FEI and TRIBE Redemption contest Findings & Analysis Report

2022-11-08

Table of contents

- Overview
    - About C4
    - Wardens
- Summary
- Scope
- Severity Criteria
- Medium Risk Findings (2)
    - [M-01] A malicious user can send tokens to the `TribeRedeemer` contract to make the redeem function work, and other users may lose assets as a result
    - [M-02] `TribeRedeemer` will start redeeming incorrectly if someone transfer redeem tokens directly to it
- Low Risk and Non-Critical Issues
    - Executive Summary
    - 01 Inconsistent usage of `hasNotSigned`
    - 02 Left the Default `MESSAGE_HASH`

    - 03 Technically the maximum amount at risk is `amountToDrip &ast; 2 - 1`
    - 04 Tautology, slippage check is ineffective
    - 05 Trust in Deployer - No guarantee that base will be `totalSupply`
    - 06 Trust in Deployer - Lack of check for same decimals
    - 07 Notice - Slippage and Interest Free Loan for Arbitrageurs
    - 08 Usual Suspects: Lack of Zero Checks for new addresses
- Disclosures

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 FEI and TRIBE Redemption smart contract system written in Solidity. The audit contest took place between September 9—September 12 2022.

Wardens

107 Wardens contributed reports to the FEI and TRIBE Redemption contest:

1. rvierdiiev
2. cccz
3. GalloDaSballo
4. pauliax
5. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder, and RoiEvenHaim)
6. 0x1f8b
7. hansfriese

8. R2
9. rbserver
10. KIntern_NA (TrungOre and duc)
11. Lambda
12. brgltd
13. horsefacts
14. Jeiwan
15. pashov
16. rotcivegaf
17. unforgiven
18. c3phas
19. csanuragjain
20. hyh
21. izhuer
22. ladboy233
23. RaymondFam
24. sorrynotsorry
25. Tointer
26. V_B (Barichek and vlad_bochok)
27. 0xNazgul
28. yixxas
29. 0xSky
30. datapunk
31. dipp
32. Junnon
33. Picodes
34. Randyyy
35. __141345__
36. _Adam

37. Aymen0909
38. cryptphi
39. d3e4
40. innertia
41. TomJ
42. 0x4non
43. 0x52
44. Deivitto
45. smiling_heretic
46. wagmi
47. djxploit
48. fatherOfBlocks
49. leosathya
50. Mohandes
51. simon135
52. Sm4rty
53. Tomo
54. oyc_109
55. rokinot
56. Samatak
57. 0xSmartContract
58. Waze
59. ak1
60. Bnke0x0
61. Chom
62. cryptonue
63. delfin454000
64. JC
65. lucacez

66. prasantgupta52
67. robee
68. Rohan16
69. scaraven
70. ajtra
71. erictee
72. lukris02
73. ReyAdmirado
74. sach1r0
75. a12jmx
76. cryptostellar5
77. Diana
78. Funen
79. Rolezn
80. 0x040
81. asutorufos
82. Chandr
83. durianSausage
84. gogo
85. mics
86. ret2basic
87. bharg4v
88. got_targ
89. ignacio
90. karanctf
91. Noah3o6
92. Ocean_Sky
93. sikorico
94. Tagir2003

95. 0x85102
96. bobirichman
97. CodingNameKiki
98. rfa
99. SnowMan
100. SooYa
101. StevenL

This contest was judged by hickuphh3.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 2 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 101 reports detailing issues with a risk rating of LOW severity or non-critical.

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

Scope

The code under review can be found within the C4 FEI and TRIBE Redemption contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 423 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](#).

Medium Risk Findings (2)

[M-01] A malicious user can send tokens to the `TribeRedeemer` contract to make the redeem function work, and other users may lose assets as a result

Submitted by cccz

In the TribeRedeemer contract, the user provides redeemedToken to redeem the tokens in the list.
The transaction will revert when the balance of tokens in the list is 0. This prevents the user from losing their redeemedToken if they redeem when there are no tokens in the contract.

```
        for (uint256 i = 0; i < tokensReceived.length; i++) {
            uint256 balance = IERC20(tokensReceived[i]).balanceO
            require(balance != 0, "ZERO_BALANCE");
            // @dev, this assumes all of `tokensReceived` and `r
            // have the same number of decimals
            uint256 redeemedAmount = (amountIn * balance) / base
            amountsOut[i] = redeemedAmount;
        }
```

However, a malicious user could send tokens in the list to the TribeRedeemer contract so that the token balance is not 0. This would allow the redeem function to work, and the user would suffer a loss when they redeem early by mistake.

