QuillAudits

Audit Report
September, 2021

For

Nord Advisory

Contents

Overview	01
Techniques and Methods	03
Issue Categories	04
Functional Testing Results	05
Issues Found	06
Closing Summary	13
Disclaimer	14

Overview

Nord Advisory
Repository: https://github.com/nordfinance/nordadvisory-v1/tree/development
Branch: Development
Commit: 2cd99074d251bf4d41cebf09d85a0827b4d0f845
Fixed In: bce8a39f19f4eecf2a2ebfbbe59021aa28633a30

Scope of the Audit

The scope of this audit was to analyse Nord Advisory smart contract's codebase for quality, security, and correctness.

Vault.sol	Vault is the user-facing smart contract. User deposits a stable token and receives the fund units (Shares) in proportion to the deposits. Users can withdraw assets anytime. Vault will handle the different fees such as Fund management(on exit), Performance fees(charged on gains monthly or at withdrawal) by maintaining HWM (High watermark) for each user. Positive HWM means the user has made some profit.
FeeHandler.sol	Handles all the calculations related to different fees. It contains multiple feesNumerators (protocol, redistribution, treasury, buyback). withdrawFeeNumerator is used to combine all the fees to be charged at times of withdrawal. Then it'll be shared across different numerators such as redistribution, treasury & buyback. FeeHandler maintains a record of the user's HWM. HWM is the share price or price per unit for the fund units. HWM is recorded/updated at each deposit with Weighted Average Price approach. On withdrawal, if the user is in profit (unit price > user's HWM), then the user is charged performance fees on that profit. FeeHandler also keeps a record of the total balance the user has deposited. This is done to ensure that the user doesn't deposit more than the designated cap.

Controller.sol	Provides a unified interface to interact with vault and Strategy. Also, it provides different access control methods over vault, feehandler, strategy contracts.
FundDivisionStrategy.sol	Handles buying/selling of assets based on the configuration provided. A fund manager can configure asset % in the fund and can rebalance the fund anytime.
PriceOracle.sol	Provides the latest price of each asset from chainlink aggregator contracts.
UniswapV2Exchange Adapter.sol	Provides calldata for swapExactTokensForTokens method to trade assets with Quickswap (which is a uniswap fork) on polygon

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return Boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

