// HALBORN

Qoda Finance - Core V1

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **June 29th, 2023 - July 28th, 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
0.3	Draft Review	07/27/2023	Gabi Urrutia
1.0	Remediation Plan	08/02/2023	Gokberk Gulgun
1.1	Remediation Plan Review	08/06/2023	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

Qoda Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on June 29th, 2023 and ending on July 28th, 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 successfully addressed by the Qoda 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)

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:

- QodaFi/v1-core

Commit ID:

- 50fbe7c35c1d75a1cfc6e1bef08e567bddbd5387

IN-SCOPE CONTRACTS :

- FixedRateMarket.sol
- LiquidityEmissionsQontroller.sol
- QAdmin.sol
- QPriceOracle.sol
- QToken.sol
- QodaLens.sol
- QollateralManager.sol
- QuoteManager.sol
- TradingEmissionsQontroller.sol

2. REMEDIATION COMMIT IDs :

- 119527170674c8c82ea53addf463595ea4719251
- d67a42652e48fd8ed297545113c9778c21af32e2
- b8617f34bafc08104e3628cce0eb4e93d40cd25e
- 993c03eee650747d41be27a57e19efb0d3c963f5
- 56d373e35efc593210fac069ae5f007ba298ffd8
- a4176933e56ba366e9a0834bf2fa477ec066eb4c
- aa3880108654ccdadd331d570b2ee6902a533c4b

- de368b549cb6ee24b56f7ad2642c2fc7e30caf1d
- ec7fdaa1fea939ac62ed84951679a4ec20db8bb7
- 4981e21206a8ee8a8b80c6ddce7e08416d4666a7
- b8617f34bafc08104e3628cce0eb4e93d40cd25e

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	8	4	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) UNAUTHORIZED QUOTE CANCELLATION THROUGH DIRECT INTERACTION WITH QUOTEMANAGER CONTRACT	Critical (10)	SOLVED - 07/23/2023
(HAL-02) UNRESTRICTED ACCESS TO CREATEQUOTE FUNCTION ALLOWS UNAUTHORIZED QUOTE CREATION	Critical (10)	SOLVED - 07/23/2023
(HAL-03) SILENT FAILURE DURING TOKEN MINTING ON DEPOSIT FUNCTION	Medium (5.5)	SOLVED - 07/23/2023
(HAL-04) SILENT FAILURE DURING TOKEN REDEMPTION ON THE WITHDRAW FUNCTION	Medium (5.5)	SOLVED - 07/23/2023
(HAL-05) ROBUST ALLOWANCE	Medium (5.5)	SOLVED - 07/24/2023
(HAL-06) INCOMPATIBILITY WITH REBASING/DEFLATIONARY/INFLATIONARY TOKENS	Medium (6.2)	SOLVED - 08/02/2023
(HAL-07) DIA ORACLE DOES NOT HAVE PROTECTION FOR ZERO VALUE	Medium (6.2)	SOLVED - 07/25/2023
(HAL-08) LATESTROUNDDATA CALL MAY RETURN STALE RESULTS	Medium (6.2)	SOLVED - 07/25/2023
(HAL-09) UNSAFE HANDLING OF ERC20 TRANSFER RESULTS	Medium (5.5)	SOLVED - 07/25/2023
(HAL-10) COMPROMISED ADMIN CAN LIQUIDATE USERS	Medium (6.2)	SOLVED - 08/02/2023
(HAL-11) IMPLEMENTATIONS CAN BE INITIALIZED	Low (2.5)	SOLVED - 07/25/2023
(HAL-12) MARKET EXISTENCE VERIFICATION ABSENCE IN SETPROTOCOLFEE FUNCTION	Low (2.5)	SOLVED - 07/25/2023
(HAL-13) USE OF DEPRECATED SETUPROLE FUNCTION	Low (2.5)	SOLVED - 07/25/2023
(HAL-14) ABSENCE OF INPUT VALIDATION LEADS TO ERRONEOUS REMOVAL FROM LINKEDLIST	Low (2.5)	SOLVED - 07/25/2023

(HAL-15) FLOATING PRAGMA	Informational (0.0)	SOLVED - 08/02/2023
(HAL-16) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS	Informational (0.0)	SOLVED - 08/02/2023
(HAL-17) INCREMENT/DECREMENT FOR LOOP VARIABLE IN AN UNCHECKED BLOCK	Informational (0.0)	SOLVED - 07/22/2023
(HAL-18) CACHE ARRAY LENGTH	Informational (0.0)	SOLVED - 08/02/2023
(HAL-19) NO NEED TO INITIALIZE VARIABLES WITH DEFAULT VALUES	Informational (0.0)	ACKNOWLEDGED
(HAL-20) UNUSED ADDRESS VARIABLE	Informational (0.0)	SOLVED - 07/22/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) UNAUTHORIZED QUOTE CANCELLATION THROUGH DIRECT INTERACTION WITH QUOTEMANAGER CONTRACT - CRITICAL(10)

Description:

It has been observed that any user can cancel quotes by directly interacting with the QuoteManager contract. The cancelQuote function does not have proper access controls to check whether the caller is authorized to cancel the quote or not. This implies that anyone who has the required information could potentially cancel a quote, irrespective of whether they are the original quoter or not.

Code Location:

