA checks and add some weaker limits on the rate limits.

#### 4.1.14 V-CSC-VUL-014: Chainlink configuration may break Lombard Bridge

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | ebfda9f |
| **Type** | Denial of Service | **Status** | Acknowledged |
| **File(s)** | `bridge/adapters/TokenPool.sol` | | |
| **Location(s)** | lockOrBurn(), releaseOrMint() | | |
| **Confirmed Fix At** | N/A | | |

The `TokenPool` contract serves as the CCIP token pool for the `LBTC` token within the Chainlink Cross-Chain Interoperability Protocol (CCIP). It implements the functions `lockOrBurn()` and `releaseOrMint()`, which are necessary to the CCIP OnRamp and OffRamp contracts for processing token transfers on the source and destination chains, respectively.

In the `lockOrBurn()` function, the `destPoolData` is returned to the CCIP OnRamp contract. When `isAttestationEnabled` is set to `false`, the entire `payload`, which exceeds 32 bytes, is returned. The CCIP system, however, enforces a strict 32-byte limit on returned data. If the `payload` exceeds this limit, the system will revert the transaction.

The `releaseOrMint()` function, invoked by the CCIP OffRamp contract, is subject to a default gas limit imposed by the CCIP configuration. This function is gas-intensive as it involves interactions with the adapter contract. If the execution of the `releaseOrMint()` function exceeds the allocated gas limit, it will fail, leading to a disruption in the token minting process on the destination chain.

**Impact** The potential issues in the `lockOrBurn()` and `releaseOrMint()` functions stem from their current implementation, which may not be compatible with the CCIP default configuration. As a result, the Lombard team may need to depend on the CCIP administrators to override these default settings specifically for the LBTC token pool. The Lombard token pool may be also be impacted by chainlink configuration changes.

**Recommendation** It is recommended to engage with the Chainlink team to discuss any special requirements for your token pool that might necessitate changes to the default CCIP configurations. While Chainlink can adjust these settings to accommodate specific needs, such changes are controlled by Chainlink only and cannot be made in a permissionless manner.

**Developer Response** The developers indicated that they are working with Chainlink to avoid such errors.

### 4.1.15 V-CSC-VUL-015: Unsafe downcast in CLAdapter

| Severity | Low | Commit | ebfda9f |
|---|---|---|---|
| Type | Data Validation | Status | Fixed |
| File(s) | bridge/adapters/CLAdapter.sol | | |
| Location(s) | initiateDeposit | | |
| Confirmed Fix At | c4b3db7 | | |

The `initiateDeposit` function in the `CLAdapter` contract is used by the LBTC Token pool for Chainlink's CCIP bridge to burn funds that are bridged to other chains. The adapter enforces that these funds must be sent through the Lombard bridge so that it can properly notarize bridge requests in cases where it is required. In doing so, however, it casts the input `receiver` from `bytes` to `bytes32` and the input `amount` from `uint256` to `uint64` without checking the length or value of the input. Doing so can result in truncation for `receiver` and can cause the `amount` value to overflow.

```
function initiateDeposit(
    uint64 remoteChainSelector,
    bytes calldata receiver,
    uint256 amount
) external returns (uint256 lastBurnedAmount, bytes memory lastPayload) {
    _onlyTokenPool();

    if (_lastPayload.length > 0) {
        ...
    } else {
        IERC20(address(lbtc())).approve(address(bridge), amount);
        (lastBurnedAmount, lastPayload) = bridge.deposit(
            getChain[remoteChainSelector],
            bytes32(receiver),
            uint64(amount)
        );
    }

    bridge.lbtc().burn(lastBurnedAmount);
}
```

**Snippet 4.12:** Definition of the `initiateDeposit` function

**Impact** While unlikely, it is possible that if receiver is not a `bytes32`, funds could be sent to the wrong recipient. Note that it is also unlikely that the amount will exceed `uint64` as long as LBTC remains pegged to bitcoin and has 8 decimals as the maximum supply would be 5376000000 satoshi. However, should LBTC be launched on a chain with a larger number of decimals, it is possible that the downcast could cause an incorrect number of funds to be bridged to another chain while the remaining funds are locked in the bridge.

