Compound Audit

OPENZEPPELIN SECURITY | AUGUST 23, 2019

Security Audits

Compound Finance is a protocol, currently deployed on the Ethereum network, for automatic, permissionless, and trust-minimized loans of Ether and various ERC20 tokens. It is one of the most widely used decentralized finance systems in the ecosystem and helps demonstrate the power of the technology.

The team asked us to review and audit a subset of the smart contracts. We reviewed the code and now publish our results.

The audited commit is `f385d71983ae5c5799faae9b2dfea43e5cf75262` and the files included in the scope were:
CarefulMath, CErc20, CEther, Comptroller, ComptrollerInterface, ComptrollerV1Storage, UnitrollerAdminStorage, CToken, EIP20Interface, EIP20NonStandardInterface, ComptrollerErrorReporter, TokenErrorReporter, Exponential, InterestRateModel, Maximillion, ReentrancyGuard, Unitroller and WhitePaperInterestRateModel.

The Price Oracle system is intended to be replaced shortly and was therefore not reviewed. During this audit we assumed that the administrator and price feeds are available, honest and not compromised.

Here we present our findings.

Critical Severity

High Severity

Interest-Free Loans

The `CToken` contract calculates the borrow balance of an account in the function `borrowBalanceStoredInternal`.

The balance is the principal scaled by the ratio of the borrow indices (which track the accumulative effect of per-block interest changes):

```
balance = principal * (current borrow index) / (original borrow index)
```

However, when the principal and ratio of borrow indices are both small the result can equal the principal, due to automatic truncation of division within solidity.

This means that a loan could accrue no actual interest, yet still be factored into calculations of `totalReserves`, `totalBorrows`, and `exchangeRates`. In the case of many small loans being taken out, the associated interest calculated for that market may not match the amount actually received when users pay off those loans.

In brief: it is possible for users to take out small, short-term, interest-free loans. Optionally, they can then resupply the borrowed assets back into Compound to receive interest.

An example attack works as follows:

1. The attacker takes out several interest-free loans. By the nature of this vulnerability, these loans must be small. However, the attacker can take out arbitrarily many of them. It is most effective if the attacker borrows an asset for which the value of this truncation error is greatest (currently, wBTC).

2. The attacker consolidates the small interest-free loans by sending all the borrowed assets to a single Ethereum account.

4. The attacker deposits the asset into Compound.

5. (Optional) The attacker can then repeat the process — leveraging up — if they so desire.

6. The attacker unwinds the trade and pays off all the small loans before the loans start "registering" interest (that is, before the interest owed is no longer "truncated away"). The result is that the attacker collects the supply-interest but does not have to pay the borrow-interest. Note that this is mathematically equivalent to the attacker simply stealing cash.

7. Repeat.

There are a few practical considerations that make this attack non-trivial to pull off on the live network. These difficulties have to do with things unrelated to the Compound protocol: gas costs, slippage, and the unpredictable behavior of other Compound users. All of these can be overcome by certain classes of attackers, and we address them below.

In practice, steps 1, 2, and 6 of this process would be gas-intensive. So much so that the attack would not be profitable if the attacker has to pay a normal gas price. However, if the attacker is a miner then their gas price is zero, and so this attack costs them nothing in gas. (An exception is if the blocks are full, in which case the attacking miner must pay the opportunity cost of not filling the block with normal, paying transactions). Note that the miner does not need much mining power to do this. The ability to mine a couple of blocks per week is technically sufficient — though the more mining power they have, the more money they can borrow with no interest, and the more likely they are to be able to unwind the trade in time.

Additionally, if the attacker wants to get the most out of the attack, they'll likely want to perform the optional step 3. This requires that they exchange one asset for another on the open market. Here they may incur fees and/or slippage as they sell one asset for another. If fees and/or slippage are greater than the profit they make on their short-term, interest-free loan, then the attack is not profitable. This limits the attackers to those that can make trades very cheaply (or to those who are willing to forgo step 3 and accept a smaller rate of return).

Finally, the attacker may be foiled if other users of Compound borrow more of the same asset the attacker did — thus driving the borrow interest rate up and causing the attacker's owed interest to

where they would be okay paying the interest anyway. That way, if the attack fails, they end up in the same position as anyone else going long on that asset — and if the attack succeeds, then they get the added bonus of not having to pay interest for the assets they used for leverage.