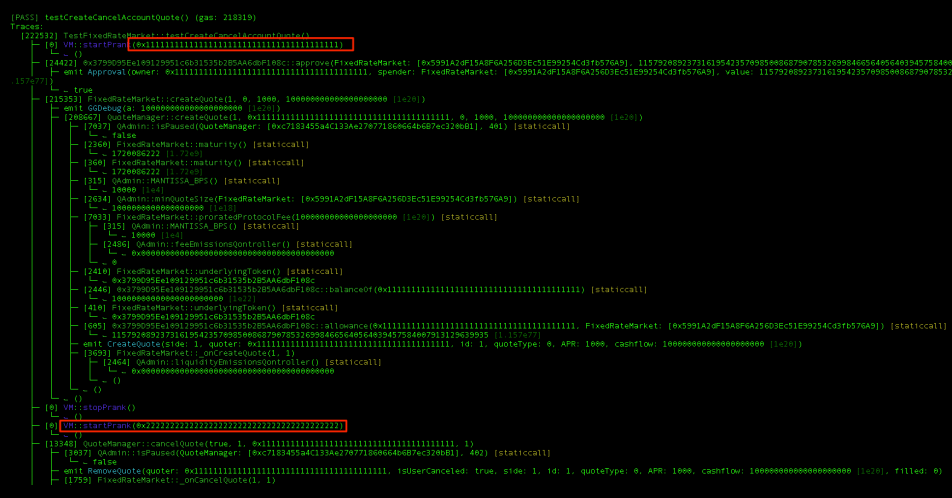
/contracts/QuoteManager.sol#L95

Listing 1

```
/// @notice Cancel `Quote` by id. Note this is a O(1) operation
 /// since `OrderbookSide` uses hashmaps under the hood. However, it is
 /// O(n) against the array of `Quote` ids by account so we should ensure
 /// that array should not grow too large in practice.
 /// @param isUserCanceled True if user actively canceled `Quote`, false otherwise
 /// @param side 0 for borrow `Quote`, 1 for lend `Quote`
 /// @param quoter Address of the `Quoter`
 /// @param id Id of the `Quote`
 function cancelQuote(bool isUserCanceled, uint8 side, address quoter, uint64 id) external whenNotPaused(402) {
     _cancelQuote(isUserCanceled, side, quoter, id);
 }
```

Proof Of Concept:

Step 1 : Deploy a new instance of QuoteManager contract in a testing environment. Record the contract address.
Step 2 : Issue a new quote using an address (let's call it Address A) via the createQuote function, or similar function if available.
Step 3 : Record the quote ID for the quote created by Address A.
Step 4 : Using a different address (Address B), call the cancelQuote function, passing the quote ID of the quote created by Address A, and other required parameters.
Step 5 : Check the status of the quote using its ID. If it shows as cancelled, this confirms that the cancelQuote function can be called by an address that did not originally create the quote



BVSS:

AO:A/AC:L/AX:L/C:C/I:C/A:C/D:C/Y:C/R:N/S:C (10)

Recommendation:

Consider implementing access control on the function.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by adding an access control to the function.

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.2 (HAL-02) UNRESTRICTED ACCESS TO CREATEQUOTE FUNCTION ALLOWS UNAUTHORIZED QUOTE CREATION - CRITICAL(10)

Description:

The createQuote function in the QuoteManager contract does not verify if the caller (msg.sender) is the same as the quoter parameter passed to the function. This potentially allows a malicious actor to create quotes on behalf of others without their consent or knowledge. If an unauthorized user can manipulate the quote creation mechanism, it can compromise the integrity and trustworthiness of the quotes' system. The unauthorized user could create misleading quotes that can be used to manipulate market behavior.

Code Location:

/contracts/QuoteManager.sol#L83

Listing 2

```
1   /** USER INTERFACE **/
2
3   /// @notice Creates a new  `Quote` and adds it to the `
↳ OrderbookSide` linked list by side
4   /// @param side 0 for borrow `Quote`, 1 for lend `Quote`
5   /// @param quoter Account of the Quoter
6   /// @param quoteType 0 for PV+APR, 1 for FV+APR
7   /// @param APR In decimal form scaled by 1e4 (ex. 1052 = 10.52%)
8   /// @param cashflow Can be PV or FV depending on `quoteType`
9   function createQuote(uint8 side, address quoter, uint8 quoteType
↳ , uint64 APR, uint cashflow) external whenNotPaused(401) {
10      _createQuote(side, quoter, quoteType, APR, cashflow);
11  }
```

Proof Of Concept:

Step 1 : Deploy a new instance of QuoteManager contract in a testing environment. Record the contract address.
Step 2 : Using an address (let's call it Address A), call the createQuote function, passing a different address (let's call it Address B) as the quoter parameter, along with other required parameters.
Step 3 : Check if a new quote is created and attributed to Address B in the system. If a quote attributed to Address B is created, it confirms that the createQuote function can be called by an address that is different from the quoter parameter.

BVSS:

AO:A/AC:L/AX:L/C:C/I:C/A:C/D:C/Y:C/R:N/S:C (10)

Recommendation:

The createQuote function should include an authorization check to ensure that the quoter parameter matches the market.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by adding a validation.

Commit ID : 1195271706 74c8c82ea53addf463595ea4719251

4.3 (HAL-03) SILENT FAILURE DURING TOKEN MINTING ON DEPOSIT FUNCTION - MEDIUM (5.5)

Description:

The `mToken.mint(actualDepositUnderlying);` 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 deposit function.

Code Location:

/contracts/QollateralManager.sol#L900

Listing 3

```
...
      MGlimmerInterface mToken = MGlimmerInterface(mTokenAddress);

      // Forward user collateral to MToken
      uint balanceBeforeMToken = mToken.balanceOf(address(this));
      mToken.mint{value: actualDepositUnderlying}();

      // Get accurate amount of MToken minted
      uint actualDepositMToken = mToken.balanceOf(address(this)) -
↳ balanceBeforeMToken;

      // Update internal account collateral balance mappings
```

```
12         _addCollateral(account, mToken, actualDepositMToken);
13 ...
```

Proof Of Concept:

Step 1 : An external actor calls the depositCollateralWithMTokenWrapWithETH() function, sending some ETH along with the transaction.
Step 2 : The function attempts to convert the sent ETH to WETH by calling weth.withdraw(actualDepositUnderlying);.
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 _addCollateral(account, mToken, actualDepositMToken);.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:M/R:P/S:C (5.5)

Recommendation:

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

Remediation Plan:

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

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.4 (HAL-04) SILENT FAILURE DURING TOKEN REDEMPTION ON THE WITHDRAW FUNCTION - MEDIUM (5.5)

Description:

In the QollateralManager contract, The `redeem` function aims to redeem `mTokens` equivalent to `mTokenRedeemAmount`. The call `mToken.redeem(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 an 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:

/contracts/QollateralManager.sol#L1052

Listing 4

```
1       MErc20Interface mToken = MErc20Interface(mTokenAddress);
2
3       // Redeem mTokens for underlying and return it to user
4       uint balanceBeforeUnderlying = underlying.balanceOf(address(
↳ this));
5       mToken.redeem(amount);
6
7       // Get accurate amount of underlying redeemed
```

```
        actualAmountUnderlying = underlying.balanceOf(address(this))
    - balanceBeforeUnderlying;

        // Update internal account collateral balance mappings
        _subtractCollateral(account, mToken, amount);
```

