Tempus Finance contest Findings & Analysis Report

2021-11-08

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 Tempus Finance contest smart contract system written in Solidity. The code contest took place between October 14—October 20 2021.

Wardens

14 Wardens contributed reports to the Tempus Finance audit contest:

1. gpersoon
2. cmichel
3. pmerkleplant
4. pants
5. hyh
6. chenyu
7. WatchPug
8. TomFrench
9. loop
10. defsec
11. Koustre
12. 0xMesaj
13. pauliax
14. yeOlde

This contest was judged by 0xean.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities and 37 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 12 non-critical recommendations and 16 gas optimizations.

Scope

The code under review can be found within the C4 Tempus Finance contest repository, and is composed of 40 smart contracts written in the Solidity programming language, and includes 3,903 lines of Solidity code.

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] Steal tokens from TempusController

Submitted by gpersoon.

Impact

The function `\_depositAndProvideLiquidity` can be used go retrieve arbitrary ERC20 tokens from the TempusController.sol contract.

As the test contract of TempusController.sol https://goerli.etherscan.io/address/0xd4330638b87f97ec1605d7ec7d67ea1de5dd7aaa shows, it has indeed ERC20 tokens.

The problem is due to the fact that you supply an arbitrary tempusAMM to depositAndProvideLiquidity and thus to `\_depositAndProvideLiquidity`. tempusAMM could be a fake contract that supplies values that are completely fake.

At the end of the function `\_depositAndProvideLiquidity`, ERC20 tokens are send to the user. If you can manipulate the variables ammTokens, mintedShares and sharesUsed you can send back any tokens held in the contract "ammTokens[0].safeTransfer(msg.sender, mintedShares - sharesUsed[0]);"

The Proof of Concept shows an approach to do this.

Proof of Concept

- https://github.com/code-423n4/2021-10-tempus/blob/63f7639aad08f2bba717830ed81e0649f7fc23ee/contracts/TempusController.sol#L73-L79
- https://github.com/code-423n4/2021-10-tempus/blob/63f7639aad08f2bba717830ed81e0649f7fc23ee/contracts/TempusController.sol#L304-L335
- Create a fake Vault contract (fakeVault) with the following functions: `fakeVault.getPoolTokens(poolId)` —-> returns {TokenToSteal1,TokenToSteal2},{fakeBalance1,fakeBalance2},0

`fakeVault.JoinPoolRequest()` —> do nothing `fakeVault.joinPool()` —> do nothing

- Create a fake Pool contract (fakePool) with the following functions: `fakePool.yieldBearingToken()` —> returns fakeYieldBearingToken `fakePool.deposit()` —> returns fakeMintedShares,…
- Create a fake ammTokens contract with the following functions: `tempusAMM.getVault()` —> returns fakeVault `tempusAMM.getPoolId()` —> returns 0 `tempusAMM.tempusPool()` —> returns fakePool
- call depositAndProvideLiquidity(fakeTempusAMM,1,false) // false -> yieldBearingToken _getAMMDetailsAndEnsureInitialized returns fakeVault,0,{token1,token2},{balance1,balance2} _deposit(fakePool,1,false) calls _depositYieldBearing which calls `fakePool.deposit()` and returns fakeMintedShares _provideLiquidity(…) calculates a vale of ammLiquidityProvisionAmounts _provideLiquidity(…) skips the safeTransferFrom because sender == address(this)) the calls to fakeVault.JoinPoolRequest() and fakeVault.joinPool() can be faked. _provideLiquidity(…) returns the value ammLiquidityProvisionAmounts

Now fakeMintedShares - ammLiquidityProvisionAmounts number of TokenToSteal1 and TokenToSteal2 are transferred to msg.sender

As you can both manipulate TokenToSteal1 and fakeMintedShares, you can transfer any token to msg.sender

Recommended Mitigation Steps

Create a whitelist for tempusAMMs

mijovic (Tempus) confirmed:

> This is a good point. However, these tokens that are locked in `TempusController` are coming from dust that was left when the user is doing early redemption. As this needs to be done with equal shares, we have a threshold parameter that is used as the maximum leftover behind redemption (usually there is a need to do a swap before redemption to make this work). So, this is going to be pennies always.

> I would not consider this as high risk, and we are not planning to fix this as steps to make this hack are too complicated to steal pennies… Also, the gas cost of doing it costs by far more than the funds that someone can steal.

mijovic (Tempus) commented:

> We changed point of view here a little bit. Will add registry of TempusAMMs and TempusPools that can be used with controller, just to prevent possible attacks with fake amms and pools.

mijovic (Tempus) patched:

> Added whitelist registry for both `TempusAMM` and `TempusPool` in this PR https://github.com/tempus-finance/tempus-protocol/pull/365 However, as amount of tokens that TempusController holds is so small (I would say this is of severity 2)

0xean (judge) commented:

> The C4 docs don't speculate on the amount of assets stolen in the TLDR of risk assessment.

```
3 — High: Assets can be stolen/lost/compromised directly (or inc
```

> Given the fact that some amount of assets could be stolen, i believe this is the correct severity for the issue.

Medium Risk Findings (2)

[M-01] `exitTempusAMM` can be made to fail

Submitted by cmichel.

