Compound III Audit

JULY 20, 2022 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Delivered to Compound on June 14th, 2022. Fix updates will be added to this document periodically.

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

As part of OpenZeppelin's Security Partnership with the Compound DAO, we have audited a new upcoming version of the

Compound lending protocol developed by Compound Labs: "Compound III", codenamed `Comet` in this report. This new version will only contain one base asset and can have multiple instances deployed on Ethereum Mainnet and other EVM-compatible networks.

Type
DeFi Lending

Timeline
From 2022-05-13 To 2022-06-14

Languages
Solidity

Scope

We audited the compound-finance/comet repository at the `0f1221967149115f50a09681eea9580879ee7720` commit.

In scope were the following contracts:

```
- Comet.sol
- Bulker.sol
- CometConfiguration.sol
- CometCore.sol
- CometExt.sol
- CometExtInterface.sol
- CometFactory.sol
- CometInterface.sol
- CometMainInterface.sol
- CometMath.sol
- CometProxyAdmin.sol
- CometRewards.sol
- CometStorage.sol
- Configurator.sol
- ConfiguratorStorage.sol
- ERC20.sol
- IWETH9.sol
- TransparentUpgradeableConfiguratorProxy.sol
- vendor/
    - access/
        - Ownable.sol
    - canonical-weth/
        - contracts/WETH9.sol
    - @chainlink/
        - contracts/src/v0.8/interfaces/AggregatorV3Interface.sol
    - interfaces/
        - draft-IERC1822.sol
    - proxy/
        - Proxy.sol
        - beacon/IBeacon.sol
        - transparent/ProxyAdmin.sol
        - transparent/TransparentUpgradeableProxy.sol
        - ERC1967/ERC1967Proxy.sol
        - ERC1967/ERC1967Upgrade.sol
    - utils/
        - Context.sol
        - StorageSlot.sol
        - Address.sol
```

General overview

The `Comet` protocol is a lending protocol where there's only one specific base asset that can be supplied and borrowed, and that accepts other assets merely as collateral for base asset borrows. The main idea is to have one or more `Comet` instances with different base assets potentially being deployed on different networks. The `Comet` contract is the core of the protocol. This contract is configured with a certain number of assets accepted as collateral and manages all deposits and borrows. It also has a fallback function that delegate calls toward the `CometExt` contract, an extension of `Comet` that defines and implements some functionalities and that will let the governance add more features to `Comet` instances without hitting size limits in the contracts.

Users' supplies and borrows are tracked in each user position under the `principal` parameter. This parameter can be positive or negative signaling whether the user is supplying base asset or borrowing it out of the protocol. Depending on its value, the `principal` accrues either supply or borrow interests over time. Users might deposit collateral assets to cover their borrow and each collateral asset is configured with proper parameters such as borrow and liquidity collateral factors, the liquidation factor, its price feed, and an eventual supply cap to limit the protocol's exposure to it. Moreover, collateral assets deposited in the protocol don't accrue supply interests.

Suppliers of both collateral assets and the base asset can transfer or withdraw those deposits to other accounts, with the only condition to not create undercollateralized positions.

Like other lending protocols, `Comet` provides supply and borrow rates so that users are incentivized either to repay their borrows or supply the base asset to the protocol. These rates are managed by function curves that use two different slope parameters depending on the current interest value and the value of a *kink* parameter. The protocol also cuts out a certain amount of a supplier's earned interest for the `reserve`.
The `reserve` is the historical supply interest put aside and managed by the protocol itself. The protocol has a global parameter `targetReserves` which is meant to be an important factor in liquidations as we will see in a moment.

The `Comet` contract defines two conditions for a user position: – isBorrowCollateralized: will return true or false depending on whether the user liquidity is positive or not. Specifically, the liquidity is calculated as the sum between the value of the base asset deposits and the value of the collateral asset balances projected with the `borrowCollateralFactor` of each asset. If the liquidity happens to be greater than zero, then the user position is currently collateralized. – isLiquidatable: will return true or false if the liquidity is negative. This time the liquidity is calculated as the sum of base asset deposits and the sum of the collateral assets balances projected with the `liquidateCollateralFactor`.

This means that there's also a third state, where a user position might be under collateralized but not liquidatable. This is given by the spread between `borrowCollateralFactor` and `liquidateCollateralFactor`. Notice that the latter is always greater than the former. In this intermediate position, the user will not be liquidated but they will not be able to borrow more of the base asset. The user can instead repay the borrow to bring the position to a safe state.

Whenever the liquidity projected with the `liquidateCollateralFactor` is negative, the liquidation process can start and it consists of two phases:

- `absorb` phase: in this phase, all the collateral assets provided by the underwater account are seized and given to the protocol. This total amount is quantified in value using the `liquidationFactor` and the current prices of the assets. This implies a close factor of 100%. However, if there's any excess after covering the debt, this excess is accounted for as base asset credit to the underwater account. On the contrary, if there's a debt excess, the `reserve` will cover it in the first instance.

- `buyCollateral` phase: if, after the `absorb`, the `reserve` is under target, the protocol will let anyone buying seized collateral do so at a discount price, incentivizing people to remove collateral from the protocol and inject more of the base asset instead, reducing the protocol's exposure to potentially dangerous assets. If the `reserve` is above the target, the protocol will retain the collateral assets and will not allow the `buyCollateral` to be trigged. This situation is meant to cover the risk of maintaining such collaterals, given the amount in the `reserve`.