/contracts/QollateralManager.sol#L1067

Listing 5

```
        // Generic logic for all other ERC20 tokens using
        // ERC20 transfer function
        MErc20Interface mToken = MErc20Interface(mTokenAddress);

        // Redeem mTokens for underlying and return it to user
        uint balanceBeforeUnderlying = underlying.balanceOf(address(
    this));
        mToken.redeem(amount);

        // Get accurate amount of underlying redeemed
        actualAmountUnderlying = underlying.balanceOf(address(this))
    - balanceBeforeUnderlying;

        // Update internal account collateral balance mappings
        _subtractCollateral(account, mToken, amount);
```

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 subtracting collateral mTokenRedeemAmount of mTokens from 'A'. This is done via the _subtractCollateral(account, mToken, amount); 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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:M/R:P/S:C (5.5)

Recommendation:

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

Remediation Plan:

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

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.5 (HAL-05) ROBUST ALLOWANCE - MEDIUM (5.5)

Description:

Non-standard tokens like USDT will revert the transaction when a contract or a user tries to approve an allowance when the spender allowance is already set to a non-zero value. For that reason, the previous allowance should be decreased before increasing allowance in the related function.

Code Location:

/contracts/QollateralManager.sol#L918

Listing 6

```
1       // Make underlying token approval
2       // NOTE: This "approve" call should be safe as it only
↳ approves the bare
3       // minimum transferred, but in general be careful of attacks
↳  here. We are
4       // relying on the above "require" check that `mTokenAddress`
↳  is a safe
5       // address, so we should make sure that only admin is able
↳ to add
6       // `mTokenAddress` as a whitelisted address or else an
↳ attacker can
7       // potentially drain funds
8       underlying.approve(mTokenAddress, actualDepositUnderlying);
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:M/R:P/S:C (5.5)

Recommendation:

Consider approving with zero first through safeApprove function.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by approving with zero first.

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.6 (HAL-06) INCOMPATIBILITY WITH REBASING/DEFLATIONARY/INFLATIONARY TOKENS - MEDIUM (6.2)

Description:

The protocol does not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest. During repaying borrow, transferTokenOrETH does not check before/after balance, which will cause insolvency in the contract due to the received amount will be less than expected (accountBorrow).

Code Location:

Utils.sol#L26C1-L71C2

Listing 7

```
function transferTokenOrETH(
                              address sender,
                              address receiver,
                              uint amount,
                              IERC20 underlying,
                              address wethAddress,
                              bool isSendingETH,
                              bool isReceivingETH
                              ) internal {
  address sender_ = sender;
  address receiver_ = receiver;

  // If it is ETH transfer, contract will send/receive on behalf
  // and do needed token wrapping/unwrapping
  if (isSendingETH) {
    sender_ = address(this);
  }
  if (isReceivingETH) {
    receiver_ = address(this);
```

```
20      }
21
22      // If sender uses ETH for transfer, token wrapping is needed
23      if (isSendingETH) {
24        IWETH weth = IWETH(wethAddress);
25        weth.deposit{ value: amount }();
26      }
27
28      // Transfer `amount` from sender to receiver
29      if (sender_ == address(this)) {
30        underlying.safeTransfer(receiver_, amount);
31      } else {
32        underlying.safeTransferFrom(sender_, receiver_, amount);
33      }
34
35      // For receiver getting ETH in transfer, token unwrapping is
↳ needed
36      if (isReceivingETH) {
37        IWETH weth = IWETH(wethAddress);
38        weth.withdraw(amount);
39        (bool success,) = receiver.call{value: amount}("");
40        if (!success) {
41          revert CustomErrors.UTL_UnsuccessfulEthTransfer();
42        }
43      }
44    }
45
46 }
```

Proof Of Concept:

Step 1 : User A calls the transferTokenOrETH function to deposit 100 tokens into the contract.
Step 2 : Due to the deflationary nature of the token, a fee (e.g., 1%) is deducted during the transfer.
Step 3 : The contract, however, records a deposit of 100 tokens, although only 99 tokens (100 tokens - 1% fee) were transferred due to the fee.

BVSS:

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

Recommendation:

Ensure that checking the previous balance/after balance equals to the amount for any rebasing/inflation/deflation. Consider adding support in contracts for such tokens before accepting user-supplied tokens. Finally, consider supporting deflationary / rebasing / etc. tokens by extra checking the balances before/after or strictly inform your users not to use such tokens if they do not want to lose them.

Remediation Plan:

SOLVED: The Qoda Finance team resolved the issue by calculating the difference in asset value before and after the transfer.

Commit ID : a4176933e56ba366e9a0834bf2fa477ec066eb4c

4.7 (HAL-07) DIA ORACLE DOES NOT HAVE PROTECTION FOR ZERO VALUE - MEDIUM (6.2)

Description:

In the implementation of the DIA Oracle, it was observed that the system can return a value of zero. Further, there appears to be no condition checking if the current block timestamp exceeds the sum of the response timestamp and the grace period. This could lead to potential inaccuracies or manipulation of the data returned by the oracle.

Here is the link to the contract code: DIA Oracle Contract

The main concern with this issue is the data integrity of the DIA Oracle. Should the Oracle return a zero value, it could potentially disrupt the functions that rely on this data. Moreover, if there is no check to ensure the block.timestamp is within the grace period, outdated data could be used, which might lead to further disruptions or even potential security risks depending on how the oracle data is used.

Code Location:

QPriceOracle.sol#L225C1-L260C4

Listing 8

```
1   /// @notice Convenience function for getting price feed from DIA
↳   oracle
2   /// @param underlyingToken Address of the underlying token
3   /// @return answer uint256, decimals uint8
4   function _priceFeedDIA(IERC20 underlyingToken) internal view
↳ returns(uint256, uint8) {
5
6     // We need to retrieve the `symbol` string from the token,
↳ which is not
7     // a part of the standard IERC20 interface
8     IERC20Metadata tokenWithSymbol = IERC20Metadata(address(
```

