QuillAudits

AUDIT REPORT

March, 2025

For

COREPOUND

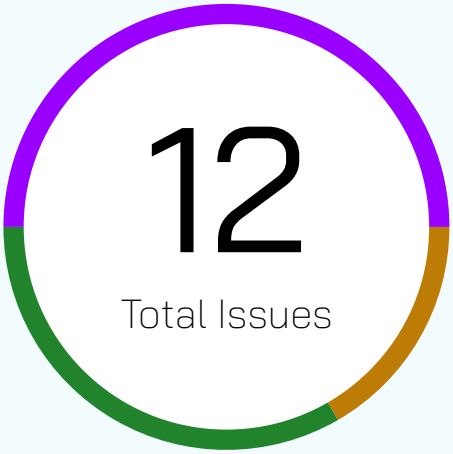
Table of Content

Executive Summary

Project name	Corepound
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 analyse the <Project Name> for quality, security, and correctness.
Project URL	https://compound.finance/
Overview	Corepound Token is a standard ERC20 token contract that mints 10 million total supply to the owner at the point of deployment. One of the utility tokens that will operate in the protocol. CorepoundFarm is specifically designed for liquidity mining in its ecosystem. It allows users to stake tokens into the contract, contributing to the liquidity pool, and in return, they earn reward tokens as an incentive for their participation. CorepoundVault smart contract provides a secure and efficient solution for managing user-deposited assets within a decentralized finance (DeFi) ecosystem. By integrating with multiple DeFi strategies, it maximizes asset yield, offering users an automated and seamless way to earn passive income on their holdings.
Contracts in Scope	CorepoundToken.sol CorepoundVault.sol CorepoundFarm.sol
Commit Hash	5b04f2e407787aa2146ec1c153a58ec4fb7f185e

Audit Scope	The scope of this Audit was to analyze the Corepound Smart Contracts for quality, security, and correctness. https://github.com/Corepound/AuditContracts/tree/main The scope of this Audit was to analyze the Corepound Smart Contracts for quality, security, and correctness. https://github.com/Corepound/AuditContracts/tree/main
Language	Solidity
Blockchain	CORE
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	January 2 - 13, 2025
Updated Code Received	18th February 2025
Review 2	25th February 2025 - 4th March 2025
Fixed In	1decaf0e9330e025475d6423929e46b6306a1cee

Number of Issues per Severity



High	0 (0.00%)
Medium	2 (16.67%)
Low	4 (33.33%)
Informational	6 (50.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	0	**2**	**2**	**5**
Acknowledged	0	0	**2**	0
Partially Resolved	0	0	0	**1**

Checked Vulnerabilities

- ✔ Access Management
- ✔ Arbitrary write to storage
- ✔ Centralization of control
- ✔ Ether theft
- ✔ Improper or missing events
- ✔ Logical issues and flaws
- ✔ Arithmetic Computations 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

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] 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 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, Foundry, Solhint, Mythril, Slither, Solidity statistical 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 Issues

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

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

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.

Medium Severity Issues

CorepoundFarm's asset accounting is not properly handled with weird ERC20 tokens

Acknowledged

Path

CorepoundVault.sol#L206

Function

_deposit()

Description

The vault will have less tokens stored in it if special ERC20 tokens are allowed to interact with the protocol. For example, if a fee-on-transfer token is used in a pool where upon every transfer there is a fee in tokens deducted from the transfer amount, the depositAssets() transaction would always revert if this is the case, and if bypassed, will record the wrong amount of tokens sent to the vault which would affect the user's withdrawal. Users would not be able to withdraw the total amount deposited. USDT on BNB chain is a popular example of a token with fees that can be set on every transfer.

Also, CorepoundStrategy's deposit function (currently out-of-scope) is called with _amount instead of _depositAmount. _depositAmount holds the value obtained from a balance check before and after a deposit - this is a valid way to handle fee-on-transfer tokens to avoid breaking protocol functionality. Instead of calling the strategy contract with a likely inaccurate value, it would be better to use the more precise _depositAmount variable precalculated.