#### Recommendation

1. Check the length of the `receiver` bytes to make sure they have a size <= 32
2. Check the value of `amount` to ensure it fits into a `uint64`

**Developer Response** The developers implemented the suggested fix.

### 4.1.16 V-CSC-VUL-016: Treasury address not set in LBTC init

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `LBTC/LBTC.sol` | | |
| **Location(s)** | __LBTC_init() | | |
| **Confirmed Fix At** | 0480d68 | | |

LBTC's initialization function `__LBTC_init()` does not set the `treasury` for the LBTC token. Instead, the contract owners are required to call `changeTreasuryAddress()` to set the treasury after deployment. This is a problem because the LBTC token requires a non-zero value for `treasury`.

**Impact** If the LBTC contract owners forget to call `changeTreasuryAddress()`, LBTC transactions will mint and transfer tokens to the zero address. This may result in an unintended loss of funds.

**Recommendation** Update `__LBTC_init()` so that it initializes the `treasury`.

**Developer Response** The developers have applied the recommendation.

### 4.1.17 V-CSC-VUL-017: Consortium _checkProof() no-ops on invalid signature lengths

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Data Validation | **Status** | Intended Behavior |
| **File(s)** | `consortium/Consortium.sol` | | |
| **Location(s)** | _checkProof() | | |
| **Confirmed Fix At** | N/A | | |

The `Consortium._checkProof()` function is used to check that a sufficient number of validators in a validator set have cryptographically signed a given message. Specifically, a "proof" consists of a list of validators, their weights, and their attached signatures. Valid signatures included in the proof should be 64 bytes long; but it is possible for a validator to not sign the payload, which is indicated by a 0-length signature. For each 64-byte length signature, `_checkProof` processes the signature, ultimately ensuring that the cumulative weight of all valid signatures is above the necessary threshold. However, `_checkProof` ignores all signatures that are not 64-bytes long. While this is correct for 0-length signatures (i.e., those indicating "no signature"), `_checkProof` does not do anything for other signature lengths. It would be better to be explicitly handle signatures with unexpected lengths.

```
bytes[] memory signatures = abi.decode(_proof, (bytes[]));

// ...

for (uint256 i; i < length; ) {
    // each signature preset R || S values
    // V is missed, because validators use Cosmos SDK keyring which is not signing in
     eth style
    if (signatures[i].length == 64) {
        // Process signature
        // ...
    }

    // No else case...
}
```

**Snippet 4.13:** Snippet from `_checkProof`

**Impact** A user, by mistake, could provide a proof that contains some signatures with invalid lengths, but such a proof would pass as long as there are enough valid signatures to overcome the weight threshold. It may not be desirable to accept such a proof.

**Recommendation** Clarify the intended behavior when a signature of invalid length is provided and made the code consistent with the intended behavior.

**Developer Response** The developers indicated that that signatures with length other than 0 or 64 are supposed to be ignored, and they have added a comment to the code explaining as such.

### 4.1.18 V-CSC-VUL-018: Missing validation on selector in Consortium setInitialValidatorSet()

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `consortium/Consortium.sol` | | |
| **Location(s)** | setInitialValidatorSet() | | |
| **Confirmed Fix At** | e806ad4 | | |

The function `Consortium.setInitalValidatorSet()` can be called by the owner to set the validator set of the first epoch. It takes a single argument, `_initialValSet`, corresponding to a ledger "payload" that contains an ABI-encoded `ValSetAction`.

```
struct ValSetAction {
    uint256 epoch;
    address[] validators;
    uint256[] weights;
    uint256 weightThreshold;
    uint256 height;
}
```

**Snippet 4.14:** Definition of `ValSetAction`

`setInitialValidatorSet()` assumes that the payload is a `ValSetAction` without checking that the selector indicates that it is one. Note that another function, `setNextValidatorSet()`, *does* perform such a check.

```
function setInitalValidatorSet(
    bytes calldata _initialValSet
) external onlyOwner {
    ConsortiumStorage storage $ = _getConsortiumStorage();

    // Veridise: No check for selector in _initialValSet...

    Actions.ValSetAction memory action = Actions.validateValSet(
        _initialValSet[4:]
    );

    // ...
}
```

**Snippet 4.15:** Snippet from `setInitalValidatorSet()`

**Impact** An admin may mistakenly call `setInitialValidatorSet()` with another action, which may result in a confusing error message when `Actions.validateValSet` is called. Worse, if the given action is similar in structure to a `ValSetAction`, the code that follows the call to `validateValSet` may still be executed even though it should not.

