// HALBORN

Moonwell Finance - Compound Vault

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 16th, 2023 - August 24th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/19/2023	Gokberk Gulgun
1.0	Remediation Plan	08/23/2023	Gokberk Gulgun
1.1	Remediation Plan Review	08/24/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Moonwell Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on August 16th, 2023 and ending on August 24th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided four weeks for the engagement and assigned a full-time security engineer to verify 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 security risks that were mostly addressed by the Moonwell Finance team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can 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)
- 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.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Foundry)

2. 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.

2.1 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
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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

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

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following contract:

- moonwell-contracts-v2

ASSESSMENTS :

1. ASSESSED PULL REQUEST:

- moonwell-contracts-v2/pull/19

COMMIT ID : 6c4ea1e30c89632f9e0ad5c3b9dd3a505a101854

- src/4626/CompoundERC4626.sol.

2. ASSESSED COMMIT ID:

COMMIT ID : 00a00f340aa5ba636501740322607b972b0921db

- src/4626/CompoundERC4626.sol.

REMEDIATION COMMIT IDs :

- 9239b4fabfbfc25fc46494ca02aaf1a0cdc160d1
- ae87166ce0634f05da0a6edd879d9c7a4c74b1e3
- b8062797ea74907c260af47b1321a8a3987b9393
- 00a00f340aa5ba636501740322607b972b0921db

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	1	1	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) COMPOUND VAULT IS VULNERABLE TO INFLATION ATTACK	High (8.8)	SOLVED - 08/24/2023
(HAL-02) INCORRECT USE OF BORROW CAP INSTEAD OF SUPPLY CAP IN MAXMINT FUNCTION	Medium (6.2)	SOLVED - 08/24/2023
(HAL-03) LACK OF TOKEN CONTRACT EXISTENCE CHECK IN SOLMATES SAFETRANSFERLIB	Low (3.8)	SOLVED - 08/24/2023
(HAL-04) ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE	Informational (1.7)	ACKNOWLEDGED
(HAL-05) CHANGE FUNCTION VISIBILITY FROM PUBLIC TO EXTERNAL	Informational (1.7)	SOLVED - 08/24/2023
(HAL-06) WELL TOKEN IS NOT UTILIZED ON THE COMPOUNDERC4626.SOL CONTRACT	Informational (1.7)	SOLVED - 08/24/2023
(HAL-07) CALLDATA IS CHEAPER THAN MEMORY	Informational (1.7)	SOLVED - 08/24/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) COMPOUND VAULT IS VULNERABLE TO INFLATION ATTACK - HIGH (8.8)

Description:

CompoundERC4626 contract follows the EIP4626 standard

This extension allows the minting and burning of **shares** (represented using the ERC20 inheritance) in exchange for underlying **assets** through standardized deposit, mint, redeem and burn workflows. But this extension also has the following problem:

When the vault is empty or nearly empty, deposits are at high risk of being stolen through front-running by inflating the share-token value through burning obtained shares. This is variously known as a donation or inflation attack and is essentially a problem of slippage.

Therefore, this issue could affect the users using the protocol that run the risk of losing a part of their deposited tokens.

Code Location:

CompoundERC4626.sol#L16C1-L16C38

Listing 1

```
1 contract CompoundERC4626 is ERC4626 {}
```

Proof Of Concept:

Step 1 : A malicious early user can deposit() with 1 wei of asset token as the first depositor of the Vault, a get 1 wei of shares token.
Step 2 : Then the attacker can send 10000e18 - 1 of Mtokens and inflate the price per share from 1,000 to an extreme value of 1.0000e22 (from

(1+10000e18 - 1)/1).

Step 3 : As a result, the future user who deposits 19999e18 will immediately lose their deposits.

BVSS:

AO:A/AC:L/AX:L/C:M/I:N/A:N/D:H/Y:N/R:N/S:U (8.8)

Recommendation:

Consider requiring a minimal amount of share tokens to be minted for the first minter, and send a part of the initial mint as a permanent reserve so that the price per share can be more resistant to manipulation.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by requiring a minimal amount of share tokens in the deployment script.

Commit ID: b8062797ea74907c260af47b1321a8a3987b9393

4.2 (HAL-02) INCORRECT USE OF BORROW CAP INSTEAD OF SUPPLY CAP IN MAXMINT FUNCTION - MEDIUM (6.2)

Description:

The maxMint function is currently designed to use the borrowCap for determining the maximum amount that can be minted, which is inconsistent with the expected behavior. Ideally, the function should be using the supplyCap to calculate this limit. This inconsistency could lead to incorrect calculations and potential imbalances in the system.

Code Location:

CompoundERC4626.sol#L159

Listing 2

```
    function maxMint(address) public view override returns (
↳ uint256) {
        if (comptroller.mintGuardianPaused(address(mToken))) {
            return 0;
        }

        uint256 borrowCap = comptroller.borrowCaps(address(mToken)
↳ );
        if (borrowCap != 0) {
            uint256 totalBorrows = mToken.totalBorrows();
            return borrowCap - totalBorrows;
        }

        return type(uint256).max;
    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:M/Y:N/R:N/S:U (6.2)

Recommendation:

Replace the use of borrowCap with supplyCap in the maxMint function to ensure accurate calculations for the maximum mintable amount.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by changing borrowCap with supplyCap.

Commit ID: ae87166ce0634f05da0a6edd879d9c7a4c74b1e3

4.3 (HAL-03) LACK OF TOKEN CONTRACT EXISTENCE CHECK IN SOLMATES SAFETRANSFERLIB - LOW (3.8)

Description:

Solmate's SafeTransferLib, which is used for transferring tokens, currently does not verify the existence of a token contract or whether the token address is the zero-address. The library explicitly states that it does not check if a token has any code, delegating that responsibility to the caller. As a result, if the token address is empty, the transfer operation will appear to succeed without actually crediting any tokens to the contract.

Code Location:

CompoundERC4626.sol#L4

Listing 3

```
1 import {SafeTransferLib} from "solmate/utils/SafeTransferLib.sol";
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:N/R:N/S:U (3.8)

Recommendation:

Consider switching to OpenZeppelin's SafeERC20 library, which includes built-in checks to verify that an address contains code. This would eliminate the need for manual checks like ensuring the address is not the zero-address or verifying that `code.length > 0`.

