Moonwell Finance - Contracts V2 Updates

Smart Contract Security Assessment

Prepared by: **Halborn**

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

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/19/2023	Gokberk Gulgun
0.2	Document Updates	07/23/2023	Gokberk Gulgun
1.0	Remediation Plan	07/23/2023	Gokberk Gulgun
1.1	Remediation Plan Review	07/24/2023	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 July 16th, 2023 and ending on July 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 two 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, Brownie)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following contract:

- moonwell-contracts-v2

Commit ID:

- c39f98bdc9dd4e448ba585923034af1d47f74dfa

IN-SCOPE CONTRACTS :

- MultiRewardDistributor.sol.
- WETHRouter.sol.
- TemporalGovernor.sol.
- ChainlinkCompositeOracle.sol.
- ChainlinkOracle.sol.

2. REMEDIATION COMMIT ID :

- 17fce574c46259cb22b8b6215b8b982169eb40e7

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	1	2	8

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) SILENT FAILURE DURING TOKEN MINTING ON THE ROUTER CONTRACT	High (8.2)	SOLVED - 07/23/2023
(HAL-02) SILENT FAILURE DURING TOKEN REDEMPTION ON THE ROUTER CONTRACT	High (8.2)	SOLVED - 07/23/2023
(HAL-03) MISSING CHAIN ID AND RECEIVER ADDRESS VERIFICATION IN EXECUTEPROPOSAL() FUNCTION	Medium (5.9)	SOLVED - 07/23/2023
(HAL-04) UNRESTRICTED RECEIVE IN WETHROUTER ENABLES EXCESS REDEMPTIONS	Low (3.1)	RISK ACCEPTED
(HAL-05) IMPLEMENTATIONS CAN BE INITIALIZED	Low (2.5)	SOLVED - 07/20/2023
(HAL-06) FLOATING PRAGMA	Informational (0.0)	SOLVED - 07/24/2023
(HAL-07) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS	Informational (0.0)	ACKNOWLEDGED
(HAL-08) INCREMENT/DECREMENT FOR LOOP VARIABLE IN AN UNCHECKED BLOCK	Informational (0.0)	ACKNOWLEDGED
(HAL-09) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN	Informational (0.0)	ACKNOWLEDGED
(HAL-10) CACHE ARRAY LENGTH	Informational (0.0)	ACKNOWLEDGED
(HAL-11) REDUNDANT SAFE CAST	Informational (0.0)	SOLVED - 07/24/2023
(HAL-12) REVERT STRING SIZE OPTIMIZATION	Informational (0.0)	ACKNOWLEDGED
(HAL-13) NO NEED TO INITIALIZE VARIABLES WITH DEFAULT VALUES	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) SILENT FAILURE DURING TOKEN MINTING ON THE ROUTER CONTRACT - HIGH (8.2)

Description:

The `mToken.mint(msg.value);` function, originating from Compound's ERC20 `mToken` contracts, is a call that does not revert on failure but returns an error code as a `uint` value instead. This behavior deviates from the standard expected of typical Solidity functions that revert on failure.

This non-standard behavior makes it difficult for calling contracts (like the one above) to correctly handle failures. As the above contract does not check the return value of `mToken.mint()`, failures in this function will not cause the overall transaction to revert.

This could lead to serious imbalances between the perceived balance of `mTokens` on the router contract and the actual supply of minted `mTokens`.

Code Location:

Listing 1

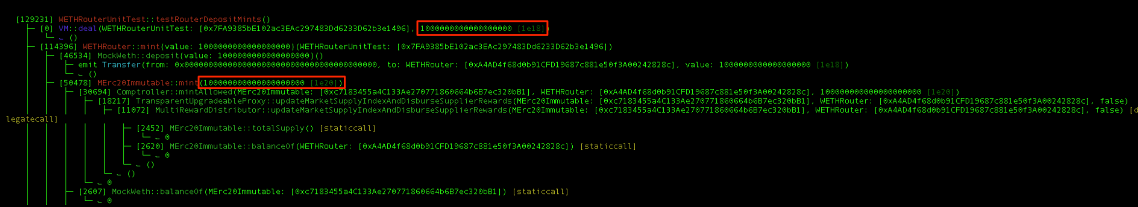
```
/// @notice Deposit ETH into the Moonwell protocol
/// @param recipient The address to receive the mToken
function mint(address recipient) external payable {
    weth.deposit{value: msg.value}();

    mToken.mint(msg.value);

    IERC20(address(mToken)).safeTransfer(
        recipient,
        mToken.balanceOf(address(this))
    );
}
```