Notice that `absorb` and `buyCollateral` are two separate functions that are not always meant to be executed at the same time, since it depends on the target level for the `reserve`. Absorbers (users initiating an `absorb`) are accounted for the amount of gas that they spent to run absorbs, and those amounts are stored as `liquidator points`, which are then redeemable for rewards in a separate contract out of scope for the current audit.

Supply and borrow

rewards

Apart from interest rates, users earn rewards for supplying and borrowing. These rewards are tracked and accrued on separate tracking indexes, which are then read by the `CometRewards` contract.

This contract has a reward token set for each `Comet` instance which will be distributed according to each user's accrued rewards in the `Comet` contract. To receive their allocated reward tokens, users will call the `claim` function in the contract.

Configurator and Bulker

As mentioned before, different `Comet` instances can exist, either with different base assets or with different supported collaterals. For this, the team developed a system to easily manage such deployments and different configurations. This is what the `Configurator` contract is meant to do.

The `Configurator` contract will let the governance set all the parameters of a general `config` struct which holds all values needed for the `Comet` instance to be deployed with proper initial parameters. Whenever the configuration is ready, anyone can call the `deploy` function in the contract. This function will make use of `CometFactory` contract, which will deploy a new `Comet` instance with the configuration set so far in the `Configurator`.

The only parameter that can't be changed in the `Configurator` is the base asset for the new `Comet` which comes in the `Configurator` initialization. For this reason, there will only be one `Configurator` contract for each base asset that `Comet` instances might use.

Finally, another useful tool is the `Bulker` contract. This is an auxiliary contract that can be used to `supply`, `withdraw`, and `transfer` assets in and from the protocol performing multiple actions in just one transaction.

Update: *The team decided to change how the `Configurator` works allowing different configurations and factories to exist at the same time in the same contract. This will allow one `Configurator` contract per network supported, avoiding having multiple `Configurator` for different base assets and/or factories. The changes are introduced in PR#437 and PR#450.*

Governance and roles

Special roles and permissions are the `governance` and the `pauseGuardian`.

The `governance`:

- Is the owner of the `CometProxyAdmin`, being able to upgrade `Configurator` and `Comet` to new implementation contracts.
- Is the `governor` of the `CometRewards` contract, being able to configure reward tokens for each `Comet` or withdraw any token from the contract.
- Is the `governor` of the `Configurator`, being able to set all config parameters before deploying a new `Comet`.
- Is the `governor` of the `Comet` contract, being able to pause functionalities, withdraw reserves or approve a spender for `Comet` assets balances.

The `pauseGuardian` instead is only able to call the `pause` function of the `Comet` contract, as the `governor`, to pause functionalities.

Upgradeability

Both the `Configurator` and the `Comet` contracts are meant to be deployed using the transparent proxy pattern which gives the possibility to separate the logic and storage layer into two different contracts, the logic implementation

(`Comet` and `Configurator`) and the proxy contract which holds the storage. In this proxy pattern, the proxy is an instance of the `TransparentUpgradeableProxy` contract. This proxy differentiates between admin actions and user actions by using a specific modifier and redirecting calls where needed. Both the `Comet` and `Configurator` have their own `TransparentUpgradeableProxy`. Specifically, the proxy admin can operate upgrades of the implementation logic layers, while users are redirected through delegate calls directly to the logic contract. The administration of the logic contract relies on the `ProxyAdmin` contract, implemented in the `CometProxyAdmin` contract. This contract abstracts away adminship actions over its managed proxies and is owned by the governance. It has a special function called `deployAndUpgradeTo` which will deploy a new `Comet` instance through the `Configurator` and upgrade the proxy to the new `Comet` implementation. In order to do so, the `CometProxyAdmin` must be able to default toward the `Configurator.deploy` function and for this, the `TransparentUpgradeableProxy` instance of the `Configurator` overrides the `_beforeFallback` function and allows such behavior. Thanks to the `deployAndUpgradeTo` function the governance can deploy and upgrade to a new `Comet` in just one transaction.

`CometExt`, `Bulker`, `CometRewards` and `CometFactory` are not upgradeable. At the end of this introduction, we show a diagram that summarizes the current architectural design.

Design

Finally, we wanted to highlight some aspects of the current system design that deserve their own discussions.

- The protocol doesn't take into account non-standard ERC20 tokens. This is explicitly assumed in the code base docstrings and it is intentional. Assets that might charge fees or that don't return booleans or that in general don't adhere to the EIP-20 standard might produce unexpected system behaviors. This implies that a proper analysis should be conducted on each supported collateral asset to determine the possible side effects of implementing them. We recommend adding this to the user documentation. Moreover, the `Comet` contract itself adheres to the EIP-20 interface but its internal logic is non-standard. An example is `Comet`'s `balanceOf` function, which is the user's principal only if it is positive, if the principal is negative, the balance is zero. The main reason why `Comet` is adhering to an ERC20 interface is because of future composability features.
- There's a novel liquidation design.