Proof of Concept

TribeRedeemer.sol#L44-L61

Recommended Mitigation Steps

Consider making the TribeRedeemer contract inherit the Pausable contract and allow users to redeem when a sufficient number of tokens have been sent to the contract.

thomas-waite (FEI and TRIBE) disputed and commented:

> Do not understand what the issue presented is. If the attacker sends funds to the contract so the balances are not 0, then the user would be able to redeem as normal. How do they suffer any loss if they 'redeem early'?

> The balances will only be 0 when all users have redeemed. Not an issue.

cccz (warden) commented:

> @thomas-waite, consider the following scenario.

1. there are no reward tokens in the TribeRedeemer contract now, (the administrator will transfer 100M reward tokens to the contract in exchange for the user's redeemedToken only after some time).
2. > when the user provides redeemedToken to call the redeem function, the transaction reverts in the previewRedeem function because the reward token balance is 0 (this prevents the user from getting 0 reward tokens).

```
        for (uint256 i = 0; i < tokensReceived.length; i++) {
            uint256 balance = IERC20(tokensReceived[i]).balanceOf(address(t
            require(balance != 0, "ZERO_BALANCE");
```

3. a malicious user can send 100 reward tokens to the contract in advance.
4. At this point, if the user calls the redeem function, the transaction will not revert and the contract will exchange the 100 reward tokens for the user's

redeemedToken. The user could have exchanged the redeemedToken for more reward tokens, but the user only got a small amount of reward tokens

hickuphh3 (judge) commented:

> Redemptions can only begin when the contract has non-zero balances for all redeemed tokens, but the start redemption time isn't explicitly stated.

> Malicious users can break this assumption by sending paltry amounts to the contract as explained above. Naiive users might begin redemptions early, thus losing out on the tokens they would've otherwise received after the full redemption token amounts have been sent to the contract.

> It is unclear if there is a time lag between the contract deployment and time at which redemption funds are sent, and if so, its duration.

> While unlikely to happen, it would be a case of users losing out on rewards they should be entitled to. Hence, I'm siding with the warden in this instance.

thomas-waite (FEI and TRIBE) commented:

> @hickuphh3 , the contract gets deployed ahead of time by necessity (the address is needed for the DAO vote). The DAO vote which funds the contract executes after deployment, there is a time lag of >= 3 days.

> I do not agree that the situation described is a vulnerability in the contract and instead it would be down to user/deployer error. Clearly, no user was encouraged to redeem before funds were available and the contract address was not publicised. So no user attempted to redeem before funds were available.

hickuphh3 (judge) commented:

> I respectfully disagree.

> There is no doubt that there are a number of prerequisites to enable this attack, which makes the likelihood low:

- Malicious actor has to know what the redeemer contract is
- Malicious actor has to deposit some tokens into redeemer contract

- Malicious actor must be able to trick users into redeeming early

> From a game theory POV, small TRIBE holders could band together to target a few large TRIBE holders.

> With these external requirements, it is only then that we will see what could be deemed as “protocol leaked value”.

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements

> The scenario thankfully remained hypothetical.

[M-02] `TribeRedeemer` will start redeeming incorrectly if someone transfer redeem tokens directly to it

Submitted by rvierdiiev

`TribeRedeemer` contract has one goal. It should take `_redeemedToken` (token that contract will accept from users), `_tokensReceived` - list of exchange tokens(this are created exactly to be changed for redeemedToken) and `_redeemBase` amount of tokens that should be redeemed(this actually should be `IERC20(_redeemedToken).totalSupply())`. After that it will start redeeming `_redeemedToken` in exchange to `_tokensReceived` tokens.

Suppose we have redeemed token `tokenA` with total supply of `10000` and `redeemBase == 10000`. And in `_tokensReceived` list we have only 1 token `tokenB` with total supply of `10000` (all tokens are controlled by `TribeRedeemer`). According to logic of `TribeRedeemer` if user wants to redeem X tokens then he will receive `(x * tokenB.balanceOf(address(this))) / redeemBase` that in our case will be just amount X. So user send X `tokenA` and receive back X `tokenB`. Now because `redeemBase == 10000` and contract balance of `tokenB` is `10000` the exchange ratio is `1:1`.

However if someone will transfer some amount of `tokenA` to `TribeRedeemer` contract directly using `ERC20.transfer` for example `500` tokens then it not call `redeem` function and `redeemBase` value will not be decreased with amount sent. That means that right now the exchange ratio should not be `1:1` as `TribeRedeemer` contract received more `500` tokens and didn't pay user for them(it should redeem that amount then with the next users, they share should be increased). So the next users will receive less amount then they should(`TribeRedeemer` should spend all `tokenB` amount in exchange of all `tokenA` tokens).

