Tally contest
Findings & Analysis Report

2021-11-19

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 the Tally smart contract system written in Solidity. The code contest took place between October 20—October 22 2021.

Wardens

15 Wardens contributed reports to the Tally contest code contest:

1. WatchPug (jtp and ming)
2. harleythedog
3. pants
4. cmichel
5. pauliax
6. leastwood
7. elprofesor
8. JMukesh
9. TomFrench
10. 0xngndev
11. Koustre
12. ye0lde

13. defsec
14. csanuragjain
15. gpersoon

This contest was judged by Oxean.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities 40 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity, 3 received a risk rating in the category of MEDIUM severity, and 9 received a risk rating in the category of LOW severity.

C4 analysis also identified 6 non-critical recommendations and 20 gas optimizations.

Scope

The code under review can be found within the C4 Tally contest repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 407 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 (2)

[H-01] Arbitrary contract call allows attackers to steal ERC20 from users' wallets

Submitted by WatchPug

Swap.sol L200-L212

```
    function fillZrxQuote(
        IERC20 zrxBuyTokenAddress,
        address payable zrxTo,
        bytes calldata zrxData,
        uint256 ethAmount
    ) internal returns (uint256, uint256) {
        uint256 originalERC20Balance = 0;
        if(!signifiesETHOrZero(address(zrxBuyTokenAddress))) {
            originalERC20Balance = zrxBuyTokenAddress.balanceOf(addr
        }
        uint256 originalETHBalance = address(this).balance;

        (bool success,) = zrxTo.call{value: ethAmount}(zrxData);
```

A call to an arbitrary contract with custom calldata is made in `fillZrxQuote()`, which means the contract can be an ERC20 token, and the calldata can be `transferFrom` a previously approved user.

Impact

The wallet balances (for the amount up to the allowance limit) of the tokens that users approved to the contract can be stolen.

PoC

Given:

- Alice has approved 1000 WETH to `Swap.sol`;

The attacker can:

```
    TallySwap.swapByQuote(
        address(WETH),
        0,
        address(WETH),
        0,
        address(0),
        address(WETH),
        abi.encodeWithSignature(
            "transferFrom(address,address,uint256)",
            address(Alice),
            address(this),
            1000 ether
        )
    )
```

As a result, 1000 WETH will be stolen from Alice and sent to the attacker.

This PoC has been tested on a forking network.

Recommendation

Consider adding a whitelist for `zrxTo` addresses.

Shadowfiend (Tally) confirmed

[H-02] Wrong calculation of `erc20Delta` and `ethDelta`

Submitted by WatchPug, also found by harleythedog

`Swap.sol` **L200-L225**

```
    function fillZrxQuote(
```

```
        IERC20 zrxBuyTokenAddress,
        address payable zrxTo,
        bytes calldata zrxData,
        uint256 ethAmount
    ) internal returns (uint256, uint256) {
        uint256 originalERC20Balance = 0;
        if(!signifiesETHOrZero(address(zrxBuyTokenAddress))) {
            originalERC20Balance = zrxBuyTokenAddress.balanceOf(addr
        }
        uint256 originalETHBalance = address(this).balance;

        (bool success,) = zrxTo.call{value: ethAmount}(zrxData);
        require(success, "Swap::fillZrxQuote: Failed to fill quote")

        uint256 ethDelta = address(this).balance.subOrZero(originalE
        uint256 erc20Delta;
        if(!signifiesETHOrZero(address(zrxBuyTokenAddress))) {
            erc20Delta = zrxBuyTokenAddress.balanceOf(address(this))
            require(erc20Delta > 0, "Swap::fillZrxQuote: Didn't rece
        } else {
            require(ethDelta > 0, "Swap::fillZrxQuote: Didn't receiv
        }

        return (erc20Delta, ethDelta);
    }
```

When a user tries to swap unwrapped ETH to ERC20, even if there is a certain amount of ETH refunded, at L215, `ethDelta` will always be `0`.

That's because `originalETHBalance` already includes the `msg.value` sent by the caller.

Let's say the ETH balance of the contract is `1 ETH` before the swap.

- A user swaps `10 ETH` to USDC;
- `originalETHBalance` will be `11 ETH`;
- If there is `1 ETH` of refund;
- `ethDelta` will be `0` as the new balance is `2 ETH` and `subOrZero(2, 11)` is `0`.

Similarly, `erc20Delta` is also computed wrong.

Consider a special case of a user trying to arbitrage from `WBTC` to `WBTC`, the `originalERC20Balance` already includes the input amount, `erc20Delta` will always be much lower than the actual delta amount.

For example, for an arb swap from `1 WBTC` to `1.1 WBTC`, the `ethDelta` will be `0.1 WBTC` while it should be `1.1 WBTC`.