**Recommendation** Add code to validate that the selector of `_initialValSet` corresponds to a `ValSetAction`.

**Developer Response** The developers have applied the recommendation.

### 4.1.19 V-CSC-VUL-019: Missing validation on the first byte of pubkeys in pubKeysToAddress

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `libs/Actions.sol` | | |
| **Location(s)** | pubKeysToAddress() | | |
| **Confirmed Fix At** | 9971034 | | |

The function `pubKeysToAddress()` converts the input ECDSA public keys to their corresponding EVM addresses. Each public key is required to be in the 65-byte long uncompressed form, which consists of one byte with the value `0x04` followed by two 32-byte values. However, `pubKeysToAddress()` does not check that the first byte is equal to `0x04`.

```
1  if (_pubKeys[i].length == 65) {
2      bytes memory data = _pubKeys[i];
3
4      // Veridise: Missing check that data[0] = 0x04...
5
6      // create a new array with length - 1 (excluding the first 0x04 byte)
7      bytes memory result = new bytes(data.length - 1);
8
9      // use inline assembly for memory manipulation
10     // ...
11
12     addresses[i] = address(uint160(uint256(keccak256(result))));
13 } else {
14     revert InvalidPublicKey(_pubKeys[i]);
15 }
```

**Snippet 4.16:** Relevant lines from `pubKeysToAddress()`

**Impact** Although it is technically safe for `pubKeysToAddress` to discard this first byte when performing the conversion, it may be useful to check that the first byte of each public key is `0x04` anyway. This would help guard against some situations where the caller provides data that is coincidentally 65-bytes in length but does not constitute an actual valid ECDSA public key.

**Recommendation** Insert code to check that the first byte of each public key is `0x04`.

**Developer Response** The developers have applied the recommendation.

### 4.1.20 V-CSC-VUL-020: Include version in Bridge Payload

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | ebfda9f |
| **Type** | Maintainability | **Status** | Acknowledged |
| **File(s)** | `bridge/Bridge.sol` | | |
| **Location(s)** | _deposit() | | |
| **Confirmed Fix At** | N/A | | |

The `Bridge` contract is designed to be upgradeable, which means its logic can be modified over time. This flexibility, however, introduces potential risks if the contract's version is not included in the payloads it creates.

The `deposit()` function constructs a payload using the `abi.encodeWithSelector()` method, which includes various parameters such as `toChain`, `toAddress`, and `amountWithoutFee`. However, it does not include a version identifier for the contract. Including a version number in the payload would help ensure that the payload is processed correctly by the appropriate version of the contract, reducing the risk of errors or vulnerabilities introduced by future upgrades.

```
bytes memory payload = abi.encodeWithSelector(
    Actions.DEPOSIT_BRIDGE_ACTION,
    bytes32(block.chainid),
    bytes32(uint256(uint160(address(this)))),
    toChain,
    config.bridgeContract,
    toAddress,
    amountWithoutFee,
    $.crossChainOperationsNonce++
);
```

**Snippet 4.17:** Code snippet from the `_deposit()` function of the `Bridge.sol` contract.

**Impact** If the contract logic changes significantly in an upgrade, there is a risk of payload clashes. This risk is particularly heightened if changes are introduced to the state of the `crossChainOperationsNonce` value.

**Recommendation** To mitigate this issue, it is recommended to include a version identifier in the payload constructed by the `_deposit()` function. This can be achieved by adding an additional parameter that specifies the current version of the contract.

**Developer Response** The developers opted not to implement the suggested fix as the payload format is encoded in the selector.

