Password Store Protocol Audit Report

Prepared by: Saurabh Kaplas

Table of Contents

Protocol Summary

The PasswordStore Protocol is designed for securely storing and retrieving user passwords on-chain. Users can store their private passwords, which are meant to be accessible only to the contract owner. The protocol allows the owner to set and access their password at any time while restricting access to unauthorized users.

Disclaimer

The "SeKurity" 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

		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 finding described in this document correspond to the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

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.

Executive Summary

This audit took 3 days to complete including the scoping, reconnaissance, vulnerability identification and report development.This is a private audit done by an individual security researcher, Saurabh Kaplas.We were able to find some critical vulnerabilities in the codebase including missing access controls, storing a raw private password on-chain and initialization timeframe vulnerability.Lookout for the #Findings section for more details.

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	0

Severity	Number of issues found
Info/Gas	2
Total	5

Findings

High

[H-1] Storing the password on-chain makes it visible to anyone, hence it is no longer private.

Description: Anyone can see the input parameters of a transaction by looking at the blockchain.When the `PasswordStore::s_password` variable is set using the `PasswordStore::setPassword` function, the password is stored as plain text on the blockchain. Even though the `PasswordStore::getPassword` function restricts access to the owner, the password is still exposed in the transaction data when it is set.

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

Impact: Anyone can monitor the blockchain and see the password when it is being set, which defeats the purpose of storing a private password.

Proof of Concept: (Proof of code)

The below test case shows how anyone can read private data off the blockchain.

1. Create a locally running chain.

```
anvil
```

2. Deploy a smart contract to the chain.

```
make deploy
```

3. Run the storage tool:

We use `1` because `s_password` is at the 1st storage slot in the `PasswordStore` contract.

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

Output will be something like this:

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

Then parse this hex to a string with:

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

And get an output of:

`myPassword`

Recommended Mitigation: Maintaining the privacy of the password is the overall aim of this protocol and that is getting breached.One way is:

```
1. Encrypt the Password Off-Chain
2. Store the Encrypted Password On-Chain
3. Retrieve and Decrypt Off-Chain
```

But this requires user to remember another password off-chain to decrypt the stored password.However, view function should also be removed as you wouldn't want the user to accidentally send a transaction with this decryption key.

[H-2] `PasswordStore::setPassword` does not enforce any access control, meaning anyone can call this function and overwrite the stored password.

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

```
    function setPassword(string memory newPassword) external {
@>      // @v-h missing access control
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Any user (not just the owner) can change the password, making it insecure and severely breaking the contract's intended functionality.

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

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

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

And then run :

```
forge test --mt test_anyone_can_set_password
```

You'll see that this fuzz test passes; proving that anyone can set the password.

Recommended Mitigation: Add a check to ensure only the owner can call the `PasswordStore::setPassword` function.

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

Medium

[M-1] Initialization Timeframe Vulnerability (Password not set during deployment)

Description: The contract does not set the password during its construction (in the constructor). As a result, when the contract is initially deployed, the password remains uninitialized, taking on the default value for a string, which is an empty string. During this initialization timeframe, the contract's password is effectively empty and can be considered a security gap.

Impact: The impact of this vulnerability is that during the initialization timeframe, the contract's password is left empty, potentially exposing the contract to unauthorized access or unintended behavior.

Recommended Mitigation: To mitigate the initialization timeframe vulnerability, consider setting a password value during the contract's deployment (in the constructor). This initial value can be passed in the constructor parameters.

```
constructor(string memory initialPassword) {
        s_owner = msg.sender;
+       s_password=initialPassword;
    }
```

Informational

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

Description:

```
/*
 * @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()` while the natspec says it should be `getPassword(string)`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line.

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

Gas

[I-2] High gas costs of storing a string

Description: Storing a string `PasswordStore::s_password` directly in the smart contract can be gas-intensive due to the variable-length nature of strings. Gas costs increase based on the length of the string, which makes it less efficient compared to storing fixed-size data types like `bytes32`.

Impact: Storing a long string as a password increases gas consumption during storage and retrieval operations.

Recommended Mitigation: Instead of storing the raw string, hash the password using keccak256 and store it as `bytes32`. This not only reduces storage costs but also enhances security by preventing direct access to the plain-text password.

```
-    string private s_password;
+    bytes32 private s_passwordHash;
```

```
function setPassword(string memory newPassword) external {
-    s_password = newPassword;
+    s_passwordHash = keccak256(abi.encodePacked(newPassword));
}
```