Maple Finance
Findings & Analysis Report

2021-05-03

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization that consists of security researchers, auditors, developers, and individuals with domain expertise in the area of smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Maple Finance's smart contract system written in Solidity. The code contest took place between April 8, 2021 and April 21, 2021.

Wardens

11 Wardens contributed reports to the Maple Finance code contest:

- 0xRajeev
- 0xsomeone
- cmichel
- jmukesh
- a_delamo
- gpersoon
- janbro
- jvaqa
- pauliax
- s1mo
- shw

This contest was judged by Nick Johnson.

Final report assembled by sockdrawermoney.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity, 3 received a risk rating in the category of MEDIUM severity, and 12 received a risk rating in the category of LOW severity.

C4 analysis also identified an aggregate total of 33 non-critical recommendations.

The Maple Finance team responded to the issues identified as result of this code contest and provided information regarding any changes to the codebase with a pull request. Links to the aforementioned PRs are appended to the issue descriptions outlined within the corresponding details described in the Issues Found By Severity section of this document. A small set of vulnerabilities and submissions were disputed by the Maple Finance team. We have selected many of the comments.

Scope

The code under review consists of smart contracts written in the Solidity programming language which can be found linked from the C4 code contest repository.

The codebase audited was the `maple-core` repository, which includes the MPL token as a submodule (`maple-token`). All technical documentation for the protocol is located in the `maple-core` wiki.

The version of the code under review, including tests and tooling, is also available at the following URLs:

- maple-core
- maple-token

Severity Criteria

C4 assesses severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 3 primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings

No high risk findings.

Medium Risk Findings

[M-01] Loans of tokens with >18 decimals can result in incorrect collateral calculation

It is possible for a user to mislead a Pool Delegate to a seemingly innocuous loan by utilizing a token with more than 18 decimals as collateral and lucrative loan terms.

The final calculation within the `collateralRequiredForDrawdown` of `LoanLib` incorrectly assumes the collateral token of a loan to be less than `18` decimals, which can not be the case as there is no sanitization conducted on the creation of a `Loan` via the factory. This can cause an underflow to the power of `10` which will cause the division to yield `0` and thus cause the `Loan` to calculate `0` as collateral required for the loan. We advise the same paradigm as `_toWad` to be applied, which is secure.

lucas-manuel (Maple) acknowledged:

> We are aware that we cannot onboard liquidityAssets or collateralAssets with more that 18 decimals of precision, and will make that part of our onboarding criteria.

Nick Johnson (Judge):

> This is 100% a legitimate issue that could be exploited against the contract, and using social mitigations (making this part of the onboarding strategy) when there's a technical mitigation (`require()` ing that the token have <= 18 decimals, or using the recommended mitigation) is insufficient and could easily lead to an exploit due to human error.

> Based on the OWASP methodogology, I'm judging this as Likelihood=Low (because of the requirement to get it past human review) and Impact=High (because of the impact of the bug if it were exploited to create a 0-collateral loan and default on it), resulting in a Severity of Medium.

[M-02] Potential huge arbitrage opportunities / MPL price decrease

When the protocol suffers a default, the BPT stakers are the first line of defense and the protocol trades the BPT pool tokens for the single-sided liquidity asset of the Balancer LIQUIDITY <> MPT pool. (`PoolLib.handleDefault`)

Note that a pool token to single-asset trade is the same as burning the LP tokens to receive an equal amount of all underlying tokens, and then trading all other tokens received for the single asset.

It's the reverse of this:

> "Depositing a single asset A to a shared pool is equivalent to depositing all pool assets proportionally and then selling more of asset A to get back all the other tokens deposited." (Balancer)

This means on each default MPT tokens are sold for the liquidity asset. As the default is potentially a huge amount that happens at once, this creates a huge arbitrage opportunity.

As the default suffered can be a huge amount and the "repayment" happens at once, this creates a huge arbitrage opportunity: The MPT token price goes down. The borrow could also be incentivised to not repay the loan and take advantage of the arbitrage opportunity, either competing themselves on-chain or through shorts/bets on the MPT price.