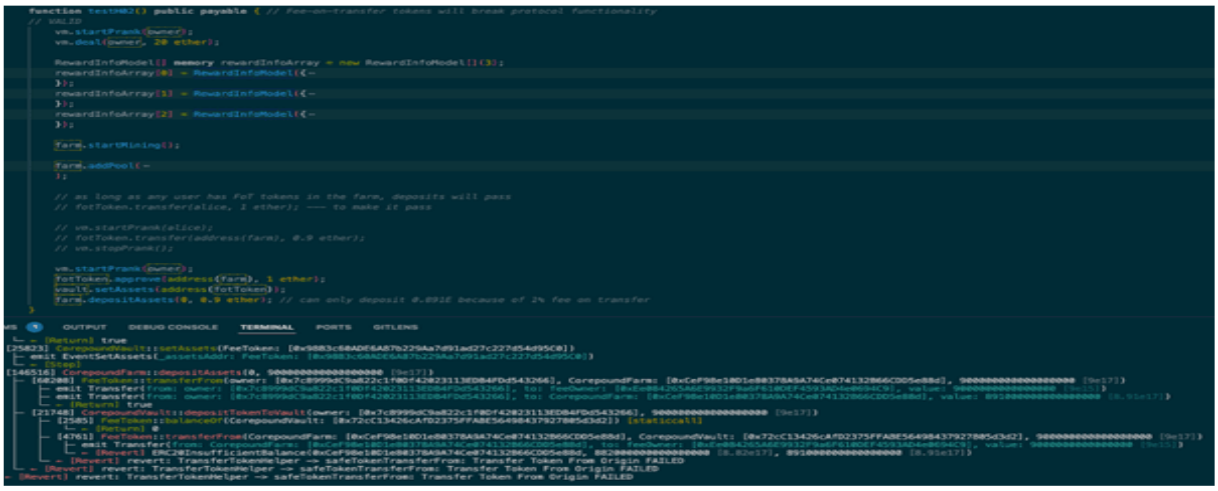
```
uint256 _poolBalance = balance();
...
uint256 _afterPoolBalance = balance();
uint256 _depositAmount = _afterPoolBalance - _poolBalance;

_userInfo.amount = _userInfo.amount + _depositAmount;
totalAssets = totalAssets + _depositAmount;


if (address(strategy) != address(0)) {
    ICorepoundStrategy(strategy).deposit(address(this), _amount);
     // @audit use `depositAmount` instead of _amount
}
```

Recommendation

The protocol can restrict the kinds of tokens used with an allowlist or properly handle token accounting and calculations as described earlier.



Old vault strategy has an unrevoked token approval even after setting to a new vault strategy

Resolved

Path

CorepoundVault.sol#L83

Function

setVaultStrategy

Description

The vault strategy contract is critical to the protocol and the core mechanism that maximizes assets in vaults. The contract owner is the only permissioned user who can invoke the setVaultStrategy and according to the documentation, there are three variants of strategy mechanism. Although, not all vaults will be linked to a strategy contract. On setting a strategy to a vault with ERC20 token, token approval is given to the strategy address. In a scenario where the contract owner intends to change to a new vault, the control flow of this function immediately reset the strategy variable, grant proper approval to the new strategy, send tokens to the new strategy, and then emit. This does not revoke the approval previously given to the old vault strategy, leaving the strategy with a large allowance.

```
/// @notice Set vault strategy
/// @param _strategy Strategy address
ftrace | funcSig
function setVaultStrategy(ICorepoundStrategy _strategy↑) external onlyOwner {
    strategy = _strategy↑;

    if (address(_strategy↑) != address(0)) {
        if (address(assets) != coreAddress) {
            IERC20(assets).approve(address(_strategy↑), 0);
            IERC20(assets).approve(address(_strategy↑), type(uint256).max);
            transferERC20ToStrategy();
        } else {
            transferNativeToStrategy();
        }
    }

    emit EventSetVaultStrategy(address(_strategy↑));
}
```

Recommendation

Revoke the approval given to old vault strategy at the point of setting a new one so that calls from old vault strategy to spend allowance of vault will revert.

```
ftrace | funcSig
function testUnrevokedApprovalForOldVaultStrategy() external {
    ICorepoundStrategy strategy1 = ICorepoundStrategy(makeAddr("Strategy-1"));
    ICorepoundStrategy strategy2 = ICorepoundStrategy(makeAddr("Strategy-2"));

    // vault sets and give uint256 max token approval to strategy 1
    vault.setVaultStrategy(strategy1);
    assertEq(token.allowance(address(vault), address(strategy1)), type(uint256).max);

    // vault resets and give uint256 max token approval to strategy 2
    vault.setVaultStrategy(strategy2);
    assertEq(token.allowance(address(vault), address(strategy2)), type(uint256).max);

    //despite the change of strategy. Strategy 1 still has allowance.
    assertEq(token.allowance(address(vault), address(strategy1)), type(uint256).max);
}
```