```
↳ underlyingToken));
 9
10      // DIA Oracle takes pair string input, e.g. `_DIAOracle.
↳ getValue("BTC/USD")`
11      string memory key = string(abi.encodePacked(tokenWithSymbol.
↳ symbol(), "/USD"));
12
13      // Catch and convert exceptions to the proper format
14      if(keccak256(abi.encodePacked(key)) == keccak256(abi.
↳ encodePacked("WBTC/USD"))) {
15        key = "BTC/USD";
16      } else if(keccak256(abi.encodePacked(key)) == keccak256(abi.
↳ encodePacked("WETH/USD"))) {
17        key = "ETH/USD";
18      } else if(keccak256(abi.encodePacked(key)) == keccak256(abi.
↳ encodePacked("WMOVR/USD"))) {
19        key = "MOVR/USD";
20      } else if(keccak256(abi.encodePacked(key)) == keccak256(abi.
↳ encodePacked("WGLMR/USD"))) {
21        key = "GLMR/USD";
22      }
23
24      // Get the value from DIA oracle
25      (uint128 answer, uint128 timestamp) = _DIAOracle.getValue(key)
↳ ;
26
27      // Ensure valid key is being used
28      if (timestamp == 0) {
29        revert CustomErrors.QPO_DIA_Key_Not_Found();
30      }
31
32      // By default, DIA oracles return the current asset price in
↳ USD with a
33      // fix-comma notation of 8 decimal places.
34      return (uint(answer), 8);
35
36    }
```

BVSS:

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

Recommendation:

To mitigate this issue, it is recommended to implement a check in the DIA Oracle to ensure that it does not return a zero value. Additionally, a condition should be introduced to verify that the current block.timestamp does not exceed the sum of the response timestamp and the grace period. This will ensure that the data returned by the Oracle is both accurate and timely.

Remediation Plan:

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

Commit ID : 56d373e35efc593210fac069ae5f007ba298ffd8

4.8 (HAL-08) LATESTROUNDDATA CALL MAY RETURN STALE RESULTS - MEDIUM (6.2)

Description:

The return values of the latestRoundData() call are:

- roundId: The round ID. Oracles provide periodic data updates to the aggregators. Data feeds are updated in rounds. Rounds are identified by their roundId, which increases with each new round. This increase may not be monotonic. Knowing the roundId of a previous round allows contracts to consume historical data.
- answer: The data that this specific feed provides, in this case, the price of an asset.
- startedAt: Timestamp of when the round started.
- updatedAt: Timestamp of when the round was updated.
- answeredInRound: The round ID of the round in which the answer was computed.

In the current implementation, There is no check for stale prices.

Code Location:

QPriceOracle.sol#L262C1-L270C4

Listing 9

```
  /// @notice Convenience function for getting price feed from
↳ Chainlink oracle
  /// @param oracleFeed Address of the chainlink oracle feed
  /// @return answer uint256, decimals uint8
  function _priceFeedChainlink(address oracleFeed) internal view
↳ returns(uint256, uint8) {
    AggregatorV3Interface aggregator = AggregatorV3Interface(
↳ oracleFeed);
    (, int256 answer,,,) =  aggregator.latestRoundData();
```

```
    uint8 decimals = aggregator.decimals();
    return (uint(answer), decimals);
  }
```

BVSS:

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

Recommendation:

_priceFeedChainlink calls out to a Chainlink oracle receiving the latestRoundData(). If there is a problem with Chainlink starting a new round and finding consensus on the new value for the oracle (e.g. Chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the Chainlink system) consumers of this contract may continue using outdated stale or incorrect data (if oracles are unable to submit no new round is started).

Remediation Plan:

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

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.9 (HAL-09) UNSAFE HANDLING OF ERC20 TRANSFER RESULTS - MEDIUM (5.5)

Description:

It was identified that the claimEmissions function in the TradingEmissionsQontroller contract does not verify the return value of the transfer function call, which facilitates the token transfer from the caller to the contract. Some tokens (e.g., ZRX) return false instead of reverting in the event of failure or insufficient balance.

Code Location:

TradingEmissionsQontroller.sol#L140

Listing 10

```
1   /// @notice Mint the unclaimed rewards to user and reset their
↳ claimable emissions
2   function claimEmissions() external whenNotPaused(601) {
3
4     // Get the emissions accrued for the user
5     uint reward = _claimableEmissions[msg.sender];
6
7     if (reward <= 0) {
8       revert CustomErrors.TEQ_ZeroRewardAmount();
9     }
10
11    // Reset user's claimable emissions
12    _claimableEmissions[msg.sender] = 0;
13
14    // Transfer the rewards directly from `
↳ TradingEmissionsQontroller` contract
15    // to user. The assumption is that total supply of reward
↳ tokens have been
16    // pre-minted to this contract address. This limits the
↳ maximum damage of
```

```
17      // a potential hack to just tokens within the contract (
↳ compared to an
18      // infinite mint of tokens if this contract were allowed to
↳ mint on a
19      // callable function
20      _underlying.transfer(msg.sender, reward);
21
22      // Emit the event
23      emit ClaimEmissions(msg.sender, reward);
24    }
25
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:M/R:P/S:C (5.5)

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20 wrapper and the safeTransfer function to transfer the tokens.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by using safeTransfer.

Commit ID : b8617f34bafc08104e3628cce0eb4e93d40cd25e

4.10 (HAL-10) COMPROMISED ADMIN CAN LIQUIDATE USERS - MEDIUM (6.2)

Description:

The function _setCollateralRatio allows an admin to set the values of _minCollateralRatio and _initCollateralRatio. While the function ensures that the minCollateralRatio_ cannot exceed initCollateralRatio_, there is currently no lower bound set for the minCollateralRatio_. This could potentially allow a malicious admin to set the _minCollateralRatio to a very low value, leading to unintended and premature liquidations of users' positions.

