Protocol Audit Report

Version 1.0

maikelordaz@gmail.com

April 22, 2024

Protocol Audit Report

Maikel Ordaz

March 30, 2024

Prepared by: Maikel Ordaz Lead Security Researcher:

- Maikel Ordaz

Table of Contents

Protocol Summary

PasswordStore is a smart contract application for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Disclaimer

Maikel Ordaz as a Blockchain Security Researcher 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		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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

- Commit Hash: 7d55682ddc4301a7b13ae9413095feffd9924566
- Solc Version: 0.8.18
- Chain(s) to deploy contract to: Ethereum

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.

Issues found

Severity	Founds
High	2
Informational	1

Findings

High

[H-1] Store the password on-chain makes it visible to anyone

Description: All data stored on-chain can be read directly from the 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 contract's owner.

We show one methhod of reading any data off chain below.

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

Proof of Concept:

The below test case shows how anyone can read the password directly from the blockchain.

1. Create a locally running chain

```
anvil
```

2. Deploy the contract to the chain

```
make deploy
```

3. Run the storage tool

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

Take from the output the storage slot related to `PasswordStore::s_password` in this case 1

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

You'll get this output

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

1. Cast the value

```
cast parse-bytes-32-string 0
    x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get the output: `myPassword`

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and then store the encrypted password on-chain. This would require the user to remember another password off-chain to decrypt the password. However, you'd also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password.

[H-2] `Password::setPassword` has no access control, meaning a non-owner could change the password

Description: The `Password::setPassword` function is set to be an `external` function, 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.`

```
1      function setPassword(string memory newPassword) external {
2  @>      // @audit: There are no access controls
3          s_password = newPassword;
4          emit SetNetPassword();
5      }
```

Impact: Anyone can set/change the password, severely breaking contract intended functionality

Proof of Concept: Add and run the following test to `PasswordStore.t.sol:PasswordStoreTest`

Test code

```
    function test_non_owner_can_set_password(address randomAddress)
        public {
        vm.assume(randomAddress != owner);

        vm.startPrank(randomAddress);
        string memory expectedPassword = "myNewPassword";
        passwordStore.setPassword(expectedPassword);
        vm.stopPrank();

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

        assertEq(actualPassword, expectedPassword);
    }
```

Recommended Mitigation: You can use `Ownable` contract from Openzeppelin, or add the next block to your function

Recommended code

```
    function setPassword(string memory newPassword) external {
@>      // @audit: New code starts here
        if (msg.sender != s_owner) {
            revert PasswordStore__NotOwner();
        }
@>      // @audit: New code ends here
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Informational

[I-1] Wrong natspec

Description: The natspec documentation is wrong

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

```
4       */
5      function getPassword() external view returns (string memory) {
6          if (msg.sender != s_owner) {
7              revert PasswordStore__NotOwner();
8          }
9          return s_password;
10     }
```

Impact: The documentation is incorrect

Recommended Mitigation:

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