Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

LBTC

Veridise Inc.

August 21, 2024

► **Prepared For:**

Lombard

► **Prepared By:**

Ajinkya Rajput
Jacob Van Greffen
Alberto Gonzalez

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

► **Version History:**

July 10, 2024	Official Report
July 4, 2024	Draft V3
Jun 27, 2024	Draft V2
May 10, 2024	Initial Draft

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
- 3.1 Audit Goals . . . 5
- 3.2 Audit Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 6

4 Vulnerability Report 9
- 4.1 Detailed Description of Issues . . . 10
  - 4.1.1 V-LOM-VUL-001: Signatures may become unverifiable . . . 10
  - 4.1.2 V-LOM-VUL-002: Repeated mints with the same data can be performed with different signatures . . . 12
  - 4.1.3 V-LOM-VUL-003: Centralization risks . . . 13
  - 4.1.4 V-LOM-VUL-004: LBTC invalid storage layout upgrade . . . 14
  - 4.1.5 V-LOM-VUL-005: withdrawFromBridge does not validate the toToken field . . . 15
  - 4.1.6 V-LOM-VUL-006: Missing event for mint . . . 16
  - 4.1.7 V-LOM-VUL-007: Inconsistency on commission limit validation . . . 17
  - 4.1.8 V-LOM-VUL-008: Constant variables named using camel case . . . 18
  - 4.1.9 V-LOM-VUL-009: Duplicate declaration . . . 19
  - 4.1.10 V-LOM-VUL-010: Unused variables and some other minor issues . . . 20

5 Appendix 21
- 5.1 Executive Summary . . . 21
- 5.2 Detailed Description of Issues . . . 22
  - 5.2.1 V-APP-VUL-001: No Decimal Check on Withdraw . . . 23
  - 5.2.2 V-APP-VUL-002: Protocol may overpay users . . . 24
  - 5.2.3 V-APP-VUL-003: Invalid Bridge Withdrawals may be Accepted . . . 25
  - 5.2.4 V-APP-VUL-004: Withdraw DoS . . . 27
  - 5.2.5 V-APP-VUL-005: Too Many Funds may be Withdrawn . . . 28
  - 5.2.6 V-APP-VUL-006: WithdrawUnsettled will always revert . . . 30
  - 5.2.7 V-APP-VUL-007: Payable functions do not credit users . . . 32
  - 5.2.8 V-APP-VUL-008: No Check that Strategy Exists when Migrated . . . 34
  - 5.2.9 V-APP-VUL-009: Lack of Inclusion of End module . . . 35
  - 5.2.10 V-APP-VUL-010: Return value should be marked as incomplete . . . 36
  - 5.2.11 V-APP-VUL-011: Potential for DoS in WaitingPool . . . 37
  - 5.2.12 V-APP-VUL-012: Approve Lazily rather than Eagerly . . . 38
  - 5.2.13 V-APP-VUL-013: Strategy can manipulate Debt . . . 39
  - 5.2.14 V-APP-VUL-014: Debt may be canceled with Fees . . . 40
  - 5.2.15 V-APP-VUL-015: Return Value does not Match Documentation . . . 41

5.2.16 V-APP-VUL-016: Value not Explicitly Returned 42
5.2.17 V-APP-VUL-017: Value can be a Boolean 43
5.2.18 V-APP-VUL-018: State changes should follow CEI pattern 44
5.2.19 V-APP-VUL-019: Get Token Decimals instead of Hardcoding 45
5.2.20 V-APP-VUL-020: Validate Yield Margin on Initialization 46
5.2.21 V-APP-VUL-021: No constructor calling _disableInitializers() in Router . 47
5.2.22 V-APP-VUL-022: Validate or Directly use Oracle Decimals 48
5.2.23 V-APP-VUL-023: Perform Item Validation 49
5.2.24 V-APP-VUL-024: Funds can potentially get stuck in bridge 50
5.2.25 V-APP-VUL-025: Incomplete Withdraw Marked as Complete 51

1 Executive Summary

From May. 6, 2024 to May. 9, 2024, Lombard engaged Veridise to review the security of their LBTC project, a BTC liquid staking protocol. This initial review covered their Ethereum-side on-chain contracts used to mint and burn LBTC based on BTC deposits or withdrawal requests. The Veridise team conducted the assessment over 6 person-days with 2 engineers, using a combination of tool-assisted analysis and extensive manual code review, focusing on the code at commit `d4b11f3`.

Later, from June 21, 2024 to June 25, 2024, Veridise was engaged again by Lombard to review an updated version of the LBTC project at commit `b42ac63`. This new version included features allowing LBTC to be transferred cross-chain on supported networks. This assessment was carried out over 3 person-days with 1 engineer, employing the same thorough approach as the initial 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. In the new version, the consortium also listens for LBTC burning events and provides a proof to the user, enabling them to mint LBTC on another supported chain. This process is referred to as a `burn/mint` bridge.

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 10 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 on chain. The low-severity issue V-LOM-VUL-004 identifies an change of storage layout between updates. Another low-severity issue V-LOM-VUL-005 reports a missing validation for the address where the proof should be verified. Also, the Veridise auditors identified 2 warnings, and 3 informational findings.

Out of 10 issues, 9 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. The auditors recommend adhering to best practices of implementing ECDSA signatures in the off-chain component.
- It was noted that the test suite tests all the user flows extensively for each contract in isolation. The auditors recommend testing with deeper sequences of user and owner actions intermixed.
- The auditors recommend adding explicit validations to ensure that mint proofs cannot be used during bridge withdrawals, and vice versa.

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
LBTC-Bridge	`b42ac63`	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
June 21 - June 25, 2024	Manual & Tools	1	3 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	4	4	4
Warning-Severity Issues	2	2	2
Informational-Severity Issues	3	2	2
TOTAL	10	9	9

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Data Validation	2
Denial of Service	1
Replay Attack	1
Centralization	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?
- Are users able to withdraw from the bridge if and only if they have valid bridging proofs?
- Are the protocol's events emitted correctly?
- Can a user cause a denial of service to the bridge?
- Can user funds be locked in the protocol?
- Can users benefit at the expense of the protocol due to changes in the configurations of the system?
- Can users use the proof for a mint operation for a bridge withdrawal operation or vice versa?

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.