Remediation Plan:

SOLVED: The Moonwell Finance team resolved the issue by ensuring that the deployment script checks the underlying token.

4.4 (HAL-04) ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE - INFORMATIONAL (1.7)

Description:

The scoped repositories make use of ERC4626 custom implementations that should follow the EIP-4626 definitions. This standard states the following security consideration:

"If implementors intend to support EOA account access directly, they should consider adding another function call for deposit/mint/withdraw /redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved."

These vault implementations do not implement a way to limit the slippage when deposits/withdraws are performed. This condition affects specially to EOA since they don't have a way to verify the amount of tokens received and revert the transaction in case they are too few compared to what was expected to be received.

Applying this security consideration would help to EOA to avoid being front-run and losing tokens in transactions towards these smart contracts.

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (1.7)

Recommendation:

It is recommended to include slippage checks in the aforementioned functions to allow EOA to set the minimum amount of tokens that they expect to receive by executing these functions.

References:

- EIP-4626: Security Considerations

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged this finding.

4.5 (HAL-05) CHANGE FUNCTION VISIBILITY FROM PUBLIC TO EXTERNAL - INFORMATIONAL (1.7)

Description:

The function claimReward() has been declared as public, but it is never called internally within the contract. It is best practice to mark such functions as external instead, as this can save gas. In cases where the function takes arguments, external functions can read the arguments directly from calldata instead of having to allocate memory.

Code Location:

CompoundERC4626.sol#L82

Listing 4

```
    /// @notice Claims liquidity mining rewards from Compound and sends it to rewardRecipient
    function claimRewards() public {
        address[] memory holders = new address[](1);
        holders[0] = address(this);

        MToken[] memory mTokens = new MToken[](1);
        mTokens[0] = MToken(address(mToken));

        comptroller.claimReward(holders, mTokens, false, true);

        uint256 amount = well.balanceOf(address(this));
        well.safeTransfer(rewardRecipient, amount);

        emit ClaimRewards(amount, address(well));
    }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (1.7)

Recommendation:

It's recommended to change the function visibility from `public` to `external` in the `claimReward` function.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by changing the function visibility.

Commit ID: b8062797ea74907c260af47b1321a8a3987b9393

4.6 (HAL-06) WELL TOKEN IS NOT UTILIZED ON THE COMPOUNDERC4626.SOL CONTRACT - INFORMATIONAL (1.7)

Description:

In the CompoundERC4626.sol contract, the Well token is defined in the constructor, but it is not used. The unused variables should be deleted from the contract.

Code Location:

CompoundERC4626.sol#L50

Listing 5

```
1 ERC20 public immutable well;
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (1.7)

Recommendation:

Consider removing redundant variables.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by removing redundant variables.

Commit ID: ae87166ce0634f05da0a6edd879d9c7a4c74b1e3

4.7 (HAL-07) CALLDATA IS CHEAPER THAN MEMORY - INFORMATIONAL (1.7)

Description:

When a function with a memory array is called externally, the **abi.decode()** step has to use a for-loop to copy each index of the **calldata** to the memory index. Each iteration of this for-loop costs at least 60 gas (i.e. 60 * .length). Using **calldata** directly, obviates the need for such a loop in the contract code and runtime execution.

If the array is passed to an internal function which passes the array to another internal function where the array is modified and therefore memory is used in the external call, it's still more gas-efficient to use **calldata** when the external function uses modifiers, since the modifiers may prevent the internal functions from being called. Some gas savings if function arguments are passed as **calldata** instead of memory. Note that in older Solidity versions, changing some function arguments from memory to calldata may cause "unimplemented feature error". This can be avoided by using a newer (0.8.*) Solidity compiler.

Code Location:

CompoundERC4626.sol#L117

Listing 6

```
    /// @notice Claims liquidity mining rewards from Compound and
↳ sends it to rewardRecipient
    /// used for edgecase where reward distribution is not yet
↳ configured or was removed
    /// the tokens were swept into the vault.
    /// @param tokens The list of tokens to sweep
    function sweepRewards(address[] calldata tokens) external {
        for (uint256 i = 0; i < tokens.length; i++) {
            ERC20 token = ERC20(tokens[i]);
            uint256 amount = token.balanceOf(address(this));
            token.safeTransfer(rewardRecipient, amount);
```

```
10              emit ClaimRewards(amount, address(token));
11          }
12      }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (1.7)

Recommendation:

Use **calldata** in the function.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by changing memory with calldata.

Commit ID: ae87166ce0634f05da0a6edd879d9c7a4c74b1e3

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

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.

Results:

- No major issues found by Slither.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

- No major issues were found by MythX.

THANK YOU FOR CHOOSING

// HALBORN