Code Location:

/contracts/QAdmin.sol#L528C1-L542C4

Listing 11

```
  function _setCollateralRatio(uint minCollateralRatio_, uint
↳ initCollateralRatio_) external onlyAdmin {
    // `minCollateralRatio_` cannot be above `initCollateralRatio_
↳ `
    if (minCollateralRatio_ > initCollateralRatio_) {
      revert CustomErrors.QA_MinCollateralRatioNotGreaterThanInit
↳ ();
    }

    // Emit the event
    emit SetCollateralRatio(_minCollateralRatio,
↳ _initCollateralRatio, minCollateralRatio_, initCollateralRatio_);

    // Set `_minCollateralRatio` to new value
    _minCollateralRatio = minCollateralRatio_;

    // Set `_initCollateralRatio` to new value
    _initCollateralRatio = initCollateralRatio_;
  }
```

BVSS:

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

Recommendation:

To avoid potential misuse and protect users, it is recommended to implement a lower bound for the _minCollateralRatio.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by comparing minCollateralRatio_ with MANTISSA_COLLATERAL_RATIO.

Commit ID : 4981e21206a8ee8a8b80c6ddce7e08416d4666a7

4.11 (HAL-11) 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 : 119527170674c8c82ea53addf463595ea4719251

4.12 (HAL-12) MARKET EXISTENCE VERIFICATION ABSENCE IN SETPROTOCOLFEE FUNCTION - LOW (2.5)

Description:

The _setProtocolFee function in the QAdmin contract does not validate the existence of the provided marketAddress. The function is designed to set the protocol fee for a given market. However, it fails to verify if the provided address indeed corresponds to an existing market. It blindly casts the provided address into an IFixedRateMarket interface, and then proceeds to set the protocol fee.

Code Location:

/contracts/QAdmin.sol#L641

Listing 12

```
  function _setProtocolFee(address marketAddress, uint
↳ protocolFee_) public onlyAdmin {

    // Max annual protocol fees of 250 basis points
    if (protocolFee_ > 250) {
      revert CustomErrors.QA_OverThreshold(protocolFee_, 250);
    }

    // Min annual protocol fees of 1 basis point
    if (protocolFee_ < 1) {
      revert CustomErrors.QA_UnderThreshold(protocolFee_, 1);
    }

    // Casting from address into corresponding interface
    IFixedRateMarket market = IFixedRateMarket(marketAddress);

    // Emit the event
    emit SetProtocolFee(_protocolFee[market], protocolFee_);

    // Set `_protocolFee` to new value
```

```
20      _protocolFee[market] = protocolFee_;
21    }
```

BVSS:

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

Recommendation:

To prevent the potential impacts, the contract should implement a mechanism to check the existence of the **marketAddress** before setting the protocol fee.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by adding a validation.

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.13 (HAL-13) USE OF DEPRECATED SETUPROLE FUNCTION - LOW (2.5)

Description:

"The contracts make use of the deprecated function _setupRole from the AccessControl contract. As per the Changelog.md, the function is deprecated.

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 use the _grantRole function instead.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by using the _grantRole function.

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.14 (HAL-14) ABSENCE OF INPUT VALIDATION LEADS TO ERRONEOUS REMOVAL FROM LINKEDLIST - LOW (2.5)

Description:

The code lacks validation checks in the remove function of its LinkedList implementation. This function does not confirm the existence of an id within the list prior to performing deletion operations.

Code Location:

LinkedList.sol#L81C1-L108C4

Listing 13

```
  /// @notice Remove the `Quote` with id `id` from the linked list
  function remove(OrderbookSide storage self, uint64 id) internal
↳ {

    QTypes.Quote memory quoteToRemove = self.quotes[id];

    if(self.head == id && self.tail == id) {
      // `OrderbookSide` only has one element. Reset both head and
↳ tail pointers
      setHeadId(self, 0);
      setTailId(self, 0);
    } else if (self.head == id) {
      // `quoteToRemove` is the current head, so set the next item
↳ in the linked list to be head
      setHeadId(self, quoteToRemove.next);
      self.quotes[quoteToRemove.next].prev = 0;
    } else if (self.tail == id) {
      // `quoteToRemove` is the current tail, so set the prev item
↳ in the linked list to be tail
      setTailId(self, quoteToRemove.prev);
      self.quotes[quoteToRemove.prev].next = 0;
    } else {
      // Link the `Quote`s before and after `quoteToRemove`
↳ together
```

```
20        link(self, quoteToRemove.prev, quoteToRemove.next);
21      }
22
23      // Ready to delete `quoteToRemove`
24      delete self.quotes[quoteToRemove.id];
25
26      // Decrement the length of the `OrderbookSide`
27      self.length--;
28    }
```

BVSS:

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

Recommendation:

Consider checking the existence of element.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by checking the existence of the element.

Commit ID : 119527170674c8c82ea53addf463595ea4719251

4.15 (HAL-15) 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:

All Contracts

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 Qoda Finance team solved the issue by changing floating pragma.

4.16 (HAL-16) 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.

Code Location:

Listing 14

```
./FixedRateMarket.sol:127:     require(address(_quoteManager) ==
↳ msg.sender, "FRM0 only quote manager");
```

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:

SOLVED: The Qoda Finance team solved the issue by using unchecked block.

4.17 (HAL-17) 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 15

```
./TradingEmissionsQontroller.sol:70:     for (uint i = 0; i < _numPhases; i++){
./TradingEmissionsQontroller.sol:76:     for (uint i = 0; i < _numPhases; i++){
./QodaLens.sol:197:     for (uint i = 0; i < assetAddresses.length; i++) {
./QodaLens.sol:205:     for (uint i = 0; i < assetAddresses.length; i++) {
./QodaLens.sol:209:       for (uint j = 0; j < maturities.length; j++) {
./QodaLens.sol:238:     for (uint i = 0; i < marketAddresses.length; i++) {
./QodaLens.sol:240:       for (uint8 side = 0; side <= 1; side++) {
./QodaLens.sol:248:     for (uint i = 0; i < marketAddresses.length; i++) {
./QodaLens.sol:250:       for (uint8 side = 0; side <= 1; side++) {
./QodaLens.sol:252:         for (uint j = 0; j < ids.length; j++) {
./QodaLens.sol:486:     for (uint i = 0; i < asset.maturities.length; i++) {
./QodaLens.sol:518:     for (uint i = 0; i < fixedRateMarkets.length; i++) {
./QodaLens.sol:527:     for (uint i = 0; i < fixedRateMarkets.length; i++) {
```

```
14 ./QodaLens.sol:540:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
15 ./QodaLens.sol:554:     for (uint i = 0; i < assets.length; i++) {
16 ./QodaLens.sol:566:     for (uint i = 0; i < assets.length; i++) {
17 ./QodaLens.sol:582:     for (uint i = 0; i < assets.length; i++) {
18 ./QuoteManager.sol:341:     for (uint i=0; i < accountQuotes.length
↳ ; i++) {
19 ./QollateralManager.sol:636:     for (uint i = 0; i <
↳ _iterableCollateralAddresses[account].length + newTokenOffset; i
↳ ++){
20 ./QollateralManager.sol:735:     for (uint i = 0; i <
↳ _iterableAccountMarkets[account].length + newMarketOffset; i++) {
21 ./QAdmin.sol:685:     for (uint i = 0; i < contractsAddr.length; i
↳ ++) {
22 ./QAdmin.sol:707:     for (uint i = 0; i < operationIds.length; i
↳ ++) {
23 ./LiquidityEmissionsQontroller.sol:378:     for (uint i = 0; i <
↳ _userInfo[account].rewardTokenList.length; i++) {
```