Review 1: The scope of the review for the first version was 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`

Review 2: The scope of the review for the bridge logic addition was limited to the solidity contracts at the commit `b42ac63`. Specifically, the following contracts were in scope:

- `contracts/consortium/LombardConsortium.sol`
- `contracts/LBTC/ILBTC.sol`
- `contracts/LBTC/LBTC.sol`
- `contracts/libs/BitcoinUtils.sol`

Review 3: During the second review, the Veridise auditors examined the differences in previously audited contracts. The original results can be found in the Appendix 5. These files are:

- `contracts/libs/CallDataRLPReader.sol`
- `contracts/libs/EthereumVerifier.sol`
- `contracts/libs/ProofParser.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 5.1 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	Centralization risks	Low	Fixed
V-LOM-VUL-004	LBTC invalid storage layout upgrade	Low	Fixed
V-LOM-VUL-005	withdrawFromBridge does not validate the toToke.	Low	Fixed
V-LOM-VUL-006	Missing event for mint	Warning	Fixed
V-LOM-VUL-007	Inconsistency on commission limit validation	Warning	Fixed
V-LOM-VUL-008	Constant variables named using camel case	Info	Fixed
V-LOM-VUL-009	Duplicate declaration	Info	Fixed
V-LOM-VUL-010	Unused variables and some other minor issues	Info	Intended Behavior

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 function, `mint()`, 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

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

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

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.

```
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()`

`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.
- Then, instead of storing `signatureHash` to prevent replay attacks, store the nonce of `data`. This will prevent replay attacks.

Developer Response The developers fixed this issue

4.1.3 V-LOM-VUL-003: Centralization risks

Severity	Low	**Commit**	b42ac63
Type	Centralization	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	N/A		

The `LBTC` and `LombardConsortium` contracts declare an owner role that is given special permissions. In particular, the owner is given abilities such as:

- Change the `thersholdAddr` which signs the withdraw bridge and mint proofs.
- Change the `consortium` contract.
- Add and remove destinations contracts.
- Update the deposit commission to 100%.

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 generate signatures that allow minting on chain and then withdraw the minted tokens to get back BTC.

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.
- It is also advisable to reduce the deposit commission value down to a more reasonable value.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.4 V-LOM-VUL-004: LBTC invalid storage layout upgrade

Severity	Low	**Commit**	b42ac63
Type	Maintainability	**Status**	Fixed
File(s)	`LBTC.sol`		
Location(s)	LBTCStorage		
Confirmed Fix At	N/A		

The storage layout defined in the struct `LBTCStorage` is different from the previous audited version. Specifically, the variable `globalNonce` has changed its order from the first slot to the last one.

```
struct LBTCStorageV1 {
    uint256 _globalNonce;
    mapping(bytes32 => bool) _usedProofs;
    string name;
    string symbol;
    bool isWithdrawalsEnabled;
    address consortium;
}

struct LBTCStorageV2 {
    mapping(bytes32 => bool) usedProofs;
    string name;
    string symbol;
    bool isWithdrawalsEnabled;
    address consortium;
    bool isWBTCEnabled;
    IERC20 wbtc;
    address treasury;
    mapping(uint256 => address) destinations;
    mapping(uint256 => uint16) depositCommission;
    mapping(bytes32 => bool) usedBridgeProofs;
    uint256 globalNonce;
}
```

Snippet 4.4: Comparison of the `LBTCStorage` struct for both versions.

Impact The proxy's implementation upgrade will have its storage layout corrupted.

Recommendation It is important not to change the order of the storage layout when making implementation upgrades. Instead, only append new variables at the end of the previous storage layout.

Developer Response The developers responded

> This variable was unused and we removed it before deployment. The actual implementation is
>
> `https://etherscan.io/address/0xa1a961ce40c9bf5c255449194a97b85439bc0122#code`

4.1.5 V-LOM-VUL-005: withdrawFromBridge does not validate the toToken field

Severity	Low	**Commit**	b42ac63
Type	Data Validation	**Status**	Fixed
File(s)	`LBTC.sol`		
Location(s)	withdrawFromBridge		
Confirmed Fix At	N/A		

The `withdrawFromBridge` function does not validate that the encoded proof is meant to be consumed by the current contract address.

Impact If more than one deployment of `LBTC` exists on the same chain, the bridge proofs can be reused.

Recommendation Add the validation `address(this) == state.toToken`.

Developer Response The developers fixed this issue

4.1.6 V-LOM-VUL-006: 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++
);
```

Developer Response The developers fixed this issue

4.1.7 V-LOM-VUL-007: Inconsistency on commission limit validation

Severity	Warning	**Commit**	b42ac63
Type	Data Validation	**Status**	Fixed
File(s)	`LBTC.sol`		
Location(s)	addDestination()		
Confirmed Fix At	N/A		

When calling `changeDepositComission` the code validates that the `newValue` is not greater than `MAX_COMMISSION`. However, the same validation is not present in the `addDestination` function.

Impact It is possible for the owner to set a `commission` greater than `MAX_COMMISSION`.

Recommendation Apply the same validation made in `changeDepositCommission`.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.8 V-LOM-VUL-008: 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 4.5: Snippet from `LombardConsortium`

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

Snippet 4.6: Snippet from `LBTC`

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

Developer Response The developers fixed this issue

4.1.9 V-LOM-VUL-009: 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 the EIP1271 standard which returns a constant value `0x1626ba7e`

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

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

Snippet 4.7: Snippet from `EIP1271SignatureUtils`

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

Snippet 4.8: 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 from that location

Developer Response The developers fixed this issue

4.1.10 V-LOM-VUL-010: Unused variables and some other minor issues

Severity	Info	**Commit**	b42ac63
Type	Maintainability	**Status**	Intended Behavior
File(s)	`LBTC.sol`		
Location(s)	LBTCStorage		
Confirmed Fix At	N/A		

The following minor issues were found in the codebase:

1. The variables `wbtc` and `isWBTCEnabled` in the `LBTCStorage` struct are never used in the logic of the code.
2. The code makes an inconsistent usage of `msg.sender` and `_msgSender()`.
3. The `usedBridgeProofs` mapping lacks an external getter in comparison with the `usedProofs` mapping.

Impact The maintainability of the contracts is affected.

Recommendation

1. Remove the unused variables or clearly document their intention to be in the storage layout, even if they are not used.
2. Remove the use of `_msgSender()` or use it instead of `msg.sender`.
3. Add an external getter for `usedBridgeProofs`.

Developer Response The developers responded

> `wbtc` variables is not used on mainnet, but kept for compatibility with testnet where it exists

Appendix 5

5.1 Executive Summary

During the second review, the Veridise auditors examined the differences in previously audited contracts. This previous review was conducted by the previous audit team: Jon Stephens and Xiangan He, from May 26, 2023, to June 22, 2023. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers and extensive manual auditing.

