PUBLIC

Code Assessment

of the Comet Smart Contracts

May 30, 2022

Produced for

by

Contents

1 Executive Summary

Dear Jared,

Thank you for trusting us to help Compound with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Comet according to Scope to support you in forming an opinion on their security risks.

Compound implements Comet, which is a gas-efficient lending platform that allows more efficient liquidity use due to a more streamlined application of borrowing stable coins against various collaterals.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	7
• Code Corrected	3
• Risk Accepted	4
Low-Severity Findings	12
• Code Corrected	9
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Comet repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 March 2022	71aec025e1fcf9c16a1aeb20a85eddae5a5d5581	Initial Version
2	29 April 2022	06ca155a2818c7c4b05356b9df439f6ff43343b5	Second Version

For the solidity smart contracts, the compiler version `0.8.11` was chosen.

2.1.1 Excluded from scope

The correctness of the following components is out of scope:

- The correct implementation of the used tokens
- The correctness of the used Chainlink oracles
- The correct setup of the governance controlling the contracts
- The economic model including the incentives of involved parties
- The contracts `Bulker` and `CometRewards` which were introduced in Version 2

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Comet is a lending platform for stable tokens (Base Token). Users can supply different tokens as collateral and, based on the particular supplied collateral, borrow a certain amount of the offered Base Token. Liquidity providers can in turn supply Base Token to Comet in order to earn the interest paid by borrowers. On the occasion that the value of a borrowing user's collateral drops under a certain threshold, his balance can be absorbed into the protocol reserves and get liquidated. Overall, the implementation is very gas efficient for users.

2.2.1 Comet

`Comet` is the main contract; all user interactions take place through this contract. It consists of one Base Token that can be borrowed and up to 15 Collateral Tokens that can be used as collateral to obtain Base Tokens. Not all tokens can be used, for more please refer to Supported Tokens. The lending flow works as follows:

- User A calls `supply` to transfer Base Token to the contract.
- User B calls `supply` to transfer Collateral Tokens to the contract.
- User B calls `withdraw` to transfer Base Token up to a certain amount from the contract to their address.
- As time progresses, User B owes interest for the borrowed tokens, decreasing his balance and in turn increasing User A's balance.
- At a certain point in time, User B has to either supply additional Collateral or repay Base Tokens. If they fail to do this, their account can be liquidated, removing all of their supplied collateral.

For each Collateral Token, two factors are defined:

- `borrowCollateralFactor`: The percentage of collateral value the user is allowed to borrow in Base Token.
- `liquidateCollateralFactor`: The percentage of collateral value the user is allowed to have in outstanding borrows before a liquidation occurs.

If the user's collateral value drops below the liquidation factor, anyone can call the `absorb` function of the contract to transfer the user's collateral into the protocol's reserves. The caller receives some points based on the amount of accounts absorbed and the gas used. The reward for these points is yet to be determined.

The contract defines a fixed value for Base Token reserves it should be holding at all times. If the current reserves are below this threshold, users can call `buyCollateral` to obtain any absorbed Collateral Tokens with a discount.

`Comet` is an `ERC20` contract. However, note that some `ERC20` functions have special behavior described in Allowances enable management of both base tokens and collateral assets. Furthermore, the contract is implementing some additional functions on top of the `ERC20` interface, these include:

- `getBorrowLiquidity`: How much more base tokens can be borrowed. However, please note this value needs to be interpreted with care. If it returns `X` not every value `Y <= X` can be borrowed due to minimum borrow requirements.
- `borrowBalanceOf`: Non-negative value of how much an account has borrowed
- `baseBalanceOf`: Integer value indicating whether an account has borrowed (`< 0`) or supplied (`> 0`) base tokens

Since `Comet` bundles a lot of functionality in a single contract, the Ethereum contract size limit is exceeded. For this reason, some of the functionality is delegated to another `CometExt` contract. However, as `CometExt` is called through a `delegatecall`, `Comet` offers all functionality through a single endpoint.