Proof Of Concept:

Step 1 : An external actor calls the `mint()` function, sending some ETH along with the transaction.
Step 2 : The function attempts to convert the sent ETH to WETH by calling `weth.deposit{value: msg.value}();`.
Step 3 : The contract calls `mToken.mint(msg.value);`, but this operation fails for some reason. However, instead of reverting the transaction, `mToken.mint()` returns an error code.
Step 4 : Ignoring the failure from `mToken.mint()`, the contract proceeds to `IERC20(address(mToken)).safeTransfer(recipient, mToken.balanceOf(address(this)));`.



BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:H/R:P/S:C (8.2)

Recommendation:

Ensure that `mToken.mint()` is successful before transferring tokens.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by adding the return value validation.

Commit ID: c39f98bdc9dd4e448ba585923034af1d47f74dfa

3.2 (HAL-02) SILENT FAILURE DURING TOKEN REDEMPTION ON THE ROUTER CONTRACT - HIGH (8.2)

Description:

In the router contract, The redeem function aims to redeem mTokens equivalent to mTokenRedeemAmount. The call mTokenRedeemAmount);` is responsible for the redemption action.

In the event of an error, the mToken.redeem() function from Compound's mToken contract doesn't revert, but instead returns a non-zero error code as a uint. This behavior deviates from the standard Solidity function behavior that typically reverts in case of an error.

The redeem() function in the MTOKEN contract doesn't check the return value of mToken.redeem(mTokenRedeemAmount);. If this redemption operation fails (returns a non-zero error code), the contract still proceeds with the remaining operations, leading to a silent failure. As a result, the contract behaves as if tokens were redeemed when they were not, creating a discrepancy between the actual and perceived balance of mtokens and eth.

Code Location:

Listing 2

```
    function redeem(uint256 mTokenRedeemAmount, address recipient) external {
        IERC20(address(mToken)).safeTransferFrom(
            msg.sender,
            address(this),
            mTokenRedeemAmount
        );

        mToken.redeem(mTokenRedeemAmount);

```

```
        weth.withdraw(weth.balanceOf(address(this)));

        (bool success, ) = payable(recipient).call{
            value: address(this).balance
        }("");
        require(success, "WETHRouter: ETH transfer failed");
    }
```

Proof Of Concept:

Step 1 : An external actor (say, an address ‘A’) calls the redeem() function with a certain mTokenRedeemAmount and recipient.

Step 2 : The function starts by transferring mTokenRedeemAmount of mTokens from ‘A’ to the contract itself. This is done via the IERC20(address(mToken)).safeTransferFrom(msg.sender, address(this), mTokenRedeemAmount); statement.

Step 3 : Next, the function attempts to redeem the mTokens that have just been transferred to the contract, using mToken.redeem(mTokenRedeemAmount);. But, for some reason, this redemption fails. In normal circumstances, this failure should cause the transaction to revert. However, due to the atypical behavior of the mToken.redeem() method (it doesn’t revert on failure but returns a non-zero uint instead), the execution continues to the next line.

Step 4 : Now, the contract attempts to convert its entire WETH balance to ETH via weth.withdraw(weth.balanceOf(address(this)));. Since the redemption in step 3 failed, this step should not result in any additional ETH being added to the contract. However, let’s assume that the contract already had some ETH balance before the transaction began.