Code assessment. The project developers provided the source code of the contracts for review. To facilitate the Veridise auditors' understanding of the code, the project developers held calls with the auditing team to answer various questions and to confirm findings.

The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables. The source code contained a test suite, which the Veridise auditors noted covers key protocol user-flows and some edge cases.

Summary of issues detected. The audit uncovered 25 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, the protocol may overpay users upon withdrawing from strategies that require time to unstake or gather the requested funds. Additionally, a lack of decimal checks on withdrawals from the project bridge may also cause potential overpaying to users.

The Veridise auditors also identified several medium-severity issues, including potential denial of service attacks to the bridge and users being sent the incorrect amount of funds from strategies.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the security of project. Most notably, the off-chain component of the bridge was not in scope of the audit. While the auditors did point out some issues that could occur if validation was not performed in the off-chain component, the list should not be considered exhaustive, nor should this component be considered secure. As the on-chain bridge contracts require trust in the off-chain logic, we highly recommend to seek a security review of this component.

Additionally, the MasterVault contract allows the project developers to add investment strategies in the future. Since the MasterVault contract makes some assumptions about the strategies, we would recommend that the developers document the properties that should hold for a strategy to properly interact with the MasterVault.

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.

5.2 Detailed Description of Issues

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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-APP-VUL-001	No Decimal Check on Withdraw	High	Fixed
V-APP-VUL-002	Protocol may overpay users	High	Fixed
V-APP-VUL-003	Invalid Bridge Withdrawals may be Accepted	High	Fixed
V-APP-VUL-004	Withdraw DoS	Medium	Acknowledged
V-APP-VUL-005	Too Many Funds may be Withdrawn	Medium	Fixed
V-APP-VUL-006	WithdrawUnsettled will always revert	Low	Fixed
V-APP-VUL-007	Payable functions do not credit users	Low	Fixed
V-APP-VUL-008	No Check that Strategy Exists when Migrated	Low	Fixed
V-APP-VUL-009	Lack of Inclusion of End module	Low	Intended Behavior
V-APP-VUL-010	Return value should be marked as incomplete	Low	Fixed
V-APP-VUL-011	Potential for DoS in WaitingPool	Low	Acknowledged
V-APP-VUL-012	Approve Lazily rather than Eagerly	Low	Acknowledged
V-APP-VUL-013	Strategy can Manipulate Debt	Low	Fixed
V-APP-VUL-014	Debt may be Canceled with Fees	Low	Fixed
V-APP-VUL-015	Return Value doesn't Match Documentation	Low	Fixed
V-APP-VUL-016	Value not Explicitly Returned	Warning	Fixed
V-APP-VUL-017	Value can be a Boolean	Warning	Intended Behavior
V-APP-VUL-018	State changes should follow CEI pattern	Warning	Fixed
V-APP-VUL-019	Get Token Decimals instead of Hardcoding	Warning	Fixed
V-APP-VUL-020	Validate Yield Margin on Initialization	Warning	Fixed
V-APP-VUL-021	No constructor calling _disableInitializers() i. . .	Warning	Fixed
V-APP-VUL-022	Validate or Directly use Oracle Decimals	Warning	Fixed
V-APP-VUL-023	Perform Item Validation	Warning	Acknowledged
V-APP-VUL-024	Funds can potentially get stuck in bridge	Warning	Fixed
V-APP-VUL-025	Incomplete Withdraw Marked as Complete	Warning	Fixed

5.2.1 V-APP-VUL-001: No Decimal Check on Withdraw

Severity	High	**Commit**	7869f69
Type	Data Validation	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	withdraw		

The ERC20 token standard allows developers to set a `decimals` field which is used to determine what should be considered to be a single unit of the token (and therefore allow users to trade partial tokens). While 18 is commonly used, tokens such as USDC use different values for these decimals. The bridge allows such tokens to be sent over the bridge, but does so by converting the tokens to their equivalent 18 decimal representation. It only does this on the deposit side of the bridge, however using the function shown below.

```
function _amountErc20Token(address fromToken, uint256 totalAmount) internal returns (
    uint256) {
    /* scale amount to 18 decimals */
    require(IERC20Extra(fromToken).decimals() <= 18, "Bridge/decimals-overflow");
    totalAmount *= (10**(18 - IERC20Extra(fromToken).decimals()));
    return totalAmount;
}
```

Figure 5.1: Function that scales an input amount to 18 decimals

Impact Most developers deploy the same contract with the same initialization on all ethereum-based chains. It is therefore likely that if the conversion to 18 decimals is required on one side of the bridge, it will need to be converted back on the other side. Without logic validating the decimals on the receiving (withdrawal) side of the bridge, we believe it is likely that this conversion will cause too many funds to be minted on the receiving side of the bridge.

Recommendation Either validate that all tokens on the withdrawal side of the chain have 18 decimals, or include logic to convert to the appropriate decimals on the receiving side of the chain as well.

5.2.2 V-APP-VUL-002: Protocol may overpay users

Severity	High	Commit	8cc37a7
Type	Logic Error	Status	Fixed
File(s)	MasterVault.sol		
Location(s)	_withdrawFromActiveStrategies		

The `_withdrawFromActiveStrategies` function allows project to retrieve funds that have been allocated to strategies. It does so by looping over all active strategies and requesting the given amount of funds from them. One thing to note though, is strategies may have differ in how they retrieve funds based on their `class`. For example, `IMMEDIATE` strategies can send funds back to the pool immediately without any delay. `DELAYED` strategies, however, require time to unstake or gather the requested funds. As a result, when making a request to a `DELAYED` strategy, project will send funds directly to the requested recipient. It should be noted, however, that this function loops over all strategies and in every iteration, the return values are overwritten if a withdrawal attempt is made.

```
1 function _withdrawFromActiveStrategies(address _recipient, uint256 _amount, Type
      class) private returns(uint256 withdrawn, bool incomplete, bool delayed) {
2     for(uint8 i = 0; i < strategies.length; i++) {
3         if(strategyParams[strategies[i]].active && (strategyParams[strategies[i]].
      class == class || class == Type.ABSTRACT) && strategyParams[strategies[i]].debt
      >= _amount) {
4             _recipient = strategyParams[strategies[i]].class == Type.DELAYED ?
      _recipient : address(this);
5             delayed = strategyParams[strategies[i]].class == Type.DELAYED ? true :
      false;
6             (withdrawn, incomplete) = _withdrawFromStrategy(_recipient, strategies[i],
      _amount);
7         }
8     }
9 }
```

Figure 5.2: Location where future iterations of the loop will overwrite data from previous iterations

Impact For `IMMEDIATE` and (presumably) `ABSTRACT` strategies, this can cause `MasterVault` to withdraw too many funds from the strategies. As in some locations, such as `cancelDebt` calls to `_withdrawFromActiveStrategies` are then followed by forwarding the entire pool balance, this can also cause too many funds to be forwarded to other locations. For `DELAYED` strategies, there is a similar effect where this can request too many funds to be forwarded to the given recipient and therefore for that recipient to be overpaid.