### 4.1.21 V-CSC-VUL-021: Remove duplicated and unused function

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | ebfda9f |
| **Type** | Maintainability | **Status** | Fixed |
| **File(s)** | `bridge/Bridge.sol` | | |
| **Location(s)** | _calcRelativeFee | | |
| **Confirmed Fix At** | 9df54c4 | | |

**Description** The `_calcRelativeFee` internal function in the `Bridge` contract is unused and duplicates the behavior of `calcRelativeFee` in the `FeeUtils` library.

**Impact** This function may become out of sync with the rest of the project, leading to errors if used in the future.

**Developer Response** The developers implemented the suggested fix.

### 4.1.22 V-CSC-VUL-022: Missing Validation in CCIP Adapter

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `bridge/adapters/AbstractAdapter.sol` | | |
| **Location(s)** | constructor, changeBridge | | |
| **Confirmed Fix At** | ec315f2 | | |

The CCIP adapter contract allows admins to manipulate the contract's configuration. The Veridise security analysts noted that some validation of the input values was missing and could prevent potential mistakes. More specifically:

1. `AbstractAdapter::constructor`
    a) This function takes as input the address to the `Bridge` contract but does not validate that the given address is not the zero address and is a smart contract (i.e. has code).
2. `AbstractAdapter::changeBridge`
    a) This function takes as input the address to the `Bridge` contract but does not validate that the given address is a smart contract (i.e. has code).

**Impact** Configuration errors could prevent the Bridge contract from operating as expected.

**Recommendation** Perform the validation listed above.

**Developer Response** The developers implemented most of the fixes suggested above but opted not to add the EOA checks.

### 4.1.23 V-CSC-VUL-023: OFTAdapter rate limit may be slightly inaccurate

| Severity | Warning | Commit | ebfda9f |
|---|---|---|---|
| Type | Logic Error | Status | Acknowledged |
| File(s) | `bridge/oft/LBTCBurnMintOFTAdapter.sol`, `bridge/oft/EfficientRateLimitedOFTAdapter.sol` | | |
| Location(s) | _debit, _credit | | |
| Confirmed Fix At | N/A | | |

To allow LBTC to be bridged to other blockchains via LayerZero, the Lombard developers implement a custom OFTAdapter. Most of the behavior is inherited from LayerZero's base OFTAdapter implementation, but the Lombard developers also add in some rate limiting functionality that restricts the amount of that are sent or received by a particular endpoint over some period of time. The rate limiting, however, is slightly inaccurate as the limit is placed on the value passed into the `_debit` function (shown below) which is greater than or equal to the amount sent. This is because the `_debitView` function removes some dust that cannot be received on the other side of the bridge and returns the amount of funds that can actually be sent. It may therefore be the case the rate limit used by the `_debit` is slightly inaccurate as it may enforce stricter restrictions depending on how frequently dust is removed from the sent amount.

```
function _debit(
    address _from,
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 _dstEid
)
    internal
    virtual
    override(OFTAdapter, EfficientRateLimitedOFTAdapter)
    returns (uint256 amountSentLD, uint256 amountReceivedLD)
{
    _checkAndUpdateRateLimit(
        _dstEid,
        _amountLD,
        RateLimitDirection.Outbound
    );
    (amountSentLD, amountReceivedLD) = _debitView(
        _amountLD,
        _minAmountLD,
        _dstEid
    );
    ILBTC(address(innerToken)).burn(_from, amountSentLD);
}
```

**Snippet 4.18:** Definition of the `_debit` function

**Impact** Depending on how frequently dust is removed from sent amount, it is possible that users may not spend funds up to the given limit.

**Recommendation** Consider implementing the following changes:

1. Place rate limits on the output of `_debitView` in `LBTCBurnMintOFTAdapter` and the output of `super._debit` in `EfficientRateLimitedOFTAdapter`.
2. While `_credit` does not reduce the amount of funds received in practice, consider rate limiting the output of `super._credit` in `EfficientRateLimitedOFTAdapter` for the purposes of consistency.

