Protocol Audit Report

Version 1.0

Luka Nikolic

January 31, 2024

Protocol Audit Report

Luka Nikolic

Jan 31, 2024

Prepared by: Luka Nikolic

Lead Security Researcher:

- Luka Nikolic

Table of Contents

Protocol Summary

PasswordStore is a protocol designed to store and retrieve user's passwords. It should be used by a single user and not by multiple users. Only the owner should be able to set and access this password.

Disclaimer

The Luka 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
	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 findings describer in this document correspond the following commit hash:

```
ff694529248699c59d991022108530247a92d56d
```

Scope

```
./src/
#--PasswordStore.sol
```

Roles

- Owner: 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

It went good.

Spent 2 days on auditing this

Issues found

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

Findings

High

[H-1] Storing the password on-chain makes it visable to anyone, and no longer private!

Description: All data stored on-chain is visible to anyone and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be a private and only accessed through the `PasswordStore::getPassword` function, which is only called by the owner of the contract.

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

Impact: Anyone can read the private password, severly 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

```
1     make anvil
```

2. Deploy contract on the chain

```
1 make deploy
```

3. Run the storage tool We use 1 because that is the storage slot of `s_password` in the contract. You have address below Return line

```
1 0: contract PasswordStore 0x....
```

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

You will get this output:

```
1 0x6d7950617373776f726400000000000000000000000000000000000000000014
```

Then you run this to parse that hex to a string:

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

And you get this :

```
1 myPassword
```

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt password off-chain, and then store the encrypted pasword on-chain. This would require the user to remembeer another password off-chain to decrypt the passworw. You also 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] `PasswordStore::setPassword` has no access controls, meaning a non owner could change password

Description: The `PasswordStore::setPassword` function is set ot 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 is no access control
3          s_password = newPassword;
```

```
4           emit SetNetPassword();
5       }
```

Impact: Anyone can break and set password and breaking contract functionality of setting and storing user passwords.

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

Code

```
1  function test_non_owner_can_set_password(address randomAddress) public
       {
2           // make sure our new address is not the owner
3           vm.assume(randomAddress != owner);
4           // setting new random address
5           vm.prank(randomAddress);
6           // creating new password
7           string memory expectedPassword = "changed password";
8           // putting our new password to store it in set password
               function
9           passwordStore.setPassword(expectedPassword);
10
11          // starting prank with actuall owner
12          vm.prank(owner);
13          // getting current owner password
14          string memory actualPassword = passwordStore.getPassword();
15
16          // check if our new changed password is same as the owner's
               password
17          assertEq(expectedPassword, actualPassword);
18      }
```

Recommended Mitigation: Add an acess control to the `setPassword` function.

```
1  if(msg.sender != s_owner){
2     revert PasswordStore__NotOwner();
3  }
```

Informational

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

Description:

```
1       /*
2        * @notice This allows only the owner to retrieve the password.
3        // @audit - there is no newPassword parameter!
```

```
4  ->    * @param newPassword The new password to set.
5        */
6      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.

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