Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.
Learn more →

PoolTogether v4 contest Findings & Analysis Report

2021-11-05

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of PoolTogether v4 contest smart contract system written in Solidity. The code contest took place between October 7—October 13 2021.

Wardens

9 Wardens contributed reports to the PoolTogether v4 contest code contest:

- WatchPug
- cmichel
- leastwood
- gpersoon
- pauliax
- pants
- yeOlde
- AC0611

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities and 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, 2 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 8 non-critical recommendations and 19 gas optimizations.

Scope

The code under review can be found within the C4 PoolTogether v4 contest repository and is composed of 20 smart contracts written in the Solidity programming language and includes 2154 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] The formula of number of prizes for a degree is wrong

Submitted by WatchPug, also found by cmichel.

The formula of the number of prizes for a degree per the document: https://v4.docs.pooltogether.com/protocol/concepts/prize-distribution/#splitting-the-prizes is:

```
Number of prizes for a degree = (2^bit range)^degree - (2^bit ra
```

Should be changed to:

```
Number of prizes for a degree = (2^bit range)^degree - (2^bit ra
```

or

```
Number of prizes for a degree = 2^(bit range * degree) - 2^(bit
```

Impact

Per the document:

> prize for a degree = total prize * degree percentage / number of prizes for a degree

Due to the miscalculation of `number of prizes for a degree`, it will be smaller than expected, as a result, `prize for a degree` will be larger than expected. Making the protocol giving out more prizes than designed.

Proof

> We will use `f(bitRange, degree)` to represent `numberOfPrizesForDegree(bitRangeSize, degree)`.

Proof: (method 1)

```
2 ^ {bitRange \times n} = f(bitRange, n) + f(bitRange, n-1) + f
f(bitRange, n) = 2 ^ {bitRange \times n} - ( f(bitRange, n-1) +
f(bitRange, n) = 2 ^ {bitRange \times n} - f(bitRange, n-1) - (
Because:

2 ^ {bitRange \times (n-1)} = f(bitRange, n-1) + f(bitRange, n-2
2 ^ {bitRange \times (n-1)} - f(bitRange, n-1) = f(bitRange, n-2

Therefore:

f(bitRange, n) = 2 ^ {bitRange \times n} - f(bitRange, n-1) - (
f(bitRange, n) = 2 ^ {bitRange \times n} - f(bitRange, n-1) - 2
f(bitRange, n) = 2 ^ {bitRange \times n} - 2 ^ {bitRange \times
```

Because `2^x = 1 << x`

Therefore, when `n > 0`:

```
f(bitRange, n) = ( 1 << bitRange * n ) - ( 1 << bitRange * (n -
```

QED.

Proof: (method 2)

By definition, `degree n` is constructed by 3 chunks:

- The first N numbers, must equal the matching numbers. Number of possible values: `1`;
- The N-th number, must not equal the N-th matching number. Number of possible values: `2^bitRange - 1`
- From N (not include) until the end. Number of possible values: `2 ^ (bitRange * (n-1))`

Therefore, total `numberOfPrizesForDegree` will be:

```
f(bitRange, n) = (2 ^ {bitRange} - 1) \times 2 ^ {bitRange \time
f(bitRange, n) = 2 ^ {bitRange} \times 2 ^ {bitRange \times (n -
f(bitRange, n) = 2 ^ {bitRange + bitRange \times (n - 1)} - 2 ^
f(bitRange, n) = 2 ^ {bitRange + bitRange \times n - bitRange} -
```

```
f(bitRange, n) = 2 ^ {bitRange \times n} - 2 ^ {bitRange \times
```

QED.

Recommendation

https://github.com/pooltogether/v4-core/blob/055335bf9b09e3f4bbe11a788710dd04d827bf37/contracts/DrawCalculator.sol#L423-L431

```
/**
    * @notice Calculates the number of prizes for a given prizeI
    * @param _bitRangeSize Bit range size for Draw
    * @param _prizeTierIndex Index of the prize tier array to ca
    * @return returns the fraction of the total prize (base 1e1ε
    */
function _numberOfPrizesForIndex(uint8 _bitRangeSize, uint256 _p
    internal
    pure
    returns (uint256)
{
    uint256 bitRangeDecimal = 2**uint256(_bitRangeSize);
    uint256 numberOfPrizesForIndex = bitRangeDecimal**_prizeTier

    while (_prizeTierIndex > 0) {
        numberOfPrizesForIndex -= bitRangeDecimal**(_prizeTierIr
        _prizeTierIndex--;
    }

    return numberOfPrizesForIndex;
}
```

L423-431 should change to:

```
if (_prizeTierIndex > 0) {
    return ( 1 << _bitRangeSize * _prizeTierIndex ) - ( 1 << _bi
} else {
    return 1;
}
```

BTW, the comment on L416 is wrong:

- seems like it's copied from `\_calculatePrizeTierFraction()`
- plus, it's not base 1e18 but base 1e9

PierrickGT (PoolTogether) confirmed and patched:

> PR: https://github.com/pooltogether/v4-core/pull/242

Alex the Entreprenerd (judge) commented:

> The warden found the usage of an incorrect formula that would cause the protocol to give out larger prizes than expected, the sponsor has mitigated in a following PR

[H-02] Miners Can Re-Roll the VRF Output to Game the Protocol

Submitted by leastwood.