**Developer Response** The developers opted not to fix this issue

### 4.1.24 V-CSC-VUL-024: Missing validation in LBTC contract

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `LBTC/LBTC.sol` | | |
| **Location(s)** | Multiple | | |
| **Confirmed Fix At** | 7801293 | | |

The CCIP adapter contract allows admins to manipulate the contract's configuration. The Veridise security analysts noted that some validation of the input values was missing and could prevent potential mistakes. More specifically:

1. `LBTC::changeConsortium`
    a) This function takes as input the address to the `Consortium` contract but does not validate that the given address is not the zero address and is a smart contract (i.e. has code).
2. `LBTC::setMintFee`
    a) This function takes as input the maximum fee but the fee is likely intended to be non-zero.
3. `LBTC::changeBurnCommission`
    a) This function takes as input the commission charged when LBTC is redeemed but it likely is not desired for this value to be excessively large.
4. `LBTC::changeDustFeeRate`
    a) This function takes as input the dust fee rate used to calculate Bitcoin's dust threshold so that users are not sent dust. It is likely not desired that this value be set such that dust is excessively large.
5. `LBTC::_validateAndMint`
    a) To prevent potential mistakes, we would recommend validating that `amountToMint <= depositAmount`.

**Impact** Configuration errors could prevent the LBTC contract from operating as expected.

**Recommendation** Perform the validation listed above.

**Developer Response** For `setMintFee`, we would like the ability to set the fee to 0. For the `changeBurnCommission` and `changeDustFeeRate` functions, the validations could prove useful but we do not want to restrict the flexibility in case of big price fluctuations in BTC. The remaining validations have been implemented.

### 4.1.25 V-CSC-VUL-025: Fee signatures can be replayed

| Severity | Warning | Commit | 109a3f2 |
|---|---|---|---|
| Type | Replay Attack | Status | Intended Behavior |
| File(s) | LBTC/LBTC.sol | | |
| Location(s) | _mintWithFee | | |
| Confirmed Fix At | N/A | | |

The LBTC contract allows claimers to mint funds on behalf of other users in return for a fee that is collected by the treasury. As the fee is collected from the user's minted funds, they must provide approval of the fee via a EIP1271 signature which is checked in the snippet below. Notably, the digest that is hashed and signed to indicate approval contains the chain ID, fee and expiration. Since the protocol does not require that signatures be unique or associated with a specific mint payload, any signature provided may be re-used until the expiration is reached.

```
function _mintWithFee(
    bytes calldata mintPayload,
    bytes calldata proof,
    bytes calldata feePayload,
    bytes calldata userSignature
) internal nonReentrant {
    ...

    {
        // Fee validation
        bytes32 digest = _hashTypedDataV4(
            keccak256(
                abi.encode(
                    Actions.FEE_APPROVAL_EIP712_ACTION,
                    block.chainid,
                    feeAction.fee,
                    feeAction.expiry
                )
            )
        );

        if (
            !EIP1271SignatureUtils.checkSignature(
                mintAction.recipient,
                digest,
                userSignature
            )
        ) {
            revert InvalidUserSignature();
        }
    }

    ...
}
```

**Snippet 4.19:** Fee validation code in `_mintWithFee`

**Impact** Since user signatures can be replayed until expiration, claimers can use previous signatures to extract higher fees from the user. More specifically if a user intends to approve some fee F1 for deposit D1 and another fee F2 for deposit D2, the claimer can use the signature for the higher fee when minting D1 and D2 for the user. Also note that if a user approves fee F1 for deposit D1 and intends to submit a proof for another deposit D2 themselves, a claimer can process D1 and D2 both with fee F1.

**Recommendation** Consider adding a field to the fee digest for the mint payload hash so that fees must be approved on every deposit.

**Developer Response** Users will be made aware that they need to pay attention to the fee expiration.

### 4.1.26 V-CSC-VUL-026: mintWithFee functions vulnerable to frontrunning

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Frontrunning | **Status** | Acknowledged |
| **File(s)** | `LBTC/LBTC.sol` | | |
| **Location(s)** | batchMintWithFee, mintWithFee | | |
| **Confirmed Fix At** | N/A | | |

Upon a Bitcoin deposit, LBTC allows users to submit a deposit proof or allow a claimer to submit the proof in return for a fee. The only difference between the two functions is that the one invoked by the claimer verifies the fee approval and mints the fees to the treasury. As a result, anyone may call the `mint` function with the same proof provided by the claimer to `mintWithFee`, allowing the `mintWithFee` and `batchMintWithFee` functions to be frontrun.