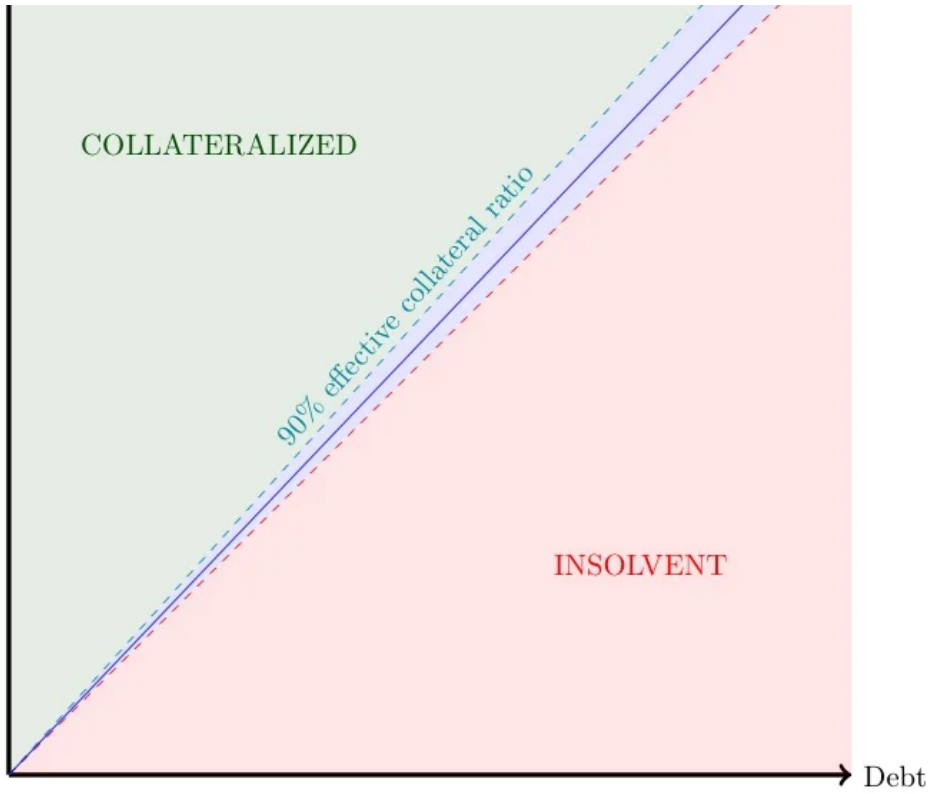
In short, there exists unavoidable error when computing interest. However, there does exist a choice over who gets the benefit of that error. The error should always be handled in a way that benefits the cToken contract, not the borrower. The small rounding error in favor of the borrower (and at the expense of the cToken contract) can be scaled up to eventually steal arbitrarily large sums of money from the cToken contract.

Consider adjusting the code such that the calculated borrow balance is always rounded *up* instead of being truncated.

Counterproductive Incentives

The protocol includes a mechanism to incentivize arbitrageurs to repay loans that are under-collateralized. This should increase the borrower's liquidity, reducing the risk of insolvency. After all, that is the point of the mechanism. However, there is a threshold where the incentives reverse and the liquidation mechanism actually pushes borrowers towards insolvency.

Consider a borrower with cTokens in various markets worth a total of `C`, and total debt worth `D`. They are located at point (`D`, `C`) in the following diagram:



After accounting for the distribution of cTokens and their corresponding collateral factors (the fraction of cTokens that can be used as collateral in each market), we can assign them an effective collateral ratio (ECR). The maximum possible value is 90%, which is labeled on the diagram. It corresponds to the minimum possible slope for remaining in the collateralized region, which is `1 / 90% = 1.11`. With these values, we can describe the important regions:

- Users in the green region above the ECR line are fully collateralized. They have enough collateral that even after reducing it by the ECR, it is still larger than their debt.

- Users in the red region are insolvent. Their debt is larger than their collateral and they no longer have any incentive to repay it. Users in this region can be a major problem for the market because they continue accruing interest while removing liquidity from the market, ensuring at least some cToken holders will be unable to redeem their assets.

- Users in the blue region are under-collateralized. They still have more assets than debt in the system, but the protocol deems them to be at risk of becoming insolvent.

