// Smart Contract Security Assessment

10.07.2024 - 10.10.2024

EVM SC

Lombard

HALBORN

EVM SC - Lombard

Prepared by: HALBORN

Last Updated 10/11/2024

Date of Engagement by: October 7th, 2024 - October 10th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	1	3	3

TABLE OF CONTENTS

1. Introduction

Lombard engaged Halborn to conduct a security assessment on their smart contracts beginning on 2024-10-07 and ending on 2024-10-11. The security assessment was scoped to the smart contracts provided in directly be the team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided four days for the engagement and assigned one full-time security engineer to check the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified several security concerns that were mostly addressed by the Lombard team.

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local or public testnet deployment (`Foundry`, `Remix IDE`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: evm-smart-contracts

(b) Assessed Commit ID: 357aa01

(c) Items in scope:

- contracts/factory/ProxyFactory.sol
- contracts/LBTC.sol
- contracts/pmm/BTCB/BTCB.sol
- contracts/libs/FeeUtils.so

Out-of-Scope:

REMEDIATION COMMIT ID:

- 6e5e9cd
- https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/pmm/BTCB/BTCB.sol
- https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/factory/ProxyFactory.sol
- https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/pmm/BTCB/BTCB.sol
https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/LBTC/LBTC.sol
- https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/LBTC/LBTC.sol

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	3	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT ASSUMPTION ON TOKEN DECIMALS LEADING TO POTENTIAL UNDERFLOWS	MEDIUM	SOLVED - 10/10/2024
CENTRALIZATION RISK IN WITHDRAWAL FUNCTIONS	LOW	RISK ACCEPTED - 10/10/2024
UNREASONABLE COMMISSION ALLOWANCE IN SETRELATIVEFEE FUNCTION	LOW	SOLVED - 10/10/2024
INCORRECT ERC20 INTERFACE	LOW	SOLVED - 10/10/2024
UNLOCKED PRAGMA COMPILER	INFORMATIONAL	SOLVED - 10/10/2024
UNUSED IMPORTS. ERRORS AND VARIABLES	INFORMATIONAL	SOLVED - 10/10/2024
UNSAFE TRANSFER METHOD IN WITHDRAW FUNCTIONS	INFORMATIONAL	SOLVED - 10/10/2024

7. FINDINGS & TECH DETAILS

7.1 INCORRECT ASSUMPTION ON TOKEN DECIMALS LEADING TO POTENTIAL UNDERFLOWS

// MEDIUM

Description

The `swapBTCBToLBTC` function defined in BTCBPMM contract facilitates the exchange of BTCB (a wrapped version of Bitcoin on Binance Smart Chain) for LBTC. The function handles the conversion process while accounting for any difference in decimals between the tokens, calculates applicable fees, and mints the resulting LBTC tokens for the user.
The function relies on the calculation of `decimalsDifference` to adjust for discrepancies in token decimals between BTCB and LBTC:

```
        function swapBTCBToLBTC(uint256 amount) external whenNotPaused {
            PMMStorage storage $ = _getPMMStorage();

            ILBTC lbtc = $.lbtc;
            IERC20Metadata btcb = $.btcb;

            uint256 decimalsDifference = 10 ** (btcb.decimals() - lbtc.decima
            uint256 amountLBTC = (amount / decimalsDifference);
            if(amountLBTC == 0) revert ZeroAmount();

            if ($.totalStake + amountLBTC > $.stakeLimit) revert StakeLimitEx

            // relative fee
            uint256 fee = FeeUtils.getRelativeFee(amountLBTC, $.relativeFee);

            $.totalStake += amountLBTC;
            btcb.safeTransferFrom(_msgSender(), address(this), amountLBTC * d
            lbtc.mint(_msgSender(), amountLBTC - fee);
            lbtc.mint(address(this), fee);
        }
```

This calculation assumes that `btcb.decimals` is always greater than or equal to `lbtc.decimals`. If this condition is not guaranteed, the subtraction can underflow, resulting in an incorrect value for `decimalsDifference`. If the underflow occurs, it may cause an unintended revert, resulting in a DoS condition, where the swap operation fails for legitimate users.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:C (6.3)

Recommendation

- Modify the code to handle cases where `btcb.decimals` is less than `lbtc.decimals`, ensuring safe computation without the risk of underflow.
- Add an explicit check to ensure that the calculation of `decimalsDifference` is safe and does not underflow:

```
require(btcb.decimals() >= lbtc.decimals(), "Token decimals mismatch may cause
```