There's a griefing attack where an attacker can make any user transaction for `TempusController.exitTempusAMM` fail. In `_exitTempusAMM`, the user exits their LP position and claims back yield and principal shares. The LP amounts to redeem are determined by the function parameter `lpTokensAmount`. A final `assert(tempusAMM.balanceOf(address(this)) == 0)` statement checks that the LP token amount of the contract is zero after the exit. This is only true if no other LP shares were already in the contract.

However, an attacker can frontrun this call and send the smallest unit of LP shares to the contract which then makes the original deposit-and-fix transaction fail.

Impact

All `exitTempusAMM` calls can be made to fail and this function becomes unusable.

Recommended Mitigation Steps

Remove the `assert` check.

mijovic (Tempus) confirmed:

> Great finding. This can block people exiting AMM via `TempusController`.

mijovic (Tempus) patched:

> Fixed in https://github.com/tempus-finance/tempus-protocol/pull/369

[M-02] `depositAndFix` can be made to fail

Submitted by cmichel.

There's a griefing attack where an attacker can make any user transaction for `TempusController.depositAndFix` fail. In `_depositAndFix`, `swapAmount` many yield shares are swapped to principal where `swapAmount` is derived from the function arguments. A final `assert(yieldShares.balanceOf(address(this)) == 0)` statement checks that the yield shares of the contract are zero after the swap. This is only true if no other yield shares were already in the contract.

However, an attacker can frontrun this call and send the smallest unit of yield shares to the contract which then makes the original deposit-and-fix transaction fail.

Impact

All `depositAndFix` calls can be made to fail and this function becomes unusable.

Recommended Mitigation Steps

Remove the `assert` check.

mijovic confirmed:

> Good catch. This can block users from doing this action via controller

mijovic patched:

> Fixed in https://github.com/tempus-finance/tempus-protocol/pull/370

Low Risk Findings (5)

- [L-01] Param `initInterestRate` in `TempusPool::constructor` should not be 0 *Submitted by pmerkleplant.*
- [L-02] Open TODOs *Submitted by pants.*
- [L-03] No `swap` slippage checks *Submitted by cmichel, also found by hyh.*
- [L-04] PermanentlyOwnable does not prevent transferring ownership to a dead address. *Submitted by chenyu.*
- [L-05] Scaling factors for token 0/1 might swap in TempusAMM constructor. *Submitted by chenyu.*

Non-Critical Findings (12)

- [N-01] Typos *Submitted by WatchPug.*
- [N-02] `transferFees` may not be the contract itself *Submitted by cmichel.*
- [N-03] `internal` functions can be `private` *Submitted by pants.*
- [N-04] `getAMMOrderedAmounts` and `_exitTempusAmmAndRedeem` functions use explicit token comparison for ordering instead of relying on Balancer's PoolTokens *Submitted by hyh.*
- [N-05] Aave/Compound pools result in liquidity mining returns being lost *Submitted by TomFrench.*

- [N-06] `depositYieldBearing` didn't check address `!= 0` *Submitted by pants.*
- [N-07] No zero address check for controller in `TempusPool` *Submitted by loop.*
- [N-08] `_setAmplificationData` should clear upper bits of values *Submitted by cmichel.*
- [N-09] cToken funds are locked if Compound's exchange rate is 0 *Submitted by pmerkleplant.*
- [N-10] Improper Access Control *Submitted by defsec.*
- [N-11] Lack of validation for Maturity Date *Submitted by Koustre.*
- [N-12] Manipulating `updateInterestRate()` in Tempus Pools to mint more Principal and Yield Tokens Than They Should *Submitted by 0xMesaj.*

Gas Optimizations (16)

- [G-01] Use of `matured` storage variable is unnecessary *Submitted by TomFrench.*
- [G-02] Repeated token transfers on deposits are unnecessary *Submitted by TomFrench.*
- [G-03] TempusAMM freezing all actions except proportional exit on maturity seems unnecessary *Submitted by TomFrench.*
- [G-04] Use of `uint8` for counter in `for` loop increases gas costs *Submitted by TomFrench, also found by pauliax.*
- [G-05] Adding unchecked directive can save gas *Submitted by WatchPug, also found by pauliax.*
- [G-06] Cache array length in for loops can save gas *Submitted by WatchPug, also found by pants.*
- [G-07] Gas: Don't store cToken twice *Submitted by cmichel.*
- [G-08] Prefix increaments are cheaper than postfix increaments *Submitted by pants.*
- [G-09] `public` functions can be `external` *Submitted by pants.*
- [G-10] Unused imports *Submitted by pauliax.*

- [G-11] `for` loop with `_TOTAL_TOKENS` *Submitted by pauliax.*
- [G-12] Long Revert Strings *Submitted by yeOlde.*
- [G-13] Named Return Issues *Submitted by yeOlde.*
- [G-14] Inheritance from `BaseGeneralPool` is unused *Submitted by TomFrench.*
- [G-15] Gas: `ERC20OwnerMintableToken.burn` should use caller *Submitted by cmichel.*
- [G-16] Make `protocolName` variables in protocol pools constant *Submitted by pmerkleplant.*

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