// Security Assessment

08.28.2025 - 09.04.2025

BTCFi Lending Vault

Coredao

HALBORN

BTCFi Lending Vault - Coredao

Prepared by: HALBORN

Last Updated 09/16/2025

Date of Engagement: August 28th, 2025 - September 4th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Introduction

CoreDAO engaged Halborn to conduct a security assessment of **the BTCFi Lending Vault contracts developed by the** b14g team from August 28th to September 4th, 2025. The scope of this assessment was limited to the smart contracts provided to the Halborn team. Commit hashes and additional details are documented in the Scope section of this report.

2. Assessment Summary

The Halborn team dedicated a total of five days to this engagement, deploying one full-time security engineer to evaluate the smart contracts' security posture.

The assigned security engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing, smart contract exploitation, and a comprehensive understanding of multiple blockchain protocols.

The objectives of this assessment were to:

- Verify that the smart contract functions operate as intended.
- Identify potential security vulnerabilities within the smart contracts.

In summary, Halborn identified several areas for improvement to reduce both the likelihood and impact of potential risks, which were partially addressed by the b14g team. The primary suggestions included:

- Implement comprehensive oracle data validation with staleness and confidence checks before using prices in calculations.
- Establish proper historical reward state tracking to prevent exploitation of uninitialized mapping slots.
- Add health factor validation after pool withdrawals to prevent liquidation risk.
- Implement proper error handling for external contract calls with try-catch blocks.
- Add event emissions for all administrative parameter changes to enhance transparency.
- Make critical addresses configurable instead of hardcoded to support network flexibility.

3. Test Approach And Methodology

Halborn employed a combination of manual, semi-automated, and automated security testing methods to ensure effectiveness, efficiency, and accuracy within the scope of this assessment. Manual testing was vital for uncovering issues related to logic, processes, and implementation details, while automated techniques enhanced code coverage and helped identify deviations from security best practices. The assessment involved the following phases and tools:

- Research into the architecture and purpose of the smart contracts.
- Manual review and walkthrough of the smart contract code.
- Manual evaluation of critical Solidity variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static security analysis of the scoped contracts and imported functions utilizing `Slither`.
- Local deployment and testing with `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:

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

EXPLOITABILITY 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

REPOSITORY

(a) Repository: BTCFiLendingVaultContract

(b) Assessed Commit ID: 8fb8301

(c) Items in scope:

- contracts/utils/WBTCSwap.sol
- contracts/marketplace/LendingVaultV2.sol
- contracts/marketplace/MergeMarketplaceStrategyV2.sol

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

REMEDIATION COMMIT ID:

- e2a2da3
- 05c7f27
- a9f2e4b

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	1	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
STALE ORACLE DATA ENABLES SLIPPAGE PROTECTION BYPASS	MEDIUM	SOLVED - 09/05/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HISTORICAL DATA LOSS ENABLES REWARD THEFT	MEDIUM	SOLVED - 09/05/2025
POOL WITHDRAWAL WITHOUT HEALTH FACTOR VALIDATION	MEDIUM	SOLVED - 09/05/2025
UNCHECKED EXTERNAL CALL RETURN VALUES	LOW	RISK ACCEPTED - 09/10/2025
OPERATOR ROLE CENTRALIZATION RISK	INFORMATIONAL	SOLVED - 09/10/2025
SILENT CHANGES REDUCE TRANSPARENCY	INFORMATIONAL	ACKNOWLEDGED - 09/10/2025
HARDCODED NETWORK ADDRESSES REDUCE FLEXIBILITY	INFORMATIONAL	ACKNOWLEDGED - 09/10/2025

7. FINDINGS & TECH DETAILS

7.1 STALE ORACLE DATA ENABLES SLIPPAGE PROTECTION BYPASS

// MEDIUM

Description

The `WBTCSwap::swap()` function calculates slippage protection using Pyth oracle prices without proper validation.

```
54 | uint256 amountOutMinimum = (msg.value * corePrice * (10000 - slippage)) / (btcPrice * 10000 * 1e10)
```

The function fetches EMA prices but doesn't verify price staleness or confidence intervals.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

Add comprehensive oracle data validation before using prices for calculations:

```
0 | require(block.timestamp - coreData.publishTime <= MAX_PRICE_AGE, "Stale price");
1 | require(coreData.conf <= MAX_CONFIDENCE_INTERVAL, "Price confidence too low");
2 | require(corePrice > 0 && btcPrice > 0, "Invalid oracle price");
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/b14glabs/BTCFiLendingVaultContract/commit/e2a2da3fb2d17f6761164d2009715816b23e24fb

7.2 HISTORICAL DATA LOSS ENABLES REWARD THEFT

// MEDIUM

Description

In `LendingVaultV2::captureReward()`, when operators skip multiple rounds and current round has zero rewards, the fallback logic copies from uninitialized mapping slots instead of finding the last valid reward state.

