Compound Comprehensive Protocol Audit

OPENZEPPELIN SECURITY | MARCH 21, 2022

Security Audits

Delivered to Compound on March 4th, 2022. Fix updates will be added to this document periodically.

Table of Contents

Summary

advisory services and monitoring solutions.

The proposal that OpenZeppelin laid out begins with an initial comprehensive audit of the current state of the protocol’s codebase and its primary components. Additionally, the Equilibria team recently submitted a proposal to refactor the `CToken` contract code, one of the main components of the protocol. This refactor was included in the comprehensive audit scope, and the findings are present in this report.

In the following sections, we give an overall summary of how the protocol works and files in scope. We would also encourage the reader to go over previous Compound audits, including past OpenZeppelin reports to learn more about how the protocol has evolved over time.

> **Type:** DeFi Lending & Governance
>
> **Timeline:** 2022-01-24 to 2022-03-04
>
> **Languages:** Solidity
>
> **Total Issues Found:** 30
>
> **Critical Severity Issues:** 1 (already resolved)
>
> **High Severity Issues:** 0
>
> **Medium Severity Issues:** 0
>
> **Low Severity Issues:** 10
>
> **Notes & Additional Information:** 19

Scope

Several parts of the currently deployed codebase have repeated contracts, which slightly differ between one another. Here we present the list of all the files in scope, including repetitions:

```
contracts
├── Comptroller
```

```
│       ├── ComptrollerInterface.sol
│       ├── ComptrollerStorage.sol
│       ├── CToken.sol
│       ├── CTokenInterfaces.sol
│       ├── EIP20Interface.sol
│       ├── EIP20NonStandardInterface.sol
│       ├── ErrorReporter.sol
│       ├── Exponential.sol
│       ├── ExponentialNoError.sol
│       ├── InterestRateModel.sol
│       ├── PriceOracle.sol
│       └── Unitroller.sol
├── CToken-Refactor by Equilibria
│       ├── CErc20.sol
│       ├── CErc20Delegate.sol
│       ├── CEther.sol
│       ├── ComptrollerInterface.sol
│       ├── CToken.sol
│       ├── CTokenInterfaces.sol
│       ├── EIP20Interface.sol
│       ├── EIP20NonStandardInterface.sol
│       ├── ErrorReporter.sol
│       ├── ExponentialNoError.sol
│       └── InterestRateModel.sol
├── GovernorBravoDelegate
│       ├── GovernorBravoDelegate.sol
│       └── GovernorBravoInterfaces.sol
├── GovernorBravoDelegator.sol
│       ├── GovernorBravoDelegator.sol
│       ├── GovernorBravoDelegatorStorage.sol
│       └── GovernorBravoEvents.sol
├── Unitroller
```

Overall Health

Our findings suggest that the protocol is robust in general as we did not detect any high severity issues, and the one critical issue detected has already been fixed. The codebase lacks order and clarity across various updates. The reported issues address this problem on a very low level, so we encourage the community to use our recommendations for starting a conversation on how to address codebase clarity for future development. We would also encourage the Compound Labs team to consider addressing our recommendations in future versions of the protocol being planned.

System Overview

The audited system is composed of three main parts:

- The `Comptroller` and `Unitroller`
- The Governance module (`GovernorBravo` , `Comp` token, and `Timelock`)
- The `CToken` and other CToken-related contracts such as `CErc20` and `CEth`

Unitroller and Comptroller

The `Comptroller` holds the implementation logic of the `Unitroller` contract. The `Unitroller` contract is the proxy and storage layer of the `Comptroller` implementation.

The `Unitroller` contract inherits from the `UnitrollerAdminStorage` contract, which defines four variables of the proxy:

- `admin` : The admin of the proxy, which can modify the comptroller implementation address. This is set in the constructor as the `msg.sender` .
- `pendingAdmin` : The admin of the `Unitroller` can be updated in two steps through the restricted `_setPendingAdmin` and `_acceptAdmin` functions. In the first step, the `pendingAdmin` address is set by the current admin through the `_setPendingAdmin` function, and is then accepted by the pending admin itself through the `_acceptAdmin` function.

implementation address is also updated in two steps through the `_setPendingImplementation` and `_acceptImplementation` functions. The former is called by the admin directly in the `Unitroller` and the latter by the admin through the old implementation as detailed below in the upgrade process section.

To avoid storage collisions, the `Comptroller` storage layout is defined through a series of incremental contracts that extend from the previous ones (namely `ComptrollerV1Storage`, `ComptrollerV2Storage`, etc.). In this way, variable and type declarations never change in order, and new variables are always added in new slots. For this reason, the `Comptroller` storage layout starts with the same variables as the `Unitroller` (`ComptrollerV1Storage` extends from `UnitrollerAdminStorage`), ensuring no collisions occur when calling the base implementation from the proxy. Moreover, all `Unitroller` functions return either `0`, in the case of correct execution, or other numbers according to the `ComptrollerErrorReporter` library.