Proof of Concept

Here is where `amounOut` is calculated.
[TribeRedeemer.sol#L58]()

This is where the `redeemBase` is decreased with redeem amount, but is not called because of direct transfer.
[TribeRedeemer.sol#L70]()

Recommended Mitigation Steps

Do not use `redeemBase` for calculation amount of tokens that are left for redeeming. Use `IERC20(_redeemedToken).totalSupply()) - IERC20(_redeemedToken).balanceOf(address(this))`.

[thomas-waite (FEI and TRIBE) disputed and commented]():

> Any `redeemedToken`s sent to this contract will be locked and effectively burned. They are not to be used in the accounting of redemptions or `redeemBase`.

[hickuphh3 (judge) commented]():

> Hmmm, the warden has a point actually. Currently, the rewards tied to TRIBE tokens that were sent directly to the contract instead of via `redeem()` becomes unrecoverable (burnt) as well.

> It's not a bad suggestion; it at least helps to split the rewards of directly sent TRIBE tokens with the remaining TRIBE holders, which acts as a form of reward recovery.

> The recommended mitigation is good, but perhaps modified to be an immutable value / constant `_redeemBase`, then the calculation becomes

```
uint256 redeemedAmount = (amountIn * balance) / (_redeemBase - I
```

> Keeping the medium severity as it's an indirect loss of rewards that can be recovered / re-distributed.

thomas-waite (FEI and TRIBE) commented:

> @hickuphh3 - All TRIBE tokens have an equal claim to the funds that are locked in this contract, irrespective of what other TRIBE tokens do.

> The situation described is just a special case of TRIBE tokens being locked out of circulation. It's not feasible for this contract to account for all locations where user circulating TRIBE is subsequently locked/removed from circulation and nor is it particularly desired - all TRIBE has an equal claim to the funds held on the contract. Dispute that this is an issue

hickuphh3 (judge) decreased severity and commented:

> Agree with the intention to keep all claims equal, and that it's not feasible to handle all outlier cases.

> Implementing the suggestion would change user behaviour to claim as late as possible, since the TRIBE amount that's mistakenly sent only increases over time.

> Downgrading to QA.

rvierdiiev (warden) commented:

> First of all I would like to say that current implementation of TribeRedeemer is not going to pay redeemers equal claims. This is important because the sponsor argues that if we change the implementation like I have suggested in submission, then next users will get more rewards. Also hickuphh3 said that this incentivize people to call redeem as late as possible to get bigger reward.

> I will explain why current implementation is not going to pay equal claims. This is because you can top up any exchange tokens directly(transfer to TribeRedeemer) and then next users will get bigger share than previous, because the formula uses exchange token balance to calculate reward amount.
> uint256 redeemedAmount = (amountIn * balance) / base;

> where balance is
> uint256 balance = IERC20(tokensReceived[i]).balanceOf(address(this));

> So the assumption of sponsor about equal claims is not true.
> And also hickuphh3 assumption that my proposed change will push users to redeem tokens later is partly wrong. They are incentivized to do so already in current implementation(as someone can transfer some exchange token to the TribeReddemer).
> Why i suggested to use redeem token balance, because i saw in the code that TribeRedeemer is going to spend all exchange tokens for the redeemed tokens.

> All exhange tokens(tokensReceived in the code) are sent to the TribeRedeemer contract before and there is no way for TribeRedeemer to return any funds that are still controlled by contract.
> That means that the purpose of TribeRedeemer is to distribute all exchange tokens to the redeemers(there is no logic to leave some part of that tokens locked in the contract, it's better to distribute everything).

> Let's see 3 situations.

1. Someone top ups exchange tokens. Then TribeRedeemer will send more reward to all next users. Finally, TribeRedeemer will send users all the exchange tokens he had(so in the end when the last redeemed token is sent to TribeRedeemer, there is no exchange tokens controlled by TribeRedeemer).
2. Someone top ups redeem token(as i described in the submission). TribeRedeemer will still send users same part of exchange tokens. Finally, when all redeem tokens are controlled by TribeRedeemer, exchange token will have funds of TribeRedeemer that is not distributed among the redeemers. This funds are locked.
3. Someone top ups both exchange token and redeem token. At this time next users will have bigger reward than previous(so sponsors assumption is broken) and in the end some exchange token will still be controlled by TribeRedeemer.