Usually, closing only a percentage of the user position is enough to cover the underwater situation and might bring the user to healthy levels without the need of seizing everything in his position. This is known as the problem of over-liquidating a user, which, depending on the scenario, might be penalized too much. However, if there's any excess credit, this is given in base asset units back to the user. In this way the protocol will automatically collect collateral assets from all users being absorbed, exposing itself to those assets' risks. The protocol is intended to set the reserve target high enough to warrant the `buyCollateral` phase immediately after a `absorb`. If not set to a high value, it could cause major problems. But this alone doesn't guarantee that the protocol will get rid of those assets. If the liquidation started because one of more collateral assets started to crash in price, the protocol heavily relies on `buyCollateral` action to occur as soon as possible. If the spread between the supplied base asset and bought collateral value is less than the fees, liquidations are unprofitable and if the collateral asset crashes too much in price, it might start to reduce attraction from liquidators. A comprehensive research study shows that cumulated sold collateral over two years in Compound v2 was worth over 350M USD. This might be an object of study for long-term reserve target values to avoid accumulating reserves too fast and have the target block collateral sell-off. If for any of the mentioned reasons, there are overdue liquidations, the protocol might become illiquid and not guarantee to all base asset suppliers the ability to withdraw.

- The upgradeability mechanism presents some complex aspects:
  - On its own, the `Comet` contract will easily hit the 24kb size limit if more features or supported assets are added.
  - The `Comet` contract is upgradeable, but due to its size, it delegates calls to `CometExt`, which implements more functionalities. However, on its own, `CometExt` is not upgradeable and its address in the `Comet` instance can't be changed after construction. This means that if `CometExt` needs to change, a new `Comet` must be constructed, forcing an upgrade mechanism to the `Comet`. This is likely to happen since `CometExt` holds the `version` variable that should change on each new `Comet` implementation.
  - Contracts have long inheritance chains that might make it difficult the task to avoid storage collisions on future added variables.

- Some parts of the system are not upgradeable and have immutable parameters that might benefit from a change at some point in the future, like the `governor` in `CometRewards`.

The suggestion here is to carefully list all the actions and steps necessary to run safe and clear upgrades of the system, reduce codebase size where possible, and keep the system modular for easy opt-in/opt-out of future features.

General feedback

We are happy to see such a quality codebase. We didn't find any critical vulnerability and are glad to have robustness across the contracts even with novel designs. The documentation is helpful and makes the task of understanding the codebase easier.

Findings

Here we present our findings.

High Severity

Locked assets in contracts

Once the protocol is deployed, the `Comet` and `Bulker` contracts will be two important pieces of the system. `Comet` is the main protocol contract while the `Bulker` is a useful tool to execute multiple protocol calls into one single transaction.

For different purposes, both the `Comet` and the `Bulker` support deposits of ETH into the contract. In the case of the `Comet` contract, the delegation toward the `CometExt` is in the fallback function, which is declared as `payable`, but there are no parts of both contracts that use ether. Moreover, there is no direct way to withdraw any ETH balance present in the contract.

In the case of the `Bulker`, the contract has a `receive` payable function to accept ETH which is needed to be used in the `ACTION_WITHDRAW_ETH` according to the docstrings. In this case, any ETH sent through the `receive` function will be again lost in the contract, with no possibility to upgrade to a new contract as the `Comet` can. The reason is that the `ACTION_WITHDRAW_ETH` is implemented into the `withdrawEthTo` function. This function unwraps WETH by sending tokens to the WETH contract and receiving back ETH in the same amount in the `Bulker` contract which are then sent to the user. In this case, the contract's ETH balance will be untouched as only ETH coming from the unwrap will be used.

Lastly, a separate reference must also be done on ERC20 tokens. The `Comet` contract has the `approveThis` function which is enough to let a `manager` move any ERC20 funds that might get lost in the `Comet` balance. However, this is not the case for the `Bulker` contract, where ERC20 tokens might also get lost.

Consider establishing mechanisms to avoid such scenarios, as it results in a direct loss of user funds.

Update: *Partially fixed in commit 3681613. In the words of the team: "We think it's a good idea to add sweep functions to the `Bulker` to prevent funds from being locked in there. As for `Comet`, we purposely made the `receive` function `payable` in case we ever wanted to support a `payable` function in `CometExt`. Doing so allows us to add a `payable` function to `CometExt` without having to also upgrade `Comet`. Since `Comet` is upgradeable, I don't think we need to support a way to sweep ETH out of the contract right off the bat".*

Medium Severity

`governor` can approve anyone to transfer the base and collateral assets within the Comet contract

The `COMP` token, the governance module

(`GovernorBravo`) and the `Timelock` are the three components that make up Compound governance, allowing for the management and upgrade of the protocol.

The `Timelock` contract, which is also the administrator of the Compound v2 protocol, is meant to be in charge of all instances of the `Comet` protocol. Each proxy, the `Configurator` implementation, the `Comet` factory, the `CometRewards` and the `Comet` implementations are all under the governance control.

Within the `Comet` implementation, there is a variable called `governor` that will be set to be the `Timelock` address. The contract also has a function called `approveThis` that only the governor can execute and it approves anyone to transfer the base and collateral assets outside the `Comet` contract. Taking into account that the possibility of a governance attack exists (Beanstalk and Curve cases), user funds could be at risk.