Recommendation Track the amount of funds removed and only remove the requested amount of funds.

5.2.3 V-APP-VUL-003: Invalid Bridge Withdrawals may be Accepted

Severity	High	**Commit**	7869f69
Type	Data Validation	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	_withdrawWarped		

To prevent invalid bridging requests, the `Bridge` contract performs some additional validation to ensure withdrawals come from known sources. As such, they validate that the depositing contract comes from a known source and that the source token is allowed to be bridged to the destination token by storing approved bridge destinations as mappings. On a withdraw, the validation does not check that the accessed index has been defined (i.e. is not `address(0)`), which could allow the bridge to accept unapproved bridging requests as shown below.

```
1  function _withdrawWarped(EthereumVerifier.State memory state, ProofParser.Proof
       memory proof) internal {
2      require(warpDestination(state.toToken, proof.chainId) == state.fromToken, "Bridge
       /bridge-from-unknown-destination");
3      IERC20Mintable(state.toToken).mint(state.toAddress, state.totalAmount);
4
5      emit WithdrawMinted(state.receiptHash, state.fromAddress, state.toAddress, state.
       fromToken, state.toToken, state.totalAmount);
6  }
7
8  function warpDestination(address fromToken, uint256 toChain) public view returns(
       address) {
9      return _warpDestinations[keccak256(abi.encodePacked(fromToken, block.chainid,
       _bridgeAddressByChainId[toChain]))];
10 }
```

Figure 5.3: Location where an unapproved destination token can be minted if `state.fromToken` is the 0 address

Impact If the off-chain consensus mechanism does not perform proper validation, this could allow invalid transaction receipts to be accepted. For example, if consensus approves and signs a receipt from an unsupported chain, where `fromToken` is `address(0)` an attacker could steal funds from the bridge.

Recommendation Ensure that the accessed mappings do not return `address(0)`

```
function withdraw(bytes calldata, /* encodedProof */ bytes calldata rawReceipt, bytes
     memory proofSignature) external override nonReentrant whenNotPaused {
    ...

    ProofParser.Proof memory proof = ProofParser.parseProof(proofOffset);
    require(_bridgeAddressByChainId[proof.chainId] == state.contractAddress, "Bridge/
    event-from-unknown-bridge");

    ...
}
```

Figure 5.4: Location where an invalid chain may be accepted if `state.contractAddress` is the 0 address

5.2.4 V-APP-VUL-004: Withdraw DoS

Severity	Medium	**Commit**	7869f69
Type	Denial of Service	**Status**	Acknowledged
File(s)		`EthereumVerifier.sol`	
Location(s)		parseTransactionReceipt	

From the provided tests, it appears that the off-chain component of the bridge will be responsible for processing transaction receipts to gather the emitted events, sign the receipt and then forward it to the requested bridge contract. In turn, the `Bridge` contract will process the emitted events to determine if any relevant bridging events were emitted. In the on-chain logic, it is then assumed that a single bridging request is present in the logs, as shown below, which could be used to perform a Denial of Service attack.

```
function parseTransactionReceipt(uint256 receiptOffset)
    internal
    pure
    returns (State memory state, PegInType pegInType)
{
    ...

    uint256 logsIter = CallDataRLPReader.beginIteration(logs);
    for (; logsIter < iter; ) {
        uint256 log = logsIter;
        logsIter = CallDataRLPReader.next(logsIter);
        /* make sure there is only one peg-in event in logs */
        PegInType logType = _decodeReceiptLogs(state, log);
        if (logType != PegInType.None) {
            require(
                pegInType == PegInType.None,
                "EthereumVerifier: multiple logs"
            );
            pegInType = logType;
        }
    }
    /* don't allow to process if peg-in type is unknown */
    require(pegInType != PegInType.None, "EthereumVerifier: missing logs");
    return (state, pegInType);
}
```

Figure 5.5: Location where receipts are processed and only a single bridge request is accepted

Impact As we do not have access to the off-chain computation performed by the bridge, we do not know if there is any post-processing done on the transaction receipt. If, however, there is not then any transaction that bridges funds twice will be rejected when performing a withdraw. Additionally, malicious developers can DOS the bridge by emitting events with the same signature as the `DepositWarped` event to inject multiple logs.

5.2.5 V-APP-VUL-005: Too Many Funds may be Withdrawn

Severity	Medium	**Commit**	8cc37a7
Type	Data Validation	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	withdrawUnderlying		

When performing a withdraw, it might be the case that the MasterVault does not have sufficient funds to pay back the user. If this occurs, it will attempt to retrieve any additional funds necessary from a strategy. As withdrawing from strategies may incur additional fees, the contract adds the amount withdrawn to the current balance of the pool. However, there is currently no validation if the return value is non-zero.

Impact A buggy strategy may cause users to be sent too few or too many funds.

Recommendation Validate that the amount withdrawn is within expected bounds.

Developer Response We will have authorized strategies only. For incomplete withdrawals, it is not possible to find bounds. However, tolerance() method has been added for a complete full withdrawal where bounds are checked.

```
function withdrawUnderlying(address _account, uint256 _amount) external payable
    override nonReentrant whenNotPaused onlyOwnerOrProvider returns (uint256 assets)
    {
    ...

    if(underlyingBalance < _amount) {
      uint256 debt = waitingPool.getUnbackedDebt();
      Type class = debt == 0 ? Type.ABSTRACT : Type.IMMEDIATE;

      (uint256 withdrawn, bool incomplete, bool delayed) =
    _withdrawFromActiveStrategies(_account, _amount + debt - underlyingBalance, class
    );

      if(withdrawn == 0 || debt != 0 || incomplete) {
        assets = _assessFee(assets, withdrawalFee);
        waitingPool.addToQueue(_account, assets);
        if(totalAssetInVault() > 0)
          SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), address(
    waitingPool), underlyingBalance);
        emit Withdraw(src, src, src, assets, _amount);
        return _amount;
      } else if(delayed) {
        assets = underlyingBalance;
      } else {
        assets = underlyingBalance + withdrawn;
      }
    }

    ...
}
```