**Impact** In both cases, frontrunning the `mintWithFee` functions will cause the claimer to waste gas and will prevent fees from being collected. In the case of the `batchMintWithFee` shown below, however, if a single deposit payload is submitted before the function invocation, then the entire `batchMintWithFee` transaction will revert since attempting to deposit twice will revert the transaction. If a malicious minter were to frontrun these transactions, they could therefore cost the claimer more gas than they would spend.

```
function batchMintWithFee(
    bytes[] calldata mintPayload,
    bytes[] calldata proof,
    bytes[] calldata feePayload,
    bytes[] calldata userSignature
) external {
    _onlyClaimer(_msgSender());

    uint256 length = mintPayload.length;
    if (
        length != proof.length ||
        length != feePayload.length ||
        length != userSignature.length
    ) {
        revert InvalidInputLength();
    }

    for (uint256 i; i < mintPayload.length; ++i) {
        _mintWithFee(
            mintPayload[i],
            proof[i],
            feePayload[i],
            userSignature[i]
        );
    }
}
```

**Snippet 4.20:** Definition of the `batchMintWithFee` function

**Recommendation** In `batchMintWithFee`, consider skipping a mint if the deposit has already been processed. We would also recommend monitoring claimer transactions to determine if they are frequently targeted by frontrunning so that fees cannot be collected.

**Developer Response** The developers will monitor the contract to determine if these functions are frequently frontrun.

### 4.1.27 V-CSC-VUL-027: Invalid consortium signatures cause revert

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 109a3f2 |
| **Type** | Usability Issue | **Status** | Fixed |
| **File(s)** | consortium/Consortium.sol | | |
| **Location(s)** | _checkProof | | |
| **Confirmed Fix At** | fc5a96e | | |

To provide some additional security, Lombard uses an off-chain consortium to validate highly sensitive actions such as minting LBTC that has been bridged from the Bitcoin blockchain or other EVM chains. If the consortium approves an action, they will submit a proof as a list of signatures where each validator in the current validator set will either provide a signature or essentially a zero value to indicate that they did not approve the action. This consortium proof is then validated by validating the signatures and ensuring that a sufficient number of validators have approved to exceed some threshold so all validators do not need to sign off on an action. As shown below, however, if a signature is of the correct size but cannot be verified, the function will revert even if a sufficient number of other validators have also provided signatures.

```
function _checkProof(
    bytes32 _payloadHash,
    bytes calldata _proof
) internal view virtual {
    ...

    for (uint256 i; i < length; ) {
        // each signature preset R || S values
        // V is missed, because validators use Cosmos SDK keyring which is not
    signing in eth style
        if (signatures[i].length == 64) {
            ...

            if (r != bytes32(0) && s != bytes32(0)) {
                // try recover with V = 27
                (address signer, ECDSA.RecoverError err, ) = ECDSA
                    .tryRecover(_payloadHash, 27, r, s);

                // revert if bad signature
                if (err != ECDSA.RecoverError.NoError) {
                    revert SignatureVerificationFailed(i, err);
                }

                // if signer doesn't match try V = 28
                if (signer != validators[i]) {
                    (signer, err, ) = ECDSA.tryRecover(
                        _payloadHash,
                        28,
                        r,
                        s
                    );
                    if (err != ECDSA.RecoverError.NoError) {
                        revert SignatureVerificationFailed(i, err);
                    }
```

```
34
35                      if (signer != validators[i]) {
36                          revert WrongSignatureReceived(signatures[i]);
37                      }
38                  }
39                  // signature accepted
40
41                  unchecked {
42                      weight += weights[i];
43                  }
44              }
45          }
46                  // ...
47      }
48      // ...
49  }
```

**Snippet 4.21:** Snippet from `_checkProof`

**Impact** If a validator's signature is corrupted or invalid for some other reason, the transaction will revert even if a sufficient number of other validators have signed the action. Note that the proof can be corrected by zero-ing out the corrupted signature but will require the user know information about the proof's structure. If this is a common occurrence it could impact the usability of the system.

