OpenZeppelin

# Compound Finance

## Multicollateral DAI and DAI Savings Rate Integration

OPENZEPPELIN SECURITY | FEBRUARY 9, 2020

Security Audits

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

## Audit history and current scope

We originally audited a subset of Compound's contracts at commit `f385d71983ae5c5799faae9b2dfea43e5cf75262` of their public repo. We then audited a patch that introduced a time delay for critical admin functions and pauseability to some other functions. That patch is reflected in commit `681833a557a282fba5441b7d49edb05153bb28ec` of Compound's public repo.

Next, we audited a refactor of the core `CToken` contract whose purpose is to accommodate underlying tokens that may extract a fee when transferring tokens (e.g., USDT). This refactor is presented in commit `2535734126c7c26e9bc452f27f45c5408acff71f` of Compound's private repository.

Then we audited the difference between the code at commit `2535734126c7c26e9bc452f27f45c5408acff71f` of Compound's private repository and commit `bcf0bc7b00e289f9b661a0ae934626e018188040` of their public repository.

In this audit, we are auditing only the difference between commit `bcf0bc7b00e289f9b661a0ae934626e018188040` and commit `9ea64ddd166a78b264ba8006f688880085eeed13` in Compound's public repository.

In particular, we are auditing all contracts in the `contracts/` directory that have been added or changed between the above two commits. These include the changes to the `JumpRateModel` contract and the two newly added files `CDaiDelegate.sol` and `DAIInterestRateModel.sol`.

We did not audit Maker's contracts.

**Here we present only the new issues that we found when auditing this latest difference between commits. Issues found in our previous reports may still apply.**

## High-level overview of the changes

This code introduces changes to accommodate Multi-collateral DAI and the new DAI Savings Rate contracts. It allows Compound to introduce a new CToken backed by DAI that will allow suppliers of DAI to earn both the DAI Savings Rate (DSR) *and* interest paid by borrowers of DAI. This is achieved by automatically sweeping all user-supplied DAI into Maker's DSS contract, so that any DAI in Compound that has not been loaned out to borrowers will be earning the DSR. Income from the DSR is distributed *pro-rata* to DAI suppliers. This logic is handled by the new `CDaiDelegate` contract.

The `CDaiDelegate` contract uses the new `DAIInterestRateModel` contract to determine the borrow rates and supply rates. The minimum value of the supply rate is equal to the DSR, with the intention being that users will always earn at least as much (non-risk-adjusted) interest by supplying their DAI to Compound as they would by supplying their DAI directly to Maker's DSS.

Additionally, the new interest rate model ensures that the borrow rate will always be strictly greater than Maker's stability fee when Compound's DAI utilization rate (the percentage of Compound's DAI that has been loaned out) meets or exceeds a value called the `kink`. The intention here is to ensure that borrowers are better off borrowing DAI directly from Maker whenever the utilization rate is above the `kink`. This provides some economic pressure on the utilization rate in the hope

This analysis assumes:

- An honest/uncompromised and live price oracle.
- For all `CToken` contracts, the `CToken` `admin` is an instance of `Timelock`.
- The `CToken` `comptroller` is an instance of `Unitroller`.
- The `Unitroller` `admin` is an instance of `Timelock`.
- The `Unitroller` `comptrollerImplementation` is an instance of `Comptroller`.
- The `Comptroller` `admin` is an instance of `Timelock`.
- The `Comptroller` `pauseGuardian` is not an instance of `Timelock`.
- The `Timelock` `admin` is not an instance of `Timelock`.
- The Maker contracts are secure, bug free, and work as intended. Note that we did not audit any of Maker's contracts!

Here we present our findings.

## Critical severity

None. 🙂

## High severity

None. 🙂

## Medium severity

### [M01] daiJoin.cage prevents withdrawals

Part of the functionality of the DSS system is the ability of the Maker admins to call the function `cage` in join.sol, which sets `live = 0`. If this happens, the call to `daiJoin.exit` within CDaiDelegate's `doTransferOut` function will revert upon reaching the `require` statement on line 169 of join.sol. Users would not be able to withdraw their funds.

contract ever be “caged”?

### [M02] Negative DSR causes unexpected reverts

On line 66 of `DAIInterestRateModel.sol`, `1e27` is subtracted from `dsr`. If for any reason, `dsr` is less than `1e27` (which corresponds to a “negative” interest rate), any calls to the `dsrPerBlock` function will revert. This includes all calls to the `poke` function and the `getSupplyRate` function.

While the Maker developers have said they do not have plans to ever allow `dsr` to be less than `1e27`, this could still happen via the Maker governance system.

Reverting on `poke` could prevent updating the `baseRatePerBlock` and `multiplierPerBlock` state variables.

Consider modifying `dsrPerBlock` such that it returns `0` when `dsr &lt; 1e27` (corresponding to a DSR of 0%). Also consider implementing a mechanism to remove DAI from the DSR contract, and to stop deposits into the DSR contract, just in case `dsr` is ever made less than `1e27`.

### [M03] Unnecessary calls to drip function of pot contract

When `rho == now`, `pot.drip` is a gas-intensive no-op. This means every call to the `CDaiDelegate` `accrueInterest` function (within a given Ethereum block and after `pot.drip` has been called) calls `pot.drip` unnecessarily. Consider wrapping line 103 of `CDaiDelegate.sol` in an `if (PotLike(potAddress).rho() != now) { ... }` statement. This could provide gas savings (on average) when cDai and/or the DSR is getting a lot of use. See this GitHub issue for more information.

## Low severity

None. 🙂

## Notes

### [N01] Negligible error in getCashPrior function

`CDaiDelegate` contract's `getCashPrior` function. That is, the return value of the `getCashPrior` function could be off by up to `chi / 1e27`. This error is negligible and should have no meaningful effect on the economics of Compound. However, developers of other projects interfacing with Compound contracts should be aware that if `chi &gt;= 1e27` then the `getCashPrior` function may not return the exact value they expect. For example, they should avoid using strict equality checks with the output of `getCashPrior` and the input parameter `amount` in `doTransferIn`.

If reducing this error is important, consider adding the balance of `vat.dai(address(this))` to the value `mul(pot.chi(), pie)` before dividing by `RAY` on line 119 of `CDaiDelegate`.

### [N02] Superfluous return

The `return` at the beginning of line 34 of `CDaiDelegate.sol` is not needed. This doesn't cause any problems but may confuse readers. To improve readability, consider removing `return` from the beginning of the line calling `_becomeImplementation(daiJoinAddress_, potAddress_);`.

### [N03] Superfluous functions in GemLike abstract contract

The `GemLike` abstract contract in `CDaiDelegate.sol` includes the two undefined functions `deposit` and `withdraw`. However, the only contract that is ever cast as `GemLike` is the `dai` contract, which does not implement a `deposit` or `withdraw` function.

If this is not intentional, consider removing the `deposit` and `withdraw` functions from the `GemLike` abstract contract to improve readability.

### [N04] Supply rate can exceed borrow rate

Unlike previous interest rate models, it is possible for the supply rate to be greater than the borrow rate with the new `DAIInterestRateModel` contract. This can happen when the reserves grow large enough.

borrow rate.

When this occurs, users are incentivized to arbitrage the rates — borrowing DAI from Compound and immediately re-supplying it back into Compound to earn a net-positive interest rate. This may be unintended behavior but is not a serious problem. As users arbitrage the rates, the utilization rate (and thus the borrow rate) increases until the supply rate is no longer greater than the borrow rate.

This seemingly "free money" would actually be coming from the DSR interest earned by the reserves. It would not come at a loss for any Compound users. We note this behavior only because it is a possibility that did not exist with previous interest rate models, and may be unintended. If this behavior is undesirable, then consider reducing the reserves whenever the supply rate approaches the borrow rate.

## [N05] Typos

- On line 144 of `CDaiDelegate.sol`, "th" should say "the".
- On line 44 of `DAIInterestRateModel.sol`, "amnount" should say "amount".
- On line 66 of `DAIInterestRateModel.sol`, "subraction" should be "subtraction".

## [N06] Misleading code comments

- On line 15 of the `DAIInterestRateModel` contract, the comment suggests that `gapPerBlock` is set to correspond to a rate of `0.05%` per block. However, the code on line 17 sets `gapPerBlock` to correspond to a rate of `0.05%` per *year*. Consider adjusting the comment to reflect the actual value of `gapPerBlock`.

- On line 81 of the `DAIInterestRateModel` contract, the comment states that the max borrow rate is determined and set by the code below. However, the code below does not set the maximum borrow rate, but rather the borrow rate when utilization rate equals `kink`. Consider changing the comment to reflect this.

## [N07] Inconsistent capitalization

### [N08] doTransferOut function may fail in an edge case

On line 163 of the `CDaiDelegate` contract's `doTransferOut` function a `1` is added when computing `pie`. This ensures that enough internal DAI will be taken from the `Pot` contract and given to the `CToken` contract that the call to `daiJoin.exit` on line 166 will succeed.

However, in the edge case where the `CToken` contract has no reserves and the last existing user attempts to withdraw their entire balance (e.g., via a call to `redeem`), the call to `doTransferOut` could unexpectedly revert at line 164 as the CToken's call to `pot.exit` may try to subtract a value greater than the `pot` contract's `pie[msg.sender]`.

This is not a serious problem, even for CToken users that are contracts (which may be hard coded to only be able to redeem their entire balances at once). Since anyone can add reserves, they can always be sure there is enough internal DAI associated with the `CDaiDelegate` contract for this call to complete.

Consider making sure there is always at least a small amount (dust) of internal DAI in the reserves.

## Conclusion

No critical or high severity issues were found. Recommendations were made to improve readability and handle cases where decisions by the Maker governance system may impact Compound users.

OpenZeppelin

## 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