The `Comptroller` base implementation is deployed at `0xbafe01ff935c7305907c33bf824352ee5979b526` and the `Unitroller` proxy at `0x3d9819210a31b4961b30ef54be2aed79b9c9cd3b`.

NOTE: The `Unitroller` is verified on Etherscan as one source file containing many contract definitions. But `Unitroller` itself only uses the `UnitrollerAdminStorage` and `ComptrollerErrorReporter` contracts. These two dependencies are also used by the `Comptroller` but with different versions of each.

Upgrade process

The upgrade process works as follows:

1. The new `Comptroller` logic implementation is deployed at a new address.
2. The `admin` of the `Unitroller` calls the `_setPendingImplementation` function in the `Unitroller` contract providing the new `Comptroller` implementation address.
3. The `admin` calls the `_become` function in the `Comptroller` contract passing the `Unitroller` address. The `Comptroller` checks that the caller is the `Unitroller`'s admin and calls the `_acceptImplementation` function in the

GovernorBravoDelegator and GovernorBravoDelegate

The `GovernorBravoDelegate` holds, similarly to the `Comptroller` contract, the implementation logic of the Governance module. The proxy, in this case, is the `GovernorBravoDelegator` contract.

The `GovernorBravoDelegate` base implementation is deployed at 0x563a63d650a5d259abae9248dddc6867813d3f87, and the `GovernorBravoDelegator` proxy is deployed at 0xc0da02939e1441f497fd74f78ce7decb17b66529.

Other relevant modules of the governance system are the `COMP` token contract, which is used for the protocol rewards and voting mechanism, and the `Timelock` contract used to queue proposals to be executed. *The `Timelock` contract was left out of scope for the current audit as it was covered in a prior OpenZeppelin audit.*

Upgrade process

The upgrade process for the governance module works as follows:

1. The new `GovernorBravoDelegate` contract is deployed.
2. The `admin` of the `GovernorBravoDelegator` calls the `_setImplementation` function on the `GovernorBravoDelegator` contract
3. The `admin` calls either the `delegateTo` function or the `fallback` function in the `GovernorBravoDelegator` contract, delegating a call to the `initialize` function of the new `GovernorBravoDelegate` contract.
4. The `admin` calls the `_initiate` function, through the proxy and with a delegate call, on the implementation contract, passing the old implementation address for continuous ID count.

CToken

The `CErc20Delegate` contract extends from the `CToken` contract. The `CErc20Delegate` contract is the base implementation contract for each existing cToken.

Upgrade process

The upgrade mechanism of cTokens is out of scope. However, it works similarly to the governance module upgrade process described above.

As of today, there are no existing `cToken`s that use the new base implementation.

Privileged Roles

From now on, we refer to the `Timelock` contract as the `admin`.

In the `Comptroller`, the `admin` can:

- Set the price oracle using the `_setPriceOracle` function.
- Set the `closeFactorMantissa` using the `_setCloseFactor` function.
- Set the `collateralFactorMantissa` for each `Market` using the `_setCollateralFactor` function.
- Set the `liquidationIncentiveMantissa` using the `_setLiquidationIncentive` function.
- Call the `fixBadAccruals` function introduced in Proposal #62.
- Call the `_grantComp` function.
- Set the comp speeds for borrowing and supplying for a list of cTokens by calling the `_setCompSpeeds` function.
- Set the comp speeds for contributors using the `_setContributorCompSpeed` function.
- Add a new market by calling the `_supportMarket` function.
- Bypass the `borrowCapGuardian` and set borrow caps for a market using the `_setMarketBorrowCaps` function.
- Change the `borrowCapGuardian` by calling the `_setBorrowCapGuardian` function.
- Change the `pauseGuardian` by calling the `_setPauseGuardian` function.
- Pause and **UNPAUSE** `mintGuardianPaused` for a specific cToken. Only the `admin` can unpause.
- Pause and **UNPAUSE** `borrowGuardianPaused` for a specific cToken. Only the `admin` can unpause.

Moreover, there is a `pauseGuardian` role which is set by `admin` that can:

- Set `borrowGuardianPaused` for a specific market.
- Set `mintGuardianPaused` for a specific market.
- Pause transfer by flagging `transferGuardianPaused` globally.
- Pause seizing by flagging `seizeGuardianPaused` globally.

Finally, there is a `borrowCapGuardian` that can set specific borrow caps for a list of markets and is also set by the `admin`.

Regarding the governance module, the `admin` can:

- Call the `initialize` function on a new `GovernorBravoDelegate` contract.
- Set a new voting delay through `_setVotingDelay`.
- Set a new voting period through `_setVotingPeriod`.
- Set a proposal threshold through `_setProposalThreshold`.
- Set the expiration time for a whitelisted account through `_setWhitelistAccountExpiration`. This can also be done by the `whitelistGuardian`.
- Set a whitelist guardian through `_setWhitelistGuardian`.
- Call the `_initiate` function.
- Set a new pending admin with `_setPendingAdmin`. The `pendingAdmin` can then call the `_acceptAdmin` function.

