Smart Contract Assessment (Updates)

Lombard

HALBORN

Report a Bug

Smart Contract Assessment (Updates) - Lombard

Prepared by: HALBORN Last Updated 12/10/2024 Date of Engagement by: December 6th, 2024 - December 9th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	1	2

1. Introduction

Lombard engaged Halborn to conduct a security assessment on their smart contracts beginning on December 6th, 2024 and ending on December 9th, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn was provided 2 days for the engagement and assigned a security engineer to evaluate 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 some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Lombard team`:

- `Implement a transferFrom for LBTC tokens when finalizing burning process.`
- `Standardize structs across all contracts.`

3. Test Approach And Methodology

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

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`,`draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

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
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: smart-contracts

(b) Assessed Commit ID: a82726b

(c) Items in scope:

- PartnerVault.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 17d5ac9
- aaa2a49

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

INFORMATIONAL

2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF LBTC TOKEN RECOVERY IN MANUAL BURN PROCESS	LOW	SOLVED - 12/09/2024
TYPE MISMATCH IN INTERFACE RETURN TYPES	INFORMATIONAL	SOLVED - 12/09/2024
CENTRALIZED ADMINISTRATIVE CONTROL	INFORMATIONAL	ACKNOWLEDGED - 12/10/2024

TABLE OF CONTENTS

7. FINDINGS & TECH DETAILS

7.1 LACK OF LBTC TOKEN RECOVERY IN MANUAL BURN PROCESS

// LOW

Description

In the `finalizeBurn` function, when `allowMintLbtc` is set to false, the contract is burning lockedFbtc (when calling `lockedFbtc.confirmRedeemFbtc()`) , then redeems FBTC tokens without ensuring the corresponding LBTC tokens are retrieved from the user :

```
159 // https://github.com/fbtc-com/fbtcX-contract/blob/main/src/LockedFB
160 function confirmRedeemFbtc(uint256 _amount) public onlyRole(MINTER_R
161         require(_amount > 0, "Amount must be greater than zero.");
162         require(fbtc.balanceOf(address(this)) >= _amount, "Insuffici
163 
164         _burn(msg.sender, _amount);
165         SafeERC20Upgradeable.safeTransfer(fbtc, msg.sender, _amount)
166 
167         emit ConfirmRedeemFbtc(msg.sender, _amount);
168     }
```

```
277 function finalizeBurn(
278     address recipient,
279     uint256 amount,
280     bytes32 depositTxId,
281     uint256 outputIndex
282 ) external nonReentrant whenNotPaused onlyRole(OPERATOR_ROLE) {
```

```
283         // ... state checks ...
284
285         // LBTC burn is skipped if allowMintLbtc is false
286         if ($.allowMintLbtc) $.lbtc.burn(recipient, amount);
287
288         $.lockedFbtc.confirmRedeemFbtc(amount);
289         $.fbtc.safeTransfer(recipient, amount);
290     }
```

So when `allowMintLbtc` is **false**, the `LBTC` tokens remain in circulation in the user wallet after `LockedFBTC` are burned from the **PartnerVault** contract and `FBTC` redemption. Nothing prevents the recipient to remove the allowance after that. This creates a situation where both `FBTC` and `LBTC` exist simultaneously for the same position. While this issue is mitigated by the `OPERATOR_ROLE` restriction, it represents an accounting inconsistency in the protocol's token lifecycle.

BVSS

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

Recommendation

It is recommended to add a LBTC token retrieval using `safeTransferFrom` when `allowMintLbtc` flag status is set to false:

```
function finalizeBurn(
    address recipient,
    uint256 amount,
    bytes32 depositTxId,
    uint256 outputIndex
) external nonReentrant whenNotPaused onlyRole(OPERATOR_ROLE) {
    // ... state checks ...
    if ($.allowMintLbtc) {
```

```
            $.lbtc.burn(recipient, amount);
        } else {
            $.lbtc.safeTransferFrom(recipient, address(this), amount);
        }
        $.lockedFbtc.confirmRedeemFbtc(amount);
        $.fbtc.safeTransfer(recipient, amount);
    }
```

Remediation

SOLVED: Recommended code has been added to Lombard smart contracts.