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:

SOLVED: The Qoda Finance team solved the issue by using unchecked block.

4.18 (HAL-18) 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 16

```
./QodaLens.sol:486:     for (uint i = 0; i < asset.maturities.
↳ length; i++) {
./QodaLens.sol:518:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
./QodaLens.sol:527:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
./QodaLens.sol:540:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
./QodaLens.sol:554:     for (uint i = 0; i < assets.length; i++) {
./QodaLens.sol:566:     for (uint i = 0; i < assets.length; i++) {
./QodaLens.sol:582:     for (uint i = 0; i < assets.length; i++) {
./QuoteManager.sol:341:     for (uint i=0; i < accountQuotes.length
↳ ; i++) {
./QollateralManager.sol:636:     for (uint i = 0; i <
```

```
↳ _iterableCollateralAddresses[account].length + newTokenOffset; i
↳ ++){
10 ./QollateralManager.sol:735:    for (uint i = 0; i <
↳ _iterableAccountMarkets[account].length + newMarketOffset; i++) {
11 ./QAdmin.sol:685:    for (uint i = 0; i < contractsAddr.length; i
↳ ++) {
12 ./QAdmin.sol:707:    for (uint i = 0; i < operationIds.length; i
↳ ++) {
13 ./LiquidityEmissionsQontroller.sol:378:    for (uint i = 0; i <
↳ _userInfo[account].rewardTokenList.length; i++)
14 ./QodaLens.sol:486:    for (uint i = 0; i < asset.maturities.
↳ length; i++) {
15 ./QodaLens.sol:518:    for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
16 ./QodaLens.sol:527:    for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
17 ./QodaLens.sol:540:    for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
18 ./QodaLens.sol:197:    for (uint i = 0; i < assetAddresses.length;
↳  i++) {
19 ./QodaLens.sol:205:    for (uint i = 0; i < assetAddresses.length;
↳  i++) {
```

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:

SOLVED: The Qoda Finance team solved the issue by caching array length.

4.19 (HAL-19) 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 17

```
./QodaLens.sol:486:    for (uint i = 0; i < asset.maturities.length; i++) {
./QodaLens.sol:518:    for (uint i = 0; i < fixedRateMarkets.length; i++) {
./QodaLens.sol:527:    for (uint i = 0; i < fixedRateMarkets.length; i++) {
./QodaLens.sol:540:    for (uint i = 0; i < fixedRateMarkets.length; i++) {
./QodaLens.sol:554:    for (uint i = 0; i < assets.length; i++) {
./QodaLens.sol:566:    for (uint i = 0; i < assets.length; i++) {
./QodaLens.sol:582:    for (uint i = 0; i < assets.length; i++) {
./QuoteManager.sol:341:    for (uint i=0; i < accountQuotes.length; i++) {
./QollateralManager.sol:636:    for (uint i = 0; i < _iterableCollateralAddresses[account].length + newTokenOffset; i++){
./QollateralManager.sol:735:    for (uint i = 0; i < _iterableAccountMarkets[account].length + newMarketOffset; i++) {
./QAdmin.sol:685:    for (uint i = 0; i < contractsAddr.length; i++) {
./QAdmin.sol:707:    for (uint i = 0; i < operationIds.length; i++) {
./LiquidityEmissionsQontroller.sol:378:    for (uint i = 0; i < _userInfo[account].rewardTokenList.length; i++)
./QodaLens.sol:486:    for (uint i = 0; i < asset.maturities.length; i++) {
```

```
15 ./QodaLens.sol:518:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
16 ./QodaLens.sol:527:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
17 ./QodaLens.sol:540:     for (uint i = 0; i < fixedRateMarkets.
↳ length; i++) {
18 ./QodaLens.sol:197:     for (uint i = 0; i < assetAddresses.length;
↳  i++) {
19 ./QodaLens.sol:205:     for (uint i = 0; i < assetAddresses.length;
↳  i++) {
```

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 Qoda Finance team acknowledged the issue.

4.20 (HAL-20) UNUSED ADDRESS VARIABLE - INFORMATIONAL (0.0)

Description:

During the code review, It has been noticed that address variable is not used and deprecated.

Code Location:

QollateralManager.sol#L35

Listing 18

```
1   /// @notice Native Token (WETH equivalent) for convenience
2   /// @custom:deprecated to be replaced by _qAdmin.WETH()
3   address constant _NATIVE_TOKEN_ADDRESS = 0
↳ x0000000000000000000000000000000000000802;
```

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 removing redundant address.

Remediation Plan:

SOLVED: The Qoda Finance team solved the issue by removing the variable.