- the close factor: the maximum fraction of the original loan that can be liquidated

- the liquidation incentive `L`: how much collateral do they receive (as a multiplier of the amount paid by the liquidator)

The Liquidation line on the diagram has slope `L` and passes through the origin. When an arbitrageur repays `x` debt, they receive `x*L` collateral from the borrower. In other words, they force the borrower to the bottom-left of the diagram on a trajectory parallel to the Liquidation line. The crucial observation is that this process can never cause a borrower to cross the liquidation line. Borrowers above the line are liquidated until they are fully collateralized, but borrowers below the line are actually pushed further towards insolvency by the liquidation process.

Ideally, they would already be liquidated before they crossed the threshold, but this would depend on various factors outside of the protocol, such as the speed of price changes, the liveness of the oracle, the congestion of Ethereum and the responsiveness and liquidity of arbitrageurs.

The Compound system currently has a liquidation incentive of `1.05`, **which means that when a borrower's debt rises to at least `1 / 1.05 = 95.2%` of the value of their cTokens, each liquidation will push them further towards insolvency**. The maximum amount that they can be liquidated in one transaction is bounded by the close factor. The current close factor in the Compound system is `0.5`, **which means that when a borrower's debt rises to at least 97.6% of the value of their cTokens, it is possible for a liquidation to force them into insolvency**.

Consider using a dynamic liquidation incentive that lowers as borrowers approach insolvency to ensure liquidations always increase solvency. Alternatively, consider treating the "incentivized insolvency" threshold of 95.2% as equivalent to insolvency, and choose the collateral factors and liquidation incentive accordingly.

Medium Severity

Interest May Not Compound

The code is designed to accrue interest as frequently as possible, but this requirement expands the responsibility of accruing interest into otherwise unrelated functions. Additionally, the size of the discrepancy between the computed and theoretical interest will depend on the volume of transactions being handled by the Compound protocol, which may change unpredictably.

To improve predictability and functional encapsulation, consider calculating interest with the compound interest formula, rather than simulating it through repeated transactions. Note that the additional gas requirements may be reduced using the `modexp` precompile. Separately, consider measuring the time between calls to `accrueInterest()` using *seconds* rather than *blocks*. This will help keep the interest rate calculation robust against changes to the average blocktime.

Incorrect Whitepaper Interest Rate

The Compound Finance whitepaper states in section 2.3:

> The interest rate earned by suppliers is *implicit*, and is equal to the borrowing interest rate multiplied by the utilization rate.

In fact, the `supplyRatePerBlock` function calculates it as the borrow interest rate multiplied by `borrowsPer` (which is not identical to the whitepaper utilization rate), and then multiplied again by (1 – the reserve factor). This is a related but not equivalent value to the specified one.

Consider updating the whitepaper or the calculation for consistency.

Outdated Interest Rates

In the `CToken` contract, borrowRatePerBlock and supplyRatePerBlock are supposed to return the current rates but they may be outdated. This is because those rates depend on variables that are updated when `accrueInterest` is called. Consider calling `accrueInterest` at the beginning of these functions.

Error Propagation

The codebase uses an error propagation mechanism that works as follows:

`Error`)

- Whenever a function returns an error, the calling function has to check it and then bubbles it up (ie. the caller returns the error tuple, possibly adding context to the error message)
- At the top level, the error/context codes are emitted in an event and then the error code is returned. The transaction is successful.
- The UI will need to convert the error codes into an error message

This scheme has the benefit of providing useful failure messages when an operation fails. However, there are serious negative properties of this pattern which we discuss below.

- First, the error propagation scheme violates the “Fail Early and Loudly” principle. Errors are not trapped when they are found. Callers have to remember to check and handle the returned error codes. Failure to do so can have serious security consequences.

- The scheme also increases both the size and the complexity of the code base. We estimate that abandoning the current scheme in favor of the traditional require/assert/revert paradigm would reduce the size of the (Solidity) codebase by 40%-60%. This would, additionally, make the code itself easier to read and understand — which is an often overlooked but important property of secure code.

- The increased code complexity is not just a matter of cognitive overhead for those attempting to read/understand the code. It also results in very large increases in gas costs due to all the extra required opcodes. From the user’s perspective, this is a strong argument against this pattern.

