PuppyRaffle Audit Report

Version 1.0

0xSnowEth

January 12, 2025

PuppyRaffle Audit Report

Snow X

January 12, 2025

Prepared by: 0xSnowEth

Lead Auditors: - Snow X

Table of Contents

- Medium
  - [M-1] Nested For-loop in `PuppyRaffle::enterRaffle` function, Enables Denial of service attacks.
  - [M-2] Smart Contract wallet Winners without a `recieve` or `fallback` function will block the start of a new contest
  - [M-3] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
  - [M-4] Unsafe cast of `PuppyRaffle::fee` loses fees
- Low
  - [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 that they have not entered the raffle.
- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached.
- Informational/Non-Crits
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Using an outdated version of solidity is not recommended.
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4]: `PuppyRaffle::selectWinner` should follow CEI.
  - [I-5]: Use of "Magic" Numbers is discouraged
  - [I-6] _isActivePlayer is never used and should be removed
  - [I-7] Potentially erroneous active player index
  - [I-8] Zero address may be erroneously considered an active player

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

Risk Classification

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

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

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

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 codebase was pretty fun to audit! It's a fun raffle system where users can enter to win an adorable dog NFT. The design is straightforward: no duplicate entries, refunds are allowed, and a winner is picked at regular intervals to mint their puppy.

Issues found

Severity	Number of issues found
High	4
Medium	4
Low	1
Info	8
Gas	2
Total	19

Findings

High

[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 `PuppyRaffle::players` array.

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


 @>       payable(msg.sender).sendValue(entranceFee);
 @>       players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }

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

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

Impact: All fees paid by raffle entrants could be stolen by malicious participant.

Proof of Concept:

1. Users enter the raffle.
2. Attacker sets up a contract with a `Fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof Of Code:

Code

Place the following to `PuppyRaffleTest.t.sol`

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

```
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           RenntrancyAttack attackerContract = new RenntrancyAttack(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          //attackerContract
17
18          vm.prank(attackUser);
19          attackerContract.attack{value: entranceFee};
20
21          console.log("starting attack contract balance:",
                startingAttackContractBalance);
22          console.log("Starting contract balance :",
                startingContractBalance);
23
24          console.log("Ending Attacker contract balance:", address(
                attackerContract).balance);
25          console.log("Ending contract balance :", address(puppyRaffle).
                balance);
26
27      }
28  }
```

And this contract as well:

```
1   contract RenntrancyAttack {
2       PuppyRaffle puppyRaffle;
3       uint256 entranceFee;
4       uint256 attackerIndex;
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = _puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18
```

```
19      }
```

Recommended Mitigation: To prevent this we should have the `PuppyRaffle:refund` function update the `players` array before making the external call, Additionally, we should move the event emission up as well.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  +        players[playerIndex] = address(0);
7  +        emit RaffleRefunded(playerAddress);
8          payable(msg.sender).sendValue(entranceFee);
9  -        players[playerIndex] = address(0);
10 -        emit RaffleRefunded(playerAddress);
11     }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinenr` function allows users to influence or predict the winner and influence or predict the winnnig puppy

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

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

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

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. see the Solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can mind/maipulate their `msg.sender` 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.

4. using on-chain values as a 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. (https://chain.link/vrf)

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

Description: In solidity prior to `0.8.0` integers were subject to integer overflow

```
1
2 uint64 myVar = type(uint64).max
3 // 18446744073709551615
4 myVar = myVar + 1
5 //myVar will  be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withDrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude the raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // aka
3 totalFees = 800000000000000000 + 17800000000000000000
4 // and this will overflow!
5 totalFees = 153255926290448384
```

4. You will not be able to withdraw, due to a line in `PuppyRaffle::withdrawFees`:

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

Allthought you could use `SelfDestruct` to send ETH to this contract in orders for the values to match and withdraw the fees this is clearly not the inteneded design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

```
1
2  function testTotalFeesOverflow() public playersEntered {
3          // We finish a raffle of 4 to collect some fees
4          vm.warp(block.timestamp + duration + 1);
5          vm.roll(block.number + 1);
6          puppyRaffle.selectWinner();
7          uint256 startingTotalFees = puppyRaffle.totalFees();
8          // startingTotalFees = 800000000000000000
9
10         // We then have 89 players enter a new raffle
11         uint256 playersNum = 89;
12         address[] memory players = new address[](playersNum);
13         for (uint256 i = 0; i < playersNum; i++) {
14             players[i] = address(i);
15         }
16         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
17         // We end the raffle
18         vm.warp(block.timestamp + duration + 1);
19         vm.roll(block.number + 1);
20
21         // And here is where the issue occurs
22         // We will now have fewer fees even though we just finished a
               second raffle
23         puppyRaffle.selectWinner();
24
25         uint256 endingTotalFees = puppyRaffle.totalFees();
26         console.log("ending total fees", endingTotalFees);
27         assert(endingTotalFees < startingTotalFees);
28
29         // We are also unable to withdraw any fees because of the
               require check
30         vm.prank(puppyRaffle.feeAddress());
31         vm.expectRevert("PuppyRaffle: There are currently players
               active!");
32         puppyRaffle.withdrawFees();
33     }
```

Recommended Mitigation: There are a few possible mitigations:

1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `safeMath` library of openZepplin for version 0.7.6 of solidity, however, you will still have a hard time with the `uint64` if too many fees are collected.
3. Remove the balance check from the `PuppyRaffle::withDrawFees`

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

There are more attack vectors with that final require, so we recommend that you remove it regardless.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

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

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Medium

[M-1] Nested For-loop in `PuppyRaffle::enterRaffle` function, Enables Denial of service attacks.

