Password Store Audit Report

Version 1.0

Dewaxindo

October 17, 2025

Prepared by: Dewaxindo

Lead Auditors:

- Dewaxindo

Table of Contents

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.

Disclaimer

The DEWAXINDO 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

	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Table 1: Severity matrix by likelihood and impact

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

Audit Details

Commit Hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

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

Executive Summary

- We found issue

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] Storing a password on-chain makes it publicly visible and not private

Description: All on-chain data is publicly visible and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be private and accessed only via `PasswordStore::getPassword()`.

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

Proof of Concept:

The following test demonstrates that anyone can read the password directly from storage.

1. Start a local node

```
make anvil
```

2. Deploy

```
make deploy
```

3. Run the storage tool Use `1` because that is the storage slot of `s_password` in the contract.

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

Example output: `0x6d7950617373776f726400000000000000000000000000000000000000000014`

Parse the hex to a string with:

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

Result: `myPassword`

Recommended Mitigation: Reconsider the contract architecture. Encrypt the password off-chain and store only the ciphertext on-chain. This requires the user to retain an off-chain decryption secret. Additionally, remove or strictly restrict any view function that could lead to inadvertently submitting the decryption secret in a transaction.

[H-2] `PasswordStore::setPassword()` Has no Access Control, Everyone Can Set New Password

Description: `setPassword(string)` does not restrict callers. Although the contract tracks an owner in `s_owner`, there is no ownership check on the write path. Any address can call `setPassword` and overwrite `s_password`.

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

Impact: Loss of integrity and confidentiality of stored data. An attacker can:

- Overwrite the password at any time (permanent loss of trusted state)

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

Code

```
  function test_non_owner_setting_password(address anyone) public {
        vm.assume(anyone != owner);
        vm.prank(anyone);

        string memory expectedPassword = "hacked-by-attacker";
        passwordStore.setPassword(expectedPassword);

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

Recommended Mitigation: Require ownership on the write path. Options:

- Add an ownership check to `setPassword`: `require(msg.sender == s_owner)` (or revert error) before writing.

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

- Or inherit an audited access control implementation such as OpenZeppelin `Ownable` and add `onlyOwner` to `setPassword`. Additionally, consider renaming the event to something accurate (e.g., `SetNewPassword`) and emitting it after a successful, authorized update.

Medium

Low

Informational

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

Description: The NatSpec for `getPassword()` includes an `@param` tag for a parameter that does not exist. This makes the documentation inaccurate and can mislead integrators and tooling.

Impact: Informational. Inaccurate NatSpec can:

- Confuse auditors and integrators
- Degrade doc-generation outputs
- Reduce maintainability and developer experience

Proof of Concept: The function has no parameters, but the NatSpec references one:

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

Recommended Mitigation: Update the NatSpec on `getPassword()`:

- Remove the incorrect `@param` tag
- Optionally add a `@return` tag describing the returned password
- Consider a `@dev` note clarifying access control

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

Gas