Step 5 : The contract then tries to transfer its entire ETH balance to the recipient specified in step 1. Despite the failed redemption, the function ends up transferring the contract’s existing ETH balance to the recipient.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:H/R:P/S:C (8.2)

Recommendation:

Ensure that `mToken.redeem()` is successful before transferring tokens.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by adding the return value validation.

Commit ID: c39f98bdc9dd4e448ba585923034af1d47f74dfa

3.3 (HAL-03) MISSING CHAIN ID AND RECEIVER ADDRESS VERIFICATION IN EXECUTEPROPOSAL() FUNCTION - MEDIUM (5.9)

Description:

The executeProposal() function in the current smart contract is responsible for parsing and verifying VAAs (Validators Aggregated Attestations) and then executing transactions based on these VAAs. The function does not verify the Chain ID or the receiver address (recipient of the transaction).

The absence of chain ID and receiver address verification could lead to significant security issues. Since the chain ID and recipient address aren't checked, an attacker can craft a VAA to target an address on another chain, causing a cross-chain replay attack.

Code Location:

Listing 3

```
function _executeProposal(bytes memory VAA, bool overrideDelay)
↪ private {
    // This call accepts single VAAs and headless VAAs
    (
        IWormhole.VM memory vm,
        bool valid,
        string memory reason
    ) = wormholeBridge.parseAndVerifyVM(VAA);

    require(valid, reason); /// ensure VAA parsing
↪ verification succeeded

    if (!overrideDelay) {
        require(
            queuedTransactions[vm.hash].queueTime != 0,
```

FINDINGS & TECH DETAILS

```
                "TemporalGovernor: tx not queued"
            );
            require(
                queuedTransactions[vm.hash].queueTime + proposalDelay <=
                    block.timestamp,
                "TemporalGovernor: timelock not finished"
            );
        } else if (queuedTransactions[vm.hash].queueTime == 0) {
            /// if queue time is 0 due to fast track execution, set it to current block timestamp
            queuedTransactions[vm.hash].queueTime = block.timestamp.toUint248();
        }



        require(
            !queuedTransactions[vm.hash].executed,
            "TemporalGovernor: tx already executed"
        );

        queuedTransactions[vm.hash].executed = true;

        address[] memory targets; /// contracts to call
        uint256[] memory values; /// native token amount to send
        bytes[] memory calldatas; /// calldata to send
        (, targets, values, calldatas) = abi.decode(
            vm.payload,
            (address, address[], uint256[], bytes[])
        );

        /// Interaction (s)

        _sanityCheckPayload(targets, values, calldatas);

        for (uint256 i = 0; i < targets.length; i++) {
            address target = targets[i];
            uint256 value = values[i];
            bytes memory data = calldatas[i];

            // Go make our call, and if it is not successful revert with the error bubbling up
            (bool success, bytes memory returnData) = target.call{
```

```
↳ value: value}(
                    data
                );

                /// revert on failure with error message if any
                require(success, string(returnData));

                emit ExecutedTransaction(target, value, data);
            }
        }
```

Proof Of Concept:

Step 1 : An attacker crafts a wormhole message that appears to be valid but is intended for a different chain (different chain ID) or is directed to an unintended recipient address.

Step 2 : The attacker submits this crafted payload to the _executeProposal() function in the smart contract.

Step 3 : Since there are no checks in place for the chain ID or recipient address, the function treats the VAA as valid and begins to execute the transaction(s) specified in the VAA payload.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:N/Y:N/R:P/S:C (5.9)

Recommendation:

Consider checking emitter Chain id, receiver on the function.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by adding the necessary validations.

Commit ID: c39f98bdc9dd4e448ba585923034af1d47f74dfa

3.4 (HAL-04) UNRESTRICTED RECEIVE IN WETHROUTER ENABLES EXCESS REDEMPTIONS - LOW (3.1)

Description:

The redeem() function in the current design of the WETHRouter smart contract is designed to handle the redemption of mToken and subsequent withdrawal of WETH. However, this function does not restrict the receipt of tokens to only WETH/mToken. As a result, any native token sent directly to the WETHRouter contract will be sent to the first redeemer.

