Protocol Audit Report

Version 1.0

Hessamedean

June 15, 2024

Protocol Audit Report

Hessamedean

June 13, 2024

Prepared by: Hessamedean Lead Auditors: - xxxxxxx

Table of Contents

Protocol Summary

The passwordStore is a protocol to save password by the owner and save it on-chain.

Disclaimer

The Hessamedean 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		
		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 discribed in this document corespond the following commit hash:

Commit Hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

```
x
./src/
#-- passwordStore.sol
```

Roles

- Owner: The user who set the password and read the password.
- Outsiders: No one else should be able to set or read the password. # Executive Summary

Add some note about how the audit went, types of things you found, etc.

Issues found

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

Findings

High

[H-1] Storing password on-chain makes it visibal to anyone in public, and no longer private

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

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

Impact: Anyone can read the private password, severly breakin the functionality of the protocol.

Proof of Concept: The below test case shows how anyone could read the password directly from the blockchain. We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1. Create a locally running chain

```
make anvil
```

2. Deploy the contract to the chain

```
make deploy
```

3. Run the storage tool

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

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

You'll get an output that looks like this:

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

You can then parse that hex to a string with:

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

And get an output of:

```
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] `PasswordStore::setPassword` has no access control, meaning a non-owner could change the password

Description: The `PasswordStore::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 only alows the owner to set the password.`

```
    function setPassword(string memory newPassword) external {
        @> // @audit- There are no access control
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Anyone can set/change the password of the contract, severly breaking the contract intended functionality.

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

Code

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

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

```
</details>
```

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

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

Informational

[I-1] The `PasswordStore::getPassword` natspac indicates a prameter that doesn't exist, causing the natspact 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) {
        if (msg.sender != s_owner) {
            revert PasswordStore__NotOwner();
        }
```

```
9           return s_password;
10      }
```

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.
```