Implementation contract can be exploited by attacker if not disabled

Acknowledged

Path

CorepoundVault.sol
CorepoundFarm.sol

Function

_disableInitializers

Description

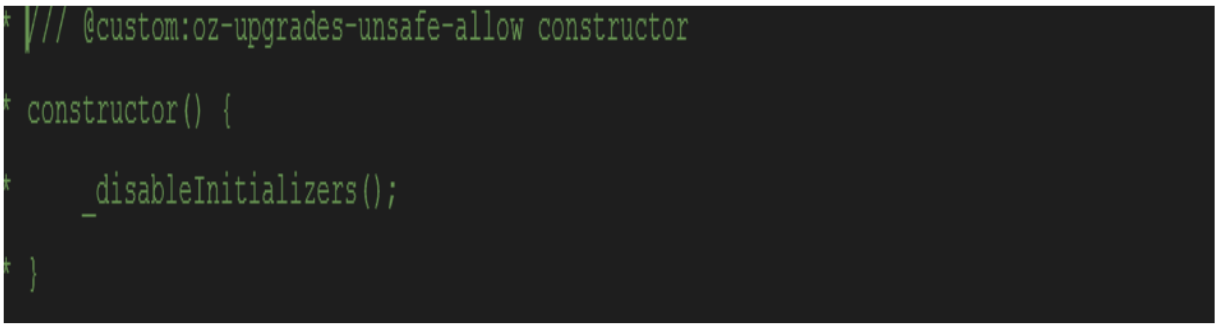
Although the implementation contracts all have the initialize function with proper modifier checks to initialize the proxy, however, an attacker can exploit these implementations because it fails to disable initialization ability for the implementation contracts themselves. According to the caution note in the OpenZeppelin Initializable.sol, it states:

```
* [CAUTION]
* ====
* Avoid leaving a contract uninitialized.
*
* An uninitialized contract can be taken over by an attacker. This applies to both a
proxy and its implementation
* contract, which may impact the proxy. To prevent the implementation contract from
being used, you should invoke
* the {_disableInitializers} function in the constructor to automatically lock it when
it is deployed:
```

Recommendation

Add the _disableInitializers function to the constructor.

Reference: https://forum.openzeppelin.com/t/what-does-disableinitializers-function-mean/28730



```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Resettable vault variables can lead to DOS for users

Resolved

Path

CorepoundVault.sol#L112,120

Function

setAssets(), setCoreAddress()

Description

When the owner gets to update the assets or coreAddress variable in the CorepoundVault, it would lead to a mismatch in function execution from the CorepoundFarm. This is because the variables would only be updated in the vault and not in the farm contract (Farm cannot update the pool's asset address). Vaults are intended to be single asset type constructs, thereby making it counterintuitive to have a setter function for the assets address.

With the configuration in the farm contract, the expected token flow through the conditional statements as well as the non-resettable assets variable would keep calling the functions to deposit native tokens or ERC20 tokens as it has been doing regardless of whether the vault state variables have been updated or not.

Anytime users call depositAssets() --> depositTokensToVault(), it would lead to a revert that cannot be handled. The same happens when users try to withdraw their tokens from the vault, as the addresses have been updated they won't be able to properly match the if...else conditional statements for withdrawNative() and withdraw() leading to their tokens locked in the vault until the owner resets the variables to the initially working values.

Recommendation

Keep the variables in CorepoundVault the same as the CorepoundFarm, and avoid resetting

Low Severity Issues

The ownable library can be removed to reduce contract bytesize

Resolved

Path

CorepoundToken.sol#L4

Function

Ownable()

Description

The CorepoundToken is a vanilla ERC20 implementation of the abstract ERC20 contract. It creates a new token with the name, symbol and supply of the token within its constructor. There is no functionality extended in the CorepoundToken contract via functions within OpenZeppelin's ERC20 library (e.g. minting or burning) thereby making the ownership of the CorepoundToken redundant as the owner will not have any privileged functions.

Recommendation

Remove the Ownable library to reduce contract bytesize upon deployment.

Potential griefing attack to protocol

Acknowledged

Path

CorepoundFarm.sol#L540, CorepoundVault.sol#L169

Function

depositAssets()
depositTokenToVault()

Description

The array stored in the poolUserList mapping could get prohibitively long if an attacker chooses to call depositAssets() with dust amounts (1 wei) repeatedly till it becomes too long to loop over in removeRewardTokenFromPool(). This is because the msg.sender is added to the array without checking for uniqueness. This is also connected to the depositTokenToVault where userList pushes every new depositor to the array.

```
489     function depositAssets(uint256 _pid, uint256 _amount) external payable nonReentrant returns (uint){
539
540         poolUserList[_pid].add(msg.sender);
541
542         emit EventDepositAsset(msg.sender, _pid, _amount);
543         return 0;
544     }
```

Recommendation

Can include a check for address uniqueness, or implement OpenZeppelin's EnumerableSet library which could be a significantly more gas-efficient alternative to looping over hundreds of addresses in an array.

Possible revert due to msg.value not being equal to amount parameters when depositing native assets

Resolved

Path

CorepoundFarm.sol#L508

Function

depositAssets()

Description

When users who intend to deposit native assets to the pool through the depositAssets function, if users pass an msg.value greater than zero, it sums up the msg.value and the incoming _amount parameter and eventually pass this updated _amount value to the depositTokenToVault. This will always revert because an inaccurate value is being sent to the vault. For instance, say users intend to deposit 10 ether worth of CORE to the pool. When they invoke the function and pass the _amount parameter as 10 ether, at L508, this adds up the 10 ether _amount parameter and the 10 ether msg.value to update _amount value to 20 ether. When it reaches L532, it would revert because it sends 20 ether rather than the 10 ether real value being sent from the users.

Recommendation

Add a check that verifies that the amount parameter to be deposited is exactly equal to the msg.value.

Timestamp dependency for reward calculation of pools can be exploited to the benefit of validators

Acknowledged

Path

CorepoundFarm

Description

Using block.timestamp for critical contract logic is not encouraged based on the fact that malicious validators/miners can exploit this to their benefit. In the case of the farm contract that utilizes the block.timestamp for the calculation of users' rewards - the more longer a user stakes into the protocol, the more rewards that is accumulated - miners can keep track of every created pools so they become earlier stakers whose assets starts gathering reward per every seconds that pass. When these malicious users withdraw large amounts, it could impact the proportional rewards. The risk of miners manipulation is very low.

Recommendation

While it is verifiable that the average time for the production of a new block on the CoreDAO network is 3 seconds and sometimes unstable, we recommended that this potential risk of manipulation be considered and switch to block.number if necessary.

Reference:
https://swcregistry.io/docs/SWC-116/
https://ethereum.stackexchange.com/questions/413/can-a-contract-safely-rely-on-block-timestamp

Informational Severity Issues

Add sufficient validation of reward token address inputs in addPool to prevent for-loops in critical functions

Resolved

Path

CorepoundFarm.sol

Function

addPool, depositAssets, withdrawAssets

Description

There is a for-loop present in the depositAssets and withdrawAssets functions that checks that the reward token address for that pool a user is to deposit to or withdraw from is not a null address. This for-loop could have been prevented by adding it to the addPool function when new pools are to be created. This will minimize the amount of gas a user has to pay and avoid for-loops.

Recommendation

Add this for-loop in the addPool function to ensure that all created pools have no reward token with null address.

Comparison with constant boolean value

Resolved

Path

CorepoundFarm.sol#L274

Function

addPool()

Description

The _isNative parameter was compared to the false boolean value which can be used directly.

Recommendation

Use directly in the if condition. In case of falsy value, use !_isNative.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality

Function visibility can be made external to optimize gas efficiency

Resolved

Description

There are functions with a public visibility modifier within the contracts that can be updated to external because they are not used internally. These functions are:

CorepoundFarm.sol
getTotalUserRevenue
getUserDepositedPools
getPoolInfo
getActionUserList
getTotalTvl
setPoolTokenRate
setStartTime
setCoreAddress

Recommendation

Set the function visibility to external.

Remove unused state variable

Resolved

Path

CorepoundFarm.sol#L274

Function

userAddrList

Description

This state variable was declared but was never used in the contract.

Recommendation

Remove the unused variable.

Use cached array length instead of referencing length method from the state array

Partially Resolved

Path

CorepoundFarm.sol

Description

There are a couple of times the pool.rewards.length value was used in performing a for-loop action in the farm contract and it was not modified during these times. For gas-efficient purposes, the length method can be cached to a local variable and used for the loop.

Recommendation

Cache array length instead of directly referencing state in order to save gas.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length

Unnecessary check in setPoolTokenRate

Resolved

Path

CorepoundFarm.sol#L82

Function

setPoolTokenRate()

Description

At no point in time will the incoming _pid parameter be lesser than 0. Moreso, there will exist a pool id of 0 when the first pool is added to the farm.

Recommendation

Remove this check to save gas.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#tautology-or-contradiction

Functional Tests

Some of the tests performed are mentioned below:

- ✔ Only owner can call privileged functions
- ✔ Native tokens and non-native tokens do not get calculated in the same pool
- ✔ Should test weird ERC20 tokens as asset for a pool in farm and vault contracts
- ✔ Should revert when depositing an unrelated token asset to a pool in the farm
- ✔ Should allow permissioned address to add several pools to farm with no duplicate
- ✔ Should successfully deposit required token asset to a pool
- ✔ Should observe reward with longer stake period into the pool
- ✔ Should observe reward accumulation for users after 24 hours of deposited USDT asset to pool
- ✔ Should observe how users rewards are harvested when depositing again
- ✔ Should withdraw all deposited USDT from vault contract through farm contract
- ✔ Should test disableInitializers to lock the possibility of reinitializing
- ✔ Should deposit native tokens to the vault with a value passed in the amount parameter
- ✘ Should handle fee-on-transfer token deposits.

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 Corepound. We performed our audit according to the procedure described above.

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

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the recieved smart contract source code

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

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

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

March, 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com