// Smart Contract Security Assessment

12.02.2024 - 12.03.2024

Thunderhead - stMOVE Contracts

Movement Labs

HALBORN

Thunderhead - stMOVE Contracts - Movement Labs

Prepared by: HALBORN

Last Updated 12/04/2024

Date of Engagement by: December 2nd, 2024 - December 3rd, 2024

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	1	2	2

TABLE OF CONTENTS

1. Introduction

Thunderhead engaged Halborn to conduct a security assessment on their smart contracts beginning on December 2nd and ending on December 3rd, 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 assigned a full-time security engineer to assess the security of the smart contracts. 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 should be addressed by the Thunderhead team. The main ones were the following:

- When updating rates, ensure that the proposed rate is greater than or equal to current one at the time of the transaction.
- Remove or deprecate the approve function and instead implement increaseAllowance and decreaseAllowance.
- Introduce a similar validation mechanism in the rebaseByShareRate function as is done in rebaseByApr.

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

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
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: stmove-contracts-eth

(b) Assessed Commit ID: b3df842

(c) Items in scope:

- src/token/fstMOVE.sol
- src/Lock.sol

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

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	2

INFORMATIONAL
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-02 - INADEQUATE SHARE RATE UPDATE VALIDATION	MEDIUM	-
HAL-03 - RACE CONDITION IN ALLOWANCE UPDATES WITH APPROVE	LOW	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-04 - POTENTIAL INCONSISTENT VALIDATION RULES FOR SHARE RATE MODIFICATIONS	LOW	-
HAL-05 - SHARE RATE EVOLUTION ALLOWS FOR MORE BURNING THAN THE ASSETS DEPOSITED	INFORMATIONAL	-
HAL-01 - FSTMOVE DESTRUCTED NOT FULLY IMPLEMENTED	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-02) INADEQUATE SHARE RATE UPDATE VALIDATION

// MEDIUM

Description

The `rebaseByShareRate` function in the **fstMove** contract is currently validating that `nextShareRate_` is not less than `lastShareRate`. However, this check might not prevent scenarios where `nextShareRate_` could still be less than the current `shareRate()` at the time of the update. This can result in the `nextShareRate` being set lower than the most recent `shareRate()`, potentially leading to inaccurate rebase calculations and impacting the financial integrity of the system.

Code Location

The `rebaseByShareRate` function is validating that `nextShareRate_` is not less than `lastShareRate`:

```
    function rebaseByShareRate(uint256 nextShareRate_, uint256 upd
        if (nextShareRate_ < lastShareRate) revert NegativeRebaseN
        if (updateEnd_ < block.timestamp) revert UpdateMustBeInFut

        lastShareRate = shareRate();
        updateStart = block.timestamp;

        updateEnd = updateEnd_;
        nextShareRate = nextShareRate_;

        emit Rebase(nextShareRate_, updateEnd_);
    }
```

BVSS

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

Recommendation

It is recommended to modify the validation within the `rebaseByShareRate` function. Specifically, the condition should be updated to ensure that `nextShareRate_` is greater than or equal to `shareRate()` at the time of the transaction.

7.2 (HAL-03) RACE CONDITION IN ALLOWANCE UPDATES WITH APPROVE

// LOW

Description

The **fstMOVE** contract includes the standard ERC-20 `approve()` function, which allows a user to set a spender's allowance. However, this function can lead to a well-known race condition vulnerability if a user updates an existing allowance without first setting it to zero. This allows a spender to exploit the window between the old and new approval to transfer tokens using the old higher allowance, potentially leading to unauthorized transfers.

Code Location

The `approve()` function can lead to a race condition vulnerability if a user updates an existing allowance without first setting it to zero:

```
    function approve(address spender, uint256 value) public virtu
        if (!_whitelisted[spender]) {
            revert NotWhitelisted();
        }

        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:M/Y:N (4.2)

Recommendation

It is recommended to either remove or deprecate the `approve()` function and instead implement `increaseAllowance()` and `decreaseAllowance()` functions.

7.3 (HAL-04) POTENTIAL INCONSISTENT VALIDATION RULES FOR SHARE RATE MODIFICATIONS

// LOW

Description

The rebaseByShareRate function in the **fstMove** contract does not include a verification step to ensure that the new share rate (nextShareRate_) adheres to a defined threshold, unlike the rebaseByApr function which checks that the annual percentage rate (APR) does not exceed the maxAprThreshold. This inconsistency might allow the setting of a nextShareRate_ that could be unexpectedly high without any boundary check, potentially leading to unintended financial implications in the contract's operations.

Code Location

The rebaseByShareRate function does not include a verification step to ensure that nextShareRate_ adheres to a defined threshold:

```
    function rebaseByShareRate(uint256 nextShareRate_, uint256 upd
        if (nextShareRate_ < lastShareRate) revert NegativeRebaseN
        if (updateEnd_ < block.timestamp) revert UpdateMustBeInFut

        lastShareRate = shareRate();
        updateStart = block.timestamp;

        updateEnd = updateEnd_;
        nextShareRate = nextShareRate_;

        emit Rebase(nextShareRate_, updateEnd_);
    }
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:M/Y:N (2.1)

Recommendation

It is recommended to introduce a similar validation mechanism in the rebaseByShareRate function as is done in rebaseByApr. This could involve defining a maximum threshold for the share rate increase or ensuring that the rate changes remain within certain limits to maintain consistency and prevent extreme modifications.

7.4 (HAL-05) SHARE RATE EVOLUTION ALLOWS FOR MORE BURNING THAN THE ASSETS DEPOSITED

// INFORMATIONAL

Description

Since the `shareRate` in the **fstMOVE** contract is increasing, burning the quantity of assets that the user deposited will not result in burning the full shares. Some would be left, possibly introducing unwanted behaviours like redeeming more than entitled in the **Lock** contract. As a result, all users trying to withdraw at the same time would cause a lack of liquidity that will leave some users unable to redeem their deposit.

Based on feedback from the **Thunderhead team**, additional MOVE will be deposited into the contract prior to enabling redemptions to mitigate any liquidity issues. However, it is important to note that verifying the implementation or effectiveness of this mechanism is outside the scope of this audit.

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:C/Y:M (1.5)

Recommendation

It is recommended to review the intention behind the rate, mint, and burn functions and make sure that users can only redeem what they are entitled to.

7.5 (HAL-01) FSTMOVE DESTRUCTED NOT FULLY IMPLEMENTED

// INFORMATIONAL

Description

The FSTMove contract is an ERC20 that has the particularity of implementing a `destructed` feature. The contract keeps a `destructed` variable and will return 0 from `balanceOf` if the variable is set to `true`.

It was found that the contract in a `destructed` state would still be able to execute functions such as `transferFrom`, `mint` and `burn`, going against the idea of a destructed contract.

Code Location

Example of destructed usage in the `balanceOf` function:

```
function balanceOf(address account) public view virtual returns (u
    if (destructed) {
        return 0;
    }

    return _shares[account] * shareRate() / BASE;
}
```

Score

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

Recommendation

It is recommended to disallow transfers, mints and burns by adding a `destructed` check in the `_update` function that is called by all affected functions.

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.