Protocol Audit Report

Version 1.0

wafardev

September 26, 2024

Protocol Audit Report

wafardev

September 24, 2024

Prepared by: wafardev

Lead Security Researcher: - wafardev

Table of Contents

Protocol Summary

PasswordStore is a password management smart contract designed to securely store and manage passwords on the blockchain. It has functionality for setting and retrieving a password via functions like `setPassword` and `getPassword`. The idea is that only the owner of the contract should be able to set or retrieve the password.

Disclaimer

Wafardev 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 the following commit hash:

```
53ca9cb1808e58d3f14d5853aada6364177f6e53
```

Scope

The following contract file was reviewed as part of the audit:

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

- Solc version: 0.8.18
- Chain(s) to deploy contract to: Ethereum
- GitHub repository: Cyfrin/3-passwordstore-audit

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

Considering the nSLOC (non-commenting Source Lines of Code), no automated tools were used in the audit; only manual review was conducted.

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] On-Chain Storage of `PasswordStore::s_password` Exposes it to the Public

Description:

All data stored on-chain is inherently public, regardless of its visibility settings in Solidity. The `PasswordStore::s_password` variable, intended to be private, is exposed to anyone who can read the blockchain data. This undermines the security model since the password, although meant to be accessed only by the owner through the `PasswordStore::getPassword` function, can easily be extracted via a JSON-RPC call that queries the contract's storage.

Impact:

The visibility of the password allows unauthorized access, effectively compromising the security and intended functionality of the protocol.

Proof of Concept:

A new test case in `PasswordStore.t.sol` demonstrates how the hidden password can be retrieved by inspecting the storage layout of the `PasswordStore` contract.

Here is the added code:

Code

```
    function test_fetch_password_onchain() public {
        string memory hiddenPassword = "myPassword";
        uint256 stringSlot = 1; // The string is stored in slot 1
        bytes32 storageData = vm.load(address(passwordStore), bytes32(
            stringSlot)); // Load the data from slot 1

        // Convert bytes32 to string
        string memory parsedData = _bytes32ToString(storageData);

        // Log the parsed data for debugging purposes
        console.log(parsedData);

        // Compare the hashed hiddenPassword and parsedData
        assertEq(keccak256(abi.encodePacked(hiddenPassword)), keccak256
            (abi.encodePacked(parsedData)));
    }

    // Helper function to convert bytes32 to string
    function _bytes32ToString(bytes32 _bytes32) internal pure returns (
        string memory) {
```

```
        uint8 i = 0;
        while (i < 32 && _bytes32[i] != 0) {
            i++;
        }
        bytes memory bytesArray = new bytes(i);
        for (uint8 j = 0; j < i; j++) {
            bytesArray[j] = _bytes32[j];
        }
        return string(bytesArray);
    }
```

Run the following command in the terminal to replicate the issue:

```
forge test --mt test_fetch_password_onchain -vv
```

The test will succeed if the password is correctly extracted and will display it in the terminal.

Recommended Mitigation:

Given the critical nature of this issue, the contract's design requires a significant overhaul. Instead of storing the password directly on-chain, consider keeping it off-chain. You could store an encrypted version of the password on-chain, with the decryption key only known by the owner, to maintain confidentiality.

[H-2] Lack of Access Control on `PasswordStore::setPassword` Allows Unauthorized Password Changes

Description:

The `PasswordStore::setPassword` function is intended to be restricted to the contract owner. However, there are no access control checks implemented to verify the caller's identity. As a result, any user can invoke this function and change the password, regardless of their authorization.

```
function setPassword(string memory newPassword) external {
@> // @audit Missing access control, anyone can set the password
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact:

The lack of access control allows any user to modify the `PasswordStore::s_password` variable. This compromises the privacy and integrity of the password, rendering the contract's intended functionality vulnerable to unauthorized interference.

Proof of Concept:

A fuzz test written in `PasswordStore.t.sol` demonstrates how any non-owner address can successfully set a new password in the `PasswordStore` contract.

Here is the added code:

Code

```
    function test_non_owner_can_set_password(address user, string
        memory newPassword) public {
        vm.startPrank(user);
        vm.assume(user != owner);
        passwordStore.setPassword(newPassword);
        vm.stopPrank();

        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();
        assertEq(actualPassword, newPassword);
    }
```

To reproduce the issue, execute the following command:

```
forge test --mt test_non_owner_can_set_password
```

The test will pass if a non-owner address is able to change the password.

Recommended Mitigation:

Introduce a check in the `setPassword` function to ensure only the owner can change the password. This can be achieved by validating `msg.sender` against the stored `owner` address.

Here is an example of the corrected function:

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

By implementing this ownership check, the issue can be mitigated effectively.

Informational

[I-1] The `PasswordStorage::getPassword` natspec indicates a non-existing parameter, 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 `PasswordStorage::getPassword` natspec indicates that the function signature should be `getPassword(string)` instead of `getPassword()`.

Impact:

The natspec is incorrect.

Recommended Mitigation:

Remove the incorrect natspec line.

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