# Protocol Audit Report

Version 1.0

*Cyfrin.io*

October 14, 2024

Prepared by: enigma
Lead Security Researcher: - enigma

## Table of Contents



## Protocol Summary

PasswordStore is a protocol dedicated to the storage and retrieval of a user's passwords. The protocol is designed to be used by a single user and is not intended for multiple users. Only the owner should be able to set and access the password.

## Disclaimer

The YOUR_NAME_HERE team makes every effort to find as many vulnerabilities in the code within the given time period, but holds no responsibility 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 focused solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| Likelihood | High | H | H/M | M |
| | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

## Audit Details

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

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

### Scope

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

### Roles

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

# Executive Summary

*Add some notes about how the audit went, types of things you found, etc.*

*We conducted the audit over the course of X hours, involving a team of Z auditors. Utilizing Y tools, we thoroughly examined the system to ensure compliance with industry standards and best practices, etc.*

## Issues Found

| Severity | Number of Issues Found |
|---|---|
| High | 2 |
| Medium | 0 |
| Low | 0 |
| Info | 1 |
| Total | 3 |

## Findings

## High

### [H-1] Variables Stored in Storage are Visible to Anyone

**Description:**
Variables stored in Solidity, regardless of the visibility keyword, are accessible to anyone. This means that passwords stored on-chain are not truly private. Specifically, the `passwordStore::s_password` variable is intended to be private and should only be accessed through the `passwordStore::getPassword` function, which is designed to be called by the contract owner. However, since all data stored on-chain is publicly accessible, the password can be read directly from the blockchain.

**Impact:**
Anyone can read the private password, severely compromising the protocol's intended functionality.

**Proof of Concept:**
The following demonstrates how anyone can read the password directly from the blockchain:

The below case shows anyone can read the password directly from the blockchain of the protocol.

1. Create a locally running chain:

```
sudo make anvil
```

2. Deploy the contract to the chain:

```
make deploy
```

3. Read the storage slot directly:

```
cast storage 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0 1 --rpc-
    url http://127.0.0.1:8545
```

4. Parse the `bytes32` string:

```
cast parse-bytes32-string 0
    x656e69676d6100000000000000000000000000000000000000000000000000c
```

5. Our output then becomes:

```
enigma
```

**Recommended Mitigation:**
Given this vulnerability, the overall contract architecture should be reconsidered. A more secure approach would be to encrypt the password off-chain and store only the encrypted version on-chain. This method would require users to keep a separate decryption key off-chain to access the original password. Additionally, it is advisable to remove any view functions that expose sensitive data, as there is a risk of users inadvertently sending their decryption key in a transaction, which could compromise security.

### [H-2] `passwordStore::setPassword` has No Access Controls, Meaning a Non-Owner Could Change the Password

**Description:**
The `passwordStore::setPassword` function is intended to be an `external` function; however, the Natspec of the function and the overall purpose of the smart contract indicate that `This function allows only the owner to set a new password.`

```
function setPassword(string memory newPassword) external {
    //@audit - There are no access controls
    s_password = newPassword;
    emit SetNetPassword();
}
```

**Impact:**
Anyone can set/change the password of the contract, severely breaking the contract's intended functionality.

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

Code

```
function test_anyone_can_set_password(address randomAddress) public {
    vm.assume(randomAddress != owner);
    vm.startPrank(randomAddress);
    string memory expectedPassword = "myNewPassword";
    passwordStore.setPassword(expectedPassword);

    vm.startPrank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
}
```

**Recommended Mitigation:**
Add an access control conditional to the `setPassword` function.

```
if (msg.sender != i_owner) {
    revert passwordStore__NotOwner();
}
```

### Low

#### [L-1] Variable Stored in Storage are Visible to Anyone

(Repeat from High findings section)

**Description:**
Variables stored in Solidity, regardless of the visibility keyword, are accessible to anyone. This means that passwords stored on-chain are not truly private. Specifically, the `passwordStore::s_password` variable is intended to be private and should only be accessed through the `passwordStore::getPassword` function, which is designed to be called by the contract owner. However, since all data stored on-chain is publicly accessible, the password can be read directly from the blockchain.

**Impact:**
Anyone can read the private password, severely compromising the protocol's intended functionality.

**Proof of Concept:**
The following demonstrates how anyone can read the password directly from the blockchain:

The below case shows anyone can read the password directly from the blockchain of the protocol. 1. Create a locally running chain: `bash sudo make anvil` 2. Deploy the contract to the chain: `bash make deploy` 3. Read the storage slot directly: `bash cast storage 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0 1 --rpc-url http://127.0.0.1:8545` 4. Parse the `bytes32` string: `bash cast parse-bytes32-string 0x656e69676d610000000000000000000000000000000000000000000000000c` 5. Our output then becomes: `plaintext enigma`

**Recommended Mitigation:**
Given this vulnerability, the overall contract architecture should be reconsidered. A more secure approach would be to encrypt the password off-chain and store only the encrypted version on-chain. This method would require users to keep a separate decryption key off-chain to access the original password. Additionally, it is advisable to remove any view functions that expose sensitive data, as there is a risk of users inadvertently sending their decryption key