Protocol Audit Report

Version 1.0

Sagar Rana

May 31, 2025

Protocol Audit Report

Sagar Rana

May 31, 2025

Prepared by: Sagar Rana Lead Auditors: - Sagar Rana

Table of Contents

Protocol Summary

PasswordStore is a protocol dedicated to storing and retrieving user's passwords securely and privately. it is designed to be used by a single user, the person who deployed the contract.

Disclaimer

The 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 to the following commit hash:
Commit Hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

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

Roles

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

Issues found

Severity	Number of issue found
Highs	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] Storing the Password Onchain Makes It Visible to Anyone

Description: All data stored onchain is publicly accessible and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be private and only accessible through the `PasswordStore::getPassword()` function, which is meant to be callable only by the contract owner.

Below, we demonstrate one method to read this private variable directly from the blockchain.

Impact: Anyone can read the supposedly private password, which completely undermines the intended confidentiality and functionality of the contract.

Proof of Concept: The following test case shows how anyone can read the password from the blockchain:

1. Start a local blockchain:

```
make anvil
```

2. Deploy the contract:

```
make deploy
```

   Note the deployed contract's address from the output.

3. Read the storage slot (assuming `s_password` is stored at slot `1`):

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

Note the output bytes.

4. Decode the bytes to a string:

```
cast parse-bytes32-string <HASH_HERE>
```

The result will be the original password used in the deployment script.

Recommended Mitigation: The current architecture needs to be redesigned. One possible solution is to store an **encrypted** version of the password onchain. This way, even if someone accesses the encrypted data, they cannot retrieve the actual password without the decryption key. However, this would require the user to remember a separate decryption password.

[H-2] Lack of Access Control in `PasswordStore::setPassword()` Allows Anyone to Modify the Password

Description: The `PasswordStore::setPassword()` function is marked `external` and does not verify if the caller is the owner. Despite the comment stating, "This function allows only the owner to set a new password," no access control is implemented.

```
function setPassword(string memory newPassword) external {
@>  // @audit Missing access control
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact: Any user can call this function and modify the password, defeating the purpose of restricting password changes to the owner.

Proof of Concept: Add the following test case to confirm that a non-owner can change the password:

```
function test_non_owner_cannot_set_password(address randomAddress)
    public {
    vm.assume(randomAddress != owner && randomAddress != address(0));
    vm.startPrank(randomAddress);
    string memory expectedPassword = "myNewPassword";
    passwordStore.setPassword(expectedPassword);
    vm.startPrank(owner);
    string memory actualPassword = passwordStore.getPassword();
```

```
8         assertEq(actualPassword, expectedPassword);
9     }
```

Then run:

```
1 forge test --mt test_non_owner_cannot_set_password
```

The test passes, indicating that a non-owner successfully modified the password.

Recommended Mitigation: Add an access control check to ensure that only the owner can call this function:

```
1 if (msg.sender != s_owner) {
2     revert PasswordStore__NotOwner();
3 }
```

Informational

[I-1] Incorrect NatSpec in `PasswordStore::getPassword()` Function

Description: The NatSpec for the `getPassword()` function contains a `@param` tag for a nonexistent parameter:

```
1 /*
2      * @notice This allows only the owner to retrieve the password.
3 @>   * @param newPassword The new password to set.
4      */
5 function getPassword() external view returns (string memory) {
```

Since `getPassword()` does not take any parameters, the documentation is misleading and incorrect.

Impact: Incorrect and misleading NatSpec documentation, which may cause confusion for developers and tools that parse it.

Recommended Mitigation: Remove the incorrect `@param` line:

```
1 /*
2      * @notice This allows only the owner to retrieve the password.
3 -    * @param newPassword The new password to set.
4      */
5 function getPassword() external view returns (string memory) {
```