Protocol Audit Report

Version 1.0

xByteNeo

August 25, 2025

Puppy Raffle Audit Report

xByteNeo

August 25, 2025

Prepared by: xByteNeo Lead Security Researcher: - xByteNeo: @xByteNeo

Table of Contents

- Medium
  * [M-1] Looping thorough players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial denial of service (DoS) attack, incrementing gas costs for future entrants
  * [M-2] Balance validation in `PuppyRaffle::withdrawFees` enables denial of service through forced ETH transfers
- Informational
  * [I-1] Floating pragmas
  * [I-2] Magic Numbers

Protocol Summary

The PasswordStore protocol is a simple smart contract that allows a user to store and later retrieve a password. The contract is designed so that only the owner (the deployer) should be able to set and get the password. The password is stored as a private string variable within the contract.

Disclaimer

The xByteNeo 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

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

- Solc version: ^0.7.6
- Chain(s) to deploy to; Ethereum

Roles

- Owner: Deployer of the protocol
  (a) can change the wallet address to which fees are sent through the changeFeeAddress function.
- Player: Participant of the raffle
  (a) can call enterRaffle function
  (b) can refund value through refund function.

Executive Summary

The PuppyRaffle contract audit revealed multiple critical security vulnerabilities that pose significant risks to user funds and protocol integrity. The contract suffers from fundamental issues including reentrancy attacks, lack of input validation, predictable randomness, arithmetic overflow/underflow vulnerabilities, and denial of service attack vectors. These vulnerabilities could lead to complete drainage of contract funds, manipulation of raffle outcomes, and prevention of normal protocol operations. We strongly recommend addressing all identified issues before deploying to mainnet, with particular focus on the high-severity vulnerabilities that directly threaten user funds.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Informational	2
Total	9

During the audit, a total of 9 issues were identified: - 4 High severity issues: lack of input validation allowing fund loss, reentrancy vulnerability enabling fund drainage, predictable randomness compromising raffle fairness, and arithmetic overflow/underflow risks. - 3 Medium severity issues: denial of service attack through gas cost manipulation, forced ETH transfer DoS vulnerability, and unsafe type casting in fee calculations. - 2 Informational issues: floating pragma versions and magic numbers usage.

The High severity issues pose immediate threats to user funds and protocol integrity, with the reentrancy vulnerability being particularly critical as it could lead to complete contract drainage. The Medium severity issues create operational risks and potential DoS scenarios that could disrupt normal protocol functionality.

Findings

High

[H-1] No validation for zero length players

Description: The `PuppyRaffle::enterRaffle` function does not check if the `newPlayers` array is empty. This allows users to send ether and not receive a position in the raffle.

```
 function enterRaffle(address[] memory newPlayers) public payable {
//@>    @audit No validation for zero length players
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
```

Impact: Users can send ethers and not receive a position in the raffle. Causing a loss of funds.

Proof of Concept: Include the test in the `PuppyRaffleTest.t.sol` file and run it with `forge test --match-test test_zeroLengthPlayers` to check the impact fisibility.

PoC

```
    function test_zeroLengthPlayers() public {
        address[] memory players = new address[](1);
        uint256 contractBalanceBefore = address(puppyRaffle).balance;
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        uint256 contractBalanceAfter = address(puppyRaffle).balance;
        assert(contractBalanceAfter > contractBalanceBefore);
    }
```

Recommended Mitigation: Add a check to the `PuppyRaffle::enterRaffle` function to check if the `newPlayers` array is empty.

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0, "PuppyRaffle: Players array
    cannot be empty");
        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]);
        }
```

[H-2] Reentrancy attack on `PuppyRaffle::refund` allowing drain of funds

Description: The `PuppyRaffle::refund` function updates the players array after making an external call. This creates a significant window for reentrancy attacks, where an attacker can call the `PuppyRaffle::refund` function multiple times and steal the contract's funds.

```
    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");

        // @audit the player is refunded before the players array is
            updated opening the reentrancy window
        payable(msg.sender).sendValue(entranceFee);

        // @audit the players array is updated after external
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: Attacker can call the `PuppyRaffle::refund` function multiple times, and steel the contract funds.

Proof of Concept: Include the test in the `PuppyRaffleTest.t.sol` file and run it with `forge test --mt test_reentrancyRefund -vvv` to check the impact fisibility.