> I hope, that I showed that current implementation can't guarantee equal rewards for users and it will be logical to use the way that i proposed to not just burn exchange tokens but distribute them among redeemers.
> If sponsor wants to pay equal claims, then he need to initialize TribeRedeemer with one more param exchangeTokensPerRedeemToken array where he can provide

> fixed amount of tokens to pay for redeemed token and use smth like this
> uint256 redeemedAmount = (amountIn * exchangeTokensPerRedeemToken) / base;

> Only in this case it's possible to get users equal claims.

[hickuphh3 (judge) increased severity to Medium and commented]():

> In the course of my investigation to see if a user has accidentally directly sent TRIBE to the redeemer contract, I have discovered that what @rvierdiiev said about equal claims is true:

> And also hickuphh3 assumption that my proposed change will push users to redeem tokens later is partly wrong. They are incentivized to do so already in current implementation(as someone can transfer some exchange token to the TribeReddemer).

> The RGT reserve and timelock sent an additional ~430k DAI to the redeemer contract recently, after redemptions have begun:

- https://etherscan.io/tx/0xe0d2a879e05c8de8ebe23cef2f380fc68c1e52b21961b6650d076e9f839128f7
- https://etherscan.io/tx/0xab80a9982d45ced06f3f63e45a4fde5def2bc9270df0e639e9a9e84ba5a24528

> This user [who redeemed close to 17M TRIBE]() missed out on about `16856213429000000000000000 / 458964340000000000000000000 * 430000 ~= 15792.45` additional DAI.

> Hence, it is a fact that not all claims are equal; later claims have already benefitted from additional tokens that were sent to the redeemer contract.

> I've flipped flopped on my decision, but in light of what I found, my rationale for the downgrade no longer stands.

> Implementing the suggestion would change user behaviour to claim as late as possible, since the TRIBE amount that's mistakenly sent only increases over time.

> As @rvierdiiev pointed out, this is already true, although I would argue that implementing this suggestion would worsen the problem.

> The situation described is just a special case of TRIBE tokens being locked out of circulation. It's not feasible for this contract to account for all locations where user circulating TRIBE is subsequently locked/removed from circulation and nor is it particularly desired.

> I agree that the protocol should not be expected to handle all outlier cases; and that accidentally sending tokens to the redeemer contract is a special case. However, it is precisely because it is a boundary / special case that the protocol is able to handle and remedy; it most cases, it doesn't have the ability to do so. It's akin to token contracts blocking `address(this)` in `to` addresses. If a path exists to recover rewards that would've been lost together with the TRIBE tokens by redistributing it, why not?

> Keeping the medium severity as it fulfils the condition: protocol leaking value (rewards lost that in this case can be remedied) with a hypothetical attack path with stated assumptions, but external requirements. This vulnerability can also be classified as a user-error bug, which have been awarded up to medium severity in previous contests Eg. overpaying with ETH.

Low Risk and Non-Critical Issues

For this contest, 101 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **GalloDaSballo** received the top score from the judge.

The following wardens also submitted reports: pauliax, CertoraInc, 0x1f8b, hansfriese, R2, rbserver, KIntern_NA, Lambda, rvierdiiev, brgltd, horsefacts, Jeiwan, pashov, rotcivegaf, unforgiven, c3phas, csanuragjain, hyh, izhuer, ladboy233, RaymondFam, sorrynotsorry, Tointer, V_B, 0xNazgul, yixxas, 0xSky, cccz, datapunk, dipp, Junnon, Picodes, Randyyy, __141345__, _Adam, Aymen0909, cryptphi, d3e4, innertia, TomJ, 0x4non, 0x52, Deivitto, smiling_heretic, wagmi, djxploit, fatherOfBlocks, leosathya, Mohandes, simon135, Sm4rty, Tomo, oyc_109, rokinot, Samatak, 0xSmartContract, Waze, ak1, Bnke0x0, Chom, cryptonue, delfin454000, JC, lucacez, prasantgupta52, robee, Rohan16, scaraven, ajtra, erictee, lukris02, ReyAdmirado, sach1r0, a12jmx, cryptostellar5, Diana, Funen, Rolezn, 0x040, asutorufos, Chandr, durianSausage, gogo, mics, ret2basic, bharg4v, got_targ, ignacio, karanctf, Noah3o6, Ocean_Sky, sikorico, Tagir2003, 0x85102, bobirichman, CodingNameKiki, rfa, SnowMan, SooYa, *and* StevenL.

Executive Summary

Personally haven't found any glaring vulnerability.

