Blocktorch

Protocol Audit Report

Version 1.0

ctlst

February 6, 2024

Protocol Audit Report

ctlst

Feb 6, 2024

Prepared by: ctlst

Lead Security Researcher: ctlst

Table of Contents

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 their password.

Disclaimer

ctlst 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:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

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

Add some notes on how the audit went, types of things you found, time spent, tools used, etc. I spent half a Tuesday **auditing** this garbage but its fine I guess it'll help with my discipline! I learned some more things about Foundry, did some markdown stuff and am going to generate a LaTex document as in good ol' university times, lmao.

Issues found

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

Findings

High

[H-1] Storing the password on-chain makes it visile to anyone, hence not private

Description: All data stored on-chain is visible to anyone and can be read directly from the blockchain storage. The `PaswordStore::s_password` variable is intended to be a private variable and only accesed through the `PaswordStore::getPassword()` function, which is intended to be called only by the owner of the contract. We show why this one is not actually the case below.

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

Proof of Concept (Proof of Code): The below test case shows how exactly anyone can read the password directly on the blockchain: 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 function in question is `external` and has no modifiers or inner `require` checks for the sender of a transaction. It will run the full logic of setting a new password for anyone triggering this function on-chain.

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

Impact: Allowing non-owner to fully execute this function without any access controls fully deminishes the whole purpose of the contrcat's functionality

Proof of Concept: Add the following fuzzing test case to the `Password.t.sol` test file:

Code

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

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

    assertEq(actualPassword, expectedPassword);
}
```

Recommended Mitigation: Add an access control conditional to the `setPassword()` function, like so:

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

Informational

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist, causing the natspec 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) {
```

The natspec for the function `PasswordStore::getPassword` indicates it should have a parameter with the signature `getPassword(string)`. However, the actual function signature is `getPassword()`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line.

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