QuillAudits

Audit Report
October, 2023

For

nealthy

Table of Content

Table of Content

Executive Summary

Project Name	Nealthy
Project URL	*https://www.nealthy.com/*
Overview	Nealthy is a VARA regulated crypto asset management company. Nealthy provides on-chain index products for KYC/KYB individuals and institutions to invest in.
Audit Scope	*https://github.com/NNNFTS/nSTBL_YieldManager_QuillAudit/tree/main/contracts*
Contracts in Scope	Branch: Main Contracts: - LoanManager.sol - LoanManagerStorage.sol - TokenLP.sol and interfaces
Commit Hash	7eba366
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated Tools, Functional Testing
Review 1	5th September 2023 - 11th September 2023
Updated Code Received	21st September 2023
Review 2	22nd September 2023 - 28th September 2023
Fixed In	*https://github.com/NNNFTS/nSTBL_YieldManager_QuillAudit/releases/tag/QuillAudit_2.0.0*

Number of Security Issues per Severity

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	1	0	2
Partially Resolved Issues	0	0	1	0
Resolved Issues	1	2	1	1

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Correctness
- Race conditions/front running
- SWC Registry
- 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
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw

Checked Vulnerabilities

- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, 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 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 analyzed 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

A 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 analyzed, 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 behavior 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

Remix IDE, Truffle, Solhint, Mythril, Slither, Solidity static analysis.

Types of Severity

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

High Severity Issues

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 be fixed before moving to a live environment.

Medium Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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 four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - LoanManager

High Severity Issues

A.1 Potential Manipulation of LP Tokens Due to Premature Calculation in **_depositMapleCash** Function

Line

175

Function - _depositMapleCash

```
function _depositMapleCash(uint256 _amount, address _asset, address _pool, address _lpToken, address _poolManager)
    internal
{
    require(isValidDepositAmount(_amount, _pool, _poolManager), "LM: Invalid amount");
    uint256 lpTokens;
    uint256 sharesReceived;
    IERC20Helper(_asset).safeTransferFrom(msg.sender, address(this), _amount);
    IERC20Helper(_asset).safeIncreaseAllowance(_pool, _amount);

    totalAssetsReceived[_asset] += _amount;
    sharesReceived = IPool(_pool).previewDeposit(_amount);
    totalSharesReceived[_asset] += sharesReceived;
    IPool(_pool).deposit(_amount, address(this));
    lpTokens = sharesReceived * 10 ** adjustedDecimals;
    totalLPTokensMinted[address(1USDC)] += lpTokens;
    IERC20Helper(_lpToken).mint(nstblHub, lpTokens);
    emit Deposit(_asset, _amount, lpTokens, sharesReceived);
}
```

POC

```
====================================================
shares calculated for LoanManager.sol contract before calling deposit function:  40
nstbl hub should receive these lptokens based on above calculated shares:  4
====================================================

Now nstbl hub deposited the same amount, but at that point of time the AUM was increased, so it will effect the
shares that LoanManager will be receiving and bcause the lptokens are calculated based on above number,
so lptokens received by nstbl hub will be less

====================================================
shares received in LoanManager.sol contract if AUM increased by 100:  140
lptokens received by nstblhub contract based on before calculated shares:  4
LP tokens that the nstbl hub contract should receive:  14
====================================================
```

POC Test Link

A.1 Potential Manipulation of LP Tokens Due to Premature Calculation in **_depositMapleCash** Function

Description

In the **_depositMapleCash** function, the calculation of LP tokens received by the **nstblHub** contract is based on the **sharesReceived** variable, which is calculated using **IPool(_pool).previewDeposit(_amount)**. This calculation occurs before the actual deposit is made into the **IPool(_pool)** contract. This design can lead to a potential vulnerability where the LP tokens received by the **nstblHub** contract can be manipulated or can vary, as the number of shares received after the **Deposit** function calculation can vary and LPtokens received by the **nstblhub** contract can be of more value or less value than what they deposited.

Remediation

To address this issue and prevent the manipulation of LP tokens, you should calculate the **sharesReceived** and LP tokens after the deposit is made into the **IPool(_pool)** contract.

Status

Fixed

A.2 Missing Redemption Status Check in **deposit** Function

Line

110

Function - deposit

```
function deposit(address _asset, uint256 _amount) public authorizedCaller nonReentrant validAsset(_asset) {
    if (_asset == usdc) {
        _depositMapleCash(_amount, usdc, mapleUSDCPool, address(lUSDC), MAPLE_POOL_MANAGER_USDC);
    } else if (_asset == usdt) {
        _depositMapleCash(_amount, usdt, mapleUSDTPool, address(lUSDT), MAPLE_POOL_MANAGER_USDT);
    }
}
```

Description

In the **deposit** function, when a user deposits assets (USDC or USDT) into the **mapleUSDCPool** or **mapleUSDTPool** contract, there is no check for the current redemption status of LP tokens. The impact of this issue is that when a redemption request is made, the USDC/USDT tokens remain in the pool. Subsequent deposits can affect the number of shares received by the loan manager, potentially leading to receiving less shares.