Hard to completely mitigate. Pool delegates should be especially careful when giving out high-value loans and demand high collateral lockup.

lucas-manuel (Maple) confirmed:

> This is a valid concern, but not something that we are going to mitigate before launch. We are going to plan for PDs to atomically liquidate and burn.

Nick Johnson (Judge):

> I think this a valid finding; whether or not it's intended to be mitigated pre-launch, the Sponsor acknowledges it's a valid concern, and not something that's declared as part of the protocol's intrinsic assumptions. These sort of findings are exactly

> what audits are intended to uncover and bring to the attention of users as caveats when using the system. I concur with the Warden's assessment of Medium.

[M-03] Bypass or reduction on the lockup period of Pool FDTs.

In `Pool.sol`, the lockup restriction of withdrawal (`Pool.sol#396`) can be bypassed or reduced if new liquidity providers cooperate with existing ones.

1. A liquidity provider, Alice, deposits liquidity assets into the pool and minted some FDTs. She then waits for `lockupPeriod` days and calls `intendToWithdraw` to pass her withdrawal window. Now she is available to receive FDTs from others.
2. A new liquidity provider, Bob, deposits liquidity assets into the pool and minted some FDTs. Currently, he is not allowed to withdraw his funds by protocol design.
3. Bob and Alice agree to cooperate with each other to reduce Bob's waiting time for withdrawal. Bob transfers his FDT to Alice via the `_transfer` function.
4. Alice calls `intendToWithdraw` and waits for the `withdrawCooldown` period. Notice that Alice's `depositDate` is updated after the transfer; however, since it is calculated using a weighted timestamp, the increased amount of lockup time should be less than `lockupPeriod`. In situations where the deposit from Alice is much larger than that from Bob, Alice could only even need to wait for the `withdrawCooldown` period before she could withdraw any funds.
5. Alice then withdraws the amount of FDT that Bob transferred to her and transfers the funds (liquidity assets) to Bob. Bob successfully reduces (or bypasses) the lockup period of withdrawal.

Recommend forcing users to wait for the lockup period when transferring FDT to others or let the `depositDate` variable record the timestamp of the last operation instead of a weighted timestamp.

lucas-manuel (Maple) confirmed:

> Addressed in this PR

Low Risk Findings

[L-01] Cross-Chain Replay Attack

The `constructor` of the Maple token calculates the `chainid` it should assign during its execution and permanently stores it in an `immutable` variable. Should Ethereum fork in the feature, the `chainid` will change however the one used by the permits will not enabling a user to use any new permits on both chains thus breaking the token on the forked chain permanently. (Please consult EIP1344 for more details.)

The calculation of the `chainid` dynamically on each `permit` invocation. As a gas optimization, the deployment pre-calculated hash for the permits can be stored to an `immutable` variable and a validation can occur on the `permit` function that ensure the current `chainid` is equal to the one of the cached hash and if not, to re-calculate it on the spot.

lucas-manuel (Maple) acknowledged:

> Not going to implement, if Ethereum forks we will not used forked MPL

[L-02] Missing check for Pool state on several functions in Pool.sol

The Pool may be in three states: Initialized, Finalized and Deactivated as indicated by the enum State variable. While a couple of functions such as fundLoan() and deposit() check against a valid Pool state i.e. Finalized using `\_isValidState(State.Finalized)` , most other functions miss this check. This could cause unexpected protocol behavior if such functions are triggered in invalid Pool states (e.g. deactivated).

Examples of such functions missing this Pool state validity check are `triggerDefault()``, claim() , `withdraw()` and `withdrawFunds()` .

Recommend adding _isValidState(State.Finalized) check to all such functions specified above.