In this setup, an unintentional or malicious transfer of arbitrary tokens to the WETHRouter contract could lead to an unexpected balance increase. When the redeem() function is called, it attempts to withdraw all ETH equivalent in the contract and sends it to the recipient. If an arbitrary amount of tokens or native ETH is sent to the contract, it would inflate the balance available for withdrawal, making it retrievable by the first redeemer.

Code Location:

Listing 4

```
    function redeem(uint256 mTokenRedeemAmount, address recipient) external {
        IERC20(address(mToken)).safeTransferFrom(
            msg.sender,
            address(this),
            mTokenRedeemAmount
        );

        mToken.redeem(100 ether);

        weth.withdraw(weth.balanceOf(address(this)));

        (bool success, ) = payable(recipient).call{
            value: address(this).balance
```

```
        }("");
        require(success, "WETHRouter: ETH transfer failed");
    }

    receive() external payable {}
```

BVSS:

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

Recommendation:

A potential solution could be to add a mechanism that isolates the withdrawal of mToken-generated WETH from the withdrawal of other tokens that might be sent to the contract. This could be achieved by storing the contract's balance before and after the mToken redemption and only allowing the withdrawal of the difference.

Remediation Plan:

RISK ACCEPTED: The Moonwell Finance team accepted the risk of the issue.

3.5 (HAL-05) IMPLEMENTATIONS CAN BE INITIALIZED - LOW (2.5)

Description:

The contracts are upgradable, inheriting from the Initializable contract. However, the current implementations are missing the _disableInitializers () function call in the constructors. Thus, an attacker can initialize the implementation. Usually, the initialized implementation has no direct impact on the proxy itself; however, it can be exploited in a phishing attack. In rare cases, the implementation might be mutable and may have an impact on the proxy.

BVSS:

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

Recommendation:

It is recommended to call _disableInitializers within the contract's constructor to prevent the implementation from being initialized.

Remediation Plan:

SOLVED: The contracts now implement the _disableInitializers() function call in the constructors.

Commit ID: c39f98bdc9dd4e448ba585923034af1d47f74dfa

3.6 (HAL-06) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

The project contains many instances of floating pragma. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too recent which has not been extensively tested.

Code Location:

The ChainlinkCompositeOracle is affected. (^0.8.0)

BVSS:

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

Recommendation:

Consider locking the pragma version with known bugs for the compiler version by removing the caret (^) symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by locking the pragma.

Commit ID: 17fce574c46259cb22b8b6215b8b982169eb40e7

3.7 (HAL-07) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL (0.0)

Description:

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also saves deployment gas.

BVSS:

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

Recommendation:

Consider replacing all revert strings with custom errors.

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

3.8 (HAL-08) INCREMENT/DECREMENT FOR LOOP VARIABLE IN AN UNCHECKED BLOCK - INFORMATIONAL (0.0)

Description:

i++ involves checked arithmetic, which is not required. This is because the value of i is always strictly less than length <= 2**256 - 1. Therefore, the theoretical maximum value of i to enter the for-loop body is 2**256 - 2. This means that the i++ in the for loop can never overflow. Regardless, the compiler performs the overflow checks.

Code Location:

Listing 5

```
File: core/Governance/TemporalGovernor.sol

60:         for (uint256 i = 0; i < _trustedSenders.length; i++) {

103:            for (uint256 i = 0; i < trustedSendersList.length; i++) {

132:            for (uint256 i = 0; i < _trustedSenders.length; i++) {

159:            for (uint256 i = 0; i < _trustedSenders.length; i++) {

380:         for (uint256 i = 0; i < targets.length; i++) {
```

Listing 6

```
File: core/MultiRewardDistributor/MultiRewardDistributor.sol

209:         for (uint256 index = 0; index < configs.length; index++) {

```

```
240:          for (uint256 index = 0; index < markets.length; index++) {

281:          for (uint256 index = 0; index < configs.length; index++) {

419:          for (uint256 index = 0; index < configs.length; index++) {

983:          for (uint256 index = 0; index < configs.length; index++) {

1009:          for (uint256 index = 0; index < configs.length; index++) {

1053:          for (uint256 index = 0; index < configs.length; index++) {

1112:          for (uint256 index = 0; index < configs.length; index++) {

1163:          for (uint256 index = 0; index < configs.length; index++) {
```

BVSS:

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

Recommendation:

Consider incrementing the for loop variable in an unchecked block.