- Implementation of the scheme makes it incompatible with battle-tested libraries (like SafeMath) and requires the use of custom math functions that reproduce the same functionality except that they return errors instead of throwing on failure.

- The default value of an uninitialized `Error` enum is the `Error.NO_ERROR` value. This means that if a new `Error` variable is declared and returned by a function without having been set, the caller will assume that there was no error. That is, the default assumption by

audited.

- It does not cover every case, leading to inconsistencies within the code base. For example, the `CEther` `mint` function reverts on failure, whereas the `CErc20` `mint` function returns an error code.

- Finally, lack of a `revert` on failure is counter to what most Ethereum users have come to expect. For instance, a failed call to `CToken.transfer` via MetaMask will result in a `success` message from MetaMask even though the transfer failed. While this may not technically violate the ERC20 standard, it is counter-intuitive and increases the probability of user error.

With all of that said, it appears that all errors *are* properly handled in the code we audited. If considering a refactor of the contracts at some point in the future, we strongly recommend moving away from this error propagation pattern and instead using off-chain tools to evaluate failed transactions in order to display meaningful error messages to users.

Unknown Repayment Amount

To repay an ERC20 loan, a borrower can call `repayBorrow` or `repayBorrowBehalf` with a specified amount to repay.

However, they accrue interest in every block, which means if they specify the value of the loan at a particular block, their loan will be slightly higher in a future block when the transaction is confirmed and they will end up leaving part of the loan unpaid.

On the other hand, they could set the repay amount to `uint256(-1)`, which is used as a signal to mean “pay the whole loan”. This prevents the user from setting an upper bound on how much to pay. It is also particularly vulnerable in the case where an address is repaying on behalf of another address, since the borrower could front-run the transaction and borrow additional funds (up to the amount that the message sender has authorized the `CToken` contract to spend), which would also be repaid in the same transaction.

Consider treating the specified repayment amount as an upper bound, where the transaction repays the minimum of that value and the size of the loan.

between interest accruals. This makes the implementation highly sensitive to changes in the average time between Ethereum blocks.

On line 30 of `WhitePaperInterestRateModel.sol` it is implicitly assumed that the time between blocks is 15 seconds. However, the average time between blocks can change dramatically.

For example, the average time between blocks may increase by significant factors due to the difficulty bomb or decrease by significant factors during the transition to Serenity.

The difference between the actual time between blocks and the assumed time between blocks causes proportional differences between the intended interest rates and the actual interest rates.

While it is possible for the admin to combat this by adjusting the interest rate model when the average time between blocks changes, such adjustments are manual and happen only after-the-fact. Errors in blocktime assumptions are cumulative, and fixing the model after-the-fact does not make users whole – it only prevents incorrect interest calculations moving forward (until the next change in blocktime).

Consider refactoring the implementation to use *seconds* rather than *blocks* to measure the time between accruals. While `block.timestamp` can be manipulated by miners within a narrow window, these errors are small and, importantly, are not cumulative. This would decouple the interest rate model from Ethereum's average blocktime.

Misleading NatSpec Comments

Since the purpose of the Ethereum Natural Specification (NatSpec) is to describe the code to the user, misleading statements should be considered a violation of the public API that may confuse or mislead users.

The `getBorrowRate` interface and implementation `@return` comments state that the rate is scaled by 10e18. In fact, it is only scaled by 1e18.

The `CToken` contract `borrowRateMaxMantissa` comment states that the maximum borrow rate per block is 0.0005% but it is actually 0.0005 (or 0.05%).

should use “bonus”, which may cause confusion for people trying to understand the code. For example, a 25% discount is equivalent to a 33% bonus. That is, “25% off” is the same as “33% more for free”.

The comment on line 6 of `Exponential.sol` says “fixed-decision” when it should say “fixed-precision”.

The comment on line 83 of `CToken.sol` describes `borrowIndex` as the accumulator of earned interest when it should be the accumulator of the earned interest rate.

Consider updating the comments appropriately.

Excessive Indirection

The `Exponential` contract represents a fixed-size decimal number with a `uint` that is scaled up so the smallest non-zero decimal value is internally represented by the number 1. However, it is also unnecessarily wrapped in a struct.