Finally, regarding each `CToken` contract, the `admin` can:

- `initialize` a new `CToken` contract.
- Set a pending admin through the `_setPendingAdmin` function. The `pendingAdmin` can then call the `_acceptAdmin` function.
- Set the `comptroller` variable through the `_setComptroller` function.
- Set a new `reserveFactorMantissa` through the `_setReserveFactor` function.
- Reduce reserves by calling `_reduceReserves`.
- Set a new interest rate model through the `_setInterestRateModel` function.

External dependencies and trust assumptions

There are three main parts of the system that depend on external actors or that make trust assumptions:

- **Oracle**: this is meant to retrieve prices of different assets. Well-known oracle attacks include price manipulation and denial of service attacks that make the price unavailable. One thing to note is that Compound has a fallback mechanism in place that should reduce the side effects of manipulation or unavailability. Moreover, when performing sensitive actions, the codebase checks for non-zero prices and reverts or returns early if needed. The oracle system was left out of scope for the current audit, but oracle upgrades may be reviewed in future proposal audits.
- **Assets**: every cToken is coupled with an underlying asset (either Ether or an ERC20 token). Market listings for new assets are proposed by the community through governance proposals. It is very important to note that the contract code of new underlying assets may introduce vulnerabilities that could impact the entire protocol.
- **Interest rate strategy**: The interest rate strategy was left out of scope for this audit. However, it makes an assumption about the number of blocks produced in a year to run calculations on interest quantities. If the protocol is to be deployed on other networks, the same assumption might not be true. Future Ethereum upgrades could also affect this assumption.

Findings

The code has been audited by three auditors over the course of 5 weeks. We present our findings in the following sections organized by severity.

Critical Severity

Anyone can steal money from other suppliers in TUSD market by creating negative interest rates

of the cToken Refactors. The OpenZeppelin team was later provided access to ChainSecurity's private report and took it as an input. After further exploring scenarios, OpenZeppelin not only confirmed Chainsecurity's findings but found that other protocols could be impacted by similar integration issues with TUSD. As a result, OpenZeppelin worked with the TUSD team to fix the issue at the source.

It is important to note that this vulnerability was in code that was out of scope for this audit and would have likely gone unnoticed if not for the excellent work of the ChainSecurity team.

Context

The `exchangeRateStoredInternal` function from the `CToken` contract calculates the exchange rate between an underlying token and its cToken as follows:

$$Rate = \frac{totalCash + totalBorrows - total}{totalSupply}$$

Where:

- `exchangeRate` : the exchange rate between the underlying and the cToken, e.g., TUSD/cTUSD
- `totalCash` : the total amount of underlying held by the cToken contract (calculated by calling `underlying.balanceOf(cToken)`)
- `totalBorrows` : the total amount of underlying borrowed
- `totalReserves` : the total reserves of the market, managed by the protocol that is not intended to be borrowed
- `totalSupply` : the total supply of the cToken minted to suppliers, otherwise known as liquidity providers (LPs)

The `mintFresh` function uses this exchange rate to calculate how many cTokens should be minted to a user that supplies underlying tokens to the market.

Timelock. This function cannot be called for the underlying token. So, for instance, if the CDAI contract holds USDT, anyone can call the `sweepToken` function on the CDAI contract, sending the USDT balance to the Timelock contract. Notably, if anyone calls this function sending the DAI address as the parameter, the call will fail since the underlying cannot be moved to the Timelock

The TUSD market case

Up until recently, the TUSD token had two different entry points: the current implementation, which lives behind a proxy deployed at , and a legacy contract that forwarded calls to the current contract, writing its storage when the `transfer`, `transferFrom`, `increaseApproval`, `decreaseApproval`, and `approve` functions were called. This introduced an undesired behavior: anyone would be able to call the `sweepToken` function of the CTUSD contract sending as the parameter the legacy contract address, effectively moving all the underlying from the CTUSD contract to the Timelock.

The issue lies in the fact that the `totalCash`, in the numerator of the `exchangeRate` formula described above, can be moved to 0 by calling the `sweepToken` function, and given the amount of underlying that is not being used for borrows in the TUSD market (roughly 50% of the TVL), the exchange rate could be moved down by around 50%.

This means that after calling the sweep function, the following would happen:

- The exchange rate, which tracks the borrow rate, and should always be an increasing function, will go down by ~50%
- Any supplier that adds TUSD to the market will receive ~2x the cTUSD amount they should. (A malicious supplier could discover this bug, call the sweep function, and then immediately add liquidity)
- Any supplier that provided liquidity to the market *before* the sweep and then redeems their liquidity after the sweep will receive roughly 50% less of the underlying asset than they should. This would only be possible if suppliers added liquidity at an inflated exchange rate after the sweep. Until the sweep is reversed, they will not be able to redeem any amount
- Even if the Timelock moves the funds back to the CTUSD contract, the relationship between the cTUSD supply and underlying assets can be permanently changed due to overminted cTUSD tokens. The interest rates would then remain negative, ultimately putting the market

The exact amounts can be found in this spreadsheet.

