QuillAudits

Audit Report
June, 2024

For

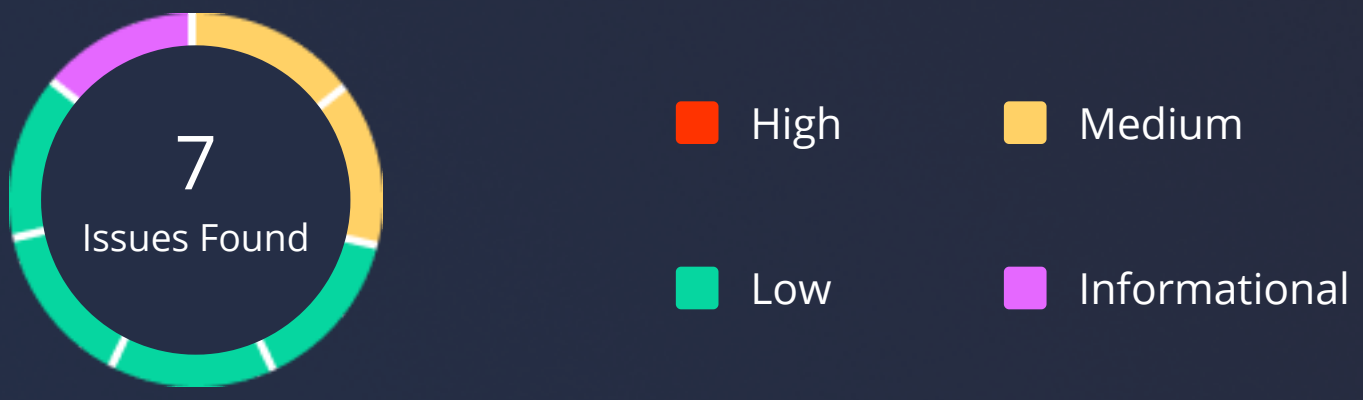
Table of Content

Table of Content

Executive Summary

Project Name	NetWeave
Overview	NetWeave protocol is a fork of well-known compound V2 contracts.
Timeline	27th May 2024 - 10th June 2024
Updated Code Received	11th June 2024
Second Review	11th June 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the NetWeave codebase for quality, security, and correctness.
Source Code	*https://github.com/NetWeaveFi/NetWeave-Finance/blob/main/contracts/RewardDistributor.sol*
Contracts In-Scope	- contracts/Comptroller.sol - contracts/RewardDistributor.sol
Branch	main
Contracts Out of Scope	In-scope contract has been audited by QuillAudits. However, these contracts inherit functionality from out-of-scope Smart contracts that were not audited. Vulnerabilities in unaudited contracts could impact in-scope Smart Contracts functionality. QuillAudits is not responsible for such vulnerabilities. Below are Out of Scope Contracts: - contracts/interfaces/* - OpenZeppelin contracts (Initializable.sol, ...)
Fixed In	*https://github.com/NetWeaveFi/NetWeave-Finance/tree/fix*
Commit	3270adb2bc932bf2e006a823440891ca3d266c8f

Number of Security Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	2	3	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	1	1

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- 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 behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC's 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

Manual Review, Slither, Hardhat.

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 - RewardDistributor

Medium Severity Issues

A1. Access control issues

Description

It was observed that most of the privileged functionality is controlled by the admin. Additional authorization levels are needed to implement the principle of least privilege, also known as least authority, which ensures only authorized processes, users, or programs can access necessary resources or information. Although the contract defines roles responsible for various actions, they can be bypassed by admin. Code Location: The owner can access those functions in Comptroller contract:

- _setRewardDistributor
- _setPriceOracle
- _setCloseFactor
- _setCollateralFactor
- _setMaxAssets
- _setLiquidationIncentive
- _supportMarket

The owner can bypass role-based access control of those functions in Comptroller:

- _setMarketBorrowCaps
- _setMintPaused
- _setBorrowPaused
- _setTransferPaused
- _setSeizePaused

Remediation

To remediate the issue, admin bypassing important issues can be removed. Also using a multi-signature wallet for admin is advised. To further increase the decentralization of the protocol, it is highly encouraged to implement a governance mechanism.

Status

Acknowledged

NetWeave Team's Comment

We will transfer all of owner/admin to timelock and safe multisig.

Low Severity Issues

A2. Use call instead of transfer for eth

Description

In contract RewardDistributor, grantRewardInternal uses transfer for eth transfer on payable user.

The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Remediation

To remediate the issue, please make sure to use call().

Status

Acknowledged

NetWeave Team's Comment

The transfer ERC20 token, not the Ether.

