Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

LBTC

Veridise Inc.
May 29, 2024

► **Prepared For:**

Lombard

► **Prepared By:**

Ajinkya Rajput
Jacob Van Greffen

► **Contact Us:** contact@veridise.com

► **Version History:**

May 29, 2024	V1
May 10, 2024	Initial Draft

Contents

1 Executive Summary

From May. 6, 2024 to May. 9, 2024, Lombard engaged Veridise to review the security of their LBTC project which is a BTC liquid staking protocol. The review covered their Ethereum-side on-chain contracts. Veridise conducted the assessment over 6 person-days, with 2 engineers reviewing code over 3 days on code at commit `d4b11f3`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as an extensive manual code review.

Project summary. LBTC is a BTC liquid staking protocol. The protocol consists of an off-chain consortium service and an on chain component. The off-chain consortium service verifies the deposit of BTC, notarizes the deposit and returns a signature to the depositor. The depositor then provides the signature to on-chain component that mints liquid staked BTC (LBTC) tokens.

Code assessment. The LBTC developers provided the source code of the LBTC contract for review. The code appears to have been developed entirely by the Lombard developers. The code is well documented. To facilitate the Veridise auditors' understanding of the code, a write-up was provided that documents the high level working of the protocol. Additionally, the code contained some in-line comments on structs and functions. The delivered source code also contained a test suite which the Veridise auditors noted tested many of the expected user-flows and much of the protocol's behavior.

Summary of issues detected. The audit uncovered 5 issues, the most severe of which is a medium severity issue (V-LOM-VUL-001) which points out that signatures provided to users may become invalid. The auditors also identified a low severity issue (V-LOM-VUL-002) which points out storing of signatures. The Veridise auditors also identified 1 warning, and 2 informational findings.

All of the 5 issues have been fixed by Lombard.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the LBTC and to avoid similar issues to those discovered in the audit in the future.

The protocol interacts with an off-chain consortium component and security of these interaction is provided via ECDSA signatures. We recommend adhering to best practices of implementing ECDSA signatures in the off-chain component. The auditors noted that the test suite tests all the user flows extensively for each contract in isolation. We recommend testing with deeper sequences of user and owner actions intermixed.

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
LBTC	`d4b11f3`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May. 6 - May. 9, 2024	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	1
Low-Severity Issues	1	1	1
Warning-Severity Issues	1	1	1
Informational-Severity Issues	2	2	2
TOTAL	5	5	5

Table 2.4: Category Breakdown.

Name	Number
Maintainability	2
Denial of Service	1
Replay Attack	1
Missing/Incorrect Events	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of LBTC's smart contracts. In our audit, we sought to answer questions such as:

- Is the protocol vulnerable to replay attacks?
- Is the protocol vulnerable to signature malleability attacks?
- Is the usage of ECDSA signatures secure?
- Is the implementation of EIP1271 secure?
- Is the signing key change safe for users?
- Is the signing key change safe for the protocol?
- Can user funds be locked in the protocol?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the solidity contracts provided in archive `smart-contracts-d4b11f36b8ba68c095464ab6990fb83e9c405638.zip` with SHA256 checksum

`1b56c9de6bce8a075a5ccc32d39ba85f31682d1ef32485bf280e68aeae5204bc`

Specifically, the following contracts were in scope,

- `contracts/consortium/LombardConsortium.sol`
- `contracts/consortium/LombardFinanceTimeLock.sol`
- `contracts/libs/EIP1271SignatureUtils.sol`
- `contracts/libs/DepositDataCodec.sol`
- `contracts/LBTC/ILBTC.sol`
- `contracts/LBTC/LBTC.sol`

Methodology. Veridise auditors reviewed the reports of previous audits for LBTC, inspected the provided tests, and read the LBTC documentation. They then began a manual review of the code assisted by property-based testing. During the audit, the Veridise auditors regularly met with the LBTC developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

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

In this case, we judge the likelihood of a vulnerability as follows in 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) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in 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 - OR - 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 - OR - 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

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-LOM-VUL-001	Signatures may become unverifiable	Medium	Fixed
V-LOM-VUL-002	Repeated mints with the same data can be perfor. .	Low	Fixed
V-LOM-VUL-003	Missing event for mint	Warning	Fixed
V-LOM-VUL-004	Constant variables named using camel case	Info	Fixed
V-LOM-VUL-005	Duplicate declaration	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-LOM-VUL-001: Signatures may become unverifiable

Severity	Medium	**Commit**	57caeb4
Type	Denial of Service	**Status**	Fixed
File(s)	`LombardConsortium.sol`		
Location(s)	changeKeyThreshold()		
Confirmed Fix At	1834e11		

`LombardConsortium` is an ERC1271 contract that can verify signatures for a given hash. This contract checks if the a given hash is signed by an address `thresholdKey`, which is stored in a state variable. The `LombardConsortium` contract allows the owner to change the threshold key by calling the privileged function `changeThresholdKey()`.

```
function isValidSignature(
    bytes32 hash,
    bytes memory signature
) external view override returns (bytes4 magicValue) {
    ConsortiumStorage storage $ = _getConsortiumStorage();

    if (ECDSA.recover(hash, signature) != $.thresholdKey) {
            revert BadSignature();
    }

    return EIP1271_MAGICVALUE;
}

function _changeThresholdKey(address newVal) internal {
    ConsortiumStorage storage $ = _getConsortiumStorage();
    emit ThresholdKeyChanged($.thresholdKey, newVal);
    $.thresholdKey = newVal;
}
```

Snippet 4.1: Snippet from `LombardConsortium`

The protocol defines an external `mint()` function that accepts a `DepositData` struct and a signature and mints `LBTC` tokens. The mint function first computes the hash of the data and checks if the the provided signature is valid for the computed hash by calling `checkSignature()` in `EIP1271SignatureUtils`, which in turn calls `isSignatureValid()` in `LombardConsortium`.