PoC

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

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

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

        uint256 endingAttackerContractBalance = address(attacker).
            balance;
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance
            ;

        console.log("Starting attacker contract balance: ",
            startingAttackerContractBalance);
        console.log("Starting puppy raffle balance: ",
            startingPuppyRaffleBalance);
        console.log("Ending attacker contract balance: ",
            endingAttackerContractBalance);
        console.log("Ending puppy raffle balance: ",
            endingPuppyRaffleBalance);

    }
```

Recommended Mitigation: To mitigate this vulnerability there are a few options:

1. move the storage update on players array prior to the external call

```
    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);
-       payable(msg.sender).sendValue(entranceFee);

+       payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

2. Use a openzeppelin ReentrancyGuard library to prevent reentrancy attacks.

```
+    function refund(uint256 playerIndex) public reentrancyGuard {
-    function refund(uint256 playerIndex) public {
```

3. Introduce a lock mechanism on the `PuppyRaffle::refund` function.

```
    function refund(uint256 playerIndex) public {

+       if (locked) {
+           revert("PuppyRaffle: Reentrancy detected");
+       }
+       locked = true;

        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);
    }
```

[H-3] `PuppyRaffle::selectWinner` is not random, and can be manipulated by the owner

Description: The logic implemented to determine the winnerIndex used in the `PuppyRaffle::selectWinner` does not guarantee its randomness. The parameters used in the keccak256 execution such as `block.timestamp` and `block.difficulty` could be manipulated by miners, and since `msg.sender` is part of the hash, the caller can predict the outcome before submitting the transaction.

Impact: The winner of the raffle will could be manipulated by an attacker severelly breaking the fairness of the raffle.

Proof of Concept: N/A

Recommended Mitigation: Consider using a more secure randomness source, such as Chainlink VRF

[H-4] `PuppyRaffle::selectWinner` totalFee underflow/overflow issue in calculations

Description: The `PuppyRaffle::selectWinner` function calculates `totalFee` without checking for potential arithmetic underflow/overflow conditions. The calculation `players.length * entranceFee` could overflow if there are many players or a high entrance fee, while the subsequent fee calculation `totalAmountCollected * fee / 100` could underflow if the parameters are not properly bounded. This is particularly concerning since Solidity versions prior to 0.8.0 don't have built-in overflow protection.

Impact: If an overflow/underflow occurs: The incorrect fee calculations would lead to wrong amounts being transferred and stored, causing the protocol fee mechanism to malfunction. This would result in permanent accounting errors in the contract and incorrect fee distributions to the owner. Such critical vulnerabilities put both the protocol and its users at risk of direct financial losses.

Proof of Concept: The following example shows how casting the maximum uint256 value results in data loss and incorrect fee calculations:

Show Proof of Concept

```
$ chisel
Welcome to Chisel! Type `!help` to show available commands.
$ chisel

$ uint256 fee = type(uint256).max
$ fee

| Hex: 0
  xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

```
| Hex (full word): 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
| Decimal: 1157920892373161954235709850086879078532699846656405640394...
$ uint64 cast64 = uint64(fee)
$ cast64

| Hex: 0x
| Hex (full word): 0x000000000000000000000000000000000000000000000000ffffffffffffffff
| Decimal: 18446744073709551615
```

Recommended Mitigation: To prevent overflow/underflow issues, upgrade to Solidity version 0.8.0 or higher which includes built-in overflow protection. For additional safety, use uint256 for all numeric values and add explicit checks before performing calculations.

Here how it could be mitigated using uint256:

```
-   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;
```

Medium

[M-1] Looping thorough players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRafflee` 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 starts 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.

```
// @audit DoS Attack
@>  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 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.

And attacker input make the `PuppyRaffle::entrants` array so big, taht no one ese enters, guarenteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players: ~6503275 - 2st 100 players: ~18995515 This is more than 3 times more expensive for the same 100 players.

PoC

```
    function test_denialOfService() public {
        vm.txGasPrice(1);

        // Let's enter 100 players
        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 * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
            );

        // now for the 2nd 100 players
        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 * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the 2nd 100 players: ", gasUsedSecond)
            ;

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same perso nfrom entering multiple times, only the same wallet address.
2. Consider using a mappping to check for duplicates. This would be more gas efficient, and would not require a loop.

```
+     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 < newPlayers.length; i++) {
+            require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+        }
+        // Check for duplicates only from the new players
+        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 OpenZeppelin's `EnumerableSet` library.

[M-2] Balance validation in `PuppyRaffle::withdrawFees` enables denial of service through forced ETH transfers

Description: The `PuppyRaffle::withdrawFees` function contains a strict validation that forces the contract's ETH balance to be exactly equal to `totalFees`. Although the contract lacks explicit payable functions to receive ETH, attackers can still manipulate the contract's balance by using `selfdestruct()` to forcefully send ETH, which circumvents this balance check and causes the validation to fail.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would cause a DoS prevening the feeAddress from withdrawing the fees.

Proof of Concept: Consider the steps bellow: - PuppyRaffle has 1 ETH in it's balance, and 1 ETH totalFees. - Attacher sends 1 wei(or any amount) via a selfdestruct - feeAddress is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the PuppyRaffle::withdrawFees function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Informational

[I-1] Floating pragmas

Description: Using floating pragma (`^0.7.6`) is dangerous. The contract can be compiled with any 0.7.x version, which may have different behaviors and security fixes. This could lead to untested or vulnerable code in production. Lock the pragma to a specific version.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

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

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

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