**Recommendation** Consider skipping validators rather than reverting when a signature cannot be verified.

**Developer Response** The developers implemented the suggested fix.

### 4.1.28 V-CSC-VUL-028: Missing argument validation in EffectiveRateLimiter

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | ebfda9f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `bridge/oft/EfficientRateLimiter.sol` | | |
| **Location(s)** | _setRateLimits | | |
| **Confirmed Fix At** | 8891939 | | |

The `EfficientRateLimiter` contract allows admins to manipulate the contract's configuration. The Veridise security analysts noted that some validation of the input values was missing and could prevent potential mistakes. More specifically:

1. `_setRateLimits`
    a) There is no validation that the given limit and window describe a valid limit. For example, if the window is set to 0 or the limit is set to UINT_MAX, then the rate limits will be ineffective which may be undesired. Similarly, if `window` is set to 0 then rate limiting will effectively be turned off.

**Impact** Configuration errors could prevent rate limits from being properly imposed.

**Recommendation** Consider adding the above validation.

**Developer Response** The developers have added some validation of the input rate limits.

### 4.1.29 V-CSC-VUL-029: Minor issues in Smart Contracts

| | | | |
|---|---|---|---|
| **Severity** | Info | **Commit** | 109a3f2 |
| **Type** | Maintainability | **Status** | Fixed |
| **File(s)** | See description | | |
| **Location(s)** | initialize() | | |
| **Confirmed Fix At** | 5b493e6 | | |

1. The `LBTC.initialize()` function sets the default `dustFeeRate` to a magic constant.
2. The `LBTC.redeem()` function duplicates the calculation logic in `calcUnstakeRequestAmount()`.
3. The `DepositBridgeAction` has a `uniqueActionData` field, which corresponds to the `Nonce` field of the `DepositBridgeMsg` in ledger. The field should be renamed for consistency.
4. The function named `setInitalValidatorSet` has a typo - should be named `setInitialValidatorSet`.

**Impact** Typos and unused code hurt code maintainability.

**Recommendation** Fix the issues described above.

**Developer Response** The developers have applied the recommendation.

### 4.1.30 V-CSC-VUL-030: Different rate limiting behavior

| | | | |
|---|---|---|---|
| **Severity** | Info | **Commit** | 109a3f2 |
| **Type** | Logic Error | **Status** | Intended Behavior |
| **File(s)** | `libs/RateLimits.sol`, `bridge/oft/EfficientRateLimiter.sol` | | |
| **Location(s)** | updateLimit, _checkAndUpdateRateLimit | | |
| **Confirmed Fix At** | N/A | | |

The Lombard contracts integrate with two different bridges: LayerZero and Chainlink's CCIP. In both cases, rate limits are imposed to restrict the amount of funds that may be sent or received from a given endpoint over a period of time. In both cases at a high level the rate limiting works by establishing an upper-limit on the number of funds that may be bridged and accumulating bridge requests towards that limit. The amount of funds that have been bridged decays linearly over a specific window of time. While the rate limits are largely the same for the two bridge integrations, there is one significant difference. Notably, the CCIP bridge tracks the rate limits for incoming and outgoing funds separately but the LayerZero rate limit reduces the number of spent funds in the opposite direction. If incoming and outgoing funds are balanced (and not too large), this can make it appear as if there is no limit.

**Recommendation** If it is intended to specifically restrict the amount of incoming and outgoing funds over a period of time, note that the LayerZero rate limit may not be effective. If the difference is intended this issue can be disregarded.

**Developer Response** The developers have indicated that this is indeed intentional

# Glossary

**ERC-20** The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 1

**Ethereum Virtual Machine** The simulated computer in which Ethereum transactions and smart contract code are executed. This is implemented in software, typically as part of an Ethereum client. Visit `https://ethereum.org/en/developers/docs/evm/` to learn more . 54

**EVM** Ethereum Virtual Machine. 1

**smart contract** A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1