This has the advantage that the scaled values can have a unique Solidity type. On the other hand, this feature is used inconsistently throughout the code base and introduces a very large overhead. Many of the functions in the `Exponential` contract implement common mathematical operations on the `Exp` type when the equivalent functions already exist for the `uint256` type. Most of the functions in the `Exponential` contract could be simplified or eliminated if the additional layer of indirection were removed. Additionally, many operations are complicated by mapping back and forth between the two representations.

Consider using the `uint256` type to internally represent the fixed-size decimal numbers. Then, consider enforcing the existing `Mantissa` suffix convention to consistently indicate whether a given variable is scaled.

Low Severity

cToken Related Issues

cTokens might not be transferable

Considering adding a related function to check if users' cTokens are transferable.

ERC20 double spend race condition

Due to `CToken`'s inheritance of ERC20's `approve` function, it is vulnerable to the ERC20 approve and double spend front running attack. Briefly, an authorized spender could spend both allowances by front running an allowance-changing transaction. Consider implementing OpenZeppelin's `decreaseAllowance` and `increaseAllowance` functions to help mitigate this.

ERC20 decimals size

The EIP20 specification optionally defines a `uint8` `decimals` field. However, the corresponding field in the `CToken` contract is a `uint256`. This is not compliant with the specification and may cause confusion when interacting with wallets and dApps. Consider setting the `decimals` type to `uint8`.

0 Address for Mints & Burns

Although not technically part of the EIP20 specification, it is common practice to use the zero address as the source for all `Transfer` events after minting, and as the destination for `Transfer`s upon burning tokens. The `Transfer` events in functions `mintFresh` and `redeemFresh` use the address of the `CToken` contract instead. Consider using the zero address instead or informing users and developers of this feature.

Mixing concerns

The `CToken` contract performs three different categories of functions:

1. ERC20 operations including transferring tokens, checking balances and setting allowances
1. Administrative operations such as setting the comptroller, interest rate model and reserve rate
1. The borrowing, loaning, repaying and liquidating operations as well as their support functions

The `CToken` administrator can call the `_reduceReserves` function to withdraw some of the reserves. However, the function requires that the administrator is the recipient address. This merges roles that should probably be distinct, particularly when the administrator is replaced with a decentralized governance process.

Consider having a separate recipient role, or allowing the administrator to choose the recipient in the function call.

Truncation-Related Issues

Throughout the compound contracts, truncation issues inherent to EVM operation result in some relatively unavoidable errors. These errors exist when minting cTokens, when redeeming cTokens for their underlying assets, and when liquidating another user's loan.

In the case of minting, the `CToken.mintFresh` function calls `divScalarByExpTruncate` to determine the number of cTokens the user will receive given their input of a certain number of underlying tokens. The result will be truncated if it is calculated to be some non-integer number of cToken units.

In the case of redeeming, the `CToken.redeemFresh` function also calls `divScalarByExpTruncate`. The number of tokens to redeem will similarly be truncated to the next lowest integer value of underlying token units.

Finally, the `Comptroller.liquidateCalculateSeizeTokens` function calls `mulScalarTruncate`. If the amount repaid is sufficiently small, the result will be rounded down to the next nearest integer value of indivisible token units.

In all cases, the user receives less than they theoretically should (either in terms of cTokens, or in terms of underlying tokens). However, the loss should never be more than 1 indivisible unit of whatever token the user is receiving. Even in the case of wBTC, where 1 indivisible unit is worth the most out of any other token involved, the loss is only roughly 1% of a USD cent (at time of writing). This issue, unlike the issue of interest-free loans, does not hurt the protocol. Instead, the users take the brunt of the (very small) loss.

enough interest that they can be profitably liquidated.

Partial Refactorization

The `getUtilizationAndAnnualBorrowRate` function of the `WhitePaperInterestRateModel` contract appears to be in a partially refactored state. The comments and choice of functions suggest that the `multiplier` variable is represented as a percentage (ie. the value 45 would imply a multiplier of 45%). However, instead of dividing by 100, it is divided by 1e18. This is consistent with how the deployed contracts treat the `multiplier` variable. Nevertheless, it uses the functions intended for `uint` values to recreate the `mulExp` function without the rounding check.

Consider using the `mulExp` function to scale the utilization rate instead of `mulScalar` and `divScalar` and update the comments to describe the correct code behaviour. In addition, consider stating the unit type in the `multiplier` and `baseRate` comments.