On February 23rd 2022, the TUSD team disallowed forwarded calls from the legacy contract to the current contract by rejecting them from the latter, ultimately fixing the issue.

High Severity

None.

Medium Severity

None.

Low Severity

cEther might fail when repaying a borrow

The `repayBorrow` function of the `CEther` contract calls internally the `repayBorrowInternal` function and the `repayBorrowFresh` function of the `CToken` contract. The `repayBorrow` passes the `msg.value` as the `repayAmount` parameter.

In line 683 of the `CToken` contract, the `repayAmountFinal` is calculated as the total borrow if `repayAmount` equals `type(uint).max`. Otherwise `repayAmountFinal` is set to `repayAmount`.

Then in line 696, the `doTransferIn` function calls back the `CEther` implementation where it checks again that `msg.value == repayAmountFinal`.

This implies the following:
– If an amount of ether equal to `type(uint).max` is passed (Which is unlikely, since it represents an enormous quantity of ETH), then the `repayAmountFinal` which is passed in the `doTransferIn` function will be different from the original `msg.value = repayAmount` and execution will revert.

So to pay an ETH loan in its entirety, one must pass `msg.value` with the exact total borrow amount. If the amount is greater, it will revert. If `msg.value == uint(type).max`, it will also revert but for different reasons.

Consider refactoring the code to avoid checking `type(uint).max` in the `CETH` case as this cannot be passed as `msg.value`. Consider requiring that `repayAmount <= accountBorrowsPrev` instead.

Possible function selector clashing

The upgradeability system implemented in the codebase does not manage function clashing between the proxy contract and the implementation contract.

Clashing can happen among functions with different names. Every function that is part of a contract's public ABI is identified, at the bytecode level, by a 4-byte identifier. This identifier depends on the name and arity of the function, but since it is only 4 bytes, there is a possibility that two different functions with different names may end up having the same identifier. The Solidity compiler tracks when this happens within the same contract, but not when the collision happens across different ones, such as between a proxy and its logic contract.

Upgradeable contract instances (or proxies) work by delegating all calls to a logic contract. However, the proxies need some functions of their own, such as `_setPendingImplementation(address)` and `_acceptImplementation()` to upgrade to a new implementation. This setup means there can be a function in the proxy contract with the same 4-byte identifier as one in the implementation contract. This issue would make it impossible to access the one defined in the implementation contract, as the call will not be handled by the fallback function but by the function with that signature in the proxy contract.

Consider thoroughly testing all functions implemented in each upgradeable contract to ensure no collisions are possible. Alternatively, consider migrating to the EIP-1822: Universal Upgradeable Proxy Standard (UUPS), which, by design, does not have this problem.

Gas inefficiencies

Arbitrary use of different `uint` types can lead to unwanted effects

Numerous unsigned integer (`uint`) values of various sizes are used. Those less than 256-bits (`uint256`) are used extensively. Non- `uint256` sizes are generally chosen to facilitate tight packing inside of `struct` s to save on storage costs. However, projects must carefully weigh the realized gas savings against the additional complexity such a design decision introduces.

Since the Ethereum Virtual Machine operates on 256-bit integers, additional operations must be performed to process non- `uint256` values. These additional operations increase the bytecode size of contracts and consume additional gas during execution.

In the Comp contract, for example, it was necessary to include some specific functions to manage these units safely but causing extra gas costs for the additional operations. Other than that, the variables are not properly packed in the slots of the contract, so the choice of picking these units is not justified.

The code in some loops is executed unnecessarily

In `GovernorBravoDelegate` , the function `cancel` should verify `ProposalState` before calling `cancelTransaction` on the `Timelock` contract to avoid unnecessary gas spending. If the proposal has not been added to the queue, it is not necessary to enter the loop to cancel the transactions, since they have not even been added to the `Timelock` yet.

Some variables are being unnecessarily initialized to their default value

Initializing a variable to its default value wastes gas uselessly. In the codebase, there are variables explicitly set to 0. this happens several times: – In `getPriorVotes` function the local variable `lower` . – In `transferTokens` function the local variable `startingAllowance` . – In `TokenErrorReporter` contract the variable `NO_ERROR` .

The loops of some functions are not properly optimized

In many loops in the codebase, the size of the array to iterate through is always read on each iteration. Here some examples: – `getHypotheticalAccountLiquidityInternal` , `_addMarketInternal` , `fixBadAccruals` and `claimComp` , in `Comptroller` . – `queue` , `execute` and `cancel` in `GovernorBravoDelegate` .

except for the first),

3. if it is a `calldata` array, this is an extra `calldataload` operation (3 additional gas for each iteration except for the first)

These extra costs are avoidable by creating a variable with the array length (caching the array length in stack).

Arguments with read-only parameters are using `memory` instead of `calldata`

Some protocol contract functions have parameters in which they use the keyboard memory. This may not be very efficient as it performs unnecessary steps if the argument is read-only. Here some examples:

- `safe32`, `safe96`, `add96` and `sub96` in `Comp.sol`
- `enterMarkets`, `updateCompBorrowIndex`, `distributeBorrowerComp`, `claimComp`, `_setCompSpeeds` in `Comptroller.sol`
- `propose` in `GovernorBravoDelegate.sol`

Consider using `calldata` instead of `memory` if the function argument is only read.