Remediation

SOLVED: The **Lombard team** solved the issue as recommended.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/6e5e9cd61e4c57e222c9d5383b0f94ce3f0ef883

References

lombard-finance/smart-contracts/contracts/pmm/BTCB/BTCB.sol#L72

7.2 CENTRALIZATION RISK IN WITHDRAWAL FUNCTIONS

// LOW

Description

The BTCBPMM contract includes two withdrawal functions that allow the administrator to withdraw the tokens held in the contract:

withdrawBTCB Function:

```
function withdrawBTCB(uint256 amount) external whenNotPaused onlyRole
    PMMStorage storage $ = _getPMMStorage();
    $.btcb.transfer($.withdrawAddress, amount);
}
```

withdrawLBTC Function:

```
function withdrawLBTC(uint256 amount) external whenNotPaused onlyRole
    PMMStorage storage $ = _getPMMStorage();
    $.lbtc.transfer($.withdrawAddress, amount);
}
```

This functions allow an address with the `DEFAULT_ADMIN_ROLE` to withdraw a specified amount of tokens from the contract. There are no limitations or conditions on the withdrawal amount. The admin can withdraw up to the entire balance of BTCB or LBTC from the contract, without any restrictions.
If the account holding the `DEFAULT_ADMIN_ROLE` is compromised or acts maliciously, they could withdraw all the funds, causing a complete loss of user assets.

The same vector applies to the `addMinter` function, which allows the owner to grant an arbitrary address the ability to mint LBTC tokens.

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:C/R:N/S:C (3.1)

Recommendation

Instead of using a single admin key, use a multi-signature wallet for the **DEFAULT_ADMIN_ROLE**.

Remediation

RISK ACCEPTED: The **Lombard team** accepted the risk of the issue.

References

lombard-finance/smart-contracts/contracts/pmm/BTCB/BTCB.sol#L87-L95

7.3 UNREASONABLE COMMISSION ALLOWANCE IN SETRELATIVEFEE FUNCTION

// LOW

Description

The FeeUtils library is used to calculate and validate fees within the contract. Specifically, the function `validateCommission` is intended to ensure that the fee (commission) rate is within a valid range before being set by the admin.

```
        function validateCommission(uint16 commission) internal pure {
            if (commission >= MAX_COMMISSION)
                revert BadCommission();
        }
```

The `MAX_COMMISSION` is set to 10000, representing 100% in basis points. This function is used to validate the commission input value to prevent unreasonable or incorrect fee settings. The validation checks that the commission is less than `MAX_COMMISSION`, meaning the fee can be set up to 100%, but not beyond.
If the fee is set to 100%, the entire staked or swapped amount could be collected as a fee, leaving zero tokens for the user. This creates an extremely unfavorable condition where the user receives nothing after the swap, which is neither user-friendly nor practical.

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:C/R:N/S:C (3.1)

Recommendation

Consider setting an upper bound on commission. To avoid unreasonable fees and ensure that users receive meaningful value after staking or swapping, update the `validateCommission` function to cap the maximum allowable fee to a more practical level.

Remediation

RISK ACCEPTED: The **Lombard team** accepted the risk of the issue.

References

lombard-finance/smart-contracts/contracts/libs/FeeUtils.sol#L22-L25

7.4 INCORRECT ERC20 INTERFACE

// LOW

Description

The ERC20 `transfer` function and `decimals` function in the `ILBTC` interface do not conform to the ERC20 standard. The current implementation is:

```
interface ILBTC {
    function mint(address to, uint256 amount) external;
    function transfer(address to, uint256 amount) external;
    function decimals() external view returns (uint256);
}
```

```
function transfer(address to, uint256 value) external returns (bool);
function decimals() external view returns (uint8);
```

The `transfer` function must return a `bool` to indicate success, and the `decimals` function should return a `uint8` instead of `uint256`. This incorrect implementation results in incompatibility with ERC20-compliant contracts and services that expect these standard function signatures.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation

Update the `transfer` function to return `bool` and the `decimals` function to return `uint8` to ensure compatibility with the ERC20 standard.

Remediation

SOLVED: The **Lombard team** solved the issue. The interface inherit from IERC20Metadata.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/pmm/BTCB/BTCB.sol

References

lombard-finance/smart-contracts/contracts/pmm/BTCB/BTCB.sol#L13-L14

7.5 UNLOCKED PRAGMA COMPILER

// INFORMATIONAL

Description

The ProxyFactory contract currently use floating pragma version `^0.8.14`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.0`, and less than `0.9.0`. It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.
Additionally, using a newer compiler version that introduces default optimizations, including unchecked overflow for gas efficiency, presents an opportunity for further optimization.