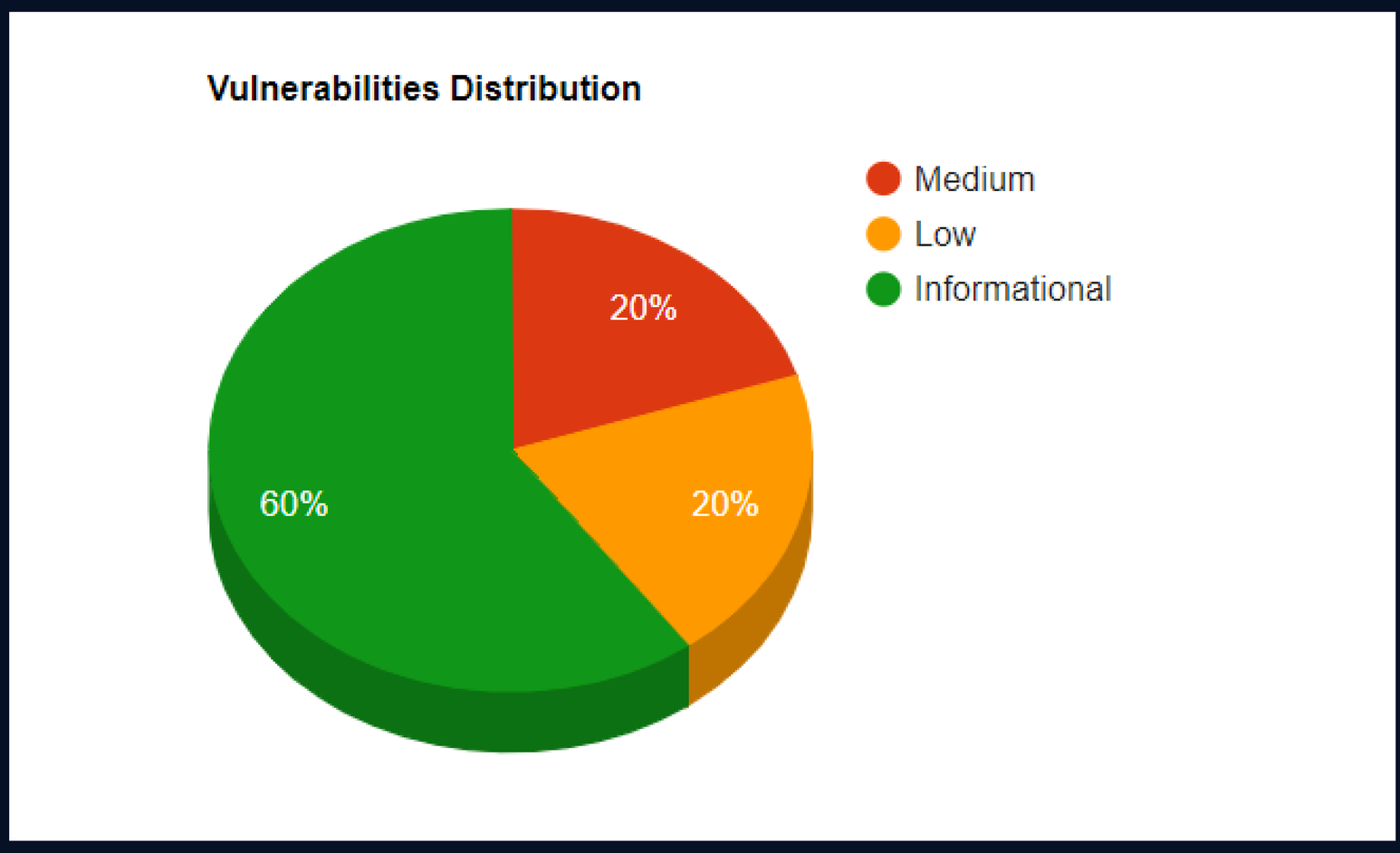
Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues to be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they should still be fixed.
Low	Low level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	1	1	3
Closed	1	2	2	2



Functional Testing Results

The complete functional report is attached below:
Nord Advisory Functional TestCase Report

Issues Found

High severity issues

- [FIXED] Improper management of userUnderlyingBalance value may lead to denial of service for a user

[#L627-685] function _deposit() updates userUnderlyingBalance of a user with the stableTokenAmount that it is trying to deposit

```
627         function _deposit(
628             uint256 stableTokenAmount,
629             address sender,
630             address beneficiary
631         ) internal {
632             require(stableTokenAmount > 0, "Cannot deposit stableToken 0");
633             require(beneficiary != address(0), "holder must be defined");
634             require(
635                 IFeeHandler(feeHandlerAddress).setUserUnderlyingBalance(
636                     beneficiary,
637                     (
638                         IFeeHandler(feeHandlerAddress)
639                             .userUnderlyingBalance(beneficiary)
640                             .add(stableTokenAmount)
641                     )
642                 ) <= maxDepositCap,
643                 "Reached max limit of deposit cap"
644             );
```

And restricts users to not deposit more than a set maxDepositCap

But while withdrawing, the withdraw function subtracts the minimum of userUnderlyingBalance and underlyingAmountToWithdraw from userUnderlyingBalance

```
608         if (underlyingAmountToWithdraw > 0) {
609             uint256 userUnderlyingBalance = IFeeHandler(feeHandlerAddress)
610                 .userUnderlyingBalance(_msgSender());
611             userUnderlyingBalance = userUnderlyingBalance.sub(
612                 Math.min(userUnderlyingBalance, underlyingAmountToWithdraw)
613             );
```

Which in some cases, the subtraction may leave some left over balance. Even after withdrawing all the shares, the userUnderlyingBalance may have some balance, which will be taken into consideration for next deposits restricting to not go beyond a set deposit limit.
So, it may lead to unintended DoS for a user, hence restricting the user to deposit to the vault. Some possible scenarios can be:

1. Withdrawing at the same share price, as it was at the time of depositing.
2. Withdrawing at a less share price, compared to what it was at the time of depositing.