A.2 Missing Redemption Status Check in **deposit** Function

Remediation

To address this issue and ensure the proper synchronization of LP tokens and shares, you should include a redemption status check before allowing users to deposit assets.

Status

Acknowledged

Medium Severity Issues

A.3 Hard-Coded LP Token Address in **_depositMapleCash** Function Will Cause Inaccurate Logging

Line

181

Function - _depositMapleCash

```
function _depositMapleCash(uint256 _amount, address _asset, address _pool, address _lpToken, address _poolManager)
    internal
{
    require(isValidDepositAmount(_amount, _pool, _poolManager), "LM: Invalid amount");
    uint256 lpTokens;
    uint256 sharesReceived;
    IERC20Helper(_asset).safeTransferFrom(msg.sender, address(this), _amount);
    IERC20Helper(_asset).safeIncreaseAllowance(_pool, _amount);

    totalAssetsReceived[_asset] += _amount;
    sharesReceived = IPool(_pool).previewDeposit(_amount);
    totalSharesReceived[_asset] += sharesReceived;
    IPool(_pool).deposit(_amount, address(this));
    lpTokens = sharesReceived * 10 ** adjustedDecimals;
    totalLPTokensMinted[address(1USDC)] += lpTokens;
    IERC20Helper(_lpToken).mint(nstblHub, lpTokens);
    emit Deposit(_asset, _amount, lpTokens, sharesReceived);
}
```

Description

In the **_depositMapleCash** function, there is an issue related to the incorrect logging of LP tokens minted. Specifically, the line of code **totalLPTokensMinted[address(1USDC)]** hard codes the address **1USDC** to track LP tokens, but there are two different LP tokens in the system. This hard coding can lead to incorrect logging and calculations if this mapping is used in the future.

A.3 Hard-Coded LP Token Address in **_depositMapleCash** Function Will Cause Inaccurate Logging

Remediation

To address this issue and ensure accurate tracking of LP tokens for different assets, you should parameterize the LP token address based on the **_lpToken** parameter provided in the function. E.g. **totalLPTokensMinted[_lpToken] += lpTokens**;

Status

Fixed

A.4 Lack of Admin Change Functionality in LoanManager.sol Contract

Line

0

Function - N/A

N/A

Description

The **LoanManager.sol** contract lacks an administrative change functionality, which is a crucial feature for many smart contracts. This absence means that there is no built-in mechanism to change or update the admin address, potentially posing security and maintenance challenges. Without an admin change functionality, the contract may become locked or non-upgradable, limiting its adaptability to changing circumstances or potential vulnerabilities.

Remediation

To address this issue and enhance the flexibility and security of the LoanManager contract, consider implementing an administrative change functionality. Here are the steps for remediation:

1. **Admin Change Function:** Create a function that allows the current admin to designate a new admin address. Ensure that this function has proper access control checks to prevent unauthorized changes.
2. **Ownership Transfer:** Optionally, consider allowing the admin to transfer ownership of the contract to a new address if needed.

Status

Fixed

A.5 Single-Step Ownership Change in TokenLP.sol Contract Poses Permanent Access Risk

Line

53

Function - setAdmin

```
    function setAdmin(address _admin) public onlyAdmin {
        admin = _admin;
    }
}
```

Description

The **setAdmin** function in the TokenLP.sol contract allows the contract's admin to be changed in a single step. This design presents a significant security risk. If the current admin (owner) mistakenly enters the wrong address while invoking setAdmin, they could permanently lose access to the contract. This scenario can lead to a critical loss of control over contract management and funds.

Remediation

It is a best practice to use a two-step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract. Consider using OpenZeppelin's **Ownable2Step** contract.

Status

Acknowledged

Low Severity Issues

A.6 Missing Zero Check

Line

88

Function - transferFee

LoanManager.constructor(address,address,address,address)._nstblHub (contracts/LoanManager.sol#88) lacks a zero-check on :
- nstblHub = _nstblHub (contracts/LoanManager.sol#89)

LoanManager.constructor(address,address,address,address)._admin (contracts/LoanManager.sol#88) lacks a zero-check on :
- admin = _admin (contracts/LoanManager.sol#90)

LoanManager.constructor(address,address,address,address)._mapleUSDCPool (contracts/LoanManager.sol#88) lacks a zero-check on :
- mapleUSDCPool = _mapleUSDCPool (contracts/LoanManager.sol#91)

