Maple Finance contest Findings & Analysis Report

2022-01-05

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in 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 smart contract system written in Solidity. The code contest took place between December 2—December 8 2021.

Wardens

13 Wardens contributed reports to the Maple Finance contest:

1. WatchPug (jtp and ming)
2. cmichel
3. gzeon
4. robee
5. hyh
6. jayjonah8
7. yeOlde
8. MetaOxNull
9. GiveMeTestEther
10. wuwe1
11. saian
12. defsec

This contest was judged by pauliax.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities and 25 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity, 2 received a risk rating in the category of MEDIUM severity, and 5 received a risk rating in the category of LOW severity.

C4 analysis also identified 7 non-critical recommendations and 10 gas optimizations.

Scope

The code under review is linked in the C4 Maple Finance contest repository README.

Severity Criteria

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

Vulnerabilities are divided into three 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 (1)

[H-01] `makePayment()` Lack of access control allows malicious `lender` to retrieve a large portion of the funds earlier, making the borrower suffer fund loss

Submitted by WatchPug

`MapleLoan.sol` L86-L93

```
    function makePayment(uint256 amount_) external override returns
        // The amount specified is an optional amount to be transfe
        require(amount_ == uint256(0) || ERC20Helper.transferFrom(_f

        ( principal_, interest_ ) = _makePayment();

        emit PaymentMade(principal_, interest_);
    }
```

The current implementation allows anyone to call `makePayment()` and repay the loan with `_drawableFunds`.

This makes it possible for a malicious `lender` to call `makePayment()` multiple times right after `fundLoan()` and retrieve most of the funds back immediately, while then `borrower` must continue to make payments or lose the `collateral`.

PoC

Given:

- `_collateralRequired` = 1 BTC
- `_principalRequested` = 12,000 USDC
- `_paymentInterval` = 30 day
- `_paymentsRemaining` = 12
- `_gracePeriod` = 1 day

- `interestRate_` = 2e17
- The borrower calls `postCollateral()` and added `1 BTC` as `_collateralAsset`;
- The lender calls `fundLoan()` and added `12,000 USDC` as `_fundsAsset`;
- The lender calls `makePayment()` 11 times, then:
- `_drawableFunds` = 96
- `_claimableFunds` = 11903
- `_principal` = 1553
- The lender calls `_claimFunds()` get 11,903 USDC of `_fundsAsset` back;

Now, for the borrower `1,579 USDC` is due, but only `96 USDC` can be used. The borrower is now forced to pay the interests for the funds that never be used or lose the collateral.

Recommendation

Change to:

```
function makePayment(uint256 amount_) external override returns
    // The amount specified is an optional amount to be transfer
    require(amount_ == uint256(0) || ERC20Helper.transferFrom(_f

    require(msg.sender == _borrower, "ML:DF:NOT_BORROWER");

    ( principal_, interest_ ) = _makePayment();

    emit PaymentMade(principal_, interest_);
}
```

deluca-mike (Maple) confirmed:

> Good catch. However, we do want accounts other than the borrower to make payments (this is actually functionality that our current borrowers use), so instead of this fix, we may enforce that a payment can only be made some window of time before it is due, to prevent anyone from prematurely paying it down.

pauliax (judge) commented:

> Great find, with a POC, deserves a severity of high as it may incur in funds lost for the borrower. The sponsor has acknowledged and mitigated the issue.

Medium Risk Findings (2)

[M-01] Anyone can call `closeLoan()` to close the loan

Submitted by WatchPug

`MapleLoan.sol` **L56-L63**

```
    function closeLoan(uint256 amount_) external override returns (u
        // The amount specified is an optional amount to be transfer
        require(amount_ == uint256(0) || ERC20Helper.transferFrom(_f

        ( principal_, interest_ ) = _closeLoan();

        emit LoanClosed(principal_, interest_);
    }
```

Based on the context, we believe that the `closeLoan()` should only be called by the `borrower`. However, the current implementation allows anyone to call `closeLoan()` anytime after `fundLoan()`.

If there is no `earlyFee`, this enables a griefing attack, causing the `borrower` and `lender` to abandon this contract and redo everything which costs more gas.

If a platform fee exits, the lender will also suffer fund loss from the platform fee charged in `fundLoan()`.

Recommendation

Change to:

```
    function closeLoan(uint256 amount_) external override returns (u
        // The amount specified is an optional amount to be transfer
        require(amount_ == uint256(0) || ERC20Helper.transferFrom(_f

        require(msg.sender == _borrower, "ML:DF:NOT_BORROWER");

        ( principal_, interest_ ) = _closeLoan();

        emit LoanClosed(principal_, interest_);
    }
```

deluca-mike (Maple) confirmed:

pauliax (judge) commented:

> Great find, missing authorization.