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:

```
INFO:Detectors:
Reentrancy in FixedRateMarket._createFixedRateLoan(uint8,address,address,uint256,uint256,uint256,uint64,bool) (contracts/FixedRateMarket.sol#717-810):
        External calls:
        - _checkRatioAndAddParticipatedMarket(borrower,lender,amountFV) (contracts/FixedRateMarket.sol#768)
                - qm._addAccountMarket(lender,currentMarket) (contracts/FixedRateMarket.sol#830)
                - qm._addAccountMarket(borrower,currentMarket) (contracts/FixedRateMarket.sol#833)
        - _repayBorrow(borrower,_qToken.balanceOf(borrower),false,true) (contracts/FixedRateMarket.sol#774)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol#122)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#135)
                - _qToken.burn(account,amount) (contracts/FixedRateMarket.sol#855)
                - Utils.transferTokenOrETH(account,address(this),amount,_underlying,_qAdmin.WETH(),isPaidInETH,false) (contracts/FixedRateMarket.sol#863)
                - weth.deposit{value: amount}() (contracts/libraries/Utils.sol#50)
                - underlying.safeTransfer(receiver_,amount) (contracts/libraries/Utils.sol#55)
                - underlying.safeTransferFrom(sender_,receiver_,amount) (contracts/libraries/Utils.sol#57)
                - weth_scope_0.withdraw(amount) (contracts/libraries/Utils.sol#63)
                - (success) = receiver.call{value: amount}() (contracts/libraries/Utils.sol#64)
        - Utils.transferTokenOrETH(lender,borrower,amountFV,_underlying,_qAdmin.WETH(),isPaidInETH && lenderInitiate,isPaidInETH && ! lenderInitiate) (contracts/FixedRateMarket.sol#786)
        - Utils.transferTokenOrETH(lender,address(feq),protocolFee_ * 2,_underlying,_qAdmin.WETH(),isPaidInETH && lenderInitiate,false) (contracts/FixedRateMarket.sol#789)
        - Utils.transferTokenOrETH(lender,borrower,amountFV - protocolFee_,_underlying,_qAdmin.WETH(),isPaidInETH && lenderInitiate,isPaidInETH && ! lenderInitiate) (contracts/FixedRateMarket.sol#790)
        - feq.receiveFees{_underlying,protocolFee_ * 2) (contracts/FixedRateMarket.sol#793)
        - _qToken.mint(lender,amountFV) (contracts/FixedRateMarket.sol#798)
        - _repayBorrow(lender,_qToken.balanceOf(lender),false,true) (contracts/FixedRateMarket.sol#801)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol#122)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#135)
                - _qToken.burn(account,amount) (contracts/FixedRateMarket.sol#855)
                - Utils.transferTokenOrETH(account,address(this),amount,_underlying,_qAdmin.WETH(),isPaidInETH,false) (contracts/FixedRateMarket.sol#863)
                - weth.deposit{value: amount}() (contracts/libraries/Utils.sol#50)
                - underlying.safeTransfer(receiver_,amount) (contracts/libraries/Utils.sol#55)
                - underlying.safeTransferFrom(sender_,receiver_,amount) (contracts/libraries/Utils.sol#57)
                - weth_scope_0.withdraw(amount) (contracts/libraries/Utils.sol#63)
                - (success) = receiver.call{value: amount}() (contracts/libraries/Utils.sol#64)
        External calls sending eth:
        - _repayBorrow(borrower,_qToken.balanceOf(borrower),false,true) (contracts/FixedRateMarket.sol#774)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#135)
                - weth.deposit{value: amount}() (contracts/libraries/Utils.sol#50)
                - (success) = receiver.call{value: amount}() (contracts/libraries/Utils.sol#64)
        - _repayBorrow(lender,_qToken.balanceOf(lender),false,true) (contracts/FixedRateMarket.sol#801)
                - (success,returndata) = target.call{value: value}(data) (lib/openzeppelin-contracts/contracts/utils/Address.sol#135)
                - weth.deposit{value: amount}() (contracts/libraries/Utils.sol#50)
                - (success) = receiver.call{value: amount}() (contracts/libraries/Utils.sol#64)
        State variables written after the call(s):
        - _repayBorrow(lender,_qToken.balanceOf(lender),false,true) (contracts/FixedRateMarket.sol#801)
                - _accountBorrows[account] -= amount (contracts/FixedRateMarket.sol#868)
        FixedRateMarket._accountBorrows (contracts/FixedRateMarket.sol#54) can be used in cross function reentrancies:
        - FixedRateMarket.accountBorrows(address) (contracts/FixedRateMarket.sol#406-408)
```

```
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/strange_setters.md
INFO:Detectors:
Function FixedRateMarket.onlyAdmin() (contracts/FixedRateMarket.sol#112-117) is an unprotected initializer.
Function FixedRateMarket.onlyQToken() (contracts/FixedRateMarket.sol#119-124) is an unprotected initializer.
Function FixedRateMarket.nonReentrant() (contracts/FixedRateMarket.sol#142-156) is an unprotected initializer.
Function LiquidityEmissionsQontroller.onlyAdmin() (contracts/LiquidityEmissionsQontroller.sol#71-76) is an unprotected initializer.
Function LiquidityEmissionsQontroller.onlyMarket() (contracts/LiquidityEmissionsQontroller.sol#78-83) is an unprotected initializer.
Function LiquidityEmissionsQontroller.whenNotPaused(uint256) (contracts/LiquidityEmissionsQontroller.sol#86-91) is an unprotected initializer.
Function QAdmin.initCollateralRatio() (contracts/QAdmin.sol#859-861) is an unprotected initializer.
Function QAdmin.initCollateralRatio(address) (contracts/QAdmin.sol#863-865) is an unprotected initializer.
Function QAdmin.onlyAdmin() (contracts/QAdmin.sol#158-163) is an unprotected initializer.
Function QPriceOracle.onlyAdmin() (contracts/QPriceOracle.sol#32-37) is an unprotected initializer.
Function QToken.onlyMarket() (contracts/QToken.sol#58-63) is an unprotected initializer.
Function QToken.whenNotPaused(uint256) (contracts/QToken.sol#66-71) is an unprotected initializer.
Function QToken.nonReentrant() (contracts/QToken.sol#76-90) is an unprotected initializer.
Function QollateralManager.initCollateralRatio() (contracts/QollateralManager.sol#478-480) is an unprotected initializer.