Storage slots read multiple times

Operations that load values onto the execution stack can be expensive. On several occasions, some variables are read multiple times and cause gas cost overruns in the execution. Here are some examples:

- `doTransferIn` in the CToken contract reads three times the `underlying` state variable.
- `propose` in the GovernorBravoDelegate contract reads three times the `targets.length` memory variable.
- `state` in Governor Bravo Delegate, the local variable `proposal` is unnecessarily defined with the keyword `storage`, causing multiple reads from storage.
- Line 254 of `cToken` contract is defining a `storage` variable when `memory` can be used instead.

Redundant validations

The `exitMarket` function performs a redundant check to verify if the sender is not already *'in'* the market. The internal function `redeemAllowedInternal` called by the former performs the same validation.

Consider removing the redundant validation.

Unnecessary extra steps

`transferTokens` has local variables and validations that are wasting gas without adding any value and reducing readability.

Consider removing intermediate variable definitions to act directly on storage variables and and synthesize the validations.

Redundant usage of the `Exp` type

Usage of `Exp` type can be avoided at lines 308-310 in favour of `uint exchangeRate = cashPlusBorrowsMinusReserves * expScale / _totalSupply;`.

`isPriceOracle` is not used

The `Comptroller` uses a `PriceOracle` contract to retrieve prices for assets. In the current implementation, the `PriceOracle` is just an interface over a `UniswapAnchorView` contract implementation. However, there's a mismatch between what the `PriceOracle` interface should implement and what is actually implemented.
Specifically, the `isPriceOracle` getter is not present in the implementation, making the `PriceOracle` definition useless.

Moreover, when setting a new oracle, there's no check on whether the new implementation implements `isPriceOracle` nor assets price is retrieved to check if the oracle is working.

Consider either refactoring the `PriceOracle` interface or adjusting the current implementation to match its interface. This improves correctness and consistency but will also avoid unexpected call failures.

should be fixed. *Some examples* are:

- In line 2458 of `Unitroller.sol`, the concept of "ComptrollerCore" is mentioned but not used nowhere else in the codebase again
- In line 2460 of `Unitroller.sol`, the comment appears to be either incomplete or the "if" should be removed from the end of the comment
- In line 17 of `Exponential.sol` says "INT" but should say "UINT"
- line 237 of `Comptroller.sol` is incorrect, since `minter` is actually used.
- In line 13 of `PriceOracle.sol` used by the `Comptroller` says that zero means price unavailable. But according to how that function works, this is not true in general. The result can be correctly 0 and be a valid price.
- In line 795 of `CToken.sol` should say "to avoid re-entrancy check"
- In line 291 of `CToken.sol`, should be changed since the function does not return error codes anymore (expect the `NO_ERROR` code in case of success)
- In line 136 of `Comptroller.sol`, "borrower" should be "supplier or borrower". Note that, in this case, the parameter name should also be changed for consistency and accuracy.
- In line 140 of `CToken.sol`, instead of mentioning that `-1` is infinite, it should say that `uint256.max` is the maximum amount allowed

Additionally, some public and external functions throughout the codebase lack documentation. The lack of documentation hinders a reviewers' understanding of the code's intention, an understanding that is essential for correctly assessing both security and correctness. Docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec). Additionally, consider reviewing all existent comments and docstrings to check whether they are complete, descriptive, and accurate, including the examples mentioned above.

Commented out or missing verify calls

- The `transferVerify`, `mintVerify` which have been commented out.
- The `repayBorrowVerify` call which was completely removed from the `cToken` code.

If any of these function calls will be re-activated in the future by changing their current implementation, this `CToken` model will need to be updated too.

To avoid compatibility issues and to independently support any `Comptroller` upgrade, consider adding those function calls back. Alternatively, as the purpose of these lines is unclear and may confuse future developers and external contributors, consider removing them from the codebase. If they are present to provide alternate implementation options, consider extracting them to a separate document where a deeper and more thorough explanation could be included.

Outdated Solidity versions

The version of Solidity used throughout the codebase is outdated, varies between contracts, and is not pinned.

The choice of Solidity version should always be informed by the features each version introduces and that the codebase could benefit from, as well as the list of known bugs associated with each version. Examples of this are:

- Unitroller and Timelock – 0.5.8 – Compiler Bugs.
- Comptroller and COMP Token – 0.5.16 – Compiler Bugs.
- GovernorBravoDelegator and GovernorBravoDelegate – 0.5.17 – Compiler Bugs.
- The new CToken contract uses ^0.8.6 – Compiler Bugs.

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase. Regardless of which version of Solidity is used, consider pinning the version consistently throughout the codebase to prevent the introduction of bugs due to incompatible future releases.

Unreachable code

In some cases, these error codes are still being checked, but the code after the check is unreachable, increases gas cost, and reduces readability.