A3. Missing re-entrancy guard for claimReward function

Description

In contract RewardDistributor, claimReward function has external call to transfer via grantRewardInternal function. transfer() itself doesn't cause any re-entrancy related issues but if claimReward cause problem then re-entrancy can occur.

claimRewards(), grantRewardInternal()

```
359     // @audit missing re-entrancy guard
        ftrace | funcSig
360     function claimReward(
361         uint8 rewardType↑,
362         address payable[] memory holders↑,
363         CToken[] memory cTokens↑,
364         bool borrowers↑,
365         bool suppliers↑
366     ) public payable {
367         require(rewardType↑ < rewardAddresses.length, "rewardType is invalid");
368         for (uint256 i = 0; i < cTokens↑.length; i++) {
369             CToken cToken = cTokens↑[i];
370             require(comptroller.isMarketListed(address(cToken)), "market must be listed");
371             if (borrowers↑ == true) {
372                 Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
373                 updateRewardBorrowIndex(rewardType↑, address(cToken), borrowIndex);
374                 for (uint256 j = 0; j < holders↑.length; j++) {
375                     distributeBorrowerReward(rewardType↑, address(cToken), holders↑[j], borrowIndex);
376                     rewardAccruedBorrow[rewardType↑][holders↑[j]] = grantRewardInternal(
377                         rewardType↑,
378                         holders↑[j],
379                         rewardAccruedBorrow[rewardType↑][holders↑[j]]
380                     );
381                 }
382             }
383             if (suppliers↑ == true) {
384                 updateRewardSupplyIndex(rewardType↑, address(cToken));
385                 for (uint256 j = 0; j < holders↑.length; j++) {
386                     distributeSupplierReward(rewardType↑, address(cToken), holders↑[j]);
387                     rewardAccruedSupply[rewardType↑][holders↑[j]] = grantRewardInternal(
388                         rewardType↑,
389                         holders↑[j],
390                         rewardAccruedSupply[rewardType↑][holders↑[j]]
391                     );
392                 }
393             }
394         }
395     }
396
```

```
405     function grantRewardInternal(uint8 rewardType↑, address payable user↑, uint256 amount↑) internal returns (uint256) {
406         address rewardAddress = rewardAddresses[rewardType↑];
407         EIP20Interface reward = EIP20Interface(rewardAddress);
408         uint256 rewardRemaining = reward.balanceOf(address(this));
409         if (amount↑ > 0 && amount↑ <= rewardRemaining) {
410             // @audit use call instead of transfer
411             reward.transfer(user↑, amount↑);
412             return 0;
413         }
414
415         return amount↑;
416     }
417
```

Remediation

To remediate the issue, please make sure to use re-entrancy guard from openzeppelin library.

Status

Resolved

Informational Issues

A4. Functions can be declared external

Description

Multiple functions are declared as public. However, they do not appear to be called from within the contract in which they are defined. Suppose a function is designed to be called by users and is not intended to be accessible internally to other functions. In that case, it is better to declare them as external to reduce the gas cost associated with their execution. Code Location: Following is the list of functions that can be declared as external.

Comptroller.sol

- enterMarkets
- getAccountLiquidity
- getHypotheticalAccountLiquidity
- _setRewardDistributor
- _setPriceOracle
- _setPauseGuardian
- _setMintPaused
- _setBorrowPaused
- _setTransferPaused
- _setSeizePaused
- _become
- getAllMarkets

Remediation

To remediate the issue, above mentioned functions can be set as external which also helps in saving gas.

Status

Resolved

B. Contract - Comptroller

Medium Severity Issues

B1. No check for staleness of the returned data from chainLinkPrice oracle

Description

In contract comptroller, in borrowAllowed() function it is checking for the price if it is 0 but it is not checking for the staleness of the returned price from the latestRoundData() in _getChainLinkPrice function in ChainLinkOraclePrice.sol

borrowAllowed(), _getChainLinkPrice()

```
355     function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint) {
356         // Pausing is a very serious situation - we revert to sound the alarms
357         require(!borrowGuardianPaused[cToken], "borrow is paused");
358
359         if (!markets[cToken].isListed) {
360             return uint(Error.MARKET_NOT_LISTED);
361         }
362
363         if (!markets[cToken].accountMembership[borrower]) {
364             // only cTokens may call borrowAllowed if borrower not in market
365             require(msg.sender == cToken, "sender must be cToken");
366
367             // attempt to add borrower to the market
368             Error err = addToMarketInternal(CToken(msg.sender), borrower);
369             if (err != Error.NO_ERROR) {
370                 return uint(err);
371             }
372
373             // it should be impossible to break the important invariant
374             assert(markets[cToken].accountMembership[borrower]);
375         }
376
377         // @audit checks for 0 value but not for staleness
378         if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
379             return uint(Error.PRICE_ERROR);
380         }
381
382
383         uint borrowCap = borrowCaps[cToken];
384         // Borrow cap of 0 corresponds to unlimited borrowing
385         if (borrowCap != 0) {
386             uint totalBorrows = CToken(cToken).totalBorrows();
387             uint nextTotalBorrows = add_(totalBorrows, borrowAmount);
388             require(nextTotalBorrows < borrowCap, "market borrow cap reached");
389         }
390
391         (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(borrower, CToken(cToken), 0, borrowAmount);
392         if (err != Error.NO_ERROR) {
393             return uint(err);
394         }
395         if (shortfall > 0) {
396             return uint(Error.INSUFFICIENT_LIQUIDITY);
397         }
398
399         // Keep the flywheel moving
400         //Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
401         uint borrowIndex = CToken(cToken).borrowIndex();
402         RewardDistributor(rewardDistributor).updateAndDistributeBorrowerRewardsForToken(cToken, borrower, borrowIndex);
403
404         return uint(Error.NO_ERROR);
405     }
406
```

```
44      function _getChainlinkPrice(IAggregatorV2V3Interface feed) internal view returns (uint) {
45          uint decimalDelta = uint(18).sub(feed.decimals());
46
47          (, int256 answer, , , ) = feed.latestRoundData();
48          if (decimalDelta > 0) {
49              return uint(answer).mul(10 ** decimalDelta);
50          } else {
51              return uint(answer);
52          }
53      }
54
```