2.2.2 CometFactory

The `Comet` contract is set up to be used as a Proxy. A `CometFactory` exists that allows the creation of a `Comet` contract with a `clone` function. During `clone` the configuration needs to be provided. The configuration is used to initialize the many `immutable` variables inside the contract. These allow a significant reduction of execution costs.

2.2.3 Roles & Trust Model

The `Comet` contract defines two addresses with special privileges:

- `pauseGuardian` is an external account that can pause certain actions on the contract.
- `governor` is the address of Compound's governance contract. It has the same privileges as `pauseGuardian`, has allowance for the contract's own funds and can call `withdrawReserves`

which allows the withdrawal of all non-user funds from the protocol. The contract is governed by holders of the **COMP** token.

The following actions can be paused by these addresses:

- `supply`, `supplyFrom`, `supplyTo`
- `withdraw`, `withdrawFrom`, `withdrawTo`
- `transfer` `transferFrom`, `transferTo`
- `absorb`
- `buyCollateral`

`Comet` contracts are deployed using OpenZeppelin's `TransparentUpgradeableProxy`. The admin managing the proxy is implemented as a `ProxyAdmin` contract.

The governance role is very powerful but fully trusted. This is an important part of the trust model as the governance has the theoretical ability to steal users' funds in different ways. These include:

1. Funds could be stolen through the function `withdrawReserves` by withdrawing all available base tokens and not just the reserves.
2. Funds could be stolen by using the function `approveThis` which would allow the transfer of all tokens including base tokens. Unlimited approvals can be given to anyone.
3. Funds could be stolen through malicious contract upgrades.

Overall, Compound states regarding the governance:

> *Governance has ultimate control over the contracts and we believe the mitigation to the risks associated with this are the timelock and the governor itself being controlled by a DAO.*

The security and usability of the protocol relies on the existence of other actors in the network, among other things these actors are expected:

- to repay loans at times of extremely high borrow rates or to supply funds at times of extremely high supply rates
- to absorb accounts in order to receive liquidator points
- to buy collateral at a discount from the protocol
- to use each contract somewhat regularly (see Regular Use Expected)

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4

- Allowances Enable Management of Both Base Tokens and Collateral Assets Risk Accepted
- Oracle Timestamps Not Checked Risk Accepted
- approve Only Allows the Values 0 and MAX Risk Accepted
- baseBorrowMin Is Not Enforced for Destination of Transfer Risk Accepted

Low-Severity Findings	3

- Balances Can Be Overflowed Risk Accepted
- Tracking Indices May Overflow on Large Tracking Speed Values Risk Accepted
- targetReserves Limit in buyCollateral Can Be Circumvented Risk Accepted

5.1 Allowances Enable Management of Both Base Tokens and Collateral Assets

Correctness Medium Version 1 Risk Accepted

Users can give privileges to other accounts through the `approve` and `allow` functions. Both functions use the `isAllowed` mapping to store this information. Accounts for which `isAllowed` is set to true have full control over both the base tokens and the collateral assets of the user.

This is problematic for the following reasons:

- It may not be clear to users who call the `approve` function (which is part of the ERC-20 interface) that this not only gives the `spender` access to their base tokens but also to their collateral assets. The function description does not specify this.
- There is no means of giving partial privileges (i.e., access only to base tokens or only to collaterals) to another account. This may force users to give unnecessary permissions to other accounts.
- Given prior experiences with ERC-20 tokens, users might expect the `approve` function to allow the spender to transfer at most `balanceOf` tokens. However, this is not true here as an approval also allows the spender to borrow funds. As a result, `balanceOf` can essentially become negative, which might not match the expectations of users or integrators.

To avoid integration issues and the compromise of user funds, these unexpected behaviors should be clearly documented.

Risk accepted:

Compound has added dev notes documenting the special behaviour.

5.2 Oracle Timestamps Not Checked

Security Medium Version 1 Risk Accepted

The function `getPrice` does not verify that the round data received from Chainlink oracles is up-to-date. If there is any problem with the oracles that results in outdated pricing data being returned. As a result critical calculations for allowed borrowing and liquidations would become inaccurate. It might be possible to liquidate safe positions or take out under-collateralized borrows.

