PasswordStore Security Review

Prepared by: Vaibhav Sutar

Table of Contents

Disclaimer

The Vaibhav Sutar audit 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

Severity	**Description**
High	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Low	Low impact and low/medium likelihood events where assets are not at risk.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

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.

Audit Details

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

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

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

Roles

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

Executive Summary

Issues found

Severity	Findings
High	2
Medium	0
Low	0
Informational	1
Gas Optimization	0
Total	3

Tools Used

No tools used. It was discovered through manual inspection of the contract.

Findings

High

[H-1] Storing the Password On-Chain Makes it Visible to Anyone

Description: All data stored on-chain in a smart contract is publicly visible and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be a private variable only accessible through the `PasswordStore::getPassword` function, which is intended to be called only by the contract owner. However, due to the nature of blockchain storage, this is not enforced.

Impact: Anyone can read the private password, severely compromising the functionality of the protocol. This undermines the intended security and privacy features.

Proof of Concept:

The following demonstrates how anyone can read the password directly from the blockchain, bypassing the intended access controls. We use Foundry's `cast` tool to read directly from the contract's storage, without requiring ownership.

Steps:

1. **Create a locally running chain:**

   ```
   make anvil
   ```

2. **Deploy the contract to the chain:**

   ```
   make deploy
   ```

3. **Read the storage slot:** Use `cast storage` to read the storage slot where the password is stored. We use `1` because that's the storage slot of `s_password` in the contract (Note: Storage slot can vary. Consult the compiler output to find the slot).

   ```
   cast storage <ADDRESS_HERE> 1 --rpc-url http://127.0.0.1:8545
   ```

   You'll get an output similar to this:

   ```
   0x6d7950617373776f726400000000000000000000000000000000000000000014
   ```

4. **Parse the hex to a string:** Use `cast parse-bytes32-string` to convert the hex representation to a readable string:

   ```
   cast parse-bytes32-string
   0x6d7950617373776f726400000000000000000000000000000000000000000014
   ```

   This will output the password:

   ```
   myPassword
   ```

Recommended Mitigation:

Due to the inherent visibility of on-chain data, the overall architecture of the contract should be rethought. Storing sensitive data like passwords directly on-chain is fundamentally insecure.

Consider these alternative approaches:

- **Off-Chain Encryption:** Encrypt the password off-chain *before* storing it on-chain. This would require the user to remember a separate encryption key (another password or a key derived from their wallet). However, it is essential to remove the `view` function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password.
- **Hashing:** Store a hash of the password instead of the password itself. This prevents direct reading of the password from the blockchain but requires implementing a secure password verification process.

[H-2] `PasswordStore::setPassword` Lacks Access Controls, Allowing Non-Owners to Change the Password

Description: The `PasswordStore::setPassword` function is declared as `external`. However, the NatSpec documentation for the function, and the overall purpose of the smart contract, indicates that *only* the contract owner should be able to set a new password.

```
    function setPassword(string memory newPassword) external {
@>      //@audit any user can set password
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact:

Any user can change the contract's password, severely compromising its intended functionality.

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

▶ Code

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

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

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

```
    if (msg.sender != s_owner) {
       revert PasswordStore_NotOwner();
    }
```

Informational

[I-1] The `PasswordStore::getPassword` NatSpec Indicates a Non-Existent Parameter, Causing Incorrect Documentation

Description:

The NatSpec documentation for the `PasswordStore::getPassword` function includes a `@param newPassword` tag, indicating that the function accepts a parameter named `newPassword`. However, the function signature `getPassword() external view returns (string memory)` shows that it takes no parameters. This discrepancy causes the NatSpec to be incorrect and misleading.

```
     /*
      * @notice This allows only the owner to retrieve the password.
      * @param newPassword The new password to set.
      */
     function getPassword() external view returns (string memory) {
```

The `PasswordStore::getPassWord` function signature is `getPassword()` which the natspec says it should be `getPassword(string)`.

Impact: The incorrect NatSpec can mislead developers using or interacting with the contract, leading to confusion about the function's intended use and potential errors.

Recommended Mitigation: Remove the incorrect `@param newPassword` line from the NatSpec documentation.

```
-* @param newPassword The new password to set.
```

This will ensure that the NatSpec accurately reflects the function's actual signature.
