88mph
Findings & Analysis Report

2021-06-14

Table of contents

Overview

About C4

Code 432n4 (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 88mph's smart contract system written in Solidity. The code contest took place between May 13 and May 19, 2021.

Wardens

5 Wardens contributed reports to the 88mph code contest:

- cmichel
- shw
- gpersoon
- Thunder
- a_delamo

This contest was judged by ghoul.sol.

Final report assembled by ninek.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 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.

Scope

The code under review can be found within the C4 code contest repository and comprises 39 smart contracts written in the Solidity programming language.

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

There were no high risk findings identified in this contest.

Medium Risk Findings

[M-01] Incompatability with deflationary / fee-on-transfer tokens

The `DInterest.deposit` function takes a `depositAmount` parameter but this parameter is not the actual transferred amount for fee-on-transfer / deflationary (or other rebasing) tokens.

The actual deposited amount might be lower than the specified `depositAmount` of the function parameter.

This would lead to wrong interest rate calculations on the principal.

Recommend transferring the tokens first and comparing pre-/after token balances to compute the actual deposited amount.

ZeframLou (88mph) acknowledged:

> While this is true, we have no plans to support fee-on-transfer or rebasing tokens.

[M-02] Unchecking the ownership of `mph` in function `distributeFundingRewards` could cause several critical functions to revert

In contract `MPHMinter`, the function `distributeFundingRewards` does not check whether the contract itself is the owner of `mph`. If the contract is not the owner of `mph`, `mph.ownerMint` could revert, causing functions such as `withdraw`, `rolloverDeposit`, `payInterestToFunders` in the contract `DInterest` to revert as well.

Recommend adding a `mph.owner() != address(this)` check as in the other functions (e.g., `mintVested`).

ZeframLou (88mph) confirmed:

> Fixed in this commit.

Low Risk Findings

[L-01] Use openzeppelin ECDA for erecover

In `Sponsorable.sol` is using erecover directly to verify the signature. Since it is a critical piece of the protocol, it is recommended to use the ECDSA from openzeppelin as it does more validations when verifying the signature.

ZeframLou (88mph) confirmed:

> Fixed in this commit.

[L-02] Anyone can withdraw vested amount on behalf of someone

The `Vesting.withdrawVested` function allows withdrawing the tokens of other users.

While the tokens are sent to the correct address, this can lead to issues with smart contracts that might rely on claiming the tokens themselves.

As one example, suppose the `_to` address corresponds to a smart contract that has a function of the following form:

```
    function withdrawAndDoSomething() {
        contract.withdrawVested(address(this), amount);
        token.transfer(externalWallet, amount)
    }
```

If the contract has no other functions to transfer out funds, they may be locked forever in this contract.

Recommend not allowing users to withdraw on behalf of other users.

ZeframLou (88mph) acknowledged:

> This is true, but `Vesting.sol` is only kept for legacy support, `Vesting02.sol` will be the main vesting contract, so we're fine with this.

[L-03] Extra precautions in updateAndQuery

The function `updateAndQuery` of EMAOracle.sol subtracts the `incomeIndex` with the previous `incomeIndex`. These `incomeIndex` values are retrieved via the moneyMarket contract from an external contract.

If, by accident, the previous `incomeIndex` is larger than the current `incomeIndex`, then the subtraction would be negative and the code halts (reverts) without an error message.

Also, the `updateAndQuery` function would not be able to execute (until the current `incomeIndex` is larger than the previous `incomeIndex`).

This situation could occur when an error occurs in one of the current or future money markets.

Recommend giving an error message when the previous `incomeIndex` is larger than the current `incomeIndex` and/or create a way to recover from this erroneous situation.

ZeframLou (88mph) confirmed:

> Fixed in this commit.

[L-04] Add extra error message in_depositRecordData

In the function `_depositRecordData` of DInterest.sol, `interestAmount` is lowered with `feeAmount`.

If, by accident, `feeAmount` happens to be larger than `interestAmount`, an error occurs and the execution stops, without an error message.

This might make troubleshooting this situation more complicated.

