Protocol Audit Report

Version 1.0

Oana Asoltanei

December 25, 2024

Protocol Audit Report

Oana Asoltanei

December 25, 2024

Prepared by: Oana Asoltanei

1. Table of Contents

2. 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. Only the owner should be able to set and access the password.

3. Disclaimer

Oana Asoltanei 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.

4. 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.

5. Audit Details

The findings described in this document correspond to the following commit hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

5.1. Scope

```
./src/
|-- PasswordStore.sol
```

5.2. Roles

Owner: The user who can set the password and read the password.

Outsides: No one else should be able to set or read the password.

6. Executive Summary

6.1. Issues found

Severity	Number of issues found
HIGH	2
MEDIUM	0
LOW	0
INFO	2
GAS	0
TOTAL	4

7. Findings

7.1. High

7.1.1. [HIGH-1] On-chain Data Visibility Allows Unauthorized Access to Sensitive Information

Description:
In the `PasswordStore::getPassword` function, the private variable `PasswordStore::`

`s_password` is used to store sensitive data. However, marking a variable as **`private`** in Solidity does not prevent it from being publicly accessible via blockchain storage. It only restricts direct access through Solidity-based function calls.

Impact:
The password stored in the contract can be read directly from the blockchain by anyone, compromising the confidentiality of the data.

Proof of Concept:
The following demonstrates how the password can be retrieved directly from blockchain storage using Foundry's cast tool:

1. Start a local blockchain instance and allow it to keep running in the background:

```
anvil
```

2. Deploy the contract to the blockchain:

```
make deploy
```

3. Retrieve the stored password: Use the `cast storage` command to read the storage slot where the password is stored. In this case, the password is stored in slot 1.

```
cast storage <ADDRESS_HERE> 1 --rpc-url http://127.0.0.1:8545
```

Example output: `0x6d795061737373776f726400000000000000000000000000000000000000000014`

4. Decode the retrieved data into a human-readable string: Use `cast parse-bytes32-string` to convert the hexadecimal output into its string representation.

```
cast parse-bytes32-string 0
    x6d795061737373776f726400000000000000000000000000000000000000000014
```

Output:

```
myPassword
```

Recommended Mitigation: Avoid storing sensitive information such as passwords in plaintext on-chain. If it is necessary to store such data, ensure it is encrypted before storage. However, note that users would need to manage and remember the decryption key separately to access the encrypted data.

7.1.2. [HIGH-2] Missing Access Control Allows Unauthorized Users to Modify the Password

Description:
The `PasswordStore::setPassword` function lacks access control mechanisms, allowing any user to invoke the function and modify the `PasswordStore::s_password` variable. This makes it possible for unauthorized users to overwrite the password set by the contract owner.

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

Impact: The absence of access control exposes the system to unauthorized access, enabling any user to overwrite the stored password, compromising the intended functionality and security of the contract.

Proof of Concept: The following test case demonstrates how an unauthorized user can modify the password:

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: Implement access control in the PasswordStore::setPassword function to restrict its execution to the contract owner. For example:

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

This ensures that only the designated owner can modify the password, maintaining the integrity of the contract's functionality.

7.2. Medium

7.3. Low

7.4. Informational

7.4.1. [INFO-1] Event Naming Convention Is Not Followed

Description:
The event `PasswordStore::SetNetPassword()` does not adhere to the recommended naming convention for Solidity events. The standard convention suggests naming events in the format `<ContractName>__<EventName>` to clearly indicate the contract they belong to, enhancing traceability and readability during debugging and analysis.

Impact:
Non-standard naming conventions can make debugging and contract analysis more challenging, especially when dealing with multiple contracts. It increases the time required to identify event origins during off-chain processing or on-chain event subscription.

Recommended Mitigation:
Update the event name from `PasswordStore::SetNetPassword()` to `PasswordStore::PasswordStore__SetNetPassword()` to align with the recommended naming convention.

```
// Rename the event to:
event PasswordStore__SetNetPassword();
```

7.4.2. [INFO-2] Incorrect NatSpec Parameter in `PasswordStorage::getPassword`

Description:
The NatSpec documentation for the `PasswordStorage::getPassword` function incorrectly references a parameter `string newPassword`, which does not exist in the function signature. The actual function, `getPassword()`, does not accept any parameters, making the current documentation misleading.

Impact:
Misinformation in NatSpec documentation can lead to confusion among developers or auditors who rely on it to understand the contract’s behavior. This could result in errors or delays during contract interaction or code review.

Recommended Mitigation:
Update the NatSpec documentation for the `getPassword` function by removing the incorrect parameter reference.

```
    /*
     * @notice This allows only the owner to retrieve the password.
-     * @param newPassword The new password to set.
     */
    function getPassword() external view returns (string memory) {
        if (msg.sender != s_owner) {
            revert PasswordStore__NotOwner();
        }
        return s_password;
    }
```

7.5. Gas