PasswordStore Initial Audit Report

Version 0.1

YASH NIMBALKAR

Date: *April 4, 2025*

PasswordStore Audit Report

Prepared by: Yash Nimbalkar

Contents

1. Disclaimer

The auditor makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the auditor is not an endorsement of the underlying business or product. The audit was timeboxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

2. Risk Classification Matrix

Impact \ Likelihood	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

3. Audit Details

- **Auditor:** Yash Nimbalkar
- **Commit Hash:** `7d55682ddc4301a7b13ae9413095feffd9924566`
- **Repository:** `https://github.com/Cyfrin/3-passwordstore-audit/tree/onboarded`
- **Audit Period:** April 3 – April 4, 2025

4. Scope

The audit covered the following contract(s):

- `src/PasswordStore.sol`

5. Protocol Summary

PasswordStore is a smart contract designed for the secure storage and retrieval of a single user's password. The protocol is intended strictly for individual use and does not support multi-user functionality. Access control is central to its design and only the designated owner should have the ability to set or retrieve the stored password.

6. Roles

- **Owner** → Can set and retrieve the password.

7. Executive Summary

Severity	Number of Issues
High	2
Medium	0
Low	0
Informational	1
Gas Optimizations	0
Total	3

8. Findings

8.1 [H-1] Passwords Stored On-Chain Are Public and Anyone Can View Them

Severity: High

Description: The contract stores the user's password in a private state variable `'s_password'`, which is expected to remain confidential and accessible only through the `'getPassword()'` function. However, in Ethereum (and most EVM-compatible chains), marking a variable as 'private' does not imply true secrecy. Private variables are only inaccessible at the code level, not at the blockchain storage level.

Impact: Any blockchain user, regardless of their permissions, can retrieve the raw password directly from the contract's storage.

Proof of Concept: The below test case shows how anyone could read the password directly from the blockchain. We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1. Create a locally running chain

```
make anvil
```

2. Deploy the contract to the chain

```
make deploy
```

3. Run the storage tool → We use the storage slot 1, because it is the slot for `s_password`

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

4. You'll get an output that looks like this:

```
0x6d7950617373776f7264000...00014
```

5. You can then parse that hex to a string with:

```
cast parse-bytes32-string
0x6d7950617373776f7264000...00014
```

6. You will get an output as the following

```
myPassword
```

Recommendation: Private data should never be stored in plaintext on-chain. Instead:

- Encrypt the password client-side using a secure symmetric algorithm (e.g., AES-GCM).
- Store only the resulting ciphertext on-chain.
- The user should retain the decryption key securely off-chain, such as in a password manager, browser extension, or secure enclave.

This architectural shift ensures the blockchain is only used as an immutable data store and not a vault for sensitive plaintext data. Note, however, that this approach moves the burden of key management to the user, which must be carefully addressed in the protocol's UX design.

8.2 [H-2] `setPassword` is Callable by Anyone

Severity: High

Description: The `'setPassword(string memory newPassword)'` function allows users to update the stored password. Although the contract's documentation (NatSpec) indicates that only the contract owner should be able to call this function, there is no actual access control implemented in the code to enforce this restriction.

Impact: Since there is no 'require' check or 'onlyOwner' modifier, any externally owned account (EOA) or smart contract can call this function and overwrite the existing password, regardless of whether they are the designated owner.

Proof of Concept: The following Foundry test demonstrates how any arbitrary address can modify the password:

```
function test_anyone_can_set_password(address attacker) public {
    vm.prank(attacker);
    string memory injectedPassword = "maliciousPass123";
    passwordStore.setPassword(injectedPassword);

    vm.prank(owner);
    string memory result = passwordStore.getPassword();
    assertEq(result, injectedPassword); // Test passes
}
```

Recommendation: Access control should be enforced using an explicit ownership check before allowing state changes. The simplest fix is to add a conditional 'require' statement or revert on unauthorized access, as shown below:

```
function setPassword(string memory newPassword) external {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    s_password = newPassword;
    emit SetNetPassword();
}
```

8.3 [I-1] Incorrect NatSpec Documentation on `getPassword()`

Severity: Informational

Description: The NatSpec comment for the 'getPassword()' function incorrectly includes a '@param' tag for 'newPassword', even though the function does not take any parameters.

Impact: While this does not affect the contract's execution, it may cause confusion for developers, auditors, or users who rely on autogenerated documentation or IDE hints to understand contract behavior.

Recommendation: Remove the incorrect '@param newPassword' line from the NatSpec comment to accurately reflect the function's signature.

9. Conclusion

The PasswordStore contract exhibits serious flaws in access control and privacy assumptions. Both identified vulnerabilities are critical and must be addressed before mainnet deployment.