Figure 5.6: Snippet of `withdrawUnderlying` that withdraws funds from strategies

5.2.6 V-APP-VUL-006: WithdrawUnsettled will always revert

Severity	Low	**Commit**	8cc37a7
Type	Logic Error	**Status**	Fixed
File(s)	`WaitingPool.sol`		
Location(s)	withdrawUnsettled		

The `WaitingPool` contract implements a queue of users to which the protocol owes funds. As this contract gains funds, the `tryRemove` function will process the queue in order to repay users. In the event that a user cannot be repaid, `withdrawUnsettled` allows users to request their funds only after `tryRemove` has attempted to serve them. However, due to the line below `withdrawUnsettled` will always revert. This is because, as shown below, `tryRemove` sets `people[_index]._settled` to

```
function withdrawUnsettled(uint256 _index) external {
    ...
    require(!people[_index]._settled && _index < index && people[_index]._address ==
    src, "WaitingPool/already-settled");
    ...
}
```

Figure 5.7: The following check will always cause `withdrawUnsettled` to revert:

`true` if `_index < index`.

Impact `withdrawUnsettled` will always revert due to the definition of `tryRemove`

Recommendation This function no longer seems to serve a purpose after removing the ability to send native tokens, we would recommend removing it altogether.

```
function tryRemove() external onlyMasterVault {
    uint256 balance;
    uint256 cap = 0;
    for(uint256 i = index; i < people.length; i++) {
        balance = getPoolBalance();
        uint256 userDebt = people[index]._debt;
        address userAddr = people[index]._address;
        if(balance >= userDebt && userDebt != 0 && !people[index]._settled && cap <
    capLimit) {
            totalDebt -= userDebt;
            people[index]._settled = true;
            emit WithdrawCompleted(userAddr, userDebt);

            cap++;
            index++;

            IERC20Upgradeable(underlying).safeTransfer(userAddr, userDebt);
        } else return;
    }
}
```

Figure 5.8: The definition of `tryRemove` which always sets `_settled` when processing a user

5.2.7 V-APP-VUL-007: Payable functions do not credit users

Severity	Low	**Commit**	8cc37a7
Type	Locked Funds	**Status**	Fixed
File(s)	depositUnderlying, withdrawUnderlying, deposit, withdraw		
Location(s)	MasterVault, CerosYieldConverterStrategyLs, ...		

Several functions in the protocol are marked as `payable`, allowing them to accept ETH deposits. If a user provides such funds, however, they will not be credited with any shares in return. Instead, the funds will eventually be locked inside the protocol as most of these contracts do not include any logic to interact with native currency.

Impact This can cause user funds to be locked in the contract.

Recommendation Either remove the payable modifier or require that `msg.value` is 0 if that is not possible.

Developer Response We've removed the payable functions.

```
function depositUnderlying(uint256 _amount) external payable override nonReentrant
    whenNotPaused onlyOwnerOrProvider returns (uint256 shares) {

        require(_amount > 0, "MasterVault/invalid-amount");
        address src = msg.sender;

        SafeERC20Upgradeable.safeTransferFrom(IERC20Upgradeable(asset()), src,
    address(this), _amount);
        shares = _assessFee(_amount, depositFee);

        uint256 waitingPoolDebt = waitingPool.totalDebt();
        uint256 waitingPoolBalance = IERC20Upgradeable(asset()).balanceOf(address(
    waitingPool));
        if(waitingPoolDebt > 0 && waitingPoolBalance < waitingPoolDebt) {
            uint256 waitingPoolDebtDiff = waitingPoolDebt - waitingPoolBalance; //
    @audit-info transfers $$ to debt pool to pay off all debt first
            uint256 poolAmount = (waitingPoolDebtDiff < shares) ? waitingPoolDebtDiff
     : shares;
            SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), address(
    waitingPool), poolAmount);
        }

        _mint(src, shares);

        if(allocateOnDeposit == 1) allocate();

        emit Deposit(src, src, _amount, shares); // @audit incorrect event emission
    }
```

Figure 5.9: The following code snippet contains no logic for minting shares to a user if they had deposited ETH.

5.2.8 V-APP-VUL-008: No Check that Strategy Exists when Migrated

Severity	Low	**Commit**	8cc37a7
Type	Data Validation	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	migrateStrategy		

The protocol allows strategies to be upgraded or migrated to new versions of the strategy. In doing so, funds from the old strategy are withdrawn it is removed from the set of active strategies. The new strategy is then added in its place. Unlike in `addStrategy`, however, an existing and already active strategy can be added a second time to the `strategies` list.

```
1  function migrateStrategy(address _oldStrategy, address _newStrategy, uint256
       _newAllocation, Type _class) external onlyOwnerOrManager {
2      require(_oldStrategy != address(0) && _newStrategy != address(0));
3
4      ...
5
6      StrategyParams memory params = StrategyParams({active: true, class: _class,
       allocation: _newAllocation, debt: 0});
7
8      bool isValidStrategy;
9      for(uint256 i = 0; i < strategies.length; i++) {
10         if(strategies[i] == _oldStrategy) {
11             isValidStrategy = true;
12             strategies[i] = _newStrategy;
13             strategyParams[_newStrategy] = params;
14
15             break;
16         }
17     }
18
19     require(isValidStrategy, "MasterVault/invalid-oldStrategy");
20     require(_deactivateStrategy(_oldStrategy),"MasterVault/cannot-deactivate");
21     require(_isValidAllocation(), "MasterVault/>100%");
22
23     emit StrategyMigrated(_oldStrategy, _newStrategy, _newAllocation);
24 }
```

Figure 5.10: The migrate strategy function which does not check that `_newStrategy` already exists

Impact This could accidentally overwrite a strategy's stored configuration parameters including, most importantly, the amount of debt that has been locked in the strategy.

Recommendation Specifically disallow migrating old strategies to existing strategies.

5.2.9 V-APP-VUL-009: Lack of Inclusion of End module

Severity	Low	**Commit**	8cc37a7
Type	Maintainability	**Status**	Intended Behavior
File(s)		N/A	
Location(s)		N/A	

project's MakerDAO fork doesn't include `end` - the emergency shutdown module used to call most of the `cage` functions located throughout. The `End`'s purpose is to coordinate a shutdown where the protocol closes down the system and reimburses stablecoin holders. This is a process that can occur during upgrades (Dai iterations), as well as for security reasons in the event that implementation flaws arise in both in the code and in the design.****

Impact Not including the `end` module would cause shutdown to become more difficult to execute.