This could lead to stale prices according to chainlink documentation:

https://docs.chain.link/data-feeds/historical-data

Remediation

To remediate the issue, please make sure to add check for the staleness of the data that it is actually returning current data.

Status

Acknowledged

Low Severity Issues

B2. Missing reward updation before changing distributor

Description

In contract comptroller, while the function updates new RewardDistributor address in _setRewardDistributor but if old RewardDistributor contract is not empty then proper rewards might not even be distributed.

_setRewardDistributor()

```
832     // @audit what happens to the rewards of the old rewardDistributor
        ftrace | funcSig
833     function _setRewardDistributor(address payable newRewardDistributor↑) public returns (uint) {
834         if (msg.sender != admin) {
835             return uint(Error.UNAUTHORIZED);
836         }
837         (bool success, ) = newRewardDistributor↑.call.value(0)(abi.encodeWithSignature("initialize()", 0));
838         if (!success) {
839             return uint(Error.REJECTION);
840         }
841
842         address oldRewardDistributor = rewardDistributor;
843         rewardDistributor = newRewardDistributor↑;
844
845         emit NewRewardDistributor(oldRewardDistributor, rewardDistributor);
846
847         return uint(Error.NO_ERROR);
848     }
```

Remediation

To remediate the issue, please make sure to updated the rewards before updating the new RewardDistributor.

Status

Acknowledged

B3. Missing check for min/max value on closeFactor

Description

In contract comptroller, in _setCloseFactor() function there should be a check range between values but it is not present which can cause issue that value might go out of mentioned range. The mentioned range in contract is between min: 0.05e18 and max: 0.09e18

_setCloseFactor()

```
82      // closeFactorMantissa must be strictly greater than this value
83      uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05
84
85      // closeFactorMantissa must not exceed this value
86      uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9
87
```

```
878     //  @audit missing check for range
        ftrace | funcSig
879     function _setCloseFactor(uint newCloseFactorMantissa1) external returns (uint) {
880         // Check caller is admin
881         require(msg.sender == admin, "only admin can set close factor");
882
883         uint oldCloseFactorMantissa = closeFactorMantissa;
884         closeFactorMantissa = newCloseFactorMantissa1;
885         emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);
886
887         return uint(Error.NO_ERROR);
888     }
889
```

Remediation

To remediate the issue, please make sure to add require check in _setCloseFactor() function to set values between closeFactorMinMantissa 0.05e18 and closeFactorMaxMantissa 0.09e18.

Status

Acknowledged

Functional Tests Cases

- ✓ claim reward for new user is 0
- ✓ only accrues rewards from when rewards enabled

Upgrade:

- ✓ Admins are able to set pending rewards
- ✓ Pending rewards match actual claim

Comtroller:

Liquidity:

- ✓ Fails if price has not been set
- ✓ Allows to borrow upto a collateral factor

getAccountLiquidity:

- ✓ returns 0 if not 'in' any markets
- ✓ allows entering 3 markets, supplying to 2 and borrowing up to collateralFactor in the 3rd
- ✓ returns collateral factor times dollar amount of tokens minted in a single market

Constructor:

- ✓ on success it sets admin to creator and pendingAdmin is unset
- ✓ on success it sets closeFactor and maxAssets as specified
- ✓ accepts a valid incentive and emits a NewLiquidationIncentive event
- ✓ accepts a valid price oracle and emits a NewPriceOracle event
- ✓ succeeds NewRewardDistributor event
- ✓ fails if factor is set without an underlying price

Functional Tests Cases

RedeemVerify:

- ✓ should allow you to redeem 0 underlying for 0 tokens
- ✓ should allow you to redeem 5 underlyig for 5 tokens
- ✓ should not allow you to redeem 5 underlying for 0 tokens

liquidateCalculateAmountSeize:

- ✓ fails if the repayAmount causes overflow
- ✓ fails if the borrowed asset price causes overflow
- ✓ reverts if it fails to calculate the exchange rate

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 NetWeave codebase. We performed our audit according to the procedure described above.

Issues of Medium, Low and Informational severity were found, suggestions and best practice 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 NetWeave 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 NetWeave 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 NetWeave 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.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
June, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com