Timeswap contest
Findings & Analysis Report

2022-05-17

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 audit 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 audit contest outlined in this document, C4 conducted an analysis of the Timeswap smart contract system written in Solidity. The audit contest took place between March 4—March 6 2022.

Wardens

21 Wardens contributed reports to the Timeswap contest:

1. IllIllI
2. WatchPug (jtp and ming)
3. TerrierLover
4. ch13fd357r0y3r
5. hyh
6. Dravee

7. gzeon
8. 0x1f8b
9. robee
10. rfa
11. kenta
12. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
13. cryptphi
14. peritoflores
15. Tomio
16. 0v3rf10w

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 10 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 12 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Timeswap contest repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 1,654 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/non-critical.

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] Wrong timing of check allows users to withdraw collateral without paying for the debt

Submitted by WatchPug, also found by IllIllI

TimeswapPair.sol#L459-L490

```
    function pay(PayParam calldata param)
        external
        override
        lock
        returns (
            uint128 assetIn,
            uint128 collateralOut
        )
    {
        require(block.timestamp < param.maturity, 'E202');
        require(param.owner != address(0), 'E201');
        require(param.to != address(0), 'E201');
        require(param.to != address(this), 'E204');
        require(param.ids.length == param.assetsIn.length, 'E205');
        require(param.ids.length == param.collateralsOut.length, 'E2

        Pool storage pool = pools[param.maturity];

        Due[] storage dues = pool.dues[param.owner];
        require(dues.length >= param.ids.length, 'E205');

        for (uint256 i; i < param.ids.length;) {
            Due storage due = dues[param.ids[i]];
            require(due.startBlock != BlockNumber.get(), 'E207');
            if (param.owner != msg.sender) require(param.collaterals
            require(uint256(assetIn) * due.collateral >= uint256(col
            due.debt -= param.assetsIn[i];
            due.collateral -= param.collateralsOut[i];
            assetIn += param.assetsIn[i];
            collateralOut += param.collateralsOut[i];
            unchecked { ++i; }
        }
        ...
```

At L484, if there is only one `id`, and for the first and only time of the for loop, `assetIn` and `collateralOut` will be `0`, therefore `require(uint256(assetIn) * due.collateral >= uint256(collateralOut) * due.debt, 'E303');` will pass.

A attacker can call `pay()` with `param.assetsIn[0] == 0` and `param.collateralsOut[i] == due.collateral`.

Proof of Concept

The attacker can:

1. `borrow()` `10,000 USDC` with `1 BTC` as `collateral`;
2. `pay()` with `0 USDC` as `assetsIn` and `1 BTC` as `collateralsOut`.

As a result, the attacker effectively stole `10,000 USDC`.

Recommended Mitigation Steps

Change to:

```
    for (uint256 i; i < param.ids.length;) {
        Due storage due = dues[param.ids[i]];
        require(due.startBlock != BlockNumber.get(), 'E207');
        if (param.owner != msg.sender) require(param.collateralsOut|
        due.debt -= param.assetsIn[i];
        due.collateral -= param.collateralsOut[i];
        assetIn += param.assetsIn[i];
        collateralOut += param.collateralsOut[i];
        unchecked { ++i; }
    }

    require(uint256(assetIn) * due.collateral >= uint256(collateral(
    ...
```

Mathepreneur (Timeswap) resolved and commented:

> Timeswap-Labs/Timeswap-V1-Core@b23b44a

0xleastwood (judge) commented:

> This is an interesting find. It appears that `assetIn` and `collateralOut` are not checked properly during the first iteration of the for loop. As a result, this functionality of this function is inherently broken as the `require` statement will always be satisfied. Nice job!

Medium Risk Findings (3)

[M-01] Underflown variable in `borrowGivenDebtETHCollateral` function

Submitted by TerrierLover

`borrowGivenDebtETHCollateral` function does never properly call `ETH.transfer` due to underflow. If `borrowGivenDebtETHCollateral` function is not deprecated, it would cause unexpected behaviors for users.

Proof of Concept

Here are codes which contain a potential issue.

