Protocol Audit Report

Version 1.0

s3bc40

November 19, 2024

Protocol Audit Report

s3bc40

November 19, 2024

Prepared by: s3bc40 Lead Security Researcher:

- s3bc40

Table of Contents

Protocol Summary

The PasswordStore smart contract is a decentralized application designed to allow users to securely store and retrieve passwords. The contract ensures that only the user who set the password can access it, preventing unauthorized access by others. This report evaluates the contract's functionality, security, and adherence to best practices.

Disclaimer

The s3bc40 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:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles6

- 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

How the audit went, things found etc.

We spent X hours with Z and Y auditors using H tool. etc.

Issues found

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

Findings

High

[H-1] Variables stored in storage on-chain are visible to anyone, no matter the solidity visibility keyword. Meaning the password is not 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 intented to be a private variable and only accessed

through the `PasswordStore::getPassword` method, intended to be called only by the owner.

Impact: Anyone can read the private password, severly breaking the functionality of the protocol.

Proof of Concept (or Proof of code): The below test case shows how anyone could read the password directly from the blockchain. We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

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 get the following password

```
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 stored password. However, you're also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with this decryption key.

[H-2] `PasswordStore::setPassword` has no access controls, meaning a non-owner could change the password

Description:

> (`PasswordStore::setPassword` is a function to set the private state variable `PasswordStore::s_password`. Right now only the owner can access to his password with the following getter `PasswordStore::setPassword`, so it is important to apply the same level of access control to set the user's password.) – MINE

The `PasswordStore::setPassword` function is set to be an `external` function, however the purpose of the smart contract and function's natspec indicate that `This function allows only the owner to set a new password.`

```
    /*
     * @notice This function allows only the owner to set a new
         password.
     * @param newPassword The new password to set.
     */
    // AUDIT access control not applying onlyOwner
    // Attack vector: missing access control
    function setPassword(string memory newPassword) external {
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Anyone can set/change the password of the contract, severly breaking the contract intented 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 != s_owner) {
    revert PasswordStore__NotOwner();
}
```

Informational

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist, causing the natspec to be incorrect.

Description:

```
/*
 * @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 says it should be `getPassword(string)`.

Impact: Natspec is incorrect.

Recommended Mitigation: Remove the icorrect natspec line.

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