LoanManager.constructor(address,address,address,address)._mapleUSDTPool (contracts/LoanManager.sol#88) lacks a zero-check on :
- mapleUSDTPool = _mapleUSDTPool (contracts/LoanManager.sol#92)

LoanManager.setAuthorizedCaller(address)._caller (contracts/LoanManager.sol#386) lacks a zero-check on :
- nstblHub = _caller (contracts/LoanManager.sol#387)

TokenLP.constructor(string,string,address)._admin (contracts/TokenLP.sol#28) lacks a zero-check on :
- admin = _admin (contracts/TokenLP.sol#29)

TokenLP.setLoanManager(address)._loanManager (contracts/TokenLP.sol#49) lacks a zero-check on :
- loanManager = _loanManager (contracts/TokenLP.sol#50)

TokenLP.setAdmin(address)._admin (contracts/TokenLP.sol#53) lacks a zero-check on :
- admin = _admin (contracts/TokenLP.sol#54)

Description

Several critical functions within the contract lack validation checks to ensure that zero addresses are not accepted as input parameters. This oversight can lead to unintended behavior, security vulnerabilities, and potential exploitation if malicious actors pass zero addresses as arguments to these functions.

Remediation

To address this issue and enhance the security of the contract, you should implement zero-address validation checks in critical functions. Here's a recommended remediation approach:

A.6 Missing Zero Check

1. **Require Non-Zero Address:** Add a validation check at the beginning of each critical function to ensure that the input addresses are non-zero. If an input address is zero, the function should revert with an error message.
2. **Input Validation:** Validate the parameters provided to the functions, especially those involving asset transfers or administrative actions, to prevent potential vulnerabilities or loss of assets.

Status

Partially Fixed

A.7 State-changing methods are missing event emissions

Line

386

Function - transferFee

LoanManager.setAuthorizedCaller(address) (contracts/LoanManager.sol#386-388) should emit an event for:
- nstblHub = _caller (contracts/LoanManager.sol#387)

TokenLP.setLoanManager(address) (contracts/TokenLP.sol#49-51) should emit an event for:
- loanManager = _loanManager (contracts/TokenLP.sol#50)

TokenLP.setAdmin(address) (contracts/TokenLP.sol#53-55) should emit an event for:
- admin = _admin (contracts/TokenLP.sol#54)

Description

Critical functions within the contract lack the emission of events. Events play a vital role in providing transparency, enabling external systems to react to changes, and offering a way to track important contract activities. The absence of events can hinder monitoring and auditing efforts, making it difficult to detect and respond to critical contract actions.

Remediation

To address this issue and improve the transparency and auditability of the contract, you should add appropriate event emissions in critical functions. These events should capture essential information about the function's execution, including input parameters and outcomes. Additionally, consider emitting events both before and after critical state changes, where applicable.

Status

Fixed

Informational Issues

A.8: Unlocked pragma (pragma solidity ^0.8.21)

Description

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, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same. Moreover, we strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor the current code base to only use stable features.

Status

Fixed

A.9: Consider using custom errors

Description

Custom errors reduce the contract size and can provide easier integration with a protocol. Consider using those instead of require statements with string error.

Status

Acknowledged

A.10: Change function visibility from public to external

Description

LM Internal Functions(except isValidDepositAmount() function) in LoanManager.sol are never called from within contracts but yet declared public. Their visibility can be made external to save gas.

Status

Acknowledged

General Recommendation

In the case of stable coins, the hierarchy of authority plays a crucial role and our team recommends that the roles with privileges must be given to the accounts with proven authority and functions like “Minting” and “Burning” must be used cautiously because once these functions are called on the mainnet then could be no minting or burning. Hence, it will be an irreversible change.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Should be able to grant Minter, Burner and Asset Protection Roles to accounts.
- ✓ Should be able to Mint and Burn tokens (from the owner’s account as well as from any other account).
- ✓ Should be able to transfer tokens.
- ✓ Should be able to transfer ownership and revert for a zero address.
- ✓ Should revert if transfer amount exceeds balance.
- ✓ Should revert if Minter and Burners don’t have desired roles.
- ✓ Should be able to set a fee receiver account.
- ✓ Should be able to deduct the fee from the transfer amount and transfer it to the fee recipient.
- ✓ Should not deduct fee if sender is owner, receiver is fee recipient, excluded from fee, or the fee percentage amount is not greater than zero.
- ✓ Should be able to freeze and unfreeze accounts and revert if the caller does not have the asset protection role.
- ✓ Owner should be able to stop minting, burning and pause/unpause the contract.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Nealthy. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Nealthy smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Nealthy smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Nealthy to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

$30B
Lines of Code Audited

Follow Our Journey

Audit Report
October, 2023

For

nealthy

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com