[Borrow.sol#L121-L127]

```
    if (maxCollateral > dueOut.collateral) {
        uint256 excess;
        unchecked {
            excess -= dueOut.collateral;
        }
        ETH.transfer(payable(msg.sender), excess);
    }
```

`excess` variable is `uint256`, and `dueOut.collateral` variable is `uint112` as shown below. Hence, both variables will never be less than 0.

[IPair.sol#L22-L26]

```
    struct Due {
        uint112 debt;
        uint112 collateral;
        uint32 startBlock;
    }
```

`uint256 excess` is initialized to 0. However, subtracting `dueOut.collateral` variable which is more than or equal to 0 from `excess` variable which is 0 will be less than 0. Hence, `excess -= dueOut.collateral` will be less than 0, and `excess` will be underflown.

Recommended Mitigation Steps

The code should properly initialize `excess` variable.

`borrowGivenPercentETHCollateral` function uses `uint256 excess = maxCollateral` at similar functionality.

[Borrow.sol#L347]

Hence, just initializing `excess` variable with `maxCollateral` can be a potential workaround to prevent the underflown.

```
    if (maxCollateral > dueOut.collateral) {
        uint256 excess = maxCollateral;
        unchecked {
            excess -= dueOut.collateral;
        }
        ETH.transfer(payable(msg.sender), excess);
    }
```

amateur-dev (Timeswap) confirmed

Mathepreneur (Timeswap) resolved and commented:

> Timeswap-Labs/Timeswap-V1-Convenience@a6f0e74

0xleastwood (judge) commented:

> Awesome find! I believe `medium` severity is the appropriate risk as this function will always revert when `maxCollateral > dueOut.collateral`.

[M-02] The `pay()` function can still be DOSed

Submitted by IllIllI

From the prior contest:

> in the pay() function users repay their debt and in line 364:
> https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L364
> it decreases their debt.

> lets say a user wants to repay all his debt, he calls the pay() function with his full debt.
> an attacker can see it and frontrun to repay a single token for his debt (since it's

> likely the token uses 18 decimals, a single token is worth almost nothing)
> and since your solidity version is above 0.8.0 the line:
> due.debt -= assetsIn[i]; will revert due to underflow

> The attacker can keep doing it everytime the user is going to pay and since 1 token is baisicly 0$ (18 decimals) the attacker doesn't lose real money

code-423n4/2022-01-timeswap-findings#86 (comment)

The sponsor said this about the fix:

> The convenience contract will implement how much asset to pay in.

code-423n4/2022-01-timeswap-findings#86 (comment)

The `pay()` function however is still DOSable. Having the `Convenience` contract contain a workaround means the `Convenience` contract is no longer a convenience but a requirement.

```
                due.debt -= param.assetsIn[i];
```

TimeswapPair.sol#L485

Proof of Concept

From the prior contest:

> A DoS on every user that repay his full debt (or enough that the difference between his total debt to what he pays his negligible)

code-423n4/2022-01-timeswap-findings#86

Recommended Mitigation Steps

Move the DOS protection to `TimeswapPair.pay()`

amateur-dev (Timeswap) confirmed

0xleastwood (judge) commented:

> I believe this is a potential security risk and `medium` risk is appropriate. It is important to ensure protocol availability cannot be negatively impacted by nefarious actors.

[M-03] NPM Dependency confusion. Unclaimed NPM Package and Scope/Org

Submitted by ch13fd357r0y3r

I discovered an npm package and the scope of the package is unclaimed on the NPM website. This will give any User to claim that package and be able to Upload a Malicious Code under that unclaimed package. This results in achieving the Remote code execution on developers/users' machine who depends on the timeswap repository to build it on local env.

Vulnerable Package Name: @timeswap-labs/timeswap-v1-core

Proof of Concept

1. Create an Organization called "timeswap-labs".
2. Create a package called "@timeswap-labs/timeswap-v1-core" under "timeswap-labs" Organization.
3. Attacker can able to upload malicious code on unclaimed npm package with a higher version like 99.99.99
4. Now if any user/timeswap developer installs it by npm install package.json. The malicious pkg will be executed.

Till now "The Package is not claimed on NPM Registry, but it's vulnerable to dependency confusion". You can read more dependency confusion here: https://dhiyaneshgeek.github.io/web/security/2021/09/04/dependency-confusion/

Recommended Mitigation Steps

Claim the Scope name called "timeswap-labs" by following the above POC Step 1.

