Protocol Audit Report

Version 1.0

cenobyte321

January 6, 2024

Protocol Audit Report

cenobyte321

January 6, 2024

Prepared by: cenobyte321 Lead Auditors: - cenobyte321

Table of Contents

Protocol Summary

PasswordStore is a protocol dedicated to storage and retreival of a user's passwords. The protocol is designed to be used by a single user, and it is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Disclaimer

Cenobyte321 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 of this document are based on the code in commit hash `0xdeadbeef`

Scope

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

Roles

- Owner: The user who can set and access the password.
- Outsiders: No one else should be able to access the password or set it.

Executive Summary

Add some notes about how the audit went here. Types of things you found, et.

We spent X hours with Z auditors using Y tools, etc.

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 visible 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 variable and only accessed through the `PasswordStore::getPassword` function, which is intended to be only called by the owner of the contract.

We show one such method of reading 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.

1. Create a locally running chain

```
make anvil
```

2. Deploy the contract to the chain

```
make deploy
```

3. Run the storage tool

Assuming the contract was deployed to `0x5fbdb2315678afecb367f032d93f642f64180aa3`, we can run the storage tool to read the password from the `s_password` storage variable, which is in the storage slot `1`.

```
cast storage 0x5fbdb2315678afecb367f032d93f642f64180aa3 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 value to a string with:

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

And you'll get an output of:

```
myPassword
```

Recommended Mitigation: Due to this, the overall architecture of the contract should be redesigned. One could encrypt the password off-chain, an dthen 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 cast parse-bytes32-string 0x6d7950617373776f726400000000000000000000000000000000000000000014 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 can set the password

Description: The `PasswordStore::setPassword` function visibility is `external`, therefore anyone can call it. The natspec comment says that only the owner can call it, but this is not enforced.

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

Impact: Anyone can set/change the password of the contract, severely breaking the contract's 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);
    }
```

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

```
    function setPassword(string memory newPassword) external {
        if (msg.sender != s_owner) {
            revert PasswordStore__NotOwner();
        }
        s_password = newPassword;
        emit SetNetPassword();
    }
```

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 `PasswordStore::getPassword` function signature is `getPassword()` which the natspec says it should be `getPassword(string)`.

Impact: The natspec is incorrect.

Proof of Concept: N/A

Recommended Mitigation: Remove the incorrect natspec line.

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