Recommendation Consider forking the `end` module.

Developer Response Since we have single entry point, we can just halt interaction rather than all of the contracts that were forked from MakerDAO.

5.2.10 V-APP-VUL-010: Return value should be marked as incomplete

Severity	Low	**Commit**	8cc37a7
Type	Logic Error	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	_withdrawFromStrategy		

As MasterVault strategies may not have sufficient funds to fully fulfill a request, it might be the case that a withdraw is partially complete. For this reason, the `_withdrawFromStrategy` function returns a boolean to indicate that the full amount could not be withdrawn. In some cases, however, the function indicates that the withdraw was complete despite no funds being withdrawn as shown below.

```
function _withdrawFromStrategy(address _recipient, address _strategy, uint256 _amount
    ) private returns(uint256, bool incomplete) {
    ...

    if(capacity <= 0 || chargedCapacity > capacity) return (0, false);
    else if(capacity < _amount) incomplete = true;

    if(params.class == Type.DELAYED && incomplete) return (0, true);

    ...

    emit WithdrawnFromStrategy(_strategy, _amount, chargedCapacity);
    return (chargedCapacity, incomplete);
}
```

Figure 5.11: Location where an incomplete withdraw is not identified as such

Impact As some parts of the code rely on this boolean to determine if the withdraw was completed successfully, this could cause funds to be tracked incorrectly.

Recommendation Only return `false` if a withdraw is completed successfully

5.2.11 V-APP-VUL-011: Potential for DoS in WaitingPool

Severity	Low	Commit	8cc37a7
Type	Denial of Service	Status	Acknowledged
File(s)	`WaitingPool.sol`		
Location(s)	tryRemove		

project uses a waiting queue for users who have yet to be paid their funds on a withdrawal. Over time, this queue is processed by sending users the funds that they are owed in the order of the queue. While this will work for most `ERC20` tokens, the developers should ensure that the underlying asset does not have user callbacks as are used by some custom tokens and `ERC20` extensions such as `ERC777`. If such a currency is used, there is the potential that a single user can perform a denial of service attack on the waiting pool.

```
function tryRemove() external onlyMasterVault {
    uint256 balance;
    uint256 cap = 0;
    for(uint256 i = index; i < people.length; i++) {
        balance = getPoolBalance();
        uint256 userDebt = people[index]._debt;
        address userAddr = people[index]._address;
        if(balance >= userDebt && userDebt != 0 && !people[index]._settled && cap <
    capLimit) {
            totalDebt -= userDebt;
            people[index]._settled = true;
            emit WithdrawCompleted(userAddr, userDebt);

            cap++;
            index++;

            IERC20Upgradeable(underlying).safeTransfer(userAddr, userDebt);
        } else return;
    }
}
```

Figure 5.12: Definition of the `tryRemove` function

Impact If there are user callbacks, a user could always revert on the transfer to `userAddr`. This would cause the entire transaction to revert, preventing users from recovering funds.

Recommendation Ensure that any underlying assets do not contain callbacks to the user on a transfer.

Developer Response We are aware of this and we will make sure the underlyings are not ERC777 or add support for them later if needed.

5.2.12 V-APP-VUL-012: Approve Lazily rather than Eagerly

Severity	Low	**Commit**	8cc37a7
Type	Access Control	**Status**	Acknowledged
File(s)	ConverterStrategyLs.sol, CerosYieldConverterStrategySp.sol, CerosRouterLs.sol, CerosRouterLsEth.sol, CerosRoute		
Location(s)	Multiple		

Many of the project contracts have to transfer funds between each other. To simplify the logic, it is a common pattern for contracts to approve other addresses to access all of the stored funds as shown below. Should there be an error in one of the approved contracts or should one of these addresses be compromised, this could allow funds to be stolen.

```
function initialize(address _destination, address _feeRecipient, address
    _underlyingToken, address _masterVault) external initializer {
    __BaseStrategy_init(_destination, _feeRecipient, _underlyingToken);

    masterVault = IMasterVault(_masterVault);
    underlying.approve(address(_destination), type(uint256).max);
    underlying.approve(address(_masterVault), type(uint256).max);
}
```

Figure 5.13: Location where a strategy approves other addresses to access all of its funds

Recommendation Rather than approve max funds on initialization, consider instead approving funds only when necessary.

Developer Response We are aware of this. The contracts are kept as restricted as possible. We will keep this in mind for future changes.

5.2.13 V-APP-VUL-013: Strategy can manipulate Debt

Severity	Low	**Commit**	8cc37a7
Type	Data Validation	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	_depositToStrategy		

The `MasterVault` contract is used to invest stored user funds using various strategies. The amount of invested funds are then tracked so that they can later be retrieved when users perform a withdraw. As some investment strategies may incur fees, however, the amount the entire deposit may not be invested. To accurately track the amount of invested funds, the protocol therefore determines two values: the strategy's capacity, and capacity after fees have been charged. The capacity is then deposited while the capacity with fees is added to a strategy's debt. There is currently no validation on the relationship between these two values though.

```
function _depositToStrategy(address _strategy, uint256 _amount) private returns (bool
     success) {
    ...

    // 'capacity' is total depositable; 'chargedCapacity' is capacity after charging
    fee
    (uint256 capacity, uint256 chargedCapacity) = IBaseStrategy(_strategy).canDeposit
    (_amount);
    if(capacity <= 0 || capacity > _amount || chargedCapacity > capacity) return
    false;

    totalDebt += chargedCapacity;
    strategyParams[_strategy].debt += chargedCapacity;

    SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), _strategy, capacity
    );
    IBaseStrategy(_strategy).deposit{value: msg.value}(capacity);

    emit DepositedToStrategy(_strategy, capacity, chargedCapacity);
    return true;
}
```

Figure 5.14: Function used to deposit funds into a strategy

Impact As there is no validation between these two values, a strategy's debt may not be accurately tracked. For example, if there is a bug in a strategy, it could be the case that it returns 0 as the capacity after fees, in which case the strategy's debt would not increase.

Recommendation Validate the allowable loss due to fees to prevent potential errors.