[amateur-dev (Timeswap) confirmed and commented](#):

> Created the organisation. Thank you.

0xleastwood (judge) commented:

> I think this is an interesting attack vector and useful find!

> I would normally mark findings unrelated to Solidity code as `invalid`, however, I think the issue here raises an interesting exploit where an attacker could inject malicious code into a smart contract dependency. As such, I think this is relevant and a valid attack path.

Low Risk and Non-Critical Issues

For this contest, 10 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hyh** received the top score from the judge.

The following wardens also submitted reports: gzeon, Dravee, 0x1f8b, WatchPug, cryptphi, peritoflores, robee, rfa, and kenta.

[L-01] CollateralizedDebt.tokenURI fails when pair's asset or collateral token have low decimals

NFTTokenURIScaffold.sol#L166

Standard ERC721 tokenURI call will fail for CollateralizedDebt pools whose underlying pair.asset() or pair.collateral() have decimals lower than 4 whenever the corresponding due quantity is lower than 1e9.

Pair's asset and collateral ERC20 can be arbitrary and some ERC20 contracts have decimals lower than 4, so such a combination is possible. In such cases current tokenURI implementation fails, which can be the issue for all integrations down the line as various systems routinely make tokenURI calls.

Placing severity to be medium per 'Assets not at direct risk, but the function of the protocol or its availability could be impacted', which is the case here as protocol availability is in question when EIP level functionality fails.

References

https://eips.ethereum.org/EIPS/eip-721

https://github.com/d-xo/weird-erc20#low-decimals

Proof of Concept

NFTTokenURIScaffold.weiToPrecisionString will fail if used for a token with decimals lower than 4 as subtraction is performed without prior checks:

NFTTokenURIScaffold.sol#L166

NFTTokenURIScaffold.weiToPrecisionString is called by NFTTokenURIScaffold.tokenURI for pair's asset and collateral ERC20:

NFTTokenURIScaffold.sol#L16-L39

NFTTokenURIScaffold.tokenURI is used in CollateralizedDebt.tokenURI:

CollateralizedDebt.sol#L46

Pair's asset and collateral tokens can be arbitrary, while tokenURI is routinely requested by a variety of external systems

Recommended Mitigation Steps

Consider adding the check and special care for low decimals case, for example add another naming rule similarly to how `significantDigits > 1e9` case is being handled

amateur-dev (Timeswap) confirmed

vhawk19 (Timeswap) commented:

> Have tested with zero decimals, yields in desired output as expected.
>
> ```
>  to default on the debt payment, the collateral will be forfeited\n\nAsset Address: 0x959922be3caee4b8cd9a407cc3ac1c251c2007b1\nCollateral Address: 0x9a9f2ccfde556a7e9ff0848998aa4a0cfd8863ae\nDebt Required: 102.00 DAI\nCollateral Locked: 23.00 ETH"
> UQyRUZ9LkZ7ZmlsbDojNTE2MEVDfS5He3N0b3AtY29sb3I6IzIwMDg3RX0uSHtmaWxsOiM1NDU3RDd9Lkl7c3RvcC1jb2xvcjojNjFGNkZGfS5Ke3N0b3AtY29sb3I6IzNDNDNGRn1dXT48L3N0eWxlPjxnIGNsaXAtcGF0aD0idXJsKCNtYWluY2xpcCkiPjxyZWN0IHdpZHRoPSIyOTAiIGhlaWdodD0iNTAwIiBmaWxsPSJ1cmwoJyNi
> ```

Mathepreneur (Timeswap) resolved

0xleastwood (judge) commented:

> Am I correct in understanding that `weiAmt` is denominated using 18 decimals and not the `decimal` value provided as an input to the `weiToPrecisionString()` function? I'm curious as to why this is not an issue.

amateur-dev (Timeswap) commented:

> Hi, we are aware of the issue highlighted. We have taken a conscious call for the decimals. We have done testing and confirm that our lend, borrow and similar transactions do not have an issue. The issue only arises once we call the URI of tokens with less than 4 decimals. In that sense, these tokens URI will not show up properly in opensea. The exposure is limited to that. We do not rely on that URI for any function call in our dapp.