Description: : the `PuppyRaffle::enterRaffle` function uses a Nested for-loop to check for Duplicate players, which creates a quadratic time complexity, in other words As the number of players increases the gas costs grows exponentially. the Vulnerable code section:

```
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");
            }
        }
```

This implementation means that each new player additon requires checking against all existing players, making the function increasingly expensive and potentially impossible to execute as the raffle grows.

Impact: Function becomes prohibitely expensive to call as player count increases. Could completely block new players from entering once array size reaches a certain length.
Function could exceed block gas limits, making the raffle effectively unusable.
Contract functionality becomes paralyzed, breaking core raffle mechanics.

Proof of Concept:

Initial state: Raffle has 100 players Attacker calls enterRaffle() with 50 new addresses Gas costs:

First loop iteration: 100 checks Second iteration: 99 checks Total comparisons: (n * (n-1)) / 2

As players[] grows, gas costs become exponentially higher When players[] reaches ~750 addresses, the function exceeds block gas limits

Recommended Mitigation: Replace the nested loop with a more efficient duplicate checking mechanism:

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

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++) {
        require(!playerExist[newPlayers[i]], "PuppyRaffle: Duplicate
           player");
        playerExist[newPlayers[i]] = true;
        players.push(newPlayers[i]);
    }

    emit RaffleEnter(newPlayers);
}
```

Alternative approaches:

Implement a maximum array size limit Use a mapping-based system instead of an array Consider breaking large player additions into smaller batches

This example shows how to structure a finding to clearly communicate the vulnerability, its impact, and solution. The title combines both the root cause (nested loop in duplicate check) and the impact (DoS vulnerability). Each section provides specific, relevant information that helps understand and address the security issue.

[M-2] Smart Contract wallet Winners without a `recieve` 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 payments, the lottery would not be able to restart.

Users could eaisly call the `selectWinner` function again and non-wallet entrants could enter, but it could cost alot 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 get their money!

Proof of Concept:

1. 10 smart contracts enter the lottery without a fallback or recieve function.
2. The lottery ends.
3. the `SelectWinner` function would'nt work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (is not recommended)
2. create a mapping of addresses -> payout amounts so winners can pull their funds themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over push

[M-3] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`

3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

[M-4] Unsafe cast of `PuppyRaffle::fee` loses fees

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

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits

3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

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

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

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

Low

[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 that they have not entered the raffle.

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

```
/// @return the index of the player in the array, if they are not active, it returns 0
```

```
2  function getActivePlayerIndex(address player) external view returns (
       uint256) {
3          for (uint256 i = 0; i < players.length; i++) {
4              if (players[i] == player) {
5                  return i;
6              }
7          }
8          return 0;
9      }
```

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

Proof of Concept:

1. Users enter the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered the raffle correctly due to the 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 funciton returns -1 if the player is not active.

Gas

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

Instances: `PuppyRaffle::raffleDuration` Should be immutable. `PuppyRaffle::commonImageUri` Should be constant. `PuppyRaffle::rareImageUri` Should be constant. `PuppyRaffle::legendaryImageUri` Should be constant.

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

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

Everytime you call players.length you read from storage, as opposed to memeory which would be more Gas-efficient.

```
1  +       uint256 playersLength = players.length;
2  -       for (uint256 i = 0; i < players.length - 1; i++) {
3  +       for (uint256 i = 0; i < players.length - 1; i++) {
4  -           for (uint256 j = i + 1; j < players.length; j++) {
```

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

Informational/Non-Crits

[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 older version prevents access to new Solidity security features. we also recommend avoiding complex pragma statements.

Recommendation: Deploy with any of the following solidity versions: `0.8.19` This Recommendation takes into account: - Risks related to recent releases. - Risks of compelx code generation changes. - Risks of new language features. - Risks of known bugs - Use a simple pragma versions 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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 70

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 203

```
        feeAddress = newFeeAddress;
```

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

It's best to keep the code clean and follow CEI (Checks, Effects, Interaction).

```
1
2 -         (bool success,) = winner.call{value: prizePool}("");
3 -         require(success, "PuppyRaffle: Failed to send prize pool to
      winner");
4           _safeMint(winner, tokenId);
5 +         (bool success,) = winner.call{value: prizePool}("");
6 +         require(success, "PuppyRaffle: Failed to send prize pool to
      winner");
7       }
```

[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:

```
1
2 uint256 prizePool = (totalAmountCollected * 80) / 100;
3         uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

```
1 uint256 public constant PRICE_POOL_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
3 uint256 public constant POOL_PRECISION = 100;
```

[I-6] _isActivePlayer is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }
```

[I-7] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.