# Password Store Audit

Version 1.0

*Dan Lipert*

October 23, 2023

# Codehawks First Flight Password Storage Audit

Dan Lipert

October 23, 2023

Prepared by: Dan Lipert

Lead Auditors:

- Dan Lipert

## Table of Contents

## Protocol Summary

The PasswordStore contract is a simplified implementation of a password storage system, which allows a user to set and retrieve a password. This password is stored on-chain, can be changed, and can be retrieved by the owner of the contract.

## Disclaimer

Dan Lipert 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

### Scope

The scope of this audit is constrained to the PasswordStore.sol smart contract. Any additional systems, applications, or contracts are not considered as part of the implementation of this protocol.

### Severity Criteria

### Summary of Findings

In summary, this protocol is not suitable for the intended use case of storing a password securely on-chain. In general, storing private information on-chain is ill-advised due to the numerous ways attackers can access the data being stored either on-chain or in-flight. It is possible that some additional out-of-scope elements may partially solve the conceptual problems of this system, but there are logic errors in this contract itself that additionally need to be solved.

### Tools Used

The smart contract was examined and tested using Paradigm's Foundry.

## High

### The Password Can Be Set By an Attacker

#### Context: PasswordStore.sol (L25-29)

#### Description: The setPassword function has no access control

```
function setPassword(string memory newPassword) external {
    s_password = newPassword;
    emit SetNetPassword();
}
```

Here, any actor can call this external function, and there is no check to see if the caller should have access to set the password.

This function needs an additional check at the beginning of the function to ensure that the caller is the owner of the contract, either through custom logic or the common onlyOwner/Ownable.sol Openzeppelin modifier.

Here is a test case PoC demonstrating the vulnerability:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.18;

import {Test, console} from "forge-std/Test.sol";
import {PasswordStore} from "../src/PasswordStore.sol";
import {DeployPasswordStore} from "../script/DeployPasswordStore.s.sol"
    ;


contract PasswordStoreTest is Test {
    PasswordStore public passwordStore;
    DeployPasswordStore public deployer;
    address public owner;

    function setUp() public {
    deployer = new DeployPasswordStore();
    passwordStore = deployer.run();
    owner = msg.sender;
    }

function test_attacker_can_set_password() public {
    // owner sets password
    vm.startPrank(owner);
    string memory expectedPassword = "noob";
    passwordStore.setPassword(expectedPassword);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
    // attacker (not owner) sets new password
    vm.startPrank(address(1));
    string memory attackPassword = "owned";
    passwordStore.setPassword(attackPassword);
    // owner retrieves password, which is not the expected password
    vm.startPrank(owner);
    string memory latestPassword = passwordStore.getPassword();
    assertNotEq(latestPassword, expectedPassword);
    assertEq(latestPassword, attackPassword);
    }
}
```

### The Password Can Be Read After Setting via Contract Storage

**Context: Contract Concept**

**Description: The password is stored unencrypted in contract storage, so it can be read by any attacker**

The password is stored in contract storage, and although it is "private", this does not imply that the data is somehow exclusively accessible by the contract owner. While other smart contracts cannot access the password string because it is set to private, the data can be read off the blockchain via the contract's storage slots.

Should this contract want to be used securely, the password should be encrypted locally by the enduser before being stored in the contract with a public/private keypair.

### The Password Can Be Read Before Setting in the Mempool or by Rublic RPC Nodes

**Context: Contract Concept**

**Description: The password is unencrypted when the setPassword function is called, so it can be read by nodes that see the transaction before it is set, or if the user uses a public RPC endpoint such as infura, flashbots, etc.**

Similarly to the Contract Storage issue, because the password is not encrypted, it can be read by any elements that sit between the enduser and the blockchain, namely ethereum nodes that can see the transaction in the mempool before it is mined, as well as any RPC providers the enduser utilizes such as Infura via Metamask, etc.

### The Password Can Be Read via Third-Party Services

**Context: Contract Concept**

**Description: The password will be visible in third-party services such as block explorers after a transaction with the setPassword function is mined**

Again, because the password is unencrypted, the transaction the enduser submits to set the password will be visible in blockchain explorers and similar services like Etherscan. The transaction bytecode will reveal the set password string to any attacker.

## Medium

No medium severity findings were found.

## Low

No low severity findings were found.

## Informational

No informational findings were found.

## Gas

No gas optimizations were found.