If the function is meant to be used for specific purposes and can't be removed, consider properly documenting this risk and establish verification measures on governance proposals to avoid misuse of this feature. To reduce the possibilities of a governance attack, be sure to always rely on a delay mechanism for proposals to be executed and eventually a pause guardian that can take action if a malicious proposal is spotted.

Update: *Fixed. The team has improved the docstrings in PR#414. In the words of the team: "The main intention for `approveThis` is to allow governance to transfer out any ERC20s accidentally sent to Comet. We do recognize the ability for governance to give approval for the base and collateral assets and transfer out user funds. However, as OpenZeppelin has noted, this will likely require a governance attack. We'd like to point out that in the case of a governance attack, the attacker would not even need `approveThis` to steal user funds as they could upgrade the implementation of Comet to whatever they please."*

The protocol may end up holding collateral assets in an unwanted manner

The Comet contract has an immutable value that defines the target amount of reserves of the base token. This value is closely related to the buyCollateral function. This function cannot be called successful if the protocol reserves are greater than or equal to this target.

If `targetReserves` is set to a small value, the contract could easily reach the level. The problem is that the absorptions can continue but the protocol will not be able to sell the collateral because the `buyCollateral` function cannot be used and the protocol could be in a situation where it would hold assets that may lose value over time.

In the opposite case, where `targetReserves` is set to a large value, the chance of reaching this level would be much lower so it could be a useless constraint.

Keeping in mind that setting this variable to a small value is more of a problem, be sure to set it to a large value. Also if the value of the target is too high to not have a useful or practical use, consider re-design the system to not make use of it.

Update: *Acknowledged. In the words of the team: "We intend for `targetReserves` to be a pretty large value so the protocol can use liquidations to build up a sizable reserve. Once reserves have reached `targetReserves`, we believe it may be advantageous for the protocol to start HODLing the collateral assets. We've run backtesting simulations to identify this as the best strategy for the protocol to build up reserves, but this strategy can definitely change as we conduct more research around liquidation auction strategies."*

Incorrect accounting of used gas

The `asborbInternal` function of

the `Comet` contract contains important logic of the protocol where users that are liquidatable have their debts absorbed by the protocol. To do this task frequently and maintain health in the system, users will call this function whenever they detect a liquidatable user position.

As a reward for doing this task recurrently, absorbers (users calling the `absorb` function) are accounted for their gas expenditure into liquidation points, with the promise of redeeming those points for reward tokens in a separate contract.

The reward mechanism for the liquidation points is out of the scope for the current audit so we can’t assess the incentives alignments in performing this task with profitable rewards.

However, how the gas used is measured doesn’t reflect entirely the actual transaction cost for the user. In particular:

- The priority fee is not taken into account. Absorbers will probably have to compete with each other and be as fast as possible in running absorbs. For this is likely to have a priority fee set as a miner’s tip in the transaction. Currently, the protocol only uses `block.basefee` but `tx.gasprice = block.basefee + priority fee` should be used instead.
- Other operations are performed after the

gas spent is measured, consuming more gas which is not taken into account.

- Potentially, a user could deposit a minimum amount of each of the collateral assets supported by the protocol to increase the cost of the transaction, since it implies iteration over all supported assets recursively. Doing so, the rewards increase proportionally to the cost and it might be used maliciously by an underwater account since they might absorb their own position to earn rewards and potentially reduce damage from the liquidation.

Consider taking these suggestions into account and changing the way the gas used is measured to improve transparency and design correctness.

Update: *Acknowledged. The team has improved docstrings in commit 14fbc27.*

Low Severity

Everyone can deploy new Comet instances

Each `Comet` contract implementation that the protocol might decide to create is meant to be deployed in the `Configurator` contract.

This contract is an intermediate logic to bootstrap the entire protocol configuration and then apply it to a newly deployed `Comet` contract.

To do this, the `Configurator` makes use of a `CometFactory` contract which has a `clone` function whose sole role is to create a new `Comet` contract and return its address.

Whenever all protocol parameters are set, the `deploy` function in the `Configurator` contract is called. This function calls the `clone` function in the factory and emits an event to signal that a new `Comet` is ready with the configuration set so far.

The `deploy` and `clone` functions are both public callable functions, so anyone can, at any moment, generate the `Comet` instance either in the factory, passing arbitrary configurations, or either through the `Configurator` with the configuration set until that moment.

Is not clear why those functions are public as there is no benefit for users to call them. Moreover, leaving open the door for anyone to deploy instances of `Comet` with arbitrary configuration or emitting arbitrary events through the `Configurator` is a concern to take into account as it might be misused by malicious actors trying to create pishing-like attacks.

Consider restricting access to those

functions and let them be called only by the governor or consider properly documenting such a design decision, raising awareness over the way it might be misused.

Update: *Not fixed. The team acknowledge the issue and in their words: "We considered making `Configurator.deploy()` a governor-only function, however it would require adding a storage slot and complicating the Configurator which does not seem worth the tradeoff. It also doesn't make sense to change `CometFactory.clone()` to be governor-only as the `CometFactory` is a stateless contract. We note for future reference that there isn't a great way to get a list of official Comet instances from on-chain, however the canonical repository does have a list (`roots.json`) for each chain, which seems sufficient until a better solution is established."*

Gas inefficiencies

There are many places throughout the codebase where changes can be made to improve gas consumption. For example:

- The `for` loops inside `absorb` and `invoke` functions do not cache the length of the array and perform unnecessary operations on each iteration.
- The `unchecked` blocks for loop counters are not used consistently throughout the

codebase. It is only used in the `Invoke` function, although it could be implemented in loops of the rest of the protocol contracts.

