Protocol Audit Report

Version 1.0

Cyfrin.io

December 7, 2025

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Dengi Lead Auditors: - dengi

Table of Contents

Protocol Summary

A smart contract applicatoin 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 DENGI 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 findings described in this document correspond to the following commit hash: Commit Hash

Scope

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

Roles

- Owner: Deployer of the contract, is the only one that can set or view password

Executive Summary

Time, tools etc

Issues found

Severity	Issues Found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] Passwords stored on-chain are visable to anyone, not matter solidity variable visibility

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.

However, anyone can direclty read this using any number of off chain methodologies

Impact: The password is not private.

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

```
1  make anvil
```

2. Deploy the contract to the chain

```
1  make deploy
```

3. Run the storage tool

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

```
1  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:

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

And get an output of:

```
1  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

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 allows only the owner to set a new password`.

```
1      function setPassword(string memory newPassword) external {
2  @>     // @audit - There are no access controls here
3          s_password = newPassword;
4          emit SetNetPassword();
5      }
```

Impact: Anyone can set/change the password of the contract.

Proof of Concept:

Add the following to the `PasswordStore.t.sol` test suite.

```
1  function test_anyone_can_set_password(address randomAddress) public {
2      vm.prank(randomAddress);
3      string memory expectedPassword = "myNewPassword";
4      passwordStore.setPassword(expectedPassword);
5      vm.prank(owner);
6      string memory actualPassword = passwordStore.getPassword();
7      assertEq(actualPassword, expectedPassword);
8  }
```

Recommended Mitigation: Add an access control modifier 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, causing the natspec to be incorrect

Description:

```
1       /*
2        * @notice This allows only the owner to retrieve the password.
3  @>    * @param newPassword The new password to set.
4        */
5      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.

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