```
196 | if (accPerShareLog[lastRoundClaim] == 0) {
197 |     accPerShareLog[lastRoundClaim] = accPerShareLog[lastRoundClaim - 1];
198 | }
```

Attack Scenario:

- Operator skips rounds 101-104, leaving `accPerShareLog[104] = 0`
- Attacker observes this state and stakes a large amount when `accPerShareLog[lastRoundClaim] = 0`
- Next round (105) generates rewards, setting `accPerShareLog[105] = X`
- Attacker receives reward calculation of `(X - 0) * large_stake`, effectively claiming rewards from round 0 despite only staking one round
- Legitimate long-term stakers receive diluted rewards due to the attacker's disproportionate share

This enables attackers to manipulate reward calculations by staking during zero-state periods.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (5.0)

Recommendation

Implement state tracking to maintain historical continuity across consecutive zero-reward rounds:

```
0  | uint256 public lastValidAccPerShare;
1  |
2  | function captureReward(address swapHelper) external onlyOperator whenNotPaused {
3  |     // ... existing reward claiming logic ...
4  |
5  |     if (totalScaledBalance > 0) {
6  |         if (reward > 0) {
7  |             // Case 1: Normal rewards - calculate new accumulation
8  |             uint256 rewardPerShare = (postScaledBalance - preScaledBalance) * 1 ether / totalScaled
9  |             lastValidAccPerShare += rewardPerShare;
10 |             accPerShareLog[lastRoundClaim] = lastValidAccPerShare;
11 |         } else {
12 |             // Case 2: Zero rewards - inherit last valid state
13 |             accPerShareLog[lastRoundClaim] = lastValidAccPerShare;
14 |         }
15 |     }
16 | }
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/b14glabs/BTCFiLendingVaultContract/commit/05c7f274c6fb4ee8f197cd86799e81c7089a54ee

7.3 POOL WITHDRAWAL WITHOUT HEALTH FACTOR VALIDATION

// MEDIUM

Description

`LendingVaultV2::claimRevenue()` performs pool withdrawals to cover revenue claims without validating the vault's health factor afterwards.

```
359 | pool.withdraw(address(stakeToken), withdrawAmount, address(this));
```

This can lead to liquidation risk as withdrawing collateral reduces the vault's health factor below safe thresholds.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (5.0)

Recommendation

Add health factor validation after pool withdrawal:

```
0 | pool.withdraw(address(stakeToken), withdrawAmount, address(this));
1 | (,,,,, uint256 currentHealthFactor) = pool.getUserAccountData(address(this));
2 | require(currentHealthFactor >= MINIMUM_HEALTH_FACTOR, "Health factor too low");
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/b14glabs/BTCFiLendingVaultContract/commit/a9f2e4b61014b037bebdbbbe349904cb1c6a3bdc

7.4 UNCHECKED EXTERNAL CALL RETURN VALUES

// LOW

Description

`LendingVaultV2::withdraw()` and `MergeMarketplaceStrategyV2::reInvest()` make external calls that don't check return values or rely on implicit revert behavior, which may not work consistently across all implementations.

File: `MergeMarketplaceStrategyV2`

```
27 | IMarketplace(marketplace).withdrawCoreProxy(withdrawData);
28 | IMarketplace(marketplace).stakeCoreProxy{value: stakeValue}(stakeData);
```

File: `LendingVaultV2`

```
298 | pool.withdraw(address(stakeToken), amount, address(this));
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (3.4)

Recommendation

Wrap external calls in try-catch blocks or verify that the called contracts properly revert on failure.

Remediation Comment

RISK ACCEPTED: The **b14g team** accepted the risk related to this finding.

7.5 OPERATOR ROLE CENTRALIZATION RISK

// INFORMATIONAL

Description

The `LendingVaultV2` contract grants excessive privileges to a single operator address without multi-signature requirements or timelock mechanisms. The operator can perform critical operations like borrowing funds (`lendingInvest`), investing in strategies (`coreInvest`), and claiming rewards (`captureReward`) without additional oversight, creating centralization risks.

Risk Scenarios:

- Compromised Operator: Single point of failure if operator key is compromised
- Malicious Operator: Could drain vault through excessive borrowing or malicious strategy investments
- Operational Risk: No backup mechanism if operator becomes unavailable

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:H (1.9)

Recommendation

Implement multi-signature requirements or timelock mechanisms for critical operator functions.

Remediation Comment

SOLVED: The **b14g team** agreed to use multisig for operator role.

7.6 SILENT CHANGES REDUCE TRANSPARENCY

// INFORMATIONAL

Description

Admin functions in `WBTCSwap::setSlippage()` and `WBTCSwap::setDeadline()` don't emit events for parameter changes, reducing transparency and making it difficult to track configuration updates.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Add event emissions to all admin parameter change functions for transparency.

Remediation Comment

ACKNOWLEDGED: This finding was acknowledged.

7.7 HARDCODED NETWORK ADDRESSES REDUCE FLEXIBILITY

// INFORMATIONAL

Description

`WBTCSwap`, `LendingVaultV2`, and `MergeMarketplaceStrategyV2` use hardcoded addresses for tokens, routers, and oracles, making them inflexible for network upgrades, migrations, or multi-chain deployments.

```
16 | address public constant WBTC = 0x5832f53d147b3d6Cd4578B9CBD62425C7ea9d0Bd;
17 | address public constant WCORE = 0x191E94fa59739e188dcE837F7f6978d84727AD01;
```

BVSS

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

Recommendation

Make critical addresses configurable through admin functions with appropriate access controls and validation checks:

```
0 | function setTokenAddresses(address _wbtc, address _wcore) external onlyOwner {
1 |     require(_wbtc != address(0) && _wcore != address(0), "Invalid addresses");
2 |     WBTC = _wbtc;
3 |     WCORE = _wcore;
4 |     emit AddressesUpdated(_wbtc, _wcore);
5 | }
```

Remediation Comment

ACKNOWLEDGED: This finding was acknowledged.

8. AUTOMATED TESTING

Halborn utilized automated testing techniques to improve coverage of specific areas within the smart contracts under review. One of the primary tools employed was `Slither`, a static analysis framework for Solidity. After successfully verifying and compiling the smart contracts in the repository into their ABI and binary formats, `Slither` was executed against the contracts. This tool performs static verification of mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs throughout the entire codebase.

The security team conducted a comprehensive review of the findings generated by the `Slither` static analysis tool. No significant issues were identified, as the reported findings were determined to be false positives.

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.