PasswordStore Audit Report

Version 1.0

Sebastian Zambrano Arango

June 23, 2024

PasswordStore Audit Report

Sebastian Zambrano Arango

June 16, 2024

Prepared and Audited by: Sebastian Zambrano Arango

Table of Contents

Protocol Summary

PasswordStore is a protocol designed to securely store and retrieve a user's password on the blockchain. The smart contract is intended for use by a single address, which has exclusive permissions to set and access the stored password.

Disclaimer

Sebastian Zambrano Arango 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

Commit Hash:

```
ff694529248699c59d991022108530247a92d56d
```

Scope

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

Roles

- **Owner:** The user that can both set and retrieve the save password.
- **Outsiders:** Any other user not having the owner role interacting with the contract.

Executive Summary

General notes about the audit, like: Hours worked, complications, eases, findings, etc..

Issues found

Severity	Number of Issues Found
High	2
Medium	0
Low	0
Informational	1
Gas	NA
Total	3

Findings

High

[H-1] Storage State Variable on-chain is publicly accessible despite visibility. Password is not private

Description: Any date stored on-chain is visible to anyone, being readable directly from Blockchain. The `PasswordStore::s_password` variable is intended to be private and only be accessed

through `PasswordStore::getPassword`, restricted to the owner of the contract, however the getter method becomes pointless.

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: The below test case shows how anyone can read the password directly from the Blockchain:

1. Create local running Blockchain:

```
make anvil
```

2. Deploy the contract to the chain:

```
make deploy
```

3. Get the storage slot for the password:

```
forge in <CONTRACT_ADDRESS> storage
```

4. Get the storage value for the slot on the local Blockchain and parse it to ascii:

```
cast storage <CONTRACT_ADDRESS> 1 --rpc-url http://127.0.0.1:8545 |
    xargs -I PASSWORD cast --to-ascii PASSWORD
```

Recommended Mitigation: Due to this finding the architecture of the contract should be rethought.

One possible wat to solve this is to encrypt the password off-chain, and save the encrypted result on-chain. However, this would require te user to remember a decryption password off-chain to get the original one. Additionally, there is caution needed when maintaining a getter for the original password, as this would require sending the decryption password creating a possible vulnerability of exposing it, meaning exposing the original one.

[H-2] High impact method 'PasswordStore::setPassword' not restricted. Anyone can set a new password

Description: Although the 'PasswordStore::getPassword' is restricted to owner only, the 'PasswordStore::setPassword' is not restricted. Allowing anyone to set a new password, although it is supposed to only be accessible to the contract's owner.

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

Impact: Anyone can set/change the password of the contract, severely breaking the Contract intended functionality.

Proof of Concept: The test `testFuzz_anyone_can_set_password` was added to `PasswordStore.t.sol` and is a proof that any random address (not being the owner) can set a new password and next time the owner get the Smart Contract address the new set password by a random address is returned:

Test

```
function testFuzz_anyone_can_set_password(address randomAddress) public {
        vm.assume(randomAddress != owner);

        string memory expectedPassword = "newPassword";
        vm.prank(randomAddress);
        passwordStore.setPassword(expectedPassword);

        vm.prank(owner);
        string memory newPassword = passwordStore.getPassword();

        assertEq(newPassword, expectedPassword);
    }
```

Recommended Mitigation: To either extend the `AccessControl` contract from `OpenZeppelin` to the Smart Contract and the usage of the `onlyRole` modifier or limit the usage with the `owner` state variable:

AccessControl

```
    function setPassword(string memory newPassword) external onlyRole(
    OWNER_ROLE) {
    s_password = newPassword;
    emit SetNetPassword();
    }
```

State Variable

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

Informational

[I-1] `PasswordStore::getPassword` natspec indicates a parameter that does not exist, leaving incorrect documentation of the code.

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

Impact: Contract documentation is incorrect.

Recommended Mitigation: Remove commented line in the Contract natspec.

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