Because of the mix of immutability and Admin Trust, end users should review the deployment settings at that time to ensure that no misconfigurations have happened.

In case any misconfigurations happens, a new deployment will be required.

Listed below are some observations of things that could be refactored to make the code more consistent, as well as some gotchas that are introduced by the choice of architecture.

[01] Inconsistent usage of `hasNotSigned`

`signAndClaimAndRedeem` has the modifier while `signAndClaim` doesn't.

RariMerkleRedeemer.sol#L108-L118

```
    function signAndClaimAndRedeem(
        bytes calldata signature,
        address[] calldata cTokens,
        uint256[] calldata amountsToClaim,
        uint256[] calldata amountsToRedeem,
        bytes32[][] calldata merkleProofs
    ) external override hasNotSigned nonReentrant {
        _sign(signature);
        _multiClaim(cTokens, amountsToClaim, merkleProofs);
        _multiRedeem(cTokens, amountsToRedeem);
    }
```

Doesn't have the modifier
RariMerkleRedeemer.sol#L88-L97

```
    function signAndClaim(
        bytes calldata signature,
        address[] calldata cTokens,
        uint256[] calldata amounts,
        bytes32[][] calldata merkleProofs
    ) external override nonReentrant {
        // both sign and claim/multiclaim will revert on invalid
        _sign(signature);
        _multiClaim(cTokens, amounts, merkleProofs);
    }
```

Mitigation Steps

Add the modifier `hasNotSigned` for consistency, or remove the modifier altogether and allow to sign multiple times the same message.

[02] Left the Default `MESSAGE_HASH`

For the in-scope code `MESSAGE` is left to the default value

MultiMerkleRedeemer.sol#L53-L54

```
    string public constant MESSAGE = "Sample message, please upd
```

This could cause the signature to be replayable in other applications that use the same message.

Mitigation Steps

Add the proper message, most likely a TOS acknowledgement or a ipfs hash to a document.

[03] Technically the maximum amount at risk is `amountToDrip &ast; 2 - 1`

While the code is meant to limit the total asset at risk, it’s important to notice that because of the following check:

MerkleRedeemerDripper.sol#L23-L24

```
            IERC20(token).balanceOf(target) < amountToDrip,
```

Any balance below `amountToDrip`, after enough time has passed, will alow to drip more, meaning the total amount at risk is not `amountToDrip` but up to `2 * amountToDrip - 1`.

Mitigation Steps

I recommend commenting this.

[04] Tautology, slippage check is ineffective

Because `amountOut = amountFeiIn;` and `amountFeiOut = amountIn;` there is no slippage, and no risk for any front-run.

The check below is always true for that reason:

SimpleFeiDaiPSM.sol#L54-L55

```
        require(amountOut >= minAmountOut, "SimpleFeiDaiPSM: Red
```

Mitigation Steps

A better require would be to check if the balance of the token that will be transferred out is sufficient to avoid a revert on the low level balance subtraction.

[05] Trust in Deployer - No guarantee that base will be `totalSupply`

TribeRedeemer.sol#L34-L35

```
        redeemBase = _redeemBase;
```

Because `redeemedToken` is known, you could just retrieve the `totalSupply` from it to ensure the claims are for all tokens available.

End users will have to verify that `redeemBase` is consistent with the Circulating Supply.

Mitigation Steps

Comment and let end users know of this, or use `totalSupply` from the token.

[06] Trust in Deployer - Lack of check for same decimals

Because TribeRedeemer assumes all assets will have the same decimals, tokens can remain stuck if they have a different amount of decimals.

A simple check for decimals in the constructor can avoid this scenario
[TribeRedeemer.sol#L33-L34]()

```
            tokensReceived = _tokensReceived;
```

In lack of a check, end users will have to verify all tokens have 18 decimals.

All of the tokens in the README have 18 decimals at this time.

[07] Notice - Slippage and Interest Free Loan for Arbitrageurs

Because `redeem` doesn't burn FEI, any caller can `mint` and `redeem` multiple times in the same tx with the goal of arbing out the FEI - DAI pair.

[SimpleFeiDaiPSM.sol#L105-L106]()

```
        function burnFeiHeld() external {
```

While FEI being tradeable for DAI is enforcing a 1-1 trade (FEI price goes up due to arbing, up to 100% + FEE), allowing the opposite swap is a easy target for arbitrageurs.

[08] Usual Suspects: Lack of Zero Checks for new addresses

Constructors don't have zero-checks, which could force a re-deployment, funds are not at risk in those cases.

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