[M-02] Unsafe implementation of `fundLoan()` allows attacker to steal collateral from an unfunded loan

Submitted by WatchPug

`MapleLoanInternals.sol` **L257-L273**

```
    uint256 treasuryFee = (fundsLent_ * ILenderLike(lender_).treasur

    // Transfer delegate fee, if any, to the pool delegate, and decr
    uint256 delegateFee = (fundsLent_ * ILenderLike(lender_).investo

    // Drawable funds is the amount funded, minus any fees.
    _drawableFunds = fundsLent_ - treasuryFee - delegateFee;

    require(
        treasuryFee == uint256(0) || ERC20Helper.transfer(_fundsAsse
        "MLI:FL:T_TRANSFER_FAILED"
    );

    require(
        delegateFee == uint256(0) || ERC20Helper.transfer(_fundsAsse
        "MLI:FL:PD_TRANSFER_FAILED"
        );
```

In the current implementation, `mapleTreasury` , `poolDelegate` and `treasuryFee` are taken from user input `lender_` , which can be faked by setting up a contract with `ILenderLike` interfaces.

This allows the attacker to set very high fees, making `_drawableFunds` near 0.

Since `mapleTreasury` and `poolDelegate` are also read from `lender_` , `treasuryFee` and `investorFee` can be retrieved back to the attacker.

As a result, the borrower won't get any `_drawableFunds` while also being unable to remove collateral.

PoC

Given:

- `_collateralRequired` = 10 BTC
- `_principalRequested` = 1,000,000 USDC
- `_paymentInterval` = 1 day
- `_paymentsRemaining` = 10
- `_gracePeriod` = 1 day
- Alice (borrower) calls `postCollateral()` and added `10 BTC` as `_collateralAsset` ;
- The attacker calls `fundLoan()` by taking `1,000,000 USDC` of flashloan and using a fake `lender` contract;
- Alice calls `drawdownFunds()` with any amount > 0 will fail;
- Alice calls `removeCollateral()` with any amount > 0 will get "MLI:DF:INSUFFICIENT_COLLATERAL" error;
- Unless Alice make payment (which is meaningless), after 2 day, the attacker can call `repossess()` and get `10 BTC` .

Recommendation

Consider reading `treasuryFee` , `investorFee` , `mapleTreasury` , `poolDelegate` from an authoritative source instead.

lucas-manuel (Maple) confirmed and disagreed with severity:

> We would consider this medium risk, since a borrower would never post collateral before a loan is funded. We can enforce this on the smart contracts level though by adding a require to the `postCollateral` function to assert that the principal amount is greater than zero.

pauliax (judge) commented:

> Great find. As per the sponsor's recommendation, this scenario is not very likely, so I am marking this issue as of medium severity.

Low Risk Findings (5)

- [L-01] Must approve 0 first *Submitted by robee*
- [L-02] Same implementation can be registerd for several versions *Submitted by cmichel*
- [L-03] Insufficient input validation *Submitted by WatchPug, also found by hyh and jayjonah8*
- [L-04] Fund stuck in `Liquidator` if `stopLiquidation` is called *Submitted by gzeon*
- [L-05] Functionality of liquidation strategies can be broken *Submitted by cmichel*

Non-Critical Findings (7)

- [N-01] Open TODOs *Submitted by robee, also found by Meta0xNull*
- [N-02] Typos *Submitted by wuwe1*
- [N-03] Floating pragma *Submitted by saian, also found by WatchPug*
- [N-04] Unchecked return value for `ERC20.approve` call *Submitted by WatchPug, also found by defsec and robee*
- [N-05] IsContract Function Usage *Submitted by defsec*
- [N-06] Consider adding storage gaps to proxied contracts *Submitted by WatchPug*

- [N-07] Function poolDelegate does not have a named return (DebtLocker.sol) *Submitted by yeOlde*

Gas Optimizations (10)

- [G-01] ”> 0” is less efficient than “!= 0” for unsigned integers *Submitted by yeOlde, also found by gzeon*
- [G-02] Short the following require messages *Submitted by robee, also found by Meta0xNull and WatchPug*
- [G-03] Adding unchecked directive can save gas *Submitted by WatchPug, also found by gzeon*
- [G-04] Storage double reading. Could save SLOAD *Submitted by robee, also found by GiveMeTestEther, hyh, and WatchPug*
- [G-05] State variables that could be set immutable *Submitted by robee, also found by WatchPug*
- [G-06] Gas Optimization: Use constant instead of block.timestamp *Submitted by gzeon*
- [G-07] Cache external call result in the stack can save gas *Submitted by WatchPug*
- [G-08] `Liquidator.sol#_locked` Switching between 1, 2 instead of true, false is more gas efficient *Submitted by WatchPug*
- [G-09] Avoid unnecessary arithmetic operations can save gas *Submitted by WatchPug*
- [G-10] Reuse arithmetic results can save gas *Submitted by WatchPug*

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.

Top