Status Update: A condition has been added in pull/12, which checks the user balance(number of shares). If the user withdraws all the balance, the userUnderlyingBalance will be set to 0.

```
609 +            // If user withdraw complete share then reset userUnderlyingBalance to 0
610 +            if (balanceOf(_msgSender()) == 0) {
611 +                userUnderlyingBalance = 0;
612 +            } else {
613 +                userUnderlyingBalance = userUnderlyingBalance.sub(
614 +                    Math.min(userUnderlyingBalance, underlyingAmountToWithdraw)
615 +                );
616 +            }
```

Also, the maxDepositCap, can be increased or decreased as per the requirements.

Medium severity issues

Vault

- **[FIXED] Distributing Fees to wrong Fees Forwarders**

  [#L214-249]Function startSaving() calls accumulateFees() function of vault with an array of feeForwarders as

```
232         IVault(_vault).accumulateFees(
233             [
234                 protocolFeesForwarder,
235                 buybackFeesForwarder,
236                 treasuryFeesForwarder,
237                 fundManagerAddress
238             ]
239         );
```

But the vault distributes fees as:
treasuryFee => feeForwarders[1] which is the buybackFeesForwarder
buybackFee => feeForwarders[2] which is the treasuryFeesForwarder

```
466             if (withdrawalFees > 0) {
467                 if (treasuryFee > 0) {
468                     IERC20Upgradeable(_underlying()).safeTransfer(
469                         feeForwarders[1],
470                         treasuryFee
471                     );
472                     emit TreasuryFeesTransferred(treasuryFee);
473                 }
474 
475                 if (buybackFee > 0) {
476                     IERC20Upgradeable(_underlying()).safeTransfer(
477                         feeForwarders[2],
478                         buybackFee
479                     );
480                     emit BuybackFeesTransferred(buybackFee);
481                 }
482 
```

As a result of which, fees will be distributed to wrong Fees Forwarders

- **[Acknowledged] Bypassing defense mechanism**

  [#L55-73] The contract implements a defense mechanism with modifier defense() with an implemented logic as: If the caller is a Contract then it should be a whitelisted address, else if its an EOA it should be a whitelisted depositor.

```
55      modifier defense() {
56          if (isContract(_msgSender())) {
57              require(
58                  // then the requirement will pass
59                  IController(controller()).whiteList(_msgSender()),
60                  "This smart contract has not been white listed" // make sure that it is on our whiteList.
61              );
62          } else {
63              if (IController(controller()).isDepositWhiteListActive()) {
64                  require(
65                      IController(controller()).whiteListedDepositor(
66                          _msgSender()
67                      ),
68                      "This address has not been whitelisted."
69                  );
70              }
71          }
72          _;
73      }
```

The defense mechanism can be bypassed by a malicious actor by calling the target function in the constructor of a contract. This way, the code size marked by extcodesize will be 0, and the contract will be considered/treated as an EOA and not a contract. This way the actor can skip the whitelist check for the contract, and also, if the deposit white list is not active, then it can bypass the whole defense mechanism.

PriceOracle

- **[FIXED] Negate Price for a token may lead to underflow**

  [#L51-61] function getLatestPriceOfCoin() fetches price of a token from chain link price feed of the corresponding token address.

```
51      function getLatestPriceOfCoin(address tokenAddress)
52          public
53          view
54          override
55          valueNotNullCoin(tokenAddress)
56          returns (int256)
57      {
58          (, int256 price, , , ) = AggregatorV3Interface(feeds[tokenAddress])
59              .latestRoundData();
60          return price;
61      }
```

As the aggregator produces an int value, it may possibly be a negative value. The type casting of negative int to an uint will underflow the uint type, hence may lead to unexpected behaviour and results. Some Appearances, that involves this type casting In FundDivisionStrategy contract.