Impact

- User can not get ETH refund for swaps from ETH to ERC20 tokens;
- Arb swap with the same input and output token will suffer the loss of almost all of their input amount unexpectedly.

Recommendation

Consider subtracting the input amount from the originalBalance.

Shadowfiend (Tally) confirmed:

> This doesn't allow explicit stealing by an attacker, but does leak value. We would suggest a (2) severity on this.

0xean (judge) commented:

> This results in a user losing assets that they will never be able to recover. Per documentation

> `3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).`

> Lost assets are a high sev.

Medium Risk Findings (3)

[M-01] Swap.sol implements potentially dangerous transfer

Submitted by elprofesor, also found by WatchPug, Koustre, cmichel, JMukesh, and pauliax

Impact

The use of `transfer()` in `Swap.sol` may have unintended outcomes on the eth being sent to the receiver. Eth may be irretrievable or undelivered if the `msg.sender` or `feeRecipient` is a smart contract. Funds can potentially be lost if;

1. The smart contract fails to implement the payable fallback function
2. The fallback function uses more than 2300 gas units

The latter situation may occur in the instance of gas cost changes. The impact would mean that any contracts receiving funds would potentially be unable to retrieve funds from the swap.

Proof of Concept

This issue directly impacts the following lines of code:

- L257
- L173
- L158

Examples of similar issues ranked as medium can be found here and here, just search for 'M04'. A detailed explanation of why relying on `payable().transfer()` may result in unexpected loss of eth can be found here

Tools Used

Manual review

Recommended Mitigation Steps

Re-entrancy has been accounted for in all functions that reference `Solidity's` transfer() `. This has been done by using a re-entrancy guard, therefore,`

we can rely on msg.sender.call.value(amount)`or using the OpenZeppelin [Address.sendValue library](#)

[Shadowfiend (Tally) acknowledged](#)

[M-02] Unused ERC20 tokens are not refunded

Submitted by WatchPug

Based on the context and comments in the code, we assume that it's possible that there will be some leftover sell tokens, not only when users are selling unwrapped ETH but also for ERC20 tokens.

However, in the current implementation, only refunded ETH is returned (L158).

Because of this, the leftover tkoens may be left in the contract unintentionally.

Swap.sol **L153-L181**

```
    if (boughtERC20Amount > 0) {
        // take the swap fee from the ERC20 proceeds and return the
        uint256 toTransfer = SWAP_FEE_DIVISOR.sub(swapFee).mul(bough
        IERC20(zrxBuyTokenAddress).safeTransfer(msg.sender, toTransf
        // return any refunded ETH
        payable(msg.sender).transfer(boughtETHAmount);

        emit SwappedTokens(
            zrxSellTokenAddress,
            zrxBuyTokenAddress,
            amountToSell,
            boughtERC20Amount,
            boughtERC20Amount.sub(toTransfer)
        );
    } else {

        // take the swap fee from the ETH proceeds and return the re
        // that if any 0x protocol fee is refunded in ETH, it also s
        // the swap fee tax
        uint256 toTransfer = SWAP_FEE_DIVISOR.sub(swapFee).mul(bough
        payable(msg.sender).transfer(toTransfer);
        emit SwappedTokens(
```

```
            zrxSellTokenAddress,
            zrxBuyTokenAddress,
            amountToSell,
            boughtETHAmount,
            boughtETHAmount.sub(toTransfer)
        );
    }
```

Shadowfiend (Tally) acknowledged:

> I believe the 0x API does in fact guarantee that we won't have any sell tokens left over, particularly since we intend to use RFQ-T for these quotes, but if not we will fix this... And we may make the change regardless to future-proof.

0xean (judge) commented:

> Downgrading to sev 2

> `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.`

> As I believe this to be a "leak value" scenario.

[M-03] Users can avoid paying fees for ETH swaps

Submitted by pants

Users can call `Swap.swapByQuote()` to execute an ETH swap (where they receive ETH) without paying swap fee for the gained ETH. They can trick the system by setting `zrxBuyTokenAddress` to an address of a malicious contract and making it think they have executed an ERC20 swap when they have actually executed an ETH swap. In this case, the system will give them the ETH they got from the swap (`boughtETHAmount`) without charging any swap fees for it, because the systems consideres this ETH as "refunded ETH" that wasn't part of the "ERC20" swap.

Impact

Users can execute ETH swap without paying swap fees for the ETH the got from the swap.

Proof of Concept

The steps of the attack are:

1. Deploy a malicious contract (denoted by `M`), that will be used for `zrxBuyTokenAddress`.
2. Call `Swap.swapByQuote()` with `zrxBuyTokenAddress=M` and `minimumAmountReceived=0`. The rest of the arguments should specify our ETH swap, nothing special here.
3. Define `M` to return `0` and `1` at the first and second times when `fillZrxQuote` calls `zrxBuyTokenAddress.balanceOf(address(this))`, respectively.
4. As a result, `boughtERC20Amount` now equals `1` and the function will "return any refunded ETH" to the caller, without charging any swap fees on it. This ETH is actually the output of that ETH swap.