- updateBasePrincipal in `Comet` should not check `principal == 0` because in that case nothing will happen and gas will be wasted.
- It is advisable that if a variable from the contract's storage is going to be read within a function several times, a copy in memory should first be created since reading to the storage directly is expensive. However, if you are only querying for the value once, as in accrueInternal and `Comet`'s fallback, it is recommended to read directly from storage without any intermediate steps.
- In `transferCollateral` within the `Comet` contract, it is recommended to consult with `getAssetInfoByAddress` at the beginning of the function so that in case the token passed by the parameters is not within the collaterals of the protocol, the function fails early and without wasting gas unnecessarily.
- Use `unchecked { ++i; }` over `unchecked { i++; }`.

When performing these changes, aim to reach an optimal tradeoff between gas optimization and readability. Having a codebase that is easy to understand reduces the chance of errors in the future and improves transparency for the community.

Update: *Partially fixed in commit 90ca4d0.*

Inconvenient use of `immutable` keyword

For accounts that use the protocol, the protocol provides a built-in system for tracking rewards. `Comet` contracts keep account of all accrued incentives for suppliers and borrowers of the base token, and users can claim them on `CometRewards` contract. It should be emphasized that all rewards from all the `Comet`s from the same chain are claimed in the same `CometRewards` contract.

The latter has a `governor` variable defined which is the role that can set the reward settings and withdraw rewards from the contract. Since the contract does not have an upgradeability mechanism, it is inconvenient to define this variable as `immutable`. If the `governor` needs to be changed, a new contract must be deployed and the old contract's state must be migrated to the new one.

Consider adding this variable to the contract storage and specifying a setter function so that the `governor` can be changed any time it is needed.

Update: *Fixed in*

commits 43b5502 and 4d1c1a4.

Logic contracts initialization allowed

The `Configurator` contract has an initializable function that is meant to be called by the proxy. However, nothing prevents users from directly calling the `initialize` function on the contract implementation. If someone does so, the state of the implementation would be initialized to some meaningful value.

We suggest adding a constructor that sets the version to an incredibly high number so that any attempt to call the implementation at the initialize function would revert with an `AlreadyInitialized` error or even a specific one to signal that initializing the implementation is prohibited.

Leaving an implementation contract not initialized is generally an insecure pattern to follow. In this case, it might lead to a situation where an attacker can take control over the implementation contract by passing himself as `governor` and try to mislead users toward malicious contracts. This is not a security issue on its own but we strongly suggest avoiding such scenarios in all implementation contracts.

As a source of inspiration, here there's an example of how the scenario can be avoided in general situations.

Update: *Fixed in commit 79f59e5.*

Missing validations

There are some places in the code base that might benefit from some sanity checks on the input provided:

- The `transferGovernor` and `setGovernor` functions of the `Configurator` are not checking the address to be non-zero.
- The `CometRewards` constructor is missing the same check over the address parameter.
- In line 260 of the `Comet` contract, the `priceFeed` is set but is not checked to retrieve a valid price.
- `withdrawAndBorrowAmount` and `repayAndSupplyAmount` functions assume certain values over the `newPrincipal` but those should be required instead.

To reduce possible errors and make the code more rodust, consider adding sanity checks where needed.

Update: *Partially fixed in PR 455, merged commit bf20ccf. `withdrawAndBorrowAmount` and `repayAndSupplyAmount` new validations but the rest of the items will not be fixed. In the words of the team: "There are arbitrarily many bad addresses which can be set, checking for the zero address seems like added complexity for little gain. In addition, while further checks in Comet.sol could be added, the contract is being* *optimized for efficiency and is up against a size limit, so we favor the current approach".*

Potential function clashes

The `TransparentUpgradeableConfiguratorProxy` overrides the `_beforeCallback` function.

Concretely, the `_beforeCallback` function is implemented in the `TransparentUpgradeableProxy` contract to avoid the `admin` of the contract to call the implementation logic directly.

This feature is removed in `TransparentUpgradeableConfiguratorProxy` contract, where the `admin` is allowed now to call directly the implementation by triggering the fallback function as a normal user would do. This is needed because of the `deployAndUpgrade` function of the `CometProxyAdmin` which sees the `admin` calling the `Configurator.deploy` function.

This is not a security issue on its own but it opens the door for potential clashes to happen. If one function is added either on the proxy or the logic contract, this can clash with any of the other contract functions. At this point, the admin will stop to be able to call the implementation contract (users will still be directed toward the implementation because of the `ifAdmin` modifier).

This article specifies how crafting clashes may not be too hard of a computational task. This article showcases how a new function in the proxy might actually enable clashes.

To avoid any unwanted behaviors, consider ensuring that the upgrade mechanism for `Configurator` always checks for potential clashes between the logic implementation and the proxy, especially if new functions are added.