In particular, the `accrueInterest` function always returns `NO_ERROR` or reverts thus these code blocks are unreachable:

- `mintInternal`
- `redeemInternal`
- `redeemUnderlyingInternal`
- `borrowInternal`
- `repayBorrowInternal`
- `repayBorrowBehalfInternal`
- `liquidateBorrowInternal` (but lines 732-735 should be kept)
- `_setReserveFactor`
- `_addReservesInternal`
- `_reduceReserves`
- `_setInterestRateModel`
- `totalBorrowsCurrent`
- `borrowBalanceCurrent`
- `exchangeRateCurrent`

The following if clauses of the `CToken` contract will never be true since the `accrueInterest` function is always called at the beginning of a mint and will update the `accrualBlockNumber` to be the latest block number.

- `mintFresh`
- `redeemFresh`
- `borrowFresh`
- `repayBorrowFresh`
- `liquidateBorrowFresh` (but lines 761-764 should be kept)
- `_setReserveFactorFresh`
- `_addReservesFresh`
- `_reduceReservesFresh`

run operations.

Unused functions and parameters

In the up-to-date version of the protocol, the `Comptroller` contract has many functions and parameters defined which are deprecated and not used anymore. In particular:

- `mintVerify`, `redeemVerify`, `borrowVerify`, `repayBorrowVerify`, `liquidateBorrowVerify`, `seizeVerify` and `transferVerify` do nothing.
- `isComped`, `maxAssets`, `_mintGuardianPaused`, `_borrowGuardianPaused`, `compSpeeds`, `compRate`, `closeFactorMinMantissa` and `closeFactorMaxMantissa` are not used anymore.

Given the distributed ownership over the `Compound` protocol, the code base is frequently revisited and maintained by many different community members and contributors. Also, the intrinsic storage layout separation pattern and the upgradeability design have important tradeoff consequences over the general readability and quality of the codebase.

We suggest revisiting the current implementation and starting a community-wide conversation over the long-term codebase development to avoid making the error more confused and less readable.

Unused return value

Line 940 of the `Comptroller` contract calls the `isCToken` getter in the `cToken` contract when listing a new market, but it does check the return value, making this check completely useless.

If the called address has no `isCToken` getter and the passed contract is not an EOA, the fallback function will be executed. Moreover, if the answer is false, the market should not be listed.

The unwanted outcome is that `Market`s can be mistakenly added but cannot be removed according to the latest deployed version.

Consider explicitly checking the return value of such calls and eventually revert with any unwanted behaviours or values returned.

If an underlying asset's collateral factor in a specific market is greater than zero and the underlying token is upgraded with malicious code, then it might be possible to set an arbitrarily large underlying balance to drain all other markets and drain them using the underlying asset as collateral. There could be other examples where a malicious upgrade of an underlying token can circumvent the entire protocol's security, and the protocol would be blind to such events.

In the short-term, consider this risk for a new token listing process and add monitoring for compromised or upgraded underlying assets. In the long-term, consider a system design where a specific underlying token can't affect the protocol as a whole to mitigate this specific risk.

Block velocity assumption

The `InterestRateStrategy` makes an assumption on the number of blocks per year to evaluate interests rate.

This will not work for many current L2 solutions or future Ethereum upgrades. Even if this doesn't pose a security issue today, it might be an issue in the future if `Compound` is deployed on many different chains.

Consider adding this to the backlog tasks when planning to deploy the protocol to a new chain.

Code style inconsistencies

Across the codebase there are several places where code has an inconsistent style. Some examples are:

- Inconsistent error handling between line 489 and line 504 in the `Comptroller` contract.
- Mixed docstrings style in the `ComptrollerInterface` used by the `Comptroller`.
- SNAKE_CASE for constants is missing for `quorumVotes` and `proposalMaxOperations` in the `GovernorBravoDelegate` contract.
- Some contracts use a `require` statement for access control and others an if and a custom function called fail that does not revert but returns a number that refers to a type of error.

style with help of linter tools such as Solhint.

Declare `uint` as `uint256`

In the audited contracts, there is a general use of unsigned integer variables declared as `uint`.

To favor explicitness, consider replacing all instances of `uint` to `uint256`.

`doTransferOut` doesn't ensure success operation

The `doTransferOut` function performs an ERC20's transfer operation taking into account that this transfer call may not return anything if the asset is not ERC20 compliant. To manage this, it uses an assembly block that is implemented to check the `returndatasize()` as follows:

- When `returndatasize() = 0`, it is assumed that the asset does not comply with the `ERC20` specification, and set the `success` variable to `true`.
- When `returndatasize() = 32`, it is assumed that the returned value was either `true` or `false`, and set the `success` variable as the returned value.
- When `returndatasize()` equals any other value, it is assumed that the asset is "an excessively non-compliant ERC20", and the function reverts.

The issue lies in the fact that when `returndatasize()` is zero, It cannot be ensured that the transfer succeeded, since the transfer may have silently failed. If the transfer call silently fails, the user's CTokens will be burned but no tokens are going to be transferred back to them.

