Protocol Audit Report

Version 1.0

Shivam Deshmukh

January 30, 2024

Protocol Audit Report

Shivam Deshmukh

March 7, 2023

Prepared by: Shivam Deshmukh

Lead Security Researcher: ShivamD21

Table of Contents

    * [L-1] `PasswordStore::s_password` remains uninitialized to the default value until `PasswordStore::setPassword()` is called.
  - Informational
    * [I-1] `PasswordStore::getPassword()` natspec mentions a parameter that does not exist.

Risk Classification

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

Disclaimer

I have made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. The security audit is not intented to be 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.

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.

Audit Details

The findings described in this document correspond to the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	1
Info	1
Gas Optimizations	0
Total	4

Findings

High

[H-1] `PasswordStore::s_password` is accessible to anyone on-chain despite private visibility modifier.

Description: The `PasswordStore::s_password` variable utilizes the **`private`** visiblity modifier and a `PasswordStore::getPassword()` function that checks for the owner, but that doesn't mean the data is inaccessible to anyone but the owner, since anyone can read any data from the blockchain.

Impact: This sacrifices the privacy of the password which is meant to be confidential.

Proof-of-Concept:

1. Run a local anvil chain

```
make anvil
```

2. Deploy the smart contract

```
make deploy
```

3. Use the `storage` option with `cast` to access the storage location of variable `PasswordStore::s_password`.

We use 1 because that's the storage slot of `s_password` in the contract.

```
cast storage <CONTRACT_ADDRESS> 1
```

It will return a bytes output like this:

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

4. Parse the bytes object to string

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

And get an output of:

```
myPassword
```

Recommended Mitigation: Since this endangers the architecture of the protocol it is advisable to onsider using some encryption methodology to encrypt the password off-chain and then store it on-chain and allow the user to use a 'key' to decrypt and access their password. It is also recommended that the view function be removed to prevent the user from sending a transaction with the password that decrypts the original password.

[H-2] Absence of access control in `PasswordStore::setPassword()` function.

Description: There is no piece of code that checks if the `msg.sender` is actually the `s_owner`, allowing anyone to call the function `PasswordStore::setPassword()` and alter the value in `PasswordStore::s_password`.

Faulty code:

```
function setPassword(string memory newPassword) external {
    // @audit no access control checks, anyone can change password
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact: This risks the integrity of the protocol as any entity can alter the password of the owner.

Proof-of-Concept: The following piece of code demonstrates the vulnerability, and needs to be copy pasted into the `PasswordStore.t.sol` test file.

Code Snippet

```
function test_non_owner_can_set_password() public {
    vm.assume(attacker != owner);
    vm.startPrank(attacker);
    passwordStore.setPassword("You have been haxed!");
    // assert will only run if the above has been executed successfully
    assertNotEq(attacker, owner);
}
```

Recommended Mitigation: Add an access modifier that checks for the `msg.sender` being the `s_owner` in the `PasswordStore::setPassword()` function.

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

Low

[L-1] `PasswordStore::s_password` remains uninitialized to the default value until `PasswordStore::setPassword()` is called.

Description: The constructor does not take any arguments and `PasswordStore::s_password` gets initialized with the default value at the time of contract deployment. It remains like that until `PasswordStore::setPassword()` is called to set some value explicitly.

Impact: This leaves a gap for unintented behaviour since the password is left empty.

Recommended Mitigation: Use the constructor to pass in an argument with the value to initialize the `PasswordStore::s_password` variable as shown in the following code.

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

Informational

[I-1] `PasswordStore::getPassword()` natspec mentions a parameter that does not exist.

Description: The natspec documentation given for the `PasswordStore::getPassword()` function states that the function takes a `newPassword` parameter, which it does not.

```
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) {
```

Impact: This makes the natspec inaccurate.

Recommended Mitigation: Remove the incorrect line from natspec.

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