5.2.14 V-APP-VUL-014: Debt may be canceled with Fees

Severity	Low	**Commit**	8cc37a7
Type	Logic Error	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	cancelDebt		

As the MasterVault may not be able to retrieve all funds at the time a request is withdrawn, project places users in a queue to be refunded as funds are made available. As this occurs, managers may cancel the debt to users which will withdraw additional funds from certain strategies and forward those funds to the waiting pool. When calculating the amount of funds that can be sent to the waiting pool, it appears that there is an error in the asset calculation. After performing a withdraw, the total assets in the vault is first queried, which determines the amount of funds in the vault without accumulated fees. After that though, if the withdraw was successful this amount is augmented with the amount withdrawn, increasing the amount to send above the amount of user funds in the pool.

```
1  function cancelDebt(Type _class) public onlyOwnerOrManager {
2      ...
3
4      (withdrawn,,delayed) = _withdrawFromActiveStrategies(address(waitingPool),
       withdrawAmount + 1, _class);
5      uint256 amount = totalAssetInVault();
6      if(withdrawn > 0 && !delayed) amount += withdrawn;
7      SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), address(waitingPool
       ), amount);
8
9      ...
10 }
```

Figure 5.15: Location where `cancelDebt` withdraws from strategies to regain funds

Impact As `totalAssetInVault` returns the total assets without fees, any additional amount added to this will either come from fees, or will cause the transfer of funds to revert, preventing the debt from being canceled.

Recommendation Only send the funds reported by `totalAssetInVault`.

5.2.15 V-APP-VUL-015: Return Value does not Match Documentation

Severity	Low	**Commit**	8cc37a7
Type	Maintainability	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	withdrawUnderlying		

When funds are withdrawn from `MasterVault` a withdrawal fee may be charged. According to the inline documentation, it appears that withdraw is intended to return the amount of funds withdrawn after fees are charged. However, currently it returns the amount of funds that the user requested to withdraw.

```
1  /** Withdraw underlying assets via projectProvider
2    * @param _account receipient
3    * @param _amount underlying assets to withdraw
4    * @return assets underlying assets excluding any fees
5    */
6  function withdrawUnderlying(address _account, uint256 _amount) external payable
       override nonReentrant whenNotPaused onlyOwnerOrProvider returns (uint256 assets)
       {
7      ...
8
9      assets = _assessFee(assets, withdrawalFee);
10     SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), _account, assets);
11
12     emit Withdraw(src, src, src, assets, _amount);
13     return _amount;
14 }
```

Figure 5.16: Location where the code is inconsistent with the documentation

Impact It is possible that incorrect values may be reported to the user as the return value of this function is later emitted in the project protocol.

Recommendation If the code is correct, update the documentation so that developers don't misuse this function.

5.2.16 V-APP-VUL-016: Value not Explicitly Returned

Severity	Warning	**Commit**	8cc37a7
Type	Maintainability	**Status**	Fixed
File(s)	MasterVault.sol		
Location(s)	_deactivateStrategy		

Solidity allows functions to return values without an explicit return statement, in which case the default value will be returned.

```
function _deactivateStrategy(address _strategy) private returns(bool success) {
    if (strategyParams[_strategy].debt <= 10) {
        strategyParams[_strategy].active = false;
        strategyParams[_strategy].debt = 0;
        return true;
    }
}
```

Figure 5.17: Example where the default value is returned on a program path

Recommendation For clarity, we recommend that all functions explicitly return a value on all program paths rather than relying on default values.

5.2.17 V-APP-VUL-017: Value can be a Boolean

Severity	Warning	**Commit**	8cc37a7
Type	Maintainability	**Status**	Intended Behavior
File(s)	function depositUnderlying		
Location(s)	MasterVault.sol		

The project developers declare the `allocateOnDeposit` variable as a `uint256` but constrain it's behavior so that it is effectively a boolean.

```
uint256 public allocateOnDeposit;

...

function changeAllocateOnDeposit(uint256 _status) external onlyOwner {
    require(_status >= 0 && _status < 2, "MasterVault/range-0-or-1");
    allocateOnDeposit = _status;

    emit AllocationOnDepositChangeed(_status);
}
```

Figure 5.18: The behavior of `allocateOnDeposit` is enforced to behave like a boolean.

Impact Changing this to a boolean renders the check unnecessary and improves code clarity.

Recommendation Implement changes as described above.

Developer Response In the future, we may add additional variable assignments.

5.2.18 V-APP-VUL-018: State changes should follow CEI pattern

Severity	Warning	**Commit**	8cc37a7
Type	Best Practices	**Status**	Fixed
File(s)	CerosRouterLs, CerosRouterLsETH, CerosRouterSp		
Location(s)	claimProfit		

Due to the prevalence of reentrancy attacks, it is considered good practice to follow the Checks-Effects-Interaction (CEI) pattern. Under this pattern functions should first perform any necessary checks and update the internal state of the contract before transferring control outside of the contract. We would recommend that the developers follow this pattern.

```
function claimProfit(address _recipient) external nonReentrant {
                ...
                s_aMATICc.transfer(_recipient, profit);  // aMATICc
        s_profits[msg.sender] -= profit;
                ...
}
```

Figure 5.19: Location where CEI should be used.

Impact Using the Checks-Effects-Interaction pattern minimizes reentrancy attack risk.

Recommendation Implement the above by switching the order of the state change and transfer.

5.2.19 V-APP-VUL-019: Get Token Decimals instead of Hardcoding

Severity	Warning	Commit	89cdc07
Type	Maintainability	Status	Fixed
File(s)	`WstETHOracle.sol`		
Location(s)	peek		

As no price oracle exists between the `MasterVault` share token and other currencies such as USD, the project developers provide one that relies on oracles for existing currencies. However, this contract relies on using hard-coded values for token decimals, which can be error prone if the code is re-used.

```
function peek() public view returns (bytes32, bool) {
    (
        /*uint80 roundID*/,
        int price,
        /*uint startedAt*/,
        /*uint timeStamp*/,
        /*uint80 answeredInRound*/
    ) = priceFeed.latestRoundData();
    if (price < 0) {
        return (0, false);
    }

    // Get stETH equivalent to 1wstETH and multiply with stETH price
    uint256 stETH = wstETH.getStETHByWstETH(1e18);
    uint256 wstETHPrice = (stETH * uint(price * (10**10))) / 1e18;

    // Get wstETH equivalent to 1share in MasterVault
    uint256 wstETH = masterVault.previewRedeem(1e18);
    uint256 sharePrice = (wstETHPrice * wstETH) / 1e18;

    return (bytes32(sharePrice), true);
}
```

Recommendation Make the token and oracle decimals immutable values and initialize them with the proper values from the contracts themselves.