Notes

Markets Can Become Insolvent

When the value of all collateral is worth less than the value of all borrowed assets, we say the market is *insolvent*. Compound does many things to reduce the risk of market insolvency, including: prudent selection of collateral-ratios, incentivizing third-party collateral liquidation, careful selection of which tokens are listed on the platform, etc. However, the risk of insolvency cannot be entirely eliminated, and there are numerous ways a market can become insolvent.

Here are five examples of things that could cause a market to become insolvent:

1. The price of the underlying (or borrowed) asset makes a big, quick move during a time of high network congestion — resulting in the market becoming insolvent before enough liquidation transactions can be mined.

2. The price oracle temporarily goes offline during a time of high market volatility. This could result in the oracle not updating the asset prices until after the market has become insolvent.

4. Miners “steal” (by not paying interest) enough funds that the market eventually becomes insolvent. (See the “Interest-Free Loans” issue).

5. Administrators list an ERC20 token with a later-discovered bug that allows minting of arbitrarily many tokens. This bad token is used as collateral to borrow funds that it never intends to repay.

In any case, the effects of an insolvent market could be disastrous. It would mean that cToken contracts would effectively be running a fractional reserve. This could result in a “run on the bank”, with the last suppliers out losing their money.

This risk is not unique to Compound. *All* collateralized loans (even non-blockchain loans) have a risk of insolvency. However, it is important to know that this risk does exist, and that it can be difficult to recover from even a small dip into insolvency.

Reserves Increased Event

The `accrueInterest` function of the `CToken` contract accrues interest and increases the reserves. However, the `AccrueInterest` event that it emits does not include the amount by which the reserves have increased. Consider adding this value to the `AccrueInterest` event or creating a new `ReservesIncreased` event to match the existing `ReservesReduced` event.

Multiple Contracts Per File

The `ErrorReporter` file contains three independent contracts: `ComptrollerErrorReporter`, `TokenErrorReporter` and `OracleErrorReporter`. This does not follow the Solidity style guide and makes the code harder to read.

Consider using a separate file for each contract.

Inconsistent NatSpec Usage

The docstrings of the contracts and functions are partially following the Ethereum Natural Specification Format (NatSpec). They use the tags sporadically. Consider adding the relevant

The comment on line 906 of `Comptroller.sol` uses the variable `newLiquidationDiscount` when it should use `newLiquidationIncentive`.

The comment on line 298 of `CToken.sol` states that the `transferVerify` function checks for under-collateralization. In fact, that check is performed at the start of the function with the `transferAllowed` hook.

The comment on line 821 of `CToken.sol` states that `redeemAmountIn` is the amount of cTokens to redeem but it is the amount of underlying.

The comments on line 821-822 of `CToken.sol` state that only one of `redeemTokensIn` or `redeemAmountIn` may be zero but in fact at least one must be zero and both may be zero.

The comment on line 322 of `Comptroller.sol` should state “Require tokens is non-zero or amount is also zero”

Consider updating the comments appropriately.

Require Statement Without Error Message

There is a `require` statement on line 111 of `CEther.sol` with no failure message. Consider adding a message to inform users in case of a revert.

Non-traditional Use of ReentrancyGuard

The NatSpec documentation on `ReentrancyGuard.sol` states:

> If you mark a function nonReentrant, you should also mark it external.

However, the following functions in `CToken.sol` have the `nonReentrant` modifier and are NOT external:

- `exchangeRateCurrent` (public)
- `mintInternal` (internal)
- `redeemInternal` (internal)
- `redeemUnderlyingInternal` (internal)

- `liquidateBorrowInternal` (internal)

Typically, the `nonReentrant` modifier would be put on their external function counterparts in `CEther.sol` and `CErc20.sol`. That said, the existing use of `ReentrancyGuard` *does* prevent reentry into the corresponding external functions, and so no changes are needed.

Unvetted Token Warning

It is important to note that if a malicious or poorly-designed token is added to Compound, it could allow someone to steal *all* funds entrusted to Compound. For example, if anyone can arbitrarily change the `totalSupply` or account balances of a listed ERC20 token faster than the price oracle can adjust the price, an attacker could use those newly minted tokens as collateral to borrow all Compound assets.