Function QollateralManager.initCollateralRatio(address) (contracts/QollateralManager.sol#485-487) is an unprotected initializer.
Function QollateralManager.onlyMarket() (contracts/QollateralManager.sol#88-93) is an unprotected initializer.
Function QollateralManager.whenNotPaused(uint256) (contracts/QollateralManager.sol#96-101) is an unprotected initializer.
Function QollateralManager.nonReentrant() (contracts/QollateralManager.sol#106-120) is an unprotected initializer.
Function QuoteManager.onlyMarket() (contracts/QuoteManager.sol#58-63) is an unprotected initializer.
Function QuoteManager.whenNotPaused(uint256) (contracts/QuoteManager.sol#68-73) is an unprotected initializer.
Function TradingEmissionsQontroller.onlyMarket() (contracts/TradingEmissionsQontroller.sol#85-90) is an unprotected initializer.
Function TradingEmissionsQontroller.whenNotPaused(uint256) (contracts/TradingEmissionsQontroller.sol#93-98) is an unprotected initializer.
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/unprotected_initialize.md
```

```
INFO:Detectors:
FixedRateMarket._repayBorrow(address,uint256,bool,bool) (contracts/FixedRateMarket.sol#843-874) uses a dangerous strict equality:
        - amount == 0 (contracts/FixedRateMarket.sol#849)
QodaLens.getEstimatedAPR(IFixedRateMarket,address,uint256,uint8,uint8) (contracts/QodaLens.sol#313-408) uses a dangerous strict equality:
        - denom == 0 (contracts/QodaLens.sol#395)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in LiquidityEmissionsQontroller._claimEmissions(address) (contracts/LiquidityEmissionsQontroller.sol#377-396):
        External calls:
        - IERC20(rewardTokenAddress).safeTransfer(account,emission) (contracts/LiquidityEmissionsQontroller.sol#388)
        State variables written after the call(s):
        - _userInfo[account].pendingReward[rewardTokenAddress] = 0 (contracts/LiquidityEmissionsQontroller.sol#385)
        LiquidityEmissionsQontroller._userInfo (contracts/LiquidityEmissionsQontroller.sol#61) can be used in cross function reentrancies:
        - LiquidityEmissionsQontroller._claimEmissions(address) (contracts/LiquidityEmissionsQontroller.sol#377-396)
        - LiquidityEmissionsQontroller._updateRewards(IFixedRateMarket,uint8,uint64) (contracts/LiquidityEmissionsQontroller.sol#299-335)
        - LiquidityEmissionsQontroller.pendingReward(address,address) (contracts/LiquidityEmissionsQontroller.sol#259-261)
        - delete _userInfo[account].rewardTokenList (contracts/LiquidityEmissionsQontroller.sol#395)
        LiquidityEmissionsQontroller._userInfo (contracts/LiquidityEmissionsQontroller.sol#61) can be used in cross function reentrancies:
        - LiquidityEmissionsQontroller._claimEmissions(address) (contracts/LiquidityEmissionsQontroller.sol#377-396)
        - LiquidityEmissionsQontroller._updateRewards(IFixedRateMarket,uint8,uint64) (contracts/LiquidityEmissionsQontroller.sol#299-335)
        - LiquidityEmissionsQontroller.pendingReward(address,address) (contracts/LiquidityEmissionsQontroller.sol#259-261)
Reentrancy in FixedRateMarket._liquidateBorrow(address,uint256,IERC20,bool) (contracts/FixedRateMarket.sol#886-946):
        External calls:
        - Utils.transferTokenOrETH(msg.sender,address(this),amount,_underlying,_qAdmin.WETH(),isPaidInETH,false) (contracts/FixedRateMarket.sol#933)
        State variables written after the call(s):
        - _accountBorrows[borrower] -= amount (contracts/FixedRateMarket.sol#936)
        FixedRateMarket._accountBorrows (contracts/FixedRateMarket.sol#54) can be used in cross function reentrancies:
        - FixedRateMarket.accountBorrows(address) (contracts/FixedRateMarket.sol#406-408)
Reentrancy in QToken._redeemQTokensByRatio(uint256,bool) (contracts/QToken.sol#202-236):
        External calls:
        - _burn(msg.sender,amount) (contracts/QToken.sol#219)
                - _fixedRateMarket._repayBorrowInQToken(to,amount) (contracts/QToken.sol#342)
        State variables written after the call(s):
        - _tokensRedeemed[msg.sender] += amount (contracts/QToken.sol#222)
        QToken._tokensRedeemed (contracts/QToken.sol#29) can be used in cross function reentrancies:
        - QToken._redeemableQTokens(address) (contracts/QToken.sol#243-251)
        - QToken._transferFrom(address,address,uint256) (contracts/QToken.sol#366-386)
        - QToken.tokensRedeemed(address) (contracts/QToken.sol#186-188)
        - QToken.tokensRedeemed() (contracts/QToken.sol#179-181)
        - _tokensRedeemedTotal += amount (contracts/QToken.sol#223)
        QToken._tokensRedeemedTotal (contracts/QToken.sol#32) can be used in cross function reentrancies:
        - QToken._redeemableQTokensByRatio(uint256) (contracts/QToken.sol#256-262)
        - QToken.tokensRedeemedTotal() (contracts/QToken.sol#191-193)
Reentrancy in FixedRateMarket._repayBorrow(address,uint256,bool,bool) (contracts/FixedRateMarket.sol#843-874):
        External calls:
        - _qToken.burn(account,amount) (contracts/FixedRateMarket.sol#855)
        - Utils.transferTokenOrETH(account,address(this),amount,_underlying,_qAdmin.WETH(),isPaidInETH,false) (contracts/FixedRateMarket.sol#863)
        State variables written after the call(s):
        - _accountBorrows[account] -= amount (contracts/FixedRateMarket.sol#868)
        FixedRateMarket._accountBorrows (contracts/FixedRateMarket.sol#54) can be used in cross function reentrancies:
        - FixedRateMarket.accountBorrows(address) (contracts/FixedRateMarket.sol#406-408)
Reentrancy in LiquidityEmissionsQontroller._setMarketInfo(address,address,uint256,uint256) (contracts/LiquidityEmissionsQontroller.sol#122-132):
        External calls:
        - marketInfo.rewardToken.safeTransferFrom(msg.sender,address(this),allocation) (contracts/LiquidityEmissionsQontroller.sol#128)
        State variables written after the call(s):
        - marketInfo.rewardPerSec = rewardPerSec  (contracts/LiquidityEmissionsQontroller.sol#129)
```

- 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