Risk accepted:

Compound acknowledges the risk and notes that even if a defense against a lack of updates was implemented, the ability to report false prices make the price oracles a primary risk vector for the protocol. Moreover, Compound encourages governance to invest in improvements upon the oracle system, especially ones which can also reduce gas costs.

5.3 `approve` Only Allows the Values 0 and MAX

Design Medium Version 1 Risk Accepted

The `approve` function only allows the values `0` and `type(uint256).max` which could lead to the following complications:

1. All approvals are infinite. In the past, infinite approvals given to buggy contracts have been exploited (e.g., in the case of Multichain). The risk of this is increased when only infinite approvals can be given.
2. All other approvals will fail. This breaks integration with existing DeFi protocols, which approve exact values. Comet Tokens would be incompatible with such protocols.

Risk accepted:

Compound accepts the risk and refers to its documentation.

5.4 `baseBorrowMin` Is Not Enforced for Destination of Transfer

Correctness Medium Version 1 Risk Accepted

Borrows have a minimum threshold, called `baseBorrowMin` to ensure that it remains worthwhile to liquidate them. However, this minimum threshold does not always hold.

If Base Tokens are transferred to another address using `transferBase` and the sender has to borrow tokens, the call reverts if the amount of borrowed tokens does not exceed the `baseBorrowMin` threshold. However, if a user receives tokens such that his balance is still negative but now violates the `baseBorrowMin` threshold, the call does not revert.

As a consequence, an attacker could intentionally set up many accounts with borrows below the threshold in order to avoid liquidation. However, setting up such accounts would also consume a lot of gas and hence is unlikely to be financially beneficial.

Risk accepted:

Compound accepts the risk with the following statement:

> The intention behind *baseBorrowMin* is to disallow initiating new borrows for which liquidation would likely not be worthwhile relative to gas costs. The destination of a transfer can only end up with 'dust' if a debt is partially repaid by another account almost fully. Both new positions would still need to be fully collateralized according to the borrow collateral factor. If some kind of griefing attack were attempted, governance could declare such tiny borrows as liquidatable at any point, and potentially seize or sell all the collateral immediately.

5.5 Balances Can Be Overflowed

Correctness | Low | Version 1 | Risk Accepted

The functions `presentValueSupply` and `presentValueBorrow` in `CometCore` allow an overflow due to unsafe casting to `uint104`. Consider the following scenario:

- A user supplies `type(int104).max` Base Tokens to the protocol.
- After some time, the `baseSupplyIndex` is equal to or greater than 2.
- The user calls any of the functions that update their balance with 0 amount.
- `totalSupplyBase` as well as the user's `principal` will be overflowed to a value smaller than the current value.

If the base token uses the maximum of 18 decimals allowed in the protocol, around 10 trillion in principal balance and an index value of at least 2 (otherwise the safe cast in `presentValue` will kick in) are needed. This is practically infeasible for base tokens pegged to the USD. However, other base tokens (or USD-based tokens after a period of extreme inflation) can bring this problem into the realm of possibilities.

Risk accepted:

Compound accepts the risk and extended the documentation to describe it.

5.6 Tracking Indices May Overflow on Large Tracking Speed Values

Correctness | Low | Version 1 | Risk Accepted

The state variables `trackingSupplyIndex`, `trackingBorrowIndex` as well as each user's `baseTrackingIndex` are of type `uint64`. The function `accrueInternal` updates these indices with the product of the passed seconds and `baseTrackingSupplySpeed` / `baseTrackingBorrowSpeed` divided by the current amount of `totalSupplyBase` / `totalBorrowBase` (without decimals). `baseTrackingSupplySpeed` / `baseTrackingBorrowSpeed` can be numbers with a decimal scale of up to 15 which will result in an overflow after a non-negligible time-frame if the amount of supplied / borrowed tokens is low and `baseMinForRewards` is set to a low value.

Suppose `trackingIndexScale`, `trackingSupplyIndex` and `trackingBorrowIndex` are all set to **1e15** (e.g. 1 COMP per second). The safe cast to `uint64` in `accrueInternal` will revert after only approximately 5 hours after the `baseMinForRewards` has been reached resulting in a denial-of-service for the whole contract. This time is multiplied by the amount of full tokens supplied.