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

3.9 (HAL-09) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN - INFORMATIONAL (0.0)

Description:

The current ownership transfer process for TemporalGovernor contract inheriting from Ownable or OwnableUpgradeable involves the current owner calling the transferOwnership() function:

Listing 7: Ownable.sol

```
function transferOwnership(address newOwner) public virtual
↳ onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the
↳ zero address");
    _setOwner(newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the onlyOwner modifier.

BVSS:

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

Recommendation:

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. This can be easily achieved by using OpenZeppelin's Ownable2Step contract instead of Ownable:

Listing 8: Ownable2Step.sol (Lines 52-56)

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.8.0) (access/Ownable2Step.sol)

pragma solidity ^0.8.0;

import "./Ownable.sol";

/**
 * @dev Contract module which provides access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership} and {acceptOwnership}.
 *
 * This module is used through inheritance. It will make available all functions
 * from parent (Ownable).
 */
abstract contract Ownable2Step is Ownable {
    address private _pendingOwner;

    event OwnershipTransferStarted(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Returns the address of the pending owner.
     */
    function pendingOwner() public view virtual returns (address) {
        return _pendingOwner;
    }

    /**
     * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there is one.
     * Can only be called by the current owner.
     */
```

```
    function transferOwnership(address newOwner) public virtual override onlyOwner {
        _pendingOwner = newOwner;
        emit OwnershipTransferStarted(owner(), newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }

    /**
     * @dev The new owner accepts the ownership transfer.
     */
    function acceptOwnership() external {
        address sender = _msgSender();
        require(pendingOwner() == sender, "Ownable2Step: caller is not the new owner");
        _transferOwnership(sender);
    }
}
```

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

FINDINGS & TECH DETAILS

3.10 (HAL-10) CACHE ARRAY LENGTH - INFORMATIONAL (0.0)

Description:

In a for loop, the length of an array can be put in a temporary variable to save some gas. This has been done already in several other locations in the code.

In the above case, the solidity compiler will always read the length of the array during each iteration. That is,

- if it is a storage array, this is an extra sload operation (100 additional extra gas (EIP-2929) for each iteration except for the first),
- if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first),
- if it is a calldata array, this is an extra calldataload operation (3 additional gas for each iteration except for the first)

Code Location:

Listing 9

```
File: core/Governance/TemporalGovernor.sol

60:         for (uint256 i = 0; i < _trustedSenders.length; i++) {

103:            for (uint256 i = 0; i < trustedSendersList.length
    ; i++) {

132:            for (uint256 i = 0; i < _trustedSenders.length; i
    ++) {

159:            for (uint256 i = 0; i < _trustedSenders.length; i
    ++) {

380:         for (uint256 i = 0; i < targets.length; i++) {
```

Listing 10

```
File: core/MultiRewardDistributor/MultiRewardDistributor.sol

209:         for (uint256 index = 0; index < configs.length; index++) {

240:         for (uint256 index = 0; index < markets.length; index++) {

281:         for (uint256 index = 0; index < configs.length; index++) {

419:         for (uint256 index = 0; index < configs.length; index++) {

983:         for (uint256 index = 0; index < configs.length; index++) {

1009:         for (uint256 index = 0; index < configs.length; index++) {

1053:         for (uint256 index = 0; index < configs.length; index++) {

1112:         for (uint256 index = 0; index < configs.length; index++) {

1163:         for (uint256 index = 0; index < configs.length; index++) {
```

BVSS:

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

Recommendation:

In a for loop, store the length of an array in a temporary variable.

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

3.11 (HAL-11) REDUNDANT SAFE CAST - INFORMATIONAL (0.0)

Description:

The TemporalGovernor contract uses the OpenZeppelin SafeCast library for type conversion operations. However, it's important to note that there's no possibility of an overflow occurring in the variable through the utilization of block.timestamp.

Code Location:

Listing 11

```
contract TemporalGovernor is ITemporalGovernor, Ownable, Pausable
↳ {
    using SafeCast for *;
}
```

BVSS:

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

Recommendation:

It is recommended to remove unnecessary SafeCast library.

Remediation Plan:

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

Commit ID: 17fce574c46259cb22b8b6215b8b982169eb40e7

3.12 (HAL-12) REVERT STRING SIZE OPTIMIZATION - INFORMATIONAL (0.0)

Description:

Shortening revert strings to fit in 32 bytes will decrease deploy time gas and will decrease runtime gas when the revert condition has been met.

Code Location:

Listing 12

```
    function setTrustedSenders(
        TrustedSender[] calldata _trustedSenders
    ) external {
        require(
            msg.sender == address(this),
            "TemporalGovernor: Only this contract can update
↳ trusted senders"
        );
}
```

BVSS:

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

Recommendation:

Shorten the revert strings to fit in 32 bytes. Alternatively, the code could be modified to use custom errors, introduced in Solidity 0.8.4.

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

3.13 (HAL-13) NO NEED TO INITIALIZE VARIABLES WITH DEFAULT VALUES - INFORMATIONAL (0.0)

Description:

Initialization to 0 or false is not necessary, as these are the default values in Solidity.

Code Location:

Listing 13

```
File: core/Governance/TemporalGovernor.sol

60:         for (uint256 i = 0; i < _trustedSenders.length; i++) {

103:             for (uint256 i = 0; i < trustedSendersList.length; i++) {

132:             for (uint256 i = 0; i < _trustedSenders.length; i++) {

159:             for (uint256 i = 0; i < _trustedSenders.length; i++) {

380:         for (uint256 i = 0; i < targets.length; i++) {
```

Listing 14

```
File: core/MultiRewardDistributor/MultiRewardDistributor.sol

209:         for (uint256 index = 0; index < configs.length; index++) {

240:         for (uint256 index = 0; index < markets.length; index++) {

```

```
281:            for (uint256 index = 0; index < configs.length; index++) {

419:            for (uint256 index = 0; index < configs.length; index++) {

983:            for (uint256 index = 0; index < configs.length; index++) {

1009:            for (uint256 index = 0; index < configs.length; index++) {

1053:            for (uint256 index = 0; index < configs.length; index++) {

1112:            for (uint256 index = 0; index < configs.length; index++) {

1163:            for (uint256 index = 0; index < configs.length; index++) {
```

BVSS:

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

Recommendation:

Remove the initialization values of 0 or false.

Remediation Plan:

ACKNOWLEDGED: The Moonwell Finance team acknowledged the issue.

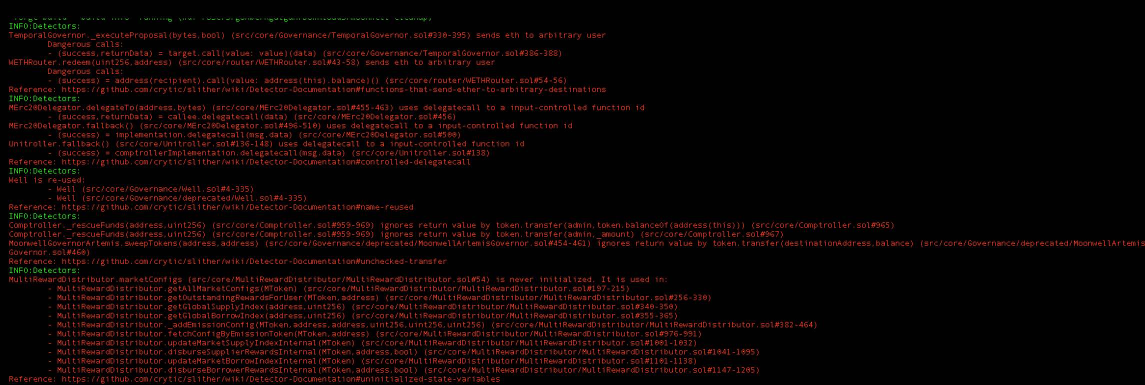
AUTOMATED TESTING

4.1 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:



```
INFO:Detectors:
TemporalGovernor._executeProposal(bytes,bool) (src/core/Governance/TemporalGovernor.sol#330-395) sends eth to arbitrary user
        Dangerous calls:
        - (success,returnData) = target.call{value: value}(data) (src/core/Governance/TemporalGovernor.sol#386-388)
WETHRouter.redeem(uint256,address) (src/core/router/WETHRouter.sol#43-58) sends eth to arbitrary user
        Dangerous calls:
        - (success) = address(recipient).call{value: address(this).balance}() (src/core/router/WETHRouter.sol#54-56)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
MErc20Delegator.delegateTo(address,bytes) (src/core/MErc20Delegator.sol#455-463) uses delegatecall to a input-controlled function id
        - (success,returnData) = callee.delegatecall(data) (src/core/MErc20Delegator.sol#456)
MErc20Delegator.fallback() (src/core/MErc20Delegator.sol#496-510) uses delegatecall to a input-controlled function id
        - (success) = implementation.delegatecall(msg.data) (src/core/MErc20Delegator.sol#500)
Unitroller.fallback() (src/core/Unitroller.sol#136-148) uses delegatecall to a input-controlled function id
        - (success) = comptrollerImplementation.delegatecall(msg.data) (src/core/Unitroller.sol#138)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#controlled-delegatecall
INFO:Detectors:
Well is re-used:
        - Well (src/core/Governance/Well.sol#4-335)
        - Well (src/core/Governance/deprecated/Well.sol#4-335)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused
INFO:Detectors:
Comptroller._rescueFunds(address,uint256) (src/core/Comptroller.sol#959-969) ignores return value by token.transfer(admin,token.balanceOf(address(this))) (src/core/Comptroller.sol#965)
Comptroller._rescueFunds(address,uint256) (src/core/Comptroller.sol#959-969) ignores return value by token.transfer(admin,_amount) (src/core/Comptroller.sol#967)
MoonwellGovernorArtemis.sweepTokens(address,address) (src/core/Governance/deprecated/MoonwellArtemisGovernor.sol#454-461) ignores return value by token.transfer(destinationAddress,balance) (src/core/Governance/deprecated/MoonwellArtemis
Governor.sol#460)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
MultiRewardDistributor.marketConfigs (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#54) is never initialized. It is used in:
        - MultiRewardDistributor.getAllMarketConfigs(MToken) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#197-215)
        - MultiRewardDistributor.getOutstandingRewardsForUser(MToken,address) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#256-330)
        - MultiRewardDistributor.getGlobalSupplyIndex(address,uint256) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#340-350)
        - MultiRewardDistributor.getGlobalBorrowIndex(address,uint256) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#355-365)
        - MultiRewardDistributor._addEmissionConfig(MToken,address,address,uint256,uint256,uint256) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#382-464)
        - MultiRewardDistributor.fetchConfigByEmissionToken(MToken,address) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#976-991)
        - MultiRewardDistributor.updateMarketSupplyIndexInternal(MToken) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#1001-1032)
        - MultiRewardDistributor.disburseSupplierRewardsInternal(MToken,address,bool) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#1041-1095)
        - MultiRewardDistributor.updateMarketBorrowIndexInternal(MToken) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#1101-1138)
        - MultiRewardDistributor.disburseBorrowerRewardsInternal(MToken,address,bool) (src/core/MultiRewardDistributor/MultiRewardDistributor.sol#1147-1205)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

- No major issues found by Slither.

AUTOMATED TESTING

4.2 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