Update: *Fixed. The team added an off-chain check to avoid collisions in PR#430. Also, the team acknowledges the issue providing the following comments: "The two types of clashes that can happen are: – New function on proxy is introduced that clashes with an existing function in the Configurator. Admin is no longer able to call the function on the Configurator. To recover: The admin should still be able to upgrade the implementation of the proxy because that function lives on the proxy. Governance can simply introduce a new version of Configurator without the clash and upgrade the proxy to this new implementation. – New function on Configurator is introduced that clashes with an existing function on the proxy. Admin cannot call this new Configurator function. To recover: Same recovery path as above. Function clashes only prevent the admin from calling a function on the Configurator, so the admin is still able to call the upgrade function* *on the proxy itself.*

In either case, the contracts are in a recoverable state and nothing malicious can happen unless the admin is malicious.”

Underwater accounts can minimize losses

In the `Comet` contract, during an `absorbeInternal` call, the `updateBasePrincipal` function is called, updating the accrued interests on the liquidated user position.

If the seizing of collateral brought the new user’s balance to positive values the user will:

- Have his excess collateral exchanged for the base asset (during an absorb this might be beneficial if the collateral price is crashing).
- Have supply interest and tracking indexes accrued straight away after the absorb, over the excess collateral converted into the base asset. Concretely, the `accrueInternal` will update interest rates and the `updateBasePrincipal` will update the tracking indexes.

Moreover, an underwater can decide to even liquidate himself, earning additional liquidation points.

Consider the extra value that an underwater can extract from his position to determine if this can be leveraged to create some attacks. If those are intended behaviors, consider improving the docstrings around the absorption mechanism to reflect these details.

Update: *Acknowledged. In the words of the team: "It sounds like the concern is that a user might absorb themselves for liquidator points and to sell off their collateral to the protocol quickly during a market downturn. In that case, it's the purpose of the collateral factor to ensure a buffer in which the account becomes liquidatable and is still profitable to the protocol to liquidate. If a user chooses to absorb themselves it would economically imply governance has somehow created an absorption incentive greater than the liquidation penalty."*

Unnecessary complexity

The `Comet` contract delegates some feature implementations into the `CometExt` contract. This contract is meant to be an actual extension of `Comet` logic and implement some functions and parameters, including the `version` parameter.

At the same time, `Comet` is built through an upgradeable proxy pattern that requires a new `Comet` version to be deployed and the proxy pointed toward the new implementation.

During an upgrade, it is convenient to

upgrade the version number to a greater value and to do so the protocol must deploy also a new `CometExt` implementation contract to indicate a new version.

This is because the `version` parameter is declared as constant and can't be changed as a result of an upgrade mechanism without actually changing it in the contract's code.

In the worst-case scenario, an error is performed in the upgrade mechanism, the version number is not incremented and potential systems integrated with the protocol might depend on that version number to establish some other logic.

To avoid deploying a new `CometExt` even if its implementation didn't change, consider moving the version parameter into the `Comet` contract code.

Eventually consider keeping `version` in both contracts to differentiate between them, as both contracts might require some implementation changes.

Update: *Acknowledged. In the words of the team: "The `version` in `CometExt` is meant specifically as part of the `EIP712Domain` of the contract. Only changes which affect this domain should be reflected in this version. Furthermore the logic relating to the domain is currently isolated to the extension contract, which is useful because it reduces the main contract size, and the size of the* *main factory."*

Notes & Additional Information

Anyone can set `baseTrackingIndex`

In the `Comet` contract, the `accrueAccount` function is publicly callable and it accrues interest rates and updates tracking indexes on the account specified as the input parameter.

Specifically, the tracking indexes are updated in the `updateBasePrincipal` function where if `newPrincipal >= 0` then the `baseTrackingIndex` is set to a meaningful value.

So anyone is able to set this value for accounts that do not even exist on the protocol. Even if it is not a security issue on its own, consider restricting this function to users that have a principal strictly greater than zero.

Update: *Not fixed. However, the team added a unit test in commit d4abcb2 to check that it doesn't affect the protocol.*

Inconsistent coding style

Inconsistencies in coding style were identified throughout the code base. Some examples are:

- constants variables `version` and `name` and not in UPPER_CASE like other constant variables
- The convention of functions named with the "_" prefix is not clear. Sometimes it is used for admin functions and other times for internal functions
- Some structs use `_reserved` to fill slots, but others do not.
- It is not clear what convention is used for the naming of codebase interfaces. Sometimes the letter `I` is used as a prefix, sometimes the word `Interface` at the end or in some cases the contract even misses both.

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

Update: *Partially fixed. Only the second item has been fixed in commit c04c056.*

CometMath is not used consistently

Many functions that could be in `CometMath`, a utility contract that contains pure mathematical functions, are scattered throughout the codebase and `min`, one of the functions within the contract, is not being used.

Consider consolidating all the helper math functions in one place and removing unused ones.

Update: *Partially fixed in commit 16d213f. The `min` function has been removed.*

Compilation warnings

While compiling the codebase contracts. The compiler raises some warnings. Specifically:

- The `totalSupply` and `totalBorrow` variables of the `getUtilization` and of the `getReserves` functions have the same name as the `totalSupply` and `totalBorrow` public functions.
- The `withdrawAndBorrowAmount` function visibility can be restricted to `pure`.