Risk accepted:

Compound accepts this risk with the following statement:

> The overflow behavior depends on several parameters which do need to be chosen carefully by governance. However, we believe these values can be chosen safely and need not often, if ever, change. In the worst case, in which governance fails to set these safely, there would be a denial of service until resolved by governance. We believe this is an acceptable risk, given that the very worst case in which bad parameters are chosen still does not result in any loss of funds.

5.7 `targetReserves` Limit in `buyCollateral` Can Be Circumvented

Security | Low | Version 1 | Risk Accepted

The `targetReserves` value describes the expected value of the protocol reserves expressed in the base asset. The function `buyCollateral` reverts if the current protocol reserves are higher than `targetReserves` as it doesn't allow the purchase of Collateral assets in this case. However, if any of the collateral tokens has a callback as part of `transferFrom` (e.g. ERC777), then this check can be circumvented with a reentrant call to `buyCollateral`. As a consequence, more collateral can be bought than intended by the protocol as the check is not correctly performed for the reentrant call.

Additionally, even without a reentrancy, the purchased amount might far exceed the value of `targetReserves`. At the time of writing it was unclear whether this is intended in all cases.

Risk accepted:

Compound accepts the risk with the following statement:

> The reserves target is a mechanism for governance to prevent the sale of collateral after a sufficient number of reserves have been reached. The risk that collateral assets may be sold and increase reserves beyond the target amount, is not a risk to the protocol health, in fact generally the opposite, as it guarantees a larger amount of reserves. The issue is that it could prevent the protocol from being as profitable as it might otherwise be in the event that assets are liquidated and sold and later become much more valuable (as has been the case previously for many crypto assets), but we are not concerned about that risk.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Wrong Computation of Borrow Balance Code Corrected

Medium-Severity Findings	3

- No Handling of Ecrecover Return on Wrong Input Code Corrected
- No Sanity Checks for liquidationFactor Code Corrected
- CometInterface Not Implemented by the Contracts Code Corrected

Low-Severity Findings	9

- Accrued Interest Not Accounted for in Balance Functions Code Corrected
- Floating Pragma Code Corrected
- Missing Constructor Sanity Checks Code Corrected
- Missing Events Code Corrected
- No Recovery of Accidental Token Transfers Possible Code Corrected
- Possible Contract Size Reductions Code Corrected
- Possible Gas Savings Code Corrected
- Rounding Errors Between User Balances and Total Balances Code Corrected
- Unused Custom Error Code Corrected

6.1 Wrong Computation of Borrow Balance

Correctness High Version 1 Code Corrected

The function `borrowBalanceOf` inside the `CometExt` uses `baseSupplyIndex` instead of `baseBorrowIndex` to compute the borrow balance:

```
function borrowBalanceOf(address account) external view returns (uint256) {
    int104 principal = userBasic[account].principal;
    return principal < 0 ? presentValueBorrow(baseSupplyIndex, unsigned104(-principal)) : 0;
```

As a consequence, the result is incorrect.

Code corrected:

The `borrowBalanceOf` function now uses the correct index (i.e., `baseBorrowIndex`). In addition, unit tests were updated to catch this issue by using a different value for *baseSupplyIndex* and *baseBorrowIndex*.

6.2 No Handling of Ecrecover Return on Wrong Input

Security Medium Version 1 Code Corrected

`ecrecover` returns 0 on error. This error value is not checked correctly within the `allowBySig` function. As a result anyone can call `allowBySig` with `owner == 0` and thereby set approvals in the name of the 0-address. Since transfers to the 0-address are possible in the contract, falsely sent funds to this address could be recovered by an attacker.

Code corrected:

The `allowBySig` function now reverts if the value returned by `ecrecover` is the 0-address.

