Protocol Audit Report

Version 1.0

meyke.xyz

March 22, 2024

Protocol Audit Report

meyke.xyz

March 22, 2024

Prepared by: Meyke

Lead Auditors:

- Carsten

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 Meyke 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 desccibed in this document correspond to the following commit hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

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 the 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
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

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 the nature of the blockchain, all data is public. The password should not be stored on-chain. Instead, the password should be stored off-chain, and the contract should only store a hash of the password. This way, the password is never stored on-chain, and the hash can be used to verify the password.

Likelihood & Impact:

- Impact: HIGH
- Likelihood: HIGH
- Severity: HIGH

[H-2] `PasswordStore::setPassword` has no access controls, 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 are no access controls
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact: Anyone can set/change the password, 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.

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

Likelihood & Impact:

- Impact: HIGH
- Likelihood: HIGH
- Severity: HIGH

Informational

[I-1] Incorrect Natspec in `PasswordStore::getPassword` Due to Non-existent Parameter

Description:

The `PasswordStore::getPassword` function's Natspec comments include a parameter that does not exist in the function's signature. This discrepancy makes the Natspec documentation incorrect. Below is the erroneous part of the code:

```
1 /*
2  * @notice This function 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 function signature of `PasswordStore::getPassword` is `getPassword()` which indicates it takes no parameters. However, the Natspec comment suggests it should be `getPassword(string newPassword)`.

Impact:

This inconsistency in documentation can lead to confusion, suggesting that the `getPassword` function might accept a parameter, which it does not.

Recommended Mitigation:

To correct the Natspec documentation, remove the line that mentions the non-existent parameter. The following change is recommended:

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

Likelihood & Impact:

- Impact: NONE
- Likelihood: NONE
- Severity: Informational