Consider resolving all compiler warnings.

Update: *Fixed in commit 9e2b195.*

Incorrect or missing docstrings

Across the codebase, some contracts lack proper documentation and docstrings. In particular, the `IWETH9` and `CometMath` functions or the `CometConfiguration` and `CometStorage` variables are having little comments or nothing at all.

Consider thoroughly documenting all functions and parameters that are part of the contracts' public API. Functions or parameters implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Moreover, the `Configurator` contract explicitly states that `BaseToken` and `TrackingIndexScale` have no setters but it doesn't specify where those are set, eventually in the `config` parameter passed as input in the `Configurator` initialization.

Consider writing this in the docstrings of the `initialize` function to improve clarity.

Update: *Partially fixed in commit 9227075. Docstring was only changed in the* `Configurator` *contract.*

Lack of indexed parameters

The `CometDeployed`, `GovernorTransferred`, `SetFactory`, `SetGovernor` lack of any indexed parameter.

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

Update: *Fixed in commit 7124b98.*

Contracts folder is not properly organized

There is no convenient structure in the `contracts` folder to easily navigate between them. All contracts, regardless of their type or module they belong to, are mixed in a single folder.

To favor the developer experience and the maintenance of the codebase, consider adding additional folders following the structure within the `vendor` directory or separating protocol components into different internal folders.

Update: *Acknowledged. In the words of the team: "The code actually is logically organized in that the protocol code sits at the top-level, with test and vendored contracts being secondary and sitting in subdirectories". In the future, the team might reorganize into subdirectories but not at this time.*

`rewardsClaimed` can be mixed between different tokens

In `CometRewards` it is possible to change the reward token of each `Comet` through `_setRewardConfig`. However, if the reward token is changed, the number of previous reward tokens claimed will persist and once someone claims their new reward asset, it will be added to `rewardsClaimed` despite being different assets.

Consider removing the ability to change the reward asset once set or changing the way the claimed rewards are stored if the reward asset changes.

Update: *Fixed in commit ced8026.*

Naming issues

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Here are some examples:

- The name `CometProxyAdmin` suggests that it is only the admin of the Comet proxy, but in reality, it will also have the same role on the `Configurator` proxy. Choose another name to avoid confusion.
- `TransparentUpgradeableConfiguratorProxy` can be called `ConfiguratorProxy`. There is no need to use the inherited contract name as a prefix.
- In `CometMath`, change `InvalidUInt` to `InvalidUint` and `toUInt` to `toUint`.
- `rescale` and `descale` have different names but the same value, consider using one variable name `scale`.

Update: *Partially fixed in commit b5a64d0.*

Use of Global imports

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

The inheritance chains within the protocol are long, and for this, the codebase would benefit from refactoring specifically which definitions are being imported in each contract.

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.

Update: *Acknowledged. In the words of the team: "We do not consider this important or worth the effort of refactoring at this time".*

Potential front-run

The `deployAndUpgrade` function of the `CometProxyAdmin` is restricted in access to be called exclusively by the governance. It deploys a new `Comet` instance through the `Configurator` and upgrades the implementation address in the proxy.

However, it doesn't call the `initializeStorage` function of the `Comet` contract through the proxy, leaving the new implementation not initialized. The function doesn't take any input parameter and it is meant to be called only once without the need to call it again on new implementation upgrades.

Whether the first initialization is performed on a separate transaction or if in the future it will be possible to re-initialize the `Comet` instance with some input values, any user can front-run any governance attempt to initialize the new deployed `Comet`.

Consider taking into account that `deployAndUpgrade` and `initializeStorage` should be done in one unique transaction to avoid the front-run scenario, especially if input parameters or re-initializations are meant to happen in future developments.

Update: *Acknowledged. The team added a scenario in PR#431 to test the governance flow for upgrading to a new version of Comet and calling its initialize function via one proposal.*

Potential reentrancies

In the codebase, we found two places where reentrancy can occur. However, those do not pose any security issue or concern but awareness should be raised:

- The `invoke` function of the `Bulker` contract can be re-entered.
- The `doTransferIn` function of the `Comet` contract is often executed at the very beginning of executions, being it

an anti-pattern to follow against reentrancy. Even if re-entering the same `doTransferIn` can't be used as vector attack in this case, one can re-enter a different function, modifying the state in an unexpected manner. Making external calls is always suggested to be done after *checks* and *effects*.

To improve clarity, consider either reducing the attack surface by making those functions non-reentrant or document and raise awarness on such scenarios.

Update: *Acknowledged. The team understands the concerns and replies: "Specific concerns due to reentrancies should be addressed with tests, static analysis or formal verification or code changes. Excessive documentation notes about universal theoretical concerns don't likely add much except make the documentation harder to read and maintain".*

Repetitive code

In many parts of the codebase, repeated or similar lines of code can be seen. Here are some examples:

- In Comet, lines 520-525, 555-561, 593-598 and 628-634 are repeated and loops after those sections have strong similarities.
- `timeElapsed > 0` check is performed

two times, when accrueInternal is called, and when internally accruedInterestIndices is called.

- `allow` and `approve` in `CometExt` do the same thing.

Consider reusing the same code defined in just one place or, if appropriate, removing duplicate code.