6.3 No Sanity Checks for `liquidationFactor`

Design Medium Version 1 Code Corrected

The `liquidationFactor` determines the liquidation penalty a user suffers based on the collateral asset. When setting the `liquidationFactor` in the function `_getPackedAsset`, it should be checked against the value of `storeFrontPriceFactor`. The `storeFrontPriceFactor` describes the discount the protocol gives when someone buys liquidated collateral. If `liquidationFactor > storeFrontPriceFactor`, then the protocol is expected to lose funds on liquidations. Any user noticing this, could perform the following attack:

Sandwich significant price updates which decrease any of the collateral prices or increase the base asset price using:

- Supply a collateral and borrow the maximum
- Absorb the account and liquidate

This would drain funds from the protocol. Hence, it should be ensured that this setting never exists.

Code corrected:

`liquidationFactor` is now assured to be smaller than or equal to `storeFrontPriceFactor`.

6.4 `CometInterface` Not Implemented by the Contracts

Correctness Medium Version 1 Code Corrected

The contracts `Comet` and `CometExt` contracts do not extend the `CometInterface`. This can lead to errors during development and integration by third parties as the interface does not match up with the implementations. One such error is that the contracts do not implement an `accrue` function even though it is defined in the `CometInterface`:

```
abstract contract CometInterface is CometCore, ERC20 {
    ...
    function accrue() virtual external;
```

Code corrected:

`CometInterface` was split into `CometMainInterface` and `CometExtInterface`. `Comet` now implements `CometMainInterface`, and `CometExt` now implements `CometExtInterface`.

6.5 Accrued Interest Not Accounted for in Balance Functions

Correctness Low Version 1 Code Corrected

The functions `balanceOf`, `borrowBalanceOf` and `baseBalanceOf` do not accrue interest before returning the respective balances. This can result in unexpected behavior. Consider the following example:

1. A contract queries its `borrowBalanceOf` of asset A. The function returns X.
2. The contract supplies X of asset A, expecting to have paid back all its borrows. However, unless `accrueInternal` has by chance been called within the same block, there will be a remaining borrow balance.

This behavior needs to be explicitly specified, currently the function descriptions do not indicate this in any way:

```
/**
 * @notice Query the current negative base balance of an account or zero
 * @param account The account whose balance to query
 * @return The present day base balance magnitude of the account, if negative
 */
```

Code corrected:

`balanceOf`, `borrowBalanceOf` and `baseBalanceOf` now calculate the current values of `baseSupplyIndex` / `baseBorrowIndex` before converting them to present values.

6.6 Floating Pragma

Design Low Version 1 Code Corrected

Comet uses the floating pragma `^0.8.11`. Contracts should be deployed with the compiler version and flags that were used during testing and auditing. Locking the pragma helps to ensure that contracts are not accidentally deployed using a different compiler version and help ensure a reproducible deployment.

Code corrected:

Compiler version has been fixed to `0.8.13`.

6.7 Missing Constructor Sanity Checks

Design Low Version 1 Code Corrected

The following sanity checks could potentially be added to the constructor:

- Base Token decimals should be at least 6 to prevent `accrualDescaleFactor` from becoming 0.

- A comment for `baseMinForRewards` suggests the value should be sufficiently large but is only checked to be non-zero.
- `reserveRate` should be lower or equal to `FACTOR_SCALE` to prevent reverting on underflow in `getSupplyRate`.
- `kink` should be lower than or equal to `FACTOR_SCALE`.

Code corrected:

- Corrected: `baseScale` is assured to be greater than or equal to `BASE_ACCRUAL_SCALE`.
- Risk accepted: Governance is trusted to choose the correct value for `baseMinForRewards`.
- Corrected: `kink` is assured to be smaller than or equal to `FACTOR_SCALE`.

6.8 Missing Events

Design | Low | Version 1 | Code Corrected

The following functions represent important state changes, for which an event might be helpful:

- `initializeStorage`
- `pause`
- `absorb`
- `buyCollateral`
- `withdrawReserves`
- `allow`
- `allowBySig`

Additionally, if absorbing an account results in a loss of funds of reserves, because the collateral did not cover the borrow, an event could be emitted as well.

Code corrected:

All mentioned functions except `initializeStorage` and `buyCollateral` now emit events. Regarding the remaining events, Compound has issued the following statement:

> Our view is that events should not be viewed as critical for tracking state changes, and that modern off-chain processors are capable of tracking all contract state transitions anyway.

6.9 No Recovery of Accidental Token Transfers Possible

Design | Low | Version 1 | Code Corrected

In case an ERC-20 token other than the base tokens or collateral tokens is sent to the contract, then it cannot be recovered. Among other reasons, this might happen due to airdrops based on the base tokens or collateral tokens.

Code corrected:

A new function `approveThis` has been introduced to allow the governance to approve any `ERC20` token to any address.

6.10 Possible Contract Size Reductions

Design | Low | Version 1 | Code Corrected

- Instead of creating an empty `AssetConfig`, and later returning `(0, 0)`, the function `_getPackedAsset` could directly return `(0, 0)`.
- The functions `isBorrowCollateralized`, `getBorrowLiquidity`, `isLiquidatable` and `getLiquidationMargin` share the same code with marginal modifications. The overlapping code could be factored out into new functions to save code size.
- The `baseScale` variable is only needed internally and is derived from `decimals` and can thus be defined as `internal` to reduce code size.
- The intialization of `trackingSupplyIndex` and `trackingBorrowIndex` to `0` in the `initializeStorage` function can be omitted.

Code corrected:

- Corrected: `__getPackedAsset` now directly returns `(0, 0)` if an `AssetConfig` element is empty.
- Not corrected: Compound claims that the compiler opimizations already account for a sufficient contract size reduction in `isBorrowCollateralized`, `getBorrowLiquidity`, `isLiquidatable` and `getLiquidationMargin`.
- Not corrected: Compound does not want to make an exception for one variable.
- Corrected: `trackingSupplyIndex` and `trackingBorrowIndex` are no longer initialized to 0.

6.11 Possible Gas Savings

Design | Low | Version 1 | Code Corrected

- The function `_getPackedAsset` calls the `decimals` function of the `asset ERC20` contract and checks if it equals the provided `decimals` variable in `AssetConfig`. Since the external call to `asset` is done anyways, there is no need to provide the decimals in the config and perform this check.
- The function `supplyCollateral` calls `getAddressInfoByAddress` and `updateAssetsIn` which calls `getAddressInfoByAddress` for the same address again.
- The function `absorbInternal` calls `isLiquidatable` and then proceeds to perform a very similar computation (including the same calls to the price oracles) again.
- The function `isBorrowCollateralized` is expected to be commonly called for contracts with a non-negative base balance, e.g., for `address(this)` in `buyCollateral`. In those cases, `isBorrowCollateralized` can return `true` as soon as `presentValue` is non-negative. Then, the call to the price oracle can be skipped.

Code corrected:

- Not corrected: The additional `decimals` value in the supplied config is used as sanity check to determine if the caller actually knows the decimals of the asset being configured.
- Corrected: `updateAssetsIn` now takes `AssetInfo` as argument and does not load asset infos itself anymore.

- Not corrected: Compound claims that the compiler already optimizes the functions.
- Corrected: `isBorrowCollateralized` now checks if the user's present value is greater than or equal to zero before performing any calculations.

6.12 Rounding Errors Between User Balances and Total Balances

Correctness Low Version 1 Code Corrected

Due to the balance calculation with indices and principal values, rounding errors can introduce an inconsistency between the user balances and the total balances. Consider the following scenario:

- `totalSupplyBase` is 100.
- `baseSupplyIndex` is 1.085 (without decimals).
- A user now supplies 10 Base Tokens with the `supply` function.
- `totalSupplyBase` gets updated to 108.
- The user's `principal` gets updated to 9.

If the protocol holds no reserves, the last user to withdraw their balance from the contract might not be able to withdraw the full amount.

Code corrected:

The calculation of totals was modified to address this issue: Indices are now no longer translated to their present values, updated and trasnlated back to their principal values. Instead, they are now updated with the delta of users' principal values.

6.13 Unused Custom Error

Design Low Version 1 Code Corrected

The `Comet` contract defines a `BadAmount` error that is never used.