0xleastwood (judge) commented:

> As per the sponsor's comment, this issue only pertains to a token's URI which is shown in opensea. The unlikely edge case raised by the warden should only occur when tokens with less than 4 decimals are used. Because of the limited exposure to an attack, I'm inclined to mark this as `1 (Low Risk)`.

Gas Optimizations

For this contest, 12 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: IllIllI, CertoraInc, 0x1f8b, Tomio, WatchPug, gzeon, TerrierLover, robee, rfa, 0v3rf10w, kenta.

Foreword

- **Storage-reading optimizations**

> The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for

> more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- `@audit` **tags**

> The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

File: CollateralizedDebt.sol

modifier onlyConvenience()

Inline a modifier that’s only used once

As `onlyConvenience()` is only used once in this contract (in `function mint()`), it should get inlined to save gas:

```
File: CollateralizedDebt.sol
80:     modifier onlyConvenience() {
81:         require(msg.sender == address(convenience), 'E403');
82:         _;
83:     }
84:
85:     function mint(address to, uint256 id) external override
86:         _safeMint(to, id);
87:     }
```

File: TimeswapPair.sol

function mint()

Use memory variables for calculation

The code can be optimized from this:

```
File: TimeswapPair.sol
185:            pool.state.x += param.xIncrease;
186:            pool.state.y += param.yIncrease;
187:            pool.state.z += param.zIncrease;
```

```
...
193:          emit Sync(param.maturity, pool.state.x, pool.state.
```

to this:

```
File: TimeswapPair.sol
185:          (uint112 _poolStateX, uint112 _poolStateY, uint112
186:
187:          pool.state.x = _poolStateX;
188:          pool.state.y = _poolStateY;
189:          pool.state.z = _poolStateZ;
...
195:          emit Sync(param.maturity, _poolStateX, _poolStateY,
```

function burn()

`> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
File: TimeswapPair.sol
225:          require(pool.state.totalLiquidity > 0, 'E206'); //
```

Also, please enable the Optimizer.

function lend()

Use memory variables for calculation

Just like in `function mint()` (Use memory variables for calculation), the code can be optimized here by caching the new values for `pool.state.x`, `pool.state.y` and `pool.state.z` :

```
File: TimeswapPair.sol
310:         pool.state.x += param.xIncrease;
311:         pool.state.y -= param.yDecrease;
312:         pool.state.z -= param.zDecrease;
...
320:         emit Sync(param.maturity, pool.state.x, pool.state.
```

The same way, the final code will look like this (with the difference that `yDecrease` and `zDecreased` are used here):

```
            (uint112 _poolStateX, uint112 _poolStateY, uint112 _poc

            pool.state.x = _poolStateX;
            pool.state.y = _poolStateY;
            pool.state.z = _poolStateZ;
...
            emit Sync(param.maturity, _poolStateX, _poolStateY, _pc
```

function borrow()

Use memory variables for calculation

Just like in `function mint()` (Use memory variables for calculation) and `function lend()`, the code can be optimized here by caching the new values for `pool.state.x`, `pool.state.y` and `pool.state.z` :

```
File: TimeswapPair.sol
432:         pool.state.x -= param.xDecrease;
433:         pool.state.y += param.yIncrease;
434:         pool.state.z += param.zIncrease;
...
444:         emit Sync(param.maturity, pool.state.x, pool.state.
```

The same way, the final code will look like this (with the difference that `xDecrease` is used here):

```
            (uint112 _poolStateX, uint112 _poolStateY, uint112 _poo

            pool.state.x = _poolStateX;
            pool.state.y = _poolStateY;
            pool.state.z = _poolStateZ;
    ...
            emit Sync(param.maturity, _poolStateX, _poolStateY, _po
```

function pay()

An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
File: TimeswapPair.sol
480:         for (uint256 i; i < param.ids.length;) { //@audit c
```

General recommendation

Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until

> now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

See original submission for a list of instances.

I suggest replacing revert strings with custom errors.

amateur-dev (Timeswap) confirmed

Mathepreneur (Timeswap) resolved and commented:

> Timeswap-Labs/Timeswap-V1-Convenience@7434c3d
> Timeswap-Labs/Timeswap-V1-Core@7055734

> Some implementation can't be implemented due to stack too deep error.

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.