Update: *Acknowledged. In the words of the team: "We gave considerable thought to how redundant sections of the code were organized and tried various permutations. Although seemingly redundant, the current form is overall the best we found in terms of the properties we were evaluating (namely clarity, code size, gas cost)".*

Typos

Line 96 of the `CometExt` contract has a typo "whi" and the number of decimals showed here is wrong, as it should be one zero less.

To improve correctness and readability, consider reviewing both the contracts and the documentation for typos.

Update: *Fixed in commit a1bd99f.*

Unnecessary return values

In the `Comet` contract, there are functions like the `supply` or `withdraw` that return calls to internal functions even if there's no actual final value returned at the end.

Consider reviewing all the occurrences where this happens and avoid returning when not necessary. This will improve readability and correctness.

Update: *Acknowledged. In the words of the team: "These are used to handoff control flow and can actually improve readability with that understanding".*

Lack of explicitness on data type sizes

The protocol heavily relies on different sized variables, which can have also positive or negative values and different scaling factors. Thanks to this, the deployment and execution of the codebase will decrease gas costs.

Given the fact that it adds some complexity and undermines the readability of the codebase, it is of utter importance to maintain explicitness and consistency across different contracts.

Specifically, implicit castings and sizes should be avoided.

To improve the overall quality of the codebase, consider reviewing it in its entirety, changing all occurrences of `uint` to `uint256` and of `int` to `int256`. Consider also reviewing implicit castings from small to bigger sizes and always use the appropriate size for each variable.

Update: *Partially fixed in PR#421. The team will continue to use the aliases `uint` and `int` as they consider them more readable. In their words: "We removed all occurrences of implicit upscaling. We accept that `uint256` and `uint` are interchangeable in Solidity and that authors are aware of this". Also, in PR#454, the team fixed an unsafe cast and improved gas usage.*

`PRICE_SCALE` constant is not used

A constant called `PRICE_SCALE` is defined in the `CometCore` contract and is supposed to be used to scale the prices returned by Chainlink aggregators.

Although the function `priceScale` returns the value stored in the bytecode, the constant is not used anywhere else.

Consider removing this constant or alternatively integrating it into the codebase.

Update: *Acknowledged. The team response: "`PRICE_SCALE` is not used internally, but it is exposed via the `priceScale` function, which could be used to understand the results of the getPrice function".*

Wrong value emitted in event

Lines 1219-1221 of the `asborbInternal` function in the `Comet` contract are:

```
uint104 debtAbsorbed = unsigned104
(newBalance - oldBalance);
uint valueOfDebtAbsorbed = mulPrice
(debtAbsorbed, basePrice, uint64(ba
seScale));
emit AbsorbDebt(absorber, account,
debtAbsorbed, valueOfDebtAbsorbed);
```

Where, during an absorb `oldBalance` is negative (otherwise the `isLiquidatable` modifier at the beginning of the `absorb` function would have exited earlier) and `newBalance >=0`. But if `newBalance > 0` the `debtAbsorbed` should be `unsigned104(-oldBalance - newBalance)` instead since the positive excess is not actual debt that has been absorbed.

Consider emitting the correct value in the `AsborbDebt` event or renaming the `debtAbsorbed` variable to reflect that it does not account only for the debt absorbed but also the excess collateral exchanged for the base asset.

Update: *Fixed in commit edc5a0a.*

Conclusions

No critical severity issues have been found, along with one high severity issue and many others lower in severity. We strongly recommend addressing even the lowest in severity since they would drastically improve the overall quality, clarity, and security of the protocol.

Update: *The team has addressed all issues either fixing them or providing improved docstrings and proper explanation arguments. Some of the changes introduced in the codebase are still open as pull requests and not incorporated into the main codebase. We assume those changes that we reviewed to be merged as they are without introducing any new change that might create new issues.*

Appendix

Monitoring Recommendation

The active monitoring of smart contracts is an important practice that can represent additional protection for a project, allowing an immediate response to unforeseen incidents. We recommend implementing monitoring for all sensitive actions and the state of critical variables. For instance:

- Monitor if `Comet` contract reaches the `targetReserves`. At that point, the collateral tokens seized with the liquidation will be retained by the protocol, and this could represent a risk if it relates to market prices precipitating in prices.
- Monitor callable governance functions, especially `approveThis`, and upgrades to malicious implementations in governance attack scenarios.
- Direct transfers of base asset to `Comet` contract (either to report an erroneous transaction or attempted manipulation).
- Large capital liquidations. Either the `absorb` function is executed for a large position or the position exceeds the `borrowCollateralFactor` and is at risk. This should inform a possible strong market price drop or the liquidation of a large entity.

RELATED POSTS

ANNOUNCEMENTS

Base Developers Can Now Access OpenZeppelin's Smart Contract Security

READ MORE

SECURITY AUDITS

Security Review – inl cargo-contract

The security review of ink! and carg contract has been deemed success and no critical issues...

READ MORE

OpenZeppelin

Email*

Get our monthly news roundup

protected by reCAPTCHA
Privacy - Terms

SIGN UP

Products

Contracts

Defender

Company

Website

About

Jobs

Logo Kit

Security

Security Audits

Learn

Docs

Forum

Ethernaut