Passwordstore Audit Report

Version 1.0

theirrationalone

December 14, 2023

Passwordstore Audit Report

theirrationalone

December 14, 2023

Prepared by: theirrationalone Lead Auditors: - theirrationalone

Table of Contents

Protocol Summary

Passwordstore 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

The theirrationalone 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

The findings described in this document correspond 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

*In this audit, I found vulnerabilities mostly about access controls, privacy, and natspec. This was an awesome experience all the way through this audit.

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 is visible to anyone. Regardless of the solidity `private` keyword, anyone can still see the password.

Description: All data stored on-chain is visible to anyone. Stored data can be read directly from the blockchain. The `PasswordStore::s_password` storage variable is intended to be private and can only be accessed by the owner of the password/protocol through the `PasswordStore::getpassword` external function.

We show one such method to read any data off-chain below.

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

Proof of Concept: (Proof of Code)

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

Interaction steps:

1. Open a blank bash terminal. Type & execute commands given below... `bash anvil`
2. Open another separated bash terminal. Type & execute commands given below... `bash forge script ./script/DeployPasswordStore.s.sol:DeployPasswordStore --rpc-url http://127.0.0.1:8545 --private-key 0xac0974bec39a17e36ba4a6b4d238ff --broadcast`
3. Copy the `Contract Address` value from the bash terminal output of step 2's execution. The value might be something like this `0x5FbDB2315678afecb367f032d93F642f64180aa3` .
4. Now on a free of processes bash terminal execute the following commands given below...
    - `cast storage 'contract_address''variable_storage_location'--rpc-url ...`

```
cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1 --rpc-url http://127.0.0.1:8545
```

5. You might get an output value of type `bytes32` something like this `0x6d7950617373776f7264000000000`
6. Put that output value (obtained in step 4) with some commands as given below...
    - Parsing: `cast parse-bytes32-string 'bytes32_output_value'`

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

7. Voila! Now you can see the private password... Password = `myPassword`, in my case as I stored this string.

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] There are no access controls on `PasswordStore::setPassword` function. Anyone can set/change/update the password.

Description: `PasswordStore::setPassword` function has no access controls which leads to the danger of high-severity exploitation. Anyone can set/update/change the password anytime, and that's a clear abuse of the functionality of the contract.

Code

```
1  function setPassword(string memory newPassword) external {
2      // no access controls to restrict (not owner) to change or set the
           password.
3  @>  s_password = newPassword;
4      emit SetNetPassword();
5  }
```

Impact: Breaks the contract's intended functionality which was only allowed the owner of the contract to set or update the password.

Proof of Concept: Put the following code into `PasswordStore.t.sol` tests Suite file. After that, Execute `forge test` command into your bash terminal.

Functionality Breaking Test Code:

```
1  function test_allow_non_owner_set_password(address anyUser) public {
2      string memory expectedPassword = "vulnerablePassword123";
3
4      vm.assume(anyUser != owner);     // to make it more clear & obvious.
5      vm.startPrank(anyUser);      // any user can set the password.
6      passwordStore.setPassword(expectedPassword);
7      vm.stopPrank();
8
9      vm.startPrank(owner);    // only owner can retrieve the password.
10     string memory actualPassword = passwordStore.getPassword();
11     vm.stopPrank();
12
13     // Indeed, Test Passes...
14     assertEq(actualPassword, expectedPassword);
15     assert(keccak256(abi.encodePacked(actualPassword)) == keccak256(abi
           .encodePacked(expectedPassword)));
16 }
```

Recommended Mitigation: We can implement a conditional which could recognize the owner of the contract and allow only them to set or update the password. Therefore, could also identify non-owner users and could restrict them. We can implement the Code snippet given below into `PasswordStore::setPassword` function, which is capable of doing so…

Mitigation Code Snippet (Place it inside `PasswordStore::setPassword` function at very top):

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

Informational

[I-1] The natspec of `PasswordStore::getPassword` function has an incorrect comment which indicates a useless invalid parameter called `newPassword`. Leads to natspec to be invalid and incorrect.

Description: Should remove that invalid parameter comment.

Comment (Invalid):

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

The `PasswordStore::getPassword` function signature is `getPassword()` but natspec says it should be `getPassword(string)` which is absolutely incorrect.

Impact: Leads to having incorrect natspec.

Recommended Mitigation: Remove the incorrect natspec line.

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