**lucas-manuel (Maple) acknowledged.

Nick Johnson (Judge):

> Submitter has not demonstrated how this can be exploited, but these seem like important checks to be omitting and may well result in invariants being violated. In the absence of a specific exploit vector, I'm awarding this as Low.

satyamakgec (Maple) commented:

> So we did an analysis again and here are some data points:
>
> - `triggerDefault()` -> It will only be called by the Pool Delegate and it internally checks whether the Pool (precicesly DebtLocker) has the 20 % of supplied LoanFDTs If yes then procced otherwise fail. So in the `Initialized` state it will not be callable as there is no fund provided to loan so not LoanFDT in the debtlocker, While if the loan get defaulted once then loan will automatically get closed so triggerDefault() will revert in that case.
> - `withdraw()` -> It has again implicit check of the pool state as there is no way to get the poolFDT if the Pool is in the `Initialized` state while if Pool is in the Finalized state then again there is no harm as user can withdraw its funds if they have otherwise it will not be possible.
> - `withdrawFunds()` -> This is simillar as above if you have entitled amount then only it will works otherwise it will be a just a gas waste of the msg.sender.
> - `claim()` -> It is a permissioned function (only be callable by Pool delegate) even if it get called during `Initialized` or after `Finalized` state then nothing happen only be the waste of gas which I think it is okay in this case as PD is very well aware about the process.
>
> We could add the statechecks eventually but it doesn't give us any extra benefit although it does increase the size of the PoolFactory contract bytecode that we don't want as we are already on the verge of `24 KB`

[L-03] Mirrored admin variables in global context, Pool, PoolFactory, Loan and LoanFactory may make it confusing for deployment and maintenance

The access control model for the different contracts and how they interact is confusing and may cause issues during deployment and maintenance. Multiple

contracts have the notion of admin(s), all of which use setAdmin function to update admin status. This mirroring and reuse of the admin variable is susceptible to accidents.

Recommend renaming the different admin variables e.g. adminGlobal, adminPool, adminLoan. Document the access control roles, hierarchy and interactions explicitly.

lucas-manuel (Maple) confirmed.

Nick Johnson (Judge):

> Unresolved, this could lead to issues (eg, forgetting to migrate admin on one contract, or accidentally granting admin permission on the wrong contract), so I'm rating this Low.

[L-04] Full payment does not consider late fees of the payment

Since the calculation of `makeFullPayment` (`Loan.sol#249`) does not consider whether the payment is late or not, the borrower can avoid paying late fees by only calling `makeFullPayment` instead of `makePayment` (`Loan.sol#238`). The borrower has no incentive to repay the loan in time and could

The full payment is calculated by `PremiumCalc`, which ignores whether the payment is late or not. A configured premium fee calculates the interest; however, it is a fixed value through time. The interest that a borrower should pay for borrowing the loan for any amount of time (e.g., a month or a year) is the same.

Recommend calculating late fees in `PremiumCalc` as in `RepaymentCalc` to let the borrower pay late fees based on the `apr` of loan.

Nick Johnson (Judge):

> Another way to look at this is that the borrower gets `gracePeriod` extra days of borrowing for free - just by deferring their final payment. Agree with Medium.

lucas-manuel (Maple) commented:

> If they deferred their final payment and did makeFullPayment instead of makePayment they would pay premiumFee on their principal, which is set to be a larger amount than a given payment plus late fee, so they would be losing money in this case.

Nick Johnson (Judge):

> Based on my understanding of the code:

> Relevant configurable parameters are the payment interval (`payment_interval`), grace period before foreclosure (`grace_period`), interest payment size (`interest_payment`), late payment fee as a percentage of interest payment size (`late_fee`), and premium fee (`premium_fee`).

> If `payment_interval * 2 < grace_period` , it's possible to be late multiple payments - in which case you pay multiple late fees.

> Without charging late payment fees on a full repayment, there are two scenarios in which the borrower can end up better off:

- If `(payment_interval / grace_period) * late_fee > premium_fee` , once the borrower is late some number of `payment_intervals` , they pay less by doing a full repayment with the premium fee than by paying off the normal way.
- If `(payment_interval / grace_period) * interest_payment > premium_fee` , they can treat the grace period as an extra loan period, and pay no more than they would have in interest (possibly less, depending on the parameters).

> Since both of these are only possible with certain parameter values, I'm downgrading this to Low. This could be remedied by either putting range checks for these parameter values in loan initialisation, or by calculating 'missed interest' and late fees in `makeFullPayment` and taking the minimum of that and the premium fee.

lucas-manuel (Maple) commented:

> We're going to leave as is and just ensure that Pool Delegates are educated around Loan terms and what they entail before funding them.

[L-05] Chainlink Price data could be stale

There is no check if the return value indicates stale data. This could lead to stale prices according to the Chainlink documentation:

- "if answeredInRound < roundId could indicate stale data."
- "A timestamp with zero value means the round is not complete and should not be used."

The price oracle might return unreliable price data which can lead to a variety of different issues in the protocol, for example, for liquidating more staker & lender tokens than required at fair market price.

Add missing checks for stale data. See example here.

lucas-manuel (Maple) confirmed:

> We will add this check, but disagree that this is a high severity bug. Especially since we will be using BTC and ETH oracles to start, it is very rare that there will be stale data.

Nick Johnson (Judge):

> Since the contract only asks for latest data, incomplete rounds should be impossible, so we can discount them. Stale data is possible; I would rate this as Likelihood=LOW (it'll be difficult to make ChainLink oracles go stale) and Impact=Medium (this could only be used to create arb opportunities on loan collateral or liquidations, which will be limited to the price change during the stale period), resulting in a Severity=Low.

[L-06] Chainlink Price oracle always assumes 8 decimals

The response from the price oracle always assumes 8 decimals (see `PoolLib.convertFromUsd`) but it's never checked if the oracle response has 8 decimals using ChainLink's `.decimals()` function. At some point, the governor

might set up a USD price feed oracle that contains more than 8 decimals leading to inflated prices everywhere.

Recommend checking `_aggregator.decimals() == 8` in `ChainlinkOracle` constructor and `changeAggregator`.

lucas-manuel (Maple) confirmed:

> We were going to do this manually, we were aware of this issue, but it is a good idea to just add a check. Disagree with severity.

Nick Johnson (Judge):

> Impact would be High if this happened, but the Likelihood is very low. Agree with Severity=Low.

[L-07] Missing check on `setManualPrice(int256 _price)`

The `ChainlinkOracle.setManualPrice` function specifies that it can only be called "if manualOverride == true".

This is not the case.

Assume an oracle failure happened, and the oracle needs to be manually set to prevent losses. The `setManualPrice` function succeeds and the owner might think that the oracle price is overwritten as the function would fail when `manualOverride` is not `true` according to specification. The protocol would still use the broken chainlink price feed and suffer losses.

Add the missing `require(manualOverride == true, "manual override not set")` check.

lucas-manuel (Maple) acknowledged:

> Not really a bug, but we will address this.

Nick Johnson (Judge):

> I would add, though, that the current configuration is unsafe. If `setManualPrice` is changed to require that `manualOverride` is true, there will be an interval between calling `setManualOverride` and `setManualPrice` during which an old price is used. Instead, a single function that enables the manual override and sets the price should probably be used.

Nick Johnson (Judge):

> On further thought, upgrading to Low based on Warden's reasoning - the success of the call to `setManualPrice` may lead the submitter to believe that the issue is resolved, when it only sets a value that is not referenced.

[L-08] Missing non-zero check

The `MapleGlobals.setPriceOracle` should check that the oracle address is not zero.

A wrong call to this function might set the oracle address to the zero address and break core oracle functionality.

Add a `require(oracle != 0)` statement.

lucas-manuel (Maple) disputed:

> We actually are not going to address this, we do not think this is a bug. The governor will manually verify non-zero addresses.

Nick Johnson (Judge):

> I think this warrants Likelihood=Low,Impact=Medium => Severity=Low. Unlike deployment misconfigurations, this mistake, while unlikely, would impact the running system, and it's easily defended against.

[L-09] MPL reward claims of balancer pools can be exploited

> When MPL tokens are added as liquidity to the Balancer pool, the Balancer pool is the owner of those tokens, which are accruing USDC interest.

Anyone can send the USDC interest to the balancer pool by calling `withdrawFundsOnBehalf(balancerPool)`.

An attacker can abuse this to capture part of this interest by doing the following steps in a single transaction:

1. Deposit MPT/USDC to the balancer pool (the initial liquidity can also be acquired by a flash loan)
2. Send USDC to the pool by calling `withdrawFundsOnBehalf(bPool)`, call `gulp`.
3. Withdraw liquidity again, the attacker will receive their initial deposit + a share of the USDC interest proportional to their LP tokens.

USDC interest that was supposed to go to MPT balancer pool stakers is stolen by attackers. Funds might be locked forever.

This is hard to prevent completely because you're sending free money to the pool. One way to reduce the risk is to only allow claiming interest by the governor / trusted parties. This would disallow attacker to perform this in a risk-free way in a single transaction, but the same attack would still be possible for miners.

Consider alternative ways of distributing the interest of balancer pools like transferring it to all MPT holders instead of liquidity providers, because:

1. it's currently not fair anyway because only the LPs that are in the pool at the time `withdrawFundsOnBehalf` was called will benefit, regardless of how long they have been providing this liquidity.
2. External arbitrageurs are the ones that benefit from this short-term pool price imbalance the most and a good chunk of the USDC interest will go to them

lucas-manuel (Maple) disputed:

> Not a bug, `distributeToHolders` and `withdrawFundsOnBehalfOf` will always be called atomically by the governor.

Nick Johnson (Judge):

> Without code to ensure that `withdrawFundsOnBehalfOf is called immediately after` distributeToHolders, this bug can still occur. I'm considering this Likelihood=Low,Impact=Medium => Severity=Low.

[L-10] MPL USDC distributions can be withdrawn by anyone

Anyone can withdraw USDC interest of another address by calling `withdrawFundsOnBehalf(addr)`.

Imagine a smart contract that has a specific function for withdrawing the USDC contract to their contract:

```
function withdrawAndTransfer() {
  mpt.withdraw();
  usdc.transfer(usdc.balanceOf(address(this), owner);
}
```

If the contract has no `skim` function to transfer out the assets, they can get stuck forever.

USDC interest can get stuck forever.

Disallow withdrawals on behalf of other users.

lucas-manuel (Maple) disputed:

> Not a bug, this was intended. Projects that integrate with MPL will have to take this functionality into account.

Nick Johnson (Judge):

> Many systems integrate support for generic ERC-20 token contracts without being able to handle per-token special cases. For example, I believe this issue would affect any Uniswap/Balancer/etc liquidity pool between MPL and any non-USDC token. Rating this as Low, per submitter, though I believe an argument could be made for making this higher severity, since it limits the ability to use MPL tokens in generic systems without losing out on dividends.

[L-11] LoanLib.unwind uses globals.fundingPeriod()

Every loan has its own fundingPeriod which is set once in the constructor: fundingPeriod = globals.fundingPeriod(); fundingPeriod in globals can change. It does not effect already deployed Loans. However, in Loan contract function unwind() calls LoanLib.unwind which checks against globals.fundingPeriod(): IGlobals globals = _globals(superFactory); // Only callable if time has passed drawdown grace period, set in MapleGlobals require(block.timestamp > createdAt.add(globals.fundingPeriod()), "Loan:FUNDING*PERIOD*NOT_FINISHED"); at this time, globals.fundingPeriod() could be different than this specific Loan's fundingPeriod.

Recommend checking expiration against local fundingPeriod.

lucas-manuel (Maple) confirmed

[L-12] Uniswap DOS

Borrowers can launch a front running/sandwich attack on triggerDefault() which manipulates the price on Uniswap outside the maxSwapSlippage range causing the function to revert and the collateral to stay in the collateralLocker There is no way to transfer the collateral out of the collateralLocker after a loan default without going through a Uniswap trade, so a borrower can lock funds indefinitely for a fraction of the locked collateral (cost of Uniswap fees) and potentially hold their collateral hostage.

Recommend using more than one source of liquidity for liquidations.

lucas-manuel (Maple) acknowledged:

> We will be upgrading liquidations post-launch.

Non-Critical Findings

- [N-01] Functions calculating the value of `BPT` is vulnerable to flash-loan attacks.
- [N-02] MapleTreasury does not emit an event when MapleGlobals address is updated

- [N-30] Function triggerDefault should call _emitBalanceUpdateEventForCollateralLocker
- [N-31] getRewardForDuration will start returning misleading results if rewardsDuration is updated
- [N-32] Oracle not checked if set for an asset
- [N-33] Default slippage value too high

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code, but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top