Remediation Hash

https://github.com/lombard-finance/smart-contracts/commit/17d5ac9c4e142655f4d9a0d28f9e7f0cee3dc7ac

References

lombard-finance/smart-contracts/contracts/fbtc/PartnerVault.sol#L277
fbtc-com/fbtcX-contract/src/LockedFBTC.sol#L160

7.2 TYPE MISMATCH IN INTERFACE RETURN TYPES

// INFORMATIONAL

Description

The contract `FBTCPartnerVault` defines an interface `LockedFBTC` that returns a struct of type `FBTCPartnerVault.Request`. However, LockedFBTC.sol also defines a request struct.

In `FBCTPartnerVault.sol`:

```
26  contract FBTCPartnerVault is PausableUpgradeable, ReentrancyGuardUpg
27  {
28      // .. //
29
30      struct Request {
31          Operation op;
32          Status status;
33          uint128 nonce;
34          bytes32 srcChain;
35          bytes srcAddress;
36          bytes32 dstChain;
37          bytes dstAddress;
38          uint256 amount;
39          uint256 fee;
40          bytes extra;
41      }
42
43      // ... //
44
45      function initializeBurn(...) {
46         // ... //
```

```
        (bytes32 hash, Request memory request) = $.lockedFbtc.redeemF
            amount,
            depositTxId,
            outputIndex
        );
        // ... //
    }
```

```
interface LockedFBTC {
    function mintLockedFbtcRequest(uint256 amount) external returns
    function redeemFbtcRequest(
        uint256 amount,
        bytes32 depositTxId,
        uint256 outputIndex
    ) external returns (bytes32, FBTCPartnerVault.Request memory);
    function confirmRedeemFbtc(uint256 amount) external;
}
```

In `LockedFBTCMock.sol`:

```
contract LockedFBTCMock {
    IERC20 public immutable fbtc;

    // ... //

    struct Request {
        Operation op;
        Status status;
        uint128 nonce;
        bytes32 srcChain;
```

```
21          bytes srcAddress;
22          bytes32 dstChain;
23          bytes dstAddress;
24          uint256 amount; // Transfer value without fee.
25          uint256 fee;
26          bytes extra;
27      }
28
29      function redeemFbtcRequest(
30          uint256 amount,
31          bytes32 depositTxId,
32          uint256 outputIndex
33      ) external pure returns (bytes32, Request memory) {
34          Request memory request = Request({
35              op: Operation.Nop,
36              status: Status.Unused,
37              nonce: 0,
38              srcChain: bytes32("test"),
39              srcAddress: bytes("test"),
40              dstChain: bytes32("test"),
41              dstAddress: bytes("test"),
42              amount: amount,
43              fee: 0,
44              extra: bytes("extra")
45          });
46
47          return (bytes32("test"), request);
48      }
```

And contract `common.sol` (out-of-scope) :

```
struct Request {
    Operation op;
    Status status;
    uint128 nonce; // Those can be packed into one slot in evm storage.
    bytes32 srcChain;
    bytes srcAddress;
    bytes32 dstChain;
    bytes dstAddress;
    uint256 amount; // Transfer value without fee.
    uint256 fee;
    bytes extra;
}
```

Struct request is declared 2 times in different parts of the code and not standardized across the contracts, which can lead to deployment problems in the future.

Score

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

Recommendation

It is recommended to define or reference a globally accessible struct type in a shared file or library. Interfaces and external contracts should return and accept only types known and accessible to all parties, ensuring seamless compilation and integration.

Remediation

SOLVED: Request struct is now imported instead of copied.

Remediation Hash

https://github.com/lombard-finance/smart-contracts/commit/aaa2a4936e0f0de4b78c78c535c61e4df5a95ec0

References

lombard-finance/smart-contracts/contracts/fbtc/PartnerVault.sol#L49
lombard-finance/smart-contracts/contracts/fbtc/PartnerVault.sol#L239

7.3 CENTRALIZED ADMINISTRATIVE CONTROL

// INFORMATIONAL

Description

The `PartnerVault` contract implements a centralized access control system through OpenZeppelin's AccessControl pattern. Critical protocol functionalities are restricted to specific roles with unilateral execution power.