Even though this does not pose a security risk in any current Compound market, it may cause problems in markets to be added in the future.

Consider adding an additional check to the `doTransferOut` function to evaluate whether the final underlying balance of the CToken is different than its initial underlying balance. Additionally, consider exhaustively reviewing the `transfer` and `transferFrom` functions of any future market addition proposal to check that they cannot silently fail.

Implementation used as interface

happens with the `CToken` contract.

Consider using interfaces instead of contract files to avoid confusion, improve readability, and reduce the codebase size.

Lack of indexed parameters in events

Over the code base, there's inconsistent use of `indexed` parameters for event definitions. Specifically, some event definitions lack completely indexed parameters. Some examples are:

- The `Failure` event of the `ComptrollerErrorReporter`.
- The event definitions of the `Unitroller`.
- Many events of the `GovernorBravoDelegator` contract.
- Many events of the `Comptroller` contract.
- Many events of the `CTokenInterfaces` contract.
- The `NewImplementation` event of the `CDelegatorInterface`.

Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Lack of input validation

In the codebase, there are several places where there's a lack of input validation. Some examples:

- Admin functions of the `Comptroller` lack of input validation.
- The `Unitroller` doesn't have input validation when setting pending implementation or admin.
- When calling `_setMarketBorrowCaps` an array of cTokens is passed to be configured with a borrow cap. However, there is no check on whether each market `isListed` before setting a borrow cap.

Consider reviewing the codebase looking for any places where an input validation might be beneficial. This will reduce attack surface, error propagations and improve overall security.

Lack of SPDX License Identifier

To avoid legal issues regarding copyright and follow best practices, consider adding the SPDX license identifier as suggested by the Solidity documentation.

Constants not declared explicitly

There are several occurrences of literal values with unexplained meaning in the Compound Protocol's contracts. *Some examples* are:

In `Comptroller.sol`: line 176, line 188, line 738, and line 1088.

In `CToken.sol`: line 71, line 405, line 521, line 584, line 670, line 752, and line 833

Literal values in the codebase without an explained meaning make the code harder to read, understand and maintain, thus hindering the experience of developers, auditors, and external contributors alike.

For the examples mentioned above in particular, all `0` constants can be replaced with either `Error.NO_ERROR` or `NO_ERROR` constants. But in general, developers should define a constant variable for every magic value used (including booleans), giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

`cEth` and `cErc20` underlying balances can be manipulated

In the `cErc20` contract, the balance of the underlying asset is calculated by calling the `balanceOf` function from the `EIP20Interface` interface. As a result, if any underlying tokens are sent directly to a `cErc20` contract, bypassing the mint function, the underlying token balance is incremented without minting the corresponding cTokens to the user.

When a market has already a notable amount of liquidity, sending funds directly to these contracts will not be profitable, and will instead mean each cToken holder can claim more tokens for themselves. However in markets where the amount of underlying in the contract is relatively low, The same is true of `cEth`, where an actor can bypass the receive function using self destruct, similarly manipulating the underlying amount.

Consider tracking the total underlying balance in a new variable, and increment/decrement it as appropriate when supplies and borrows happen, to avoid adverse actors from being able to bypass predefined functions and manipulating the market's exchange rate.

Markets can't be unlisted

The current implementation does not allow for a `Market`s `isListed` flag to be turned off. However, markets can become `deprecated` but will stay in the `allMarkets` storage variable.

Moreover, the current implementation often performs `for` loops around the `allMarkets` array. This altogether means that markets can't be effectively unlisted nor removed from the array.

If markets are added in the future or if some get deprecated, they will still incur higher gas prices for normal operations since the calculations will run through all the `markets` array independently.

Consider starting a discussion on how to effectively refactor the code to improve the current design in order to improve general gas cost and optimize storage usage.

Improper imports style

Non-explicit imports are used throughout all protocol contracts. This reduces code readability and could lead to conflicts between names defined locally and the ones imported.

Furthermore, many contracts are defined inside the same Solidity files, making the issue even more significant.

Some examples of multiple contracts and interfaces within the same file are:

- `GovernorBravoEvents`, `GovernorBravoDelegatorStorage`, `GovernorBravoDelegateStorageV1`, `GovernorBravoDelegateStorageV2`, `CDelegationStorage`, `CDelegatorInterface`, `CDelegateInterface` are in CTokenInterfaces.sol

On the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported and avoid having multiple implementations in the same file.

Typos

Accross the codebase there are different typos. Some examples are:

- "contructor" should be "constructor".
- "arity" should be "parity".
- "tather" should be "rather".

Consider correcting these typos and review the codebase to check for more to improve code readability.

Unclear and inconsistent naming

There are some places in the codebase that might benefit from some naming changes:

- The `UnitrollerAdminStorage.sol` not only holds the admin storage variables but also the implementation variables (`comptrollerImplementation`, `pendingComptrollerImplementation`), so naming it `UnitrollerStorage` would be more general and consistent.
- In general, there is no clear and consistent naming system for functions. Some contracts follow the convention that `internal` ones are prefixed with "_", but in other cases this prefix is used for admin functions that can be `public`, `external` or, `internal`. Some examples are:
- In the Comp token contract the prefix is used for `internal` functions although not consistently as `safe32`, `safe96`, `add96`, `sub96` and `getChainId` are `internal` and do not carry the prefix.

Consider being consistent to improve code readability, clarity, and quality. An inconsistent naming system can confuse users and is prone to error. Moreover, consider reviewing the entire codebase for potential other occurrences.

Unnecessary assembly code

Functions `getChainId` and `getChainIdInternal` use assembly code to query for the identifier of the blockchain in which the contract is deployed.

For future implementations, consider using the global variable `block.chainid` to improve readability, reduce execution gas and optimize bytecode size.

Unnecessary checks

Throughout the codebase, there are some unnecessary checks that can be avoided to save gas and improve readability. For example:

- The `_acceptAdmin` function and the `_acceptImplementation` function in `Unitroller.sol` validate that the `msg.sender` is not the zero address (that is, the new admin who calls the function is not the zero address and the address of the new Comptroller implementation is not the zero address), which is an unfeasible scenario, and is anyway validated in both `if` clauses in the first evaluated condition
- The `borrowAllowed` function in `Comptroller.sol` redundantly checks whether the borrower is part of the `accountMembership` mapping, since this flag is properly set in the `addToMarketInternal` function, and if that fails, it will return beforehand.

Wrong or missing visibility in functions and variables

The following functions are defined as `public` but are never locally used:

- in `Comp.sol`: `delegate`, `delegateBySig` and `getPriorVotes`.
- in `Comptroller.sol`: `enterMarkets`, `getAccountLiquidity`, `getHypotheticalAccountLiquidity`, `_setPriceOracle`, `_setPauseGuardian`, `_setMintPaused`, `_setBorrowPaused`,

- in `GovernorBravoDelegate.sol`: `propose`
- all functions in Unitroller contract.

Moreover, in the `ExponentialNoError` contract, all the constants are implicitly using the default visibility.

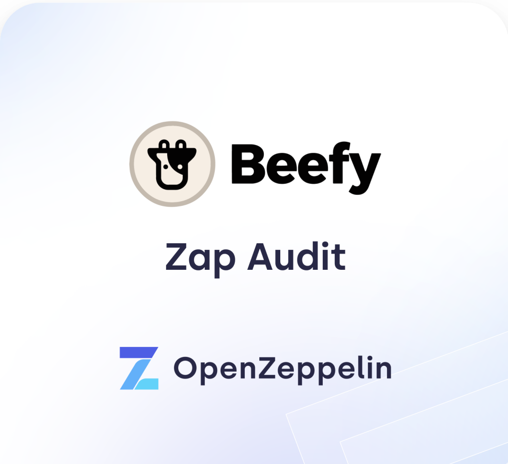
To clarify intent and favor readability, consider explicitly declaring the visibility of all constants and state variables. Also, consider changing the visibility of the aforementioned functions to `external` to reduce gas costs.

Conclusions

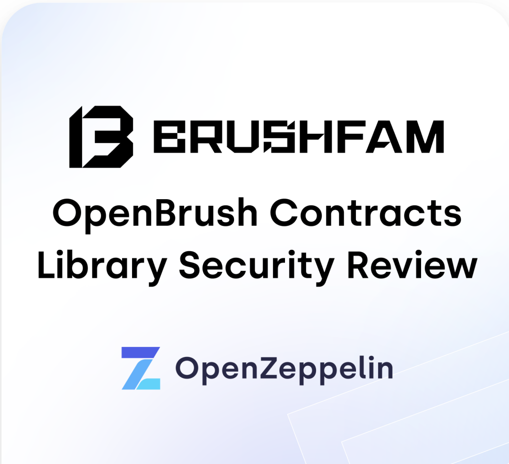
One critical issue was found and resolved alongside numerous issues of lower severity. Recommendations on how to fix each issue have been provided, along with suggestions to improve overall code quality.

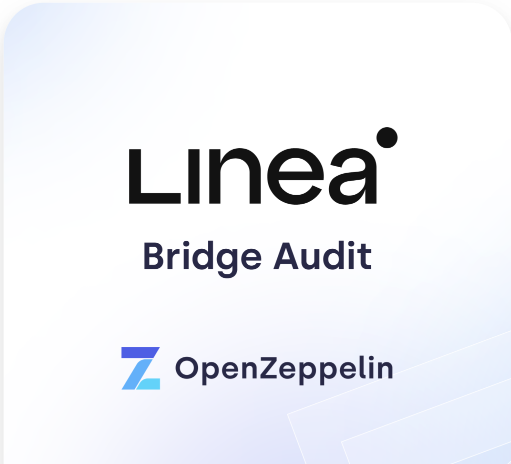
We recommend that the Compound Labs team and Compound community contributors consider addressing our recommendations in future proposal upgrades and protocol versions.

Related Posts



Beefy Zap Audit





Linea Bridge Audit

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Security Audits

Security Audits

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs

Privacy | Terms of Use