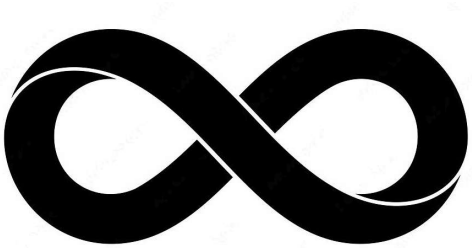

Protocol Audit Report

Version 1.0

Muhammad Kashif

September 19, 2025

Protocol Audit Report

Muhammad Kashif

September 19, 2025

Prepared by: Muhammad Kashif

Security Researcher: Muhammad Kashif

Table of Contents

Protocol Summary

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

Disclaimer

I 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 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

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

Audit Details

Commit Hash

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

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

Issues found

Severity	No of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] Storing the password on-chain makes it visisble to anyone, and no longer private.

Description: All data stored on-chain is visible to anyone and can be read directly from blockchain. The `PasswordStore::s_password` variable is intended to be a private variable and only accessed through the `PasswordStore::getPassword` function, which is intended to be only called by the owner of the contract.

We show one such method of reading any data off chain below.

Impact: Anyone can read the private password, serverly breaking the functionality of the protocol.

Proof of Concept:

1. Create a locally running chain.

```
make anvil
```

2. Deploy the contract to the chain.

```
make deploy
```

3. Run the storage tool. We use 1 because thats the storage slot of `PasswordStore::s_password` in the contract.

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

You will get the output like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

This is hex representation, And if you convert this you will get the password string.

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

You will get the output like this.

```
myPassword
```

Recommended Mitigation: The practice of storing plain-text passwords on-chain creates a severe security risk due to the public nature of the EVM. Sensitive data must never be stored on-chain. To meet the business requirement for decentralization traits of EVM Chain while avoiding a single point of failure, the architecture should be revised to store only a hashed password (using a strong, anti-brute force algorithm) on-chain. It is critical to remember that this shifts the security risk to the management of the decryption key, which must also be handled with a secure, off-chain solution.

[H-2] Missing Access Control - The `PasswordStore::setPassword()` has no access control, Anyone can change the password.

Description: The function `PasswordStore::setPassword(string memory)` is set to be an external, However, the natspec of the function and overall purpose of the smart contract is that `This function allows only the owner to set a new password.`

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

Impact: Anyone can set/change the password of the contract, serverly breaking the integrity of the Smart Contract.

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

Code

```
function test_anyOneCanSetPassword(address randomAddress) public {
        vm.assume(randomAddress != owner);
        vm.prank(randomAddress);
        string memory expectedPasswd = "myNewPasswd";
        passwordStore.setPassword(expectedPasswd);

        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();

        assertEq(expectedPasswd, actualPassword);
    }
```

Recommended Mitigation: Add an access control conditional to the `setPassword1` function.

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

Informational

[I-1] The `PasswordStore::getPassword` netspec indicates a parameter that doesn't exist, causing the netspec to be incorrect.

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

```
    /*
     * @notice This allows only the owner to retrieve the password.
     * @param newPassword The new password to set.
     */


    function getPassword() external view returns (string memory) {
        if (msg.sender != s_owner) {
            revert PasswordStore__NotOwner();
        }
        return s_password;
    }
```

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect netspec line.

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

[I-1] Gas Optimization: Immutable Owner Address `PasswordStore::s_owner`

Description The code declares the contract owner using a standard state variable: `s_owner`. This variable is stored in a storage slot, meaning every read operation (SLOAD) incurs a significant gas cost.

Impact: The current implementation incurs unnecessary gas overhead. Reading from an immutable variable costs ~3 gas (a PUSH opcode), while reading from storage costs at least 100 gas (for a warm access) and 2100 gas (for a cold access).

Recommended Mitigation: Change the owner variable from a storage variable to an immutable variable.