Suggest adding something like:

```
    require(interestAmount >= feeAmount,"DInterest: fee too large");
```

ZeframLou (88mph) acknowledged

[L-05] function payInterestToFunders does not have a re-entrancy modifier

Function `payInterestToFunders` does not have a re-entrancy modifier. I expect to see this modifier because similar functions (including sponsored version) have it.

Add ‘nonReentrant’ to function `payInterestToFunders`.

ZeframLou (88mph) confirmed:

> Fixed in this commit.

Non-Critical Findings

[N-01] zero amount of token value can be entered for creating vest object in vesting.sol

The impact will be on the end-user; if they entered the zero amount by mistake, they would end paying a gas fee for nothing.

In the `vest` function, there is checking of `vestPeriodInSeconds` but no checking of amount, due to which amount can be set to 0.

Recommend adding a condition to check for function parameter.

ZeframLou (88mph) acknowledged:

> We're ok with this.

ghoul.sol commented:

> Passing 0 doesn't break anything. This is a non-critical issue.

[N-02] lack of zero address validation in constructor

Due to the lack of zero address validation, there is a chance of losing funds.

Dumper.sol

OneSplitDumper.sol

Recommend adding zero address check.

ZeframLou (88mph) acknowledged:

> We're fine with this.

[N-03] Multiple definitions of PRECISION

There are multiple definitions of PRECISION.

This risk is that is someone (a new developer?) would change the value of PRECISION on one location and might forget to change it in one of the other places. Also, 2 of them are defined as public while the rest is internal.

Recommend defining PRECISION once and import this in all the other contracts.

ZeframLou (88mph) acknowledged

[N-04] contract AaveMarket function setRewards has a misleading revert message

AaveMarket function `setRewards` has a misleading revert message:

```
    require(newValue.isContract(), "HarvestMarket: not contract");
```

Recommend naming it 'AaveMarket', not 'HarvestMarket'.

ZeframLou (88mph) confirmed:

> Fixed in this commit.

[N-05] Mismatch between the comment and the actual code

Here the comment says that it should transfer from msg.sender, but it actually transfers from the sender which is not always the msg.sender (e.g., sponsored txs):

```
    // Transfer `fundAmount` stablecoins from msg.sender
    stablecoin.safeTransferFrom(sender, address(this), fundAmount);
```

Update the comment to match the code.

ZeframLou (88mph) confirmed:

> Fixed in this commit.

Gas Optimizations

[G-01] Gas optimizations by using external over public

Some functions could use external instead of public in order to save gas.

If we run the following methods on Remix, we can see the difference:

```
    //  transaction cost  21448 gas
    //  execution cost    176 gas
    function tt() external returns(uint256) {
        return 0;
    }

    //  transaction cost  21558 gas
    //  execution cost    286 gas
    function tt_public() public returns(uint256) {
        return 0;
    }
```

[ZeframLou (88mph) disputed]():

> Many of the functions listed are also called within the contract, so changing their visibility to `public` will break things.

[ghoul.sol commented]():

> Even though out of the whole list, only a few functions are good candidates to be changed, it's technically a valid suggestion.

[ZeframLou (88mph) confirmed]():

> Addressed in [this commit]().

[G-02] Gas optimizations - storage over memory

Some functions are using memory to read state variables when using storage is more gas efficient.

[Solidity doc reference]()

[ZeframLou (88mph) disputed]():

> The memory location of those variables is intentionally set to be `memory` to only load the data from storage once to save gas. If changed to `storage`, there will be

> multiple SLOAD's for accessing the same variable, which is expensive.

ghoul.sol commented:

> In case of the `withdrawableAmountOfDeposit` function, the warden is right. `_getDeposit` makes 7 `SLOAD` calls in this case because it's pulling the whole object from storage. Using storage would make only 5 `SLOAD` calls.

> In the case of `_getVestWithdrawableAmount`, either way uses 6 `SLOAD` calls.

> Warden is correct in at least one case, so the reports stand.

ZeframLou (88mph) confirmed:

> Addressed in this commit.

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