Consider making a formal list of properties that a “safe” token should and should not have, and be sure each new token is safe before listing it on Compound.

Transparent Proxy Pattern

The `Unitroller` contract uses a proxy pattern to redirect transactions to functions it does not recognize. However, it will not be able to proxy calls to functions with the same function signature as one of its own functions, including its inherited functions.

Consider using the transparent proxy pattern for increased generality.

Unused Return Value

The `getUtilizationAndAnnualBorrowRate` function of the `WhitePaperInterestRateModel` contract returns the `utilization rate` in addition to the borrow rate. However, the only function that calls it discards the utilization rate. Consider removing the utilization rate from the return values.

collateralFactorMantissa Needs to be Set

In `Comptroller.sol`, `collateralFactorMantissa` for each market is initialized implicitly to 0 until it is set within `_setCollateralFactor`. This means that whenever a new

Consider Adding Syntax Highlighting to GitHub Repository

Consider adding Solidity syntax highlighting to the GitHub repository by putting the line `*.sol linguist-language=Solidity` in a `.gitattributes` file in the root of the repository.

This helps improve readability for users inspecting contract code on GitHub.

Accrue Zero Interest Event

In the accrueInterest function of `CToken`, the `AccrueInterest` event is still emitted when there is no interest (because it was already accrued in this block). Consider checking the `accrualBlockNumber` before emitting the event (or indeed, executing the rest of the function).

Unexpected ERC20's

Within each of the cToken ERC20 contracts, there is no way to check for unexpected sends of ERC20 tokens to the contract. However, the ERC20 balance is used in the `exchangeRate` calculation via the variable `totalCash`, which is equal to the ERC20 token balance.

This results in cToken values increasing with ERC20 sends, but importantly, it *does not* result in the value of borrows increasing, as `borrowIndex` is unaffected by total underlying balance.

Consider adding a state variable to track internal balances, which would help identify any unexpected ERC20 tokens.

Avoid Overloaded Functions

Within `Exponential.sol` there are two definitions for `mulExp()` (here, and here). Consider changing the name of one of the functions for improved readability of code.

Use ABI Encoder

The `requireNoError` function in the `CEther` contract appends a string to another string one byte at a time. Consider commenting the operation for clarity and using `abi.encodePacked` for brevity.

Missing require

Consider mapping it in a `require` statement.

Default Visibility

The following state variables and constants are using the default visibility:

- In `Comptroller`:
  `closeFactorMinMantissa`, `closeFactorMaxMantissa`, `collateralFactorMaxMantissa`, `liquidationIncentiveMinMantissa`, `liquidationIncentiveMaxMantissa`

- In `CToken`:
  `borrowRateMaxMantissa`, `reserveFactorMaxMantissa`, `accountTokens`, `transferAllowances`, `accountBorrows`

- In `Exponential`:
  `expScale`, `halfExpScale`, `mantissaOne`

For readability, consider explicitly declaring the visibility of all state variables.

TODO's in Code

The codebase still contains `TODO` statements (in `Exponential`, `Unitroller`, and `CToken` lines 1306, 1405, 1455, 1465 and 1525), despite being used in production.

Consider resolving and removing them.

Superfluous Code in acceptAdmin

On line 1333 of `CToken.sol` there is the conditional statement:

```
if (msg.sender != pendingAdmin || msg.sender == address(0)) ...
```

The comment immediately above this line is:

```
// Check caller is pendingAdmin and pendingAdmin ≠ address(0)
```

In either case, however, the second half of the conditional is superfluous. It is infeasible for `msg.sender` to be the zero address. Thus, the conditional statement can be reduced to:

`if (msg.sender != pendingAdmin) ...`.

Change uint to uint256

Throughout the code base, some variables are declared as `uint`. To favor explicitness, consider changing all instances of `uint` to `uint256`.

Use assert

The `require` statement on line 1476 of `CToken.sol` is confirming a property that should never fail for any user input. In such a situation, consider using an `assert` statement instead.

Conclusion

No critical and two high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.

Summary

If you are interested in a non-technical overview of this audit, we present a summary of the system as it relates to the audit as well as a couple of interesting findings in our summary article.

Related Posts

Beefy

BRUSHFAM
OpenBrush Contracts

Linea

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

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

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits