Protocol Audit Report

Version 1.0

yudistira19

August 7, 2024

Protocol Audit Report

Yudistira19

August 7, 2024

Prepared by: Yudistira Adi Wahyudi R Lead Auditors:

- yudistira19

Table of Contents

Protocol Summary

A 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 AKKSES 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
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 finding 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

Add some notes about how yhe audit went, types of things you found, etc.

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

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Informational	1
Total	5

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 blockchain. The `PasswordStore::s_password` variable is intended to be 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 contract.

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 allows only the owner to set a new password.`

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

```
5       }
```

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

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

Code

```
1       function test_anyone_can_set_password(address randomAddress) public
            {
2           vm.assume(randomAddress != owner);
3           vm.prank(randomAddress);
4           string memory expectedPassword = "myNewPassword";
5           passwordStore.setPassword(expectedPassword);
6
7           vm.prank(owner);
8           string memory actualPassword = passwordStore.getPassword();
9           assertEq(actualPassword, expectedPassword);
10      }
```

Recommended Mitigation: Add an access control conditional 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:

The `getPassword` function in the `PasswordStore` contract has an incorrect NatSpec parameter. The function signature is `getPassword()`, indicating it takes no parameters. However, the NatSpec documentation incorrectly specifies a `newPassword` parameter.

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

Impact: The incorrect NatSpec parameter can lead to confusion for developers using the contract, as it provides misleading information about the function's inputs

Recommended Mitigation: Remove the incorrect `@param` line from the NatSpec to accurately reflect the function signature.

```
- *@param newPassword the new password to set
```