Impact

Miners are able to rewrite a chain's history if they dislike the VRF output used by the protocol. Consider the following example:

- A miner or well-funded user is participating in the PoolTogether protocol.
- A VRF request is made and fulfilled in the same block.
- The protocol participant does not benefit from the VRF output and therefore wants to increase their chances of winning by including the output in another block, producing an entirely new VRF output. This is done by re-orging the chain, i.e. following a new canonical chain where the VRF output has not been included in a block.
- This attack can be continued as long as the attacker controls 51% of the network. The miner itself could control a much smaller proportion of the network and still be able to mine a few blocks in succession, although this is of low probability but entirely possible.

- A well-funded user could also pay miners to re-org the chain on their behalf in the form of MEV to achieve the same benefit.

The PoolTogether team is aware of this issue but is yet to mitigate this attack vector fully.

Proof of Concept

- https://docs.chain.link/docs/vrf-security-considerations/#choose-a-safe-block-confirmation-time-which-will-vary-between-blockchains
- https://github.com/pooltogether/pooltogether-rng-contracts/blob/master/contracts/RNGChainlink.sol
- https://github.com/pooltogether/v4-core/blob/master/contracts/DrawBeacon.sol#L311-L324
- https://github.com/pooltogether/v4-core/blob/master/contracts/DrawBeacon.sol#L218-L232
- https://github.com/pooltogether/blockhash-analysis-simulation

Tools Used

- Manual code review
- Discussions with Brendan

Recommended Mitigation Steps

Consider adding a confirmation time between when the actual VRF request was made and when it was later fulfilled on-chain. This could be as few as 5 blocks, reducing the probability of an effective chain reorganization to extremely close to 0.

asselstine (PoolTogether) acknowledged:

> Yes, this is something I've known for awhile. The VRF operator whose signature we are requesting *could* collude with a miner to manipulate the blockhash that is being fed to the VRF.

> We're using the original VRF implementation by Chainlink. VRF 2.0 is rolling out soon, and we'll explore confirmation times with their team.

Alex the Entreprenerd (judge) commented:

> The warden has identified a “supply chain” attack on the VRF, while the finding may seem innocuous, it does pose the fundamental question as to whether Chainlink’s VRF is a source of truly verifiable randomness that won’t be gamed for personal gains.

> The sponsor does agree on the attack vector, the miner and the chainlink provider could collude with the purpose of gaming the system.

> I find it hard to leave this as a High Severity finding, because it implies that Chainlink’s VRF Service is flawed in it’s design

> However, if trace of proof of collusion between miners and chainlink operators where to be found, this would question the impartiality of their service.

Alex the Entreprenerd (judge) commented:

> After a day of thinking I’ve decided to leave the finding as high risk. Not as a statement to Chainlink’s security model, as that’s outside of scope, but because the finding is objectively true.

> A malicious operator can, in conjunction with a miner, re-roll the VRF to pursue their own gains.

> I’m not sure what could be done to mitigate this at the chain level, it seems to me that at this time, there may be no source of true randomness that can be achieved on-chain without assuming a degree of counter-party risk

Medium Risk Findings (2)

[M-01] Deposits don’t work with fee-on transfer tokens

Submitted by cmichel.

There are ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()` . Others are rebasing tokens that increase in value over time like Aave’s aTokens (`balanceOf` changes over time).

Impact

The `PrizePool._depositTo()` function will try to supply more `_amount` than was actually transferred. The tx will revert and these tokens cannot be used.

Recommended Mitigation Steps

One possible mitigation is to measure the asset change right before and after the asset-transferring routines

[asselstine (PoolTogether) acknowledged]():

> We don’t plan on incorporating fee-on-transfer tokens, so I think we can safely ignore this.

[Alex the Entreprenerd (judge) commented]():

> The sponsor acknowledges the finding, simple mitigation is to not use `feeOnTransfer` tokens in the protocol

[M-02] `PrizePool.awardExternalERC721()` Erroneously Emits Events

Submitted by leastwood.