Code corrected:

Unused errors `BadAmount` in `Comet` and `Unauthorized` in `CometExt` have been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Event Reordering Possible

Note Version 1

`doTransferIn` and `doTransferOut` are always called before events are emitted. If the respective `ERC20` tokens that are called implement callbacks to the sender or receiver, events could possibly be reordered due to reentrancy. While this is not problematic for the contract itself, this can introduce errors in third-party applications that make certain assumptions about the emitted events.

7.2 Magic Numbers

Note Version 1

The functions `_getPackedAsset` and `getAssetInfo` use the same magic numbers for packing as well as descale and rescale factors:

```
uint256 word_a = (uint160(asset) << 0 |
                  uint256(borrowCollateralFactor) << 160 |
                  uint256(liquidateCollateralFactor) << 176 |
                  uint256(liquidationFactor) << 192);
```

These numbers should be defined as constants to avoid errors during development.

7.3 Potential Incentive to Withdraw Supply

Note Version 1

In certain circumstances, users might have an incentive to actually withdraw parts of their supplied base tokens. Consider the following scenario:

- `kink` is set to 80%.
- `interestRateSlopeLow` is set to 10%.
- `interestRateSlopeHigh` is set to 300%.
- `interestRateBase` is set to 5%
- For simplification, `reserveRate` is set to 0.
- User A has supplied 100 base tokens to the contract.
- User B has borrowed 80 of those base tokens, resulting in 80% utilization.
- User A currently receives 10.4 base tokens interest per year.
- User A now withdraws 20 base tokens such that utilization becomes 100%
- User A now receives 58.4 base tokens interest per year, even though they have reduced their balance.

If User A holds a significant stake in supplied base tokens, they might be incentivized to withdraw some of their supply for as long as the utilization is high enough so that they earn more than 10.4 base tokens per year. However, obviously such a scenario incentivizes others to supply liquidity or repay borrows, so that it is unlikely to last.

7.4 Regular Use Expected

Note **Version 1**

For the sake of security, the protocol assumes that each contract is used somewhat regularly. This is required so that the function `accrueInternal` is called regularly. If there is **no regular usage**, e.g., if the contract is not called for a year, the following issue arises:

Collateral that normally would be liquidatable can still be transferred / withdrawn. This is because the interest needs to be explicitly accrued to update the indexes. Transfers and withdrawals of collateral are allowed without explicit accrual and hence rely on recent actions. Theoretically, this can lead to under-collateralized accounts, but given typical configurations, this would take years of inactivity. The authors are aware of this requirement and added the following comment:

```
// Note: no accrue interest, BorrowCF < LiquidationCF covers small changes
```

7.5 Supported Tokens

Note **Version 1**

Not all ERC20 tokens can act as base and collateral tokens for Comet contracts. In particular, the following tokens are **not** supported:

- Tokens with more than 18 decimals
- Tokens with less than 6 decimals, e.g., GUSD
- Tokens with transfer fees
- Tokens where the balance can change without a transfer, these include:
  - Interest bearing tokens that increase balances
  - Deflationary tokens that decrease balances
  - Rebasing tokens
- Tokens with a missing return value on `transfer` or `transferFrom` (e.g., USDT)
- Tokens that require certain receiver functions to be implemented in contracts, e.g., `ERC223`
- Tokens with rapidly increasing/positively manipulatable prices (cannot be used as base token)
- Tokens with rapidly decreasing/negatively manipulatable prices (cannot be used as collateral token)
- Tokens with multiple entry points for which more than one entry point has been added to the contract's collateral assets.

Additionally the following tokens can break the protocol depending on their use:

- Tokens with blacklisting in case a Comet contract is blacklisted
- Pausable tokens when paused
- Upgradable tokens that later introduce one of the problematic features

7.6 The Fallback Function Is Payable

Note Version 1

The fallback function of `Comet` is payable even though none of the functions of `CometExt` are payable. Hence, there is no reason for the fallback function to be payable.

7.7 Transfers to 0-Address Allowed

Note Version 1

The functions `transferInternal` and `withdrawInternal` do not revert on transfers to the 0-address. As a consequence, the base asset and the collateral assets might accidentally be transferred to the 0-address.