Compound COMP Distribution System Audit

OPENZEPPELIN SECURITY | MAY 22, 2020

Security Audits

The Compound team engaged us to audit their new COMP distribution system, which will be added to the Comptroller. The audited commit is `a303a3259286869d0a4aae967d7cb9e55db39c37`, and the majority of the changes are in the `Comptroller.sol` file together with necessary storage updates in `ComptrollerStorage.sol`.

High-level overview of the changes

The new COMP distribution system is designed to distribute COMP tokens to users based on their usage of the protocol. The protocol first allocates a `compRate` amount of COMP tokens per block, which are then allocated to all eligible markets in proportion to `compSpeed`. In each market, COMP is then distributed to all borrowers and suppliers according to their borrow and supply amount. Every market has a “Supply Index” and “Borrow Index” for tracking the accrued COMP token per unit in this market (the unit is borrow token for borrowers and `cToken` for suppliers).

Additionally, in every market there is a “Supplier Index” and a “Borrower Index” to track each user’s progress with corresponding indices. Every time a user takes an action that accrues COMP, the delta between the “Supply Index” and “Supplier Index” is multiplied with the user’s unit to calculate how much COMP they’ve accrued. The system will automatically distribute COMP to users once the amount they’ve accrued is over a threshold.

in the given commit. We outline below some of the steps in the process we followed.

We closely inspected how the COMP distribution system is designed and how COMP was tracked throughout the life cycle of participants. The design of the system requires an update on `CompBorrowState` and `ComSupplyState` whenever the borrow and/or supply balance of a market changes. We carefully inspected the code to ensure that all related actions correctly trigger the update of those states. Similarly, we checked to ensure the `distributeBorrowComp` and `distributeSupplierComp` functions are called whenever a user's borrow and/or supply balance may change.

We looked into the mechanism of COMP distribution to ensure balances are properly tracked, as well as for the possibility of reentrancy during this process. We found an edge case of COMP distribution when `userAccrued > compRemaining; userAccrued > threshold`, in which case the system could return `userAccrued.sub(compRemaining)` which will send the user the remaining COMP. We decided not to report this as an issue because it ultimately complies with the specification.

The first time a supplier interacts with the contracts, their `supplierIndex.mantissa` is changed from 0 to `compInitialIndex`. This means they have a positive supply index delta during their first interaction with the contract after the COMP distribution system is deployed. So users who were supplying assets before the COMP distribution system was launched will earn an appropriate amount of COMP on the supply they had during the time after the distribution system was deployed and the time of the user's first interaction with the contracts.

Since COMP distribution happens before new cTokens are minted (and before any cToken transfers complete), this initial positive index delta cannot be exploited to get an unfair amount of COMP. For example, an attacker may attempt to flash-borrow a lot of some underlying asset, and then deposit it into Compound as supply during their initial interaction with Compound, in the hopes that the large supply would be multiplied by the positive initial index delta and result in them earning COMP that they didn't deserve. But this attack would not succeed because the calculation of the number of COMP tokens to reward is done before the attacker's new cTokens would be minted. In other words, we found no possible way for a user to increase their cToken balance after

We think this release is well structured, the code is clean and easy to read. The Compound team has done a good job implementing the functionalities required by the COMP distribution system. We were unable to identify any further issues at this stage.

Critical severity

None.

High severity

None.

Medium severity

None.

Low severity

None.

Notes & Additional Information

[N01] Unnecessary public visibility in some functions

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

- In `Dripper.sol`, the `drip` function
- In `Comptroller.sol`, the `claimComp`, `_dropCompMarket`, and `getCompMarkets` functions

To reduce gas costs, consider changing the visibility of these functions to `external`.

Conclusion

Related Posts

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

Threat Monitoring
Incident Response
Operation and Automation

Zero Knowledge Proof Practice

Blog

Company

About us
Jobs
Blog

Contracts Library

Docs