Tool Used

Manual code review.

Recommended Mitigation Steps

Charge swap fees for the "refunded ETH" on ERC20 swaps (when `boughtERC20Amount > 0`), or require `boughtETHAmount == 0`.

Shadowfiend (Tally) acknowledged:

> Still working through whether this is an issue we're truly worried about; in particular, if you want to do this, you probably might as well use the 0x API to swap directly.

> Nonetheless, it's overshadowed by #37, which will likely lead to changes that will make this impractical as well.

0xean (judge) commented:

> Downgrading to severity 2 as this would lead to “leaked value” as only the fees are lost by the protocol in this attack vector and customer assets aren’t being stolen.

Low Risk Findings (9)

- [L-01] `Ownable` Contract Does Not Implement Two-Step Transfer Ownership Pattern *Submitted by leastwood, also found by elprofesor and WatchPug*
- [L-02] Wrong value for `SwappedTokens` event parameter *Submitted by WatchPug*
- [L-03] Insufficient input validation *Submitted by WatchPug, also found by cmichel, leastwood, and pants*
- [L-04] Incorrect `FeesSwept` amount being emitted in `sweepFees` function *Submitted by harleythedog, also found by pauliax*
- [L-05] Token Can Deny Execution of `sweepFees()` Function *Submitted by leastwood, also found by JMukesh and 0xngndev*
- [L-06] Swap fee can be set to 100% *Submitted by TomFrench, also found by pauliax*
- [L-07] Contract does not work well with fee-on transfer tokens *Submitted by cmichel*
- [L-08] Events not indexed *Submitted by harleythedog*
- [L-09] Open TODOs *Submitted by pants*

Non-Critical Findings (6)

- [N-01] use of floating pragma *Submitted by JMukesh*
- [N-02] Consider removing `Math.sol` *Submitted by WatchPug*
- [N-03] Inclusive check *Submitted by pauliax*
- [N-04] `Swap.setSwapFee()` emits a `NewSwapFee` when the swap fee hasn’t changed *Submitted by pants*
- [N-05] `Swap.setFeeRecipient()` emits a `NewFeeRecipient` when the fee recipient hasn’t changed *Submitted by pants*
- [N-06] frontrun swapByQuote or abuse high allowance - replacement *Submitted by gpersoon*

Gas Optimizations (20)

- [G-01] Unnecessary `SLOAD` in `Swap.setSwapFee()` *Submitted by pants*
- [G-02] Unnecessary conditions causing Over Gas consumption *Submitted by csanuragjain*
- [G-03] Gas: Math library could be “unchecked” *Submitted by cmichel, also found by WatchPug*
- [G-04] Gas: `SafeMath` is not needed when using Solidity version 0.8 *Submitted by cmichel, also found by WatchPug, 0xngndev, defsec, and pants*
- [G-05] Gas: `minReceived` check can be simplified *Submitted by cmichel, also found by TomFrench*
- [G-06] Unnecessary array boundaries check when loading an array element twice *Submitted by pants, also found by yeOlde*
- [G-07] Cache or use existing memory versions of state variables (`feeRecipient` , `swapFee`) *Submitted by yeOlde*
- [G-08] Gas Optimization: Reduce the size of error messages. *Submitted by 0xngndev, also found by yeOlde*
- [G-09] Check if `boughtETHAmount > 0` can save gas *Submitted by WatchPug, also found by pauliax*
- [G-10] Use of uint8 for counter in for loop increases gas costs *Submitted by TomFrench, also found by pauliax*
- [G-11] Remove duplicate reads of storage variables *Submitted by harleythedog, also found by pants*
- [G-12] Unnecessary `CALLDATALOAD` s in for-each loops *Submitted by pants, also found by WatchPug*
- [G-13] Unnecessary checked arithmetic in for loops *Submitted by pants*
- [G-14] Prefix increaments are cheaper than postfix increaments *Submitted by pants*
- [G-15] `internal` functions can be `private` *Submitted by pants*
- [G-16] Unnecessary require statement in `Swap` 's constructor *Submitted by pants*

- [G-17] Unnecessary `SLOAD`s in `EmergencyGovernable.onlyTimelockOrEmergencyGovernance()` *Submitted by pants*
- [G-18] Emit `feeRecipient__(memory)` instead of `feeRecipient` (storage) *Submitted by harleythedog, also found by pants*
- [G-19] double reading calldata variable inside a loop *Submitted by pants*
- [G-20] Upgrade pragma to at least 0.8.4 *Submitted by defsec*

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