Impact In the event that

- An user obtains a signature
- And the owner changes the key before the user has used the signature by calling the `mint()`.

The user will not be able to mint tokens for his signature as the `isSignatureValid()` will revert.

```
function mint(
    bytes calldata data,
    bytes memory proofSignature
) external nonReentrant {
    LBTCStorage storage $ = _getLBTCStorage();

    bytes32 proofHash = keccak256(data);

    // The problem is if we will change signer its open ability to reuse same
    signatures
    // But Consortium save signature forever and it will not be changed if we change
    signer
    bytes32 signatureHash = keccak256(proofSignature);

    if ($._usedProofs[signatureHash]) {
        revert ProofAlreadyUsed();
    }

    // we can trust data only if proof is signed by Consortium
    EIP1271SignatureUtils.checkSignature($.consortium, proofHash, proofSignature);
```

Snippet 4.2: Snippet from `mint()`

Recommendation Consortium could maintain a list of signature that are generated and not verified by the user and define an privileged external function that only the signature generator can call to update the said list.

`LombardConsortium` should also disallow changing the threshold keys until there are outstanding signatures that are not verified for a grace period of time since the last outstanding signature is reported.

Developer Response The consortium service can provide a new signature after the `thresholdKey` changes

4.1.2 V-LOM-VUL-002: Repeated mints with the same data can be performed with different signatures

Severity	Low	**Commit**	57caeb4
Type	Replay Attack	**Status**	Fixed
File(s)	`LBTC.sol`		
Location(s)	mint		
Confirmed Fix At	1834e11		

So long as the signature passed into `mint()` is distinct, multiple mints can be executed over the same data.

```
function mint(
        bytes calldata data,
        bytes memory proofSignature
    ) external nonReentrant {
        LBTCStorage storage $ = _getLBTCStorage();

        bytes32 proofHash = keccak256(data);

        // The problem is if we will change signer its open ability to reuse same signatures
        // But Consortium save signature forever and it will not be changed if we change signer
        bytes32 signatureHash = keccak256(proofSignature);

        if ($._usedProofs[signatureHash]) {
            revert ProofAlreadyUsed();
        }

        ...
}
```

Snippet 4.3: Snippet from `mint()`

If the attacker can generate multiple signatures for the same `data` field, they can replay the `mint()` transaction multiple times.

Impact Attackers that can generate multiple distinct signatures for the same minting data can replay their `mint()` multiple times. If `data` represents a deposit into an account controlled by the attacker, this amounts to stealing funds.

Recommendation Add a nonce to the `DepositData` structure and require that users pass in `data` with a unique nonce. This will mean that the hash (and thus the signature) for `data` with the same `chainId`, `to`, and `amount` fields will be different values. Then, instead of storing `signatureHash` to prevent replay attacks, store the nonce of `data`. This will prevent replay attacks whether or not the attacker uses the same signature or a different signature for `data`.

4.1.3 V-LOM-VUL-003: Missing event for mint

Severity	Warning	**Commit**	57caeb4
Type	Missing/Incorrect Eve	**Status**	Fixed
File(s)	`LBTC.sol`		
Location(s)	mint()		
Confirmed Fix At	1834e11		

The `mint` transaction does not emit any event when executing. As a side effect, the global nonce is never incremented during `mint`.

Impact Forgoing `mint` events hurts the greater accessibility of LBTC. Without `mint` events, external users and applications will have a much harder time tracking the state of LBTC.

Recommendation Add a definition for the `Minted` event to `ILBTC.sol`. Also, at the end of the `mint` function, emit the following event:

```
emit Minted(
    depositData.chainId,
    depositData.to,
    depositData.amount,
    $._globalNonce++
);
```

4.1.4 V-LOM-VUL-004: Constant variables named using camel case

Severity	Info	Commit	57caeb4
Type	Maintainability	Status	Fixed
File(s)	See Description		
Location(s)	N/A		
Confirmed Fix At	1834e11		

The protocol stores state of the contract in a struct at a constant slot as shown below

```
bytes32 private constant ConsortiumStorageLocation =
        0xbac09a3ab0e06910f94a49c10c16eb53146536ec1a9e948951735cde3a58b500;
```

Snippet 4.4: Snippet from `LombardConsortium`

```
bytes32 private constant LBTCStorageLocation = 0
    xa9a2395ec4edf6682d754acb293b04902817fdb5829dd13adb0367ab3a26c700;
```

Snippet 4.5: Snippet from `LBTC`

Recommendation According to convention the variables `ConsortiumStorageLocation` and `LBTCStorageLocation` should be all caps.

4.1.5 V-LOM-VUL-005: Duplicate declaration

Severity	Info	**Commit**	57caeb4
Type	Maintainability	**Status**	Fixed
File(s)	See Description		
Location(s)	N/A		
Confirmed Fix At	1834e11		

The protocol uses EIP1271 standard which returns an constant value `0x1626ba7e`

The protocol stores this value in an internal constants variable `EIP1271_MAGICVALUE` in `LombardConsortium` and `EIP1271SignatureUtils` as shown below.

```
1 bytes4 internal constant EIP1271_MAGICVALUE = 0x1626ba7e;
```

Snippet 4.6: Snippet from `EIP1271SignatureUtils`

```
1 bytes4 internal constant EIP1271_MAGICVALUE = 0x1626ba7e;
```

Snippet 4.7: Snippet from `LombardConsortium`

Impact The value might be updated in future versions of protocol. In such a case, the the value will need to be updated in both locations. If the developers mistakenly do not update the value in both locations, the protocol will become in operable

Recommendation Define the constant only in one location and reference it in from that location