Score

Impact:

Likelihood:

Recommendation

Lock the pragma version to the same version used during development and testing.

Remediation

SOLVED: The **Lombard team** solved the issue as recommended.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/factory/ProxyFactory.sol

References

lombard-finance/smart-contracts/contracts/Factory/ProxyFactory.sol#L2

7.6 UNUSED IMPORTS. ERRORS AND VARIABLES

// INFORMATIONAL

Description

Throughout the code in scope, there are several instances where the imports, errors, and events are declared but never used.
In `LBTC.sol` :

- `import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";`
- `import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";`
- `bool isWBTCEnabled;`
- `IERC20 wbtc;`
- `mapping(uint256 => address) __removed_destinations;`
- `mapping(uint256 => uint16) __removed_depositCommission;`
- `mapping(bytes32 => bool) __removed_usedBridgeProofs;`
- `uint256 __removed_globalNonce;`

In `BTCB.sol` :

- `import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";`
- `error UnauthorizedAccount(address account);`

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider removing all unused imports, errors.

Remediation

SOLVED: The **Lombard team** solved the issue as recommended.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/pmm/BTCB/BTCB.sol https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/LBTC/LBTC.sol

7.7 UNSAFE TRANSFER METHOD IN WITHDRAW FUNCTIONS

// INFORMATIONAL

Description

The `withdrawLBTC` and `withdrawBTCB` functions use direct calls to the `transfer` function instead of using `safeTransfer`, as implemented in `swapBTCBToLBTC`.

withdrawBTCB Function:

```
        function withdrawBTCB(uint256 amount) external whenNotPaused onlyRole
            PMMStorage storage $ = _getPMMStorage();
            $.btcb.transfer($.withdrawAddress, amount);
        }
```

withdrawLBTC Function:

```
        function withdrawLBTC(uint256 amount) external whenNotPaused onlyRole
            PMMStorage storage $ = _getPMMStorage();
            $.lbtc.transfer($.withdrawAddress, amount);
        }
```

Direct use of transfer can lead to potential issues, such as failed transfers not being correctly handled, which can introduce vulnerabilities if the tokens do not conform strictly to the ERC20 standard.

Score

Impact:

Likelihood:

Recommendation

Consider replacing the usage of `transfer` with `safeTransfer` in the `withdrawLBTC` and `withdrawBTCB` functions.

Remediation

SOLVED: The **Lombard team** solved the issue as recommended.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/blob/438ddbfbde18df01972fbc0e30e53a1b3b2dfb3a/contracts/LBTC/LBTC.sol

References

lombard-finance/smart-contracts/contracts/pmm/BTCB/BTCB.sol#L87-L95

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
BTCBPMM.withdrawBTCB(uint256) (src/BTCB.sol#87-90) ignores return value by $.btcb.transfer($.withdrawAddress,amount) (src/BTCB.sol#89)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
BTCBPMM.swapBTCBToLBTC(uint256) (src/BTCB.sol#66-85) performs a multiplication on the result of a division:
        - amountLBTC = (amount / decimalsDifference) (src/BTCB.sol#73)
        - btcb.safeTransferFrom(_msgSender(),address(this),amountLBTC * decimalsDifference) (src/BTCB.sol#82)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
ILBTC (src/BTCB.sol#11-15) has incorrect ERC20 function interface:ILBTC.transfer(address,uint256) (src/BTCB.sol#13)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-erc20-interface
```

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Slither:src/FeeUtils.sol analyzed (2 contracts with 73 detectors), 9 result(s) found
```

```
INFO:Detectors:
TransparentUpgradeableProxy._fallback() (lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol#93-103) calls Proxy._fallback() (lib/openzeppelin-contracts/contracts/proxy/Proxy.sol#
58-60) which halt the execution return(uint256,uint256)(0,returndatasize()()) (lib/openzeppelin-contracts/contracts/proxy/Proxy.sol#42)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-return-in-assembly
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return value by Address.functionDelegateCall(newImplementation,data) (lib/openzeppeli
n-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores return value by Address.functionDelegateCall(IBeacon(newBeacon).implementatio
n(),data) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#178)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
TransparentUpgradeableProxy.constructor(address,address,bytes).initialOwner (lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol#77) lacks a zero-check on :
                - _admin = address(new ProxyAdmin(initialOwner)) (lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol#78)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.