Impact

The `awardExternalERC721()` function uses solidity’s try and catch statement to ensure a single tokenId cannot deny function execution. If the try statement fails, an `ErrorAwardingExternalERC721` event is emitted with the relevant error, however, the failed tokenId is not removed from the list of tokenIds emitted at the end of function execution. As a result, the `AwardedExternalERC721` is emitted with the entire list of tokenIds, regardless of failure. An off-chain script or user could therefore be tricked into thinking an ERC721 tokenId was successfully awarded.

Proof of Concept

https://github.com/pooltogether/v4-core/blob/master/contracts/prize-pool/PrizePool.sol#L250-L270

Tools Used

Manual code review

Recommended Mitigation Steps

Consider emitting only successfully transferred tokenIds in the `AwardedExternalERC721` event.

PierrickGT (PoolTogether) confirmed and patched:

> PR: https://github.com/pooltogether/v4-core/pull/246

Alex the Entreprenerd (judge) commented:

> The sponsor acknowledged and mitigated by actively managing a list of `_awardedTokenIds` to keep track of the tokens that didn't go through the `catch` part of the error hadling

Low Risk Findings (10)

- [L-01] `PrizeSplit.sol#distribute()` The value of the event parameter is wrong *Submitted by WatchPug, also found by leastwood and cmichel.*
- [L-02] Usage of deprecated `safeApprove` *Submitted by cmichel.*
- [L-03] Should `safeApprove(0)` first *Submitted by cmichel.*
- [L-04] Reserve does not correctly implement RingBuffer *Submitted by cmichel.*
- [L-05] `PrizePool` uses `ERC20` for `ERC721` *Submitted by cmichel.*
- [L-06] PrizeSplit uint8 limits *Submitted by gpersoon.*
- [L-07] Inaccurate Revert Message *Submitted by leastwood.*
- [L-08] calculateNextBeaconPeriodStartTime casts timestamp to uint64 *Submitted by pauliax.*
- [L-09] staticcall may return true for an invalid _yieldSource *Submitted by pauliax.*

Non-Critical Findings (8)

- [N-01] `YieldSourcePrizePool._canAwardExternal()` Does Not Prevent the Deposit Token From Being Withdrawn *Submitted by leastwood.*
- [N-02] Unbounded iteration over picks when `claim` ing draws *Submitted by cmichel.*
- [N-03] Lack of Pause Mechanism *Submitted by leastwood.*
- [N-04] PrizePoolHarness._supply -a return without specifying return value *Submitted by pants.*
- [N-05] Unnecessary imports *Submitted by pauliax.*
- [N-06] Wrong comment regarding decimal precision of `_calculatePrizeTierFraction` *Submitted by cmichel.*
- [N-07] Comment Typos *Submitted by leastwood.*
- [N-08] Style issues *Submitted by pauliax.*

Gas Optimizations (19)

- [G-01] `PrizePool.sol#setTicket()` Remove unnecessary variable can make the code simpler and save some gas *Submitted by WatchPug.*
- [G-02] `PrizePool.sol#_canDeposit()` Remove redundant code can make the code simpler and save some gas *Submitted by WatchPug.*
- [G-03] Adding unchecked directive can save gas *Submitted by WatchPug.*
- [G-04] Immutable variables *Submitted by pauliax, also found by WatchPug and pants.*
- [G-05] `PrizeSplit.sol#_totalPrizeSplitPercentageAmount()` Avoid unnecessary copy from storage to memory can save gas *Submitted by WatchPug.*
- [G-06] `PrizeDistributor.sol#claim()` Remove redundant check can save gas *Submitted by WatchPug, also found by yeOlde.*
- [G-07] Gas: Default case of `_calculateTierIndex` can return `0` *Submitted by cmichel.*
- [G-08] Gas: Bitmasks creation can be simplified *Submitted by cmichel.*

- [G-09] Gas: `PrizePool.captureAwardBalance` computation can be simplified *Submitted by cmichel.*
- [G-10] Gas improvement _transferTwab *Submitted by gpersoon.*
- [G-11] No need to put ReentrnacyGaurd on PrizePool.constructor. *Submitted by pants.*
- [G-12] double reading from memory inside a for loop. *Submitted by pants.*
- [G-13] ++i is more gas efficient than i++ for loops. *Submitted by pants, also found by yeOlde.*
- [G-14] function _getPrizeSplitAmount can be refactored *Submitted by pauliax.*
- [G-15] Less than 256 uints are not efficient *Submitted by pauliax.*
- [G-16] unchecked arithmetics *Submitted by pauliax.*
- [G-17] Unnecessary Multiple Return Statements (PrizePool.sol) *Submitted by yeOlde.*
- [G-18] Unnecessary Addition In Loop (PrizeDistributionBuffer.sol) *Submitted by yeOlde.*
- [G-19] Unnecessary decrement (DrawCalculator.sol) *Submitted by yeOlde.*

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