5.2.20 V-APP-VUL-020: Validate Yield Margin on Initialization

Severity	Warning	**Commit**	89cdc07
Type	Maintainability	**Status**	Fixed
File(s)	`MasterVault_V2.sol`		
Location(s)	initialize		

The `MasterVault_V2` contract stores `wstETH` liquid staking tokens so that users may utilize them as collateral for project. During the time that these tokens are staked with project, they will also generate rewards as one `wstETH` will be worth more `stETH`. As part of the project protocol, they will take a cut of those generated rewards as determined by `yieldMargin` while they hold the `wstETH` as collateral. When this value is set on initialization no validation is performed, as shown below, unlike when it is set in `changeYieldMargin`.

```
function initialize(string memory _name, string memory _symbol, uint256 _yieldMargin,
     address _underlying) external initializer {
    __ERC4626_init(IERC20MetadataUpgradeable(_underlying));
    __ERC20_init(_name, _symbol);
    __Ownable_init();
    __Pausable_init();
    __ReentrancyGuard_init();

    yieldMargin = _yieldMargin;
    yieldBalance = 0;
}
```

Figure 5.20: The `initialize` function with no validation of the yield margin

Impact There is the potential that this could cause initialization issues.

Recommendation Perform similar validation in `initialize` that is performed in `changeYieldMargin`

5.2.21 V-APP-VUL-021: No constructor calling _disableInitializers() in Router

Severity	Warning	Commit	8cc37a7
Type	Best Practices	Status	Fixed
File(s)	CerosRouterSP		
Location(s)	constructor		

`_disableInitializers()` prevents initialization of the implementation contract itself, providing an extra line of defense against attackers looking to initialize the contract as an attack against the proxy.

Impact If the initializer is not called upon deployment, attackers can call the contract to initialize the contract.

Recommendation Add the constructor and `_disableInitializers()`, similar to what many of the other contracts in scope already have.

5.2.22 V-APP-VUL-022: Validate or Directly use Oracle Decimals

Severity	Warning	**Commit**	8cc37a7
Type	Data Validation	**Status**	Fixed
File(s)	`ManticOracle.sol`		
Location(s)	peek		

project implements several price oracles that fetch data from chainlink. While these chainlink oracle pairs have predictable decimals for the returned prices, they can vary depending on the pair used. For example, USD pairs use 8 decimals while ETH pairs use 18 decimals.

```
function peek() public view returns (bytes32, bool) {
    (
        /*uint80 roundID*/,
        int price,
        /*uint startedAt*/,
        /*uint timeStamp*/,
        /*uint80 answeredInRound*/
    ) = priceFeed.latestRoundData();
    if (price < 0) {
        return (0, false);
    }
    return (bytes32(uint(price * (10**10))), true);
}
```

Figure 5.21: Code that fetches a chainlink price and converts it to 18 decimals

Impact Should the wrong pair is used, it is possible for the oracles to return incorrect prices.

Recommendation Either use the pair's decimals directly or validate the pair's decimals on initialization

5.2.23 V-APP-VUL-023: Perform Item Validation

Severity	Warning	**Commit**	7869f69
Type	Data Validation	**Status**	Acknowledged
File(s)	`CallDataRLPReader.sol`		
Location(s)	toUintStrict, toUint, beginIteration		

The project Bridge encodes transaction data in the RLP format, which is then parsed by the on-chain contracts. Importantly, under this format most items (except for small bytes) are associated with a length indicating the size of the item. project's RLP reader uses such lengths to determine the location of items, but currently does not use them to validate user parsing requests, such as when a user requests to convert an item's data to an integer as shown below.

```
function toUintStrict(uint256 ptr) internal pure returns (uint256) {
    // one byte prefix
    uint256 result;
    assembly {
        result := calldataload(add(ptr, 1))
    }
    return result;
}
```

Figure 5.22: Location where no validation is performed when reading an integer

Impact The lack of validation could cause the reader to misinterpret data or improperly use data.

Recommendation For functions such as `beginIteration`, ensure that the current item is a list. For items such as `toUint` and `toUintStrict` validate that the items are of an appropriate size.

5.2.24 V-APP-VUL-024: Funds can potentially get stuck in bridge

Severity	Warning	**Commit**	89cdc07
Type	Locked Funds	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	receive()		

The project bridge implements a 'primitive' to receive ETH. Funds that get sent in this way, however, do not get accounted for.

```
1 // --- Primitives ---
2     receive() external payable {}
```

Figure 5.23: The bridge can receive ETH via. this receive function

Impact As a result, users that send ETH to the bridge will have their funds locked inside the bridge.

Recommendation In the short term, remove `receive()` since the bridge only seems to be supporting USDC / ERC20 inputs of funds.

In the long term, the team can implement support for bridging ETH via. wETH, eliminating the need to take in native ETH while maintaining the same interface as today.

5.2.25 V-APP-VUL-025: Incomplete Withdraw Marked as Complete

Severity	Warning	**Commit**	8cc37a7
Type	Maintainability	**Status**	Fixed
File(s)	`MasterVault.sol`		
Location(s)	_withdrawFromActiveStrategies		

When the MasterVault contract requires additional funds, it may need to withdraw funds from its investment strategies using the `_withdrawFromActiveStrategies` function. As the investment strategies may not have sufficient funds, this function may return that the withdrawal request is incomplete, or rather it was unable to retrieve the desired amount of funds. In the current version of the source code, though, when a strategy could not be found to withdraw from, the withdrawal will be marked as complete (or rather not incomplete) due to the fact that the function will return the default value for the boolean.

```
1 function _withdrawFromActiveStrategies(address _recipient, uint256 _amount, Type
      class) private returns(uint256 withdrawn, bool incomplete, bool delayed) {
2     for(uint8 i = 0; i < strategies.length; i++) {
3         if(strategyParams[strategies[i]].active && (strategyParams[strategies[i]].
      class == class || class == Type.ABSTRACT) && strategyParams[strategies[i]].debt
      >= _amount) {
4             _recipient = strategyParams[strategies[i]].class == Type.DELAYED ?
      _recipient : address(this);
5             delayed = strategyParams[strategies[i]].class == Type.DELAYED ? true :
      false;
6             (withdrawn, incomplete) = _withdrawFromStrategy(_recipient, strategies[i],
      _amount);
7         }
8     }
9 }
```

Figure 5.24: Source code of the `_withdrawFromActiveStrategies` function

Impact Currently callers of this function validate both `incomplete` and the amount withdrawn. If future code does not do so, however, and relies only on the `incomplete` flag, it could cause the protocol to misbehave.

Recommendation If no withdraw was made, return that it is incomplete.