1. To calculate tokenPrice

```
439     function getTokenPriceAndUnit(address token)
440         internal
441         view
442         returns (uint256, uint256)
443     {
444         uint256 tokenPrice = uint256(
445             IPriceOracle(priceOracle).getLatestPriceOfCoin(token)
446         );
447         uint256 tokenUnit = 10**uint256(ERC20(token).decimals());
448         return (tokenPrice, tokenUnit);
449     }
```

2. To calculate underlyingValue

```
255     function _investedUnderlyingBalance() internal view returns (uint256) {
256         uint256 result = 0;
257         for (uint256 i = 0; i < activeAssets.length; i++) {
258             (uint256 assetPrice, uint256 assetUnit) = getTokenPriceAndUnit(
259                 activeAssets[i]
260             );
261             result = result.add(
262                 assetPrice
263                     .mul(IERC20(activeAssets[i]).balanceOf(address(this)))
264                     .div(assetUnit)
265             );
266         }
267         uint256 underlyingValue = uint256(
268             IPriceOracle(priceOracle).getLatestPriceOfCoin(underlying())
269         );
270         uint256 underlyingUnit = 10**uint256(ERC20(underlying()).decimals());
271         result = result.add(
272             underlyingValue
273                 .mul(IERC20(underlyingERC).balanceOf(address(this)))
274                 .div(underlyingUnit)
275         );
276         return result;
277     }
```

Low level severity issues

PriceOracle

- **[Acknowledged] [#L28-41]Function addInstancesOfCoin()**

  As there are no checks for aggregatorAddress, whether it actually belongs to the corresponding tokenAddress or not, hence a mismatch between the token and its price feed may happen.

UniswapV2ExchangeAdapter

- **[FIXED] Missing Zero Address Validation**

  [#L27-29] function constructor(): Missing zero address check for _router address

- **[FIXED] Missing setter to update router address**

  Status Update: Exchange Adaptor can be replaced.

Informational

- [Acknowledged] Public functions that are never called by the contract should be declared external to save gas.

FundDivisionStrategy

- [FIXED] Unset masterAssetDecimal value may lead to unoperational **rebalancing of assets.**

[#L478-547] function handleAssetRebalance()

```
497                 if (buy) {
498                     if (computedAssetBalance > assetBalance) {
499                         uint256 actualAssetToBuy = (
500                             (computedAssetBalance.sub(assetBalance)).mul(
501                                 underlyingUnit
502                             )
503                         ).div(priceOracleMasterAssetUnit);
```

As masterAssetDecimal is involved in the calculation of actualAssetToBuy as a Divisor. If not set, it will lead to divide by zero issue and as a result rebalancing of assets will not be possible

- **[Acknowledged] Too many operations inside a for loop with unknown upper bound may exceed block gas limit and result in DoS for business critical functions.**

  Some of these business critical functions are:
  [#L282-310] function investAllUnderlying()
  [#L355-372] function withdrawAllAssets()
  [#L377-437] function withdrawToVault()
  [#L478-547] function handleAssetRebalance()

Vault

- [FIXED] Function getEstimatedWithdrawalAmount() does the same calculation for estimatedWithdrawal and realTimeCalculatedValue values

```
285     function getEstimatedWithdrawalAmount(uint256 numberOfShares)
286         public
287         view
288         returns (uint256 estimatedWithdrawal, uint256 realTimeCalculatedValue)
289     {
290         uint256 calculatedSharePrice = getPricePerFullShare();
291         return (
292             numberOfShares.mul(calculatedSharePrice).div(underlyingUnit()),
293             numberOfShares.mul(calculatedSharePrice).div(underlyingUnit())
294         );
295     }
```

- [Acknowledged] Too many operations inside a for loop with unknown upper bound may exceed block gas limit and result in DoS for business critical functions.

Some of these business critical functions are:
[#L687-717] function collectPerformanceFeeForUsers()

Closing Summary

The audit showed several high, medium, low, and informational severity issues. In the end, the majority of the issues were fixed or acknowledged by the Auditee. Some suggestions have also been made to improve the code quality and gas optimisation.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the code. Besides a security audit, please don’t consider this report as investment advice.

Audit Report
August, 2021

For

Nord Advisory

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com