Relevant code:

```
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
bytes32 public constant OPERATOR_ROLE = keccak256("OPERATOR_ROLE");
```

```
function initialize (
    address admin,
    address fbtc_,
    address lbtc_,
    uint256 stakeLimit_
) external initializer {
    __Pausable_init();
    __ReentrancyGuard_init();
    __AccessControl_init();
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
}
```

Key privileged functions include:

```
function setLockedFbtc(address lockedFbtc_) external onlyRole(DEFAULT_ADM
function setAllowMintLbtc(bool shouldMint) external onlyRole(DEFAULT_ADMI
function setStakeLimit(uint256 newStakeLimit) external onlyRole(OPERATOR_
function pause() external onlyRole(PAUSER_ROLE)
function unpause() external onlyRole(DEFAULT_ADMIN_ROLE)
function removeWithdrawalRequest(...) external onlyRole(OPERATOR_ROLE)
function finalizeBurn(...) external onlyRole(OPERATOR_ROLE)
function initializeBurn(...) external onlyRole(OPERATOR_ROLE)
```

Score

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

Recommendation

It is recommended to add a time-delayed, multi-signature governance structure.

Remediation

ACKNOWLEDGED: A multi sig will be used for the contract handling.

References

lombard-finance/smart-contracts/contracts/fbtc/PartnerVault.sol#L26

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Slither`, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the 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.

```
➜  smart-contracts git:(fbtc) ✗ slither . --include-paths ./contracts/fbtc/PartnerVault.sol
'npx hardhat clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/smart-contracts)
'npx hardhat clean --global' running (wd: /Users/liliancariou/Desktop/Halborn/audits/smart-contracts)
'npx hardhat compile --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/smart-contracts)
INFO:Detectors:
FBTCPartnerVault.mint(uint256) (contracts/fbtc/PartnerVault.sol#151-178) ignores return value by $.fbtc.approve(address($.lockedFbtc),amount) (contracts/fbtc/PartnerVault.sol#162
)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
FBTCPartnerVault._getPartnerVaultStorage() (contracts/fbtc/PartnerVault.sol#285-290) uses assembly
        - INLINE ASM (contracts/fbtc/PartnerVault.sol#287-289)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity are used:
        - Version used: ['0.8.24', '>=0.8.0', '>=0.8.0<0.9.0', '^0.8.0', '^0.8.1', '^0.8.19', '^0.8.20', '^0.8.22', '^0.8.4']
        - 0.8.24 (node_modules/@chainlink/contracts-ccip/src/v0.8/ccip/pools/TokenPool.sol#2)
        - 0.8.24 (node_modules/@chainlink/contracts-ccip/src/v0.8/ccip/test/mocks/MockRMN.sol#2)
        - 0.8.24 (contracts/LBTC/ILBTC.sol#2)
        - 0.8.24 (contracts/LBTC/LBTC.sol#2)
        - 0.8.24 (contracts/PoR/IPoR.sol#2)
        - 0.8.24 (contracts/PoR/PoR.sol#2)
        - 0.8.24 (contracts/bascule/Bascule.sol#3)
        - 0.8.24 (contracts/bascule/BasculeV2.sol#3)
        - 0.8.24 (contracts/bascule/interfaces/IBascule.sol#3)
        - 0.8.24 (contracts/bridge/Bridge.sol#2)
        - 0.8.24 (contracts/bridge/IBridge.sol#2)
        - 0.8.24 (contracts/bridge/adapters/AbstractAdapter.sol#2)
        - 0.8.24 (contracts/bridge/adapters/CLAdapter.sol#2)
        - 0.8.24 (contracts/bridge/adapters/IAdapter.sol#2)
        - 0.8.24 (contracts/bridge/adapters/TokenPool.sol#2)
        - 0.8.24 (contracts/bridge/oft/EfficientRateLimitedOFTAdapter.sol#2)
        - 0.8.24 (contracts/bridge/oft/EfficientRateLimiter.sol#2)
        - 0.8.24 (contracts/bridge/oft/LBTCBurnMintOFTAdapter.sol#2)
        - 0.8.24 (contracts/bridge/oft/LBTCOFTAdapter.sol#2)
        - 0.8.24 (contracts/consortium/Consortium.sol#2)
        - 0.8.24 (contracts/consortium/ILombardTimelockController.sol#4)
        - 0.8.24 (contracts/consortium/INotaryConsortium.sol#2)
        - 0.8.24 (contracts/consortium/LombardTimeLock.sol#2)
        - 0.8.24 (contracts/factory/ProxyFactory.sol#2)
        - 0.8.24 (contracts/fbtc/PartnerVault.sol#2)
        - 0.8.24 (contracts/interfaces/IConsortiumConsumer.sol#2)
        - 0.8.24 (contracts/libs/Actions.sol#2)
```

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.

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.