Protocol Audit Report

Version 1.0

Tigress

August 9, 2024

Protocol Audit Report

Tigress

March 7, 2023

Prepared by: Tigress Lead Security Researcher: - Himanshi Vyas

Table of Contents

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, and is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Disclaimer

The Tigress 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 described in this document correspond the following commit hash:

Scope

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

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.

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	2
Total	4

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 have shown one such method of reading any data off chain below.

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

Proof of Concept:(Proof of Code)

The below test case shows how we can read 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

```
cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1 --rpc-url
    http://127.0.0.1:8545
```

you will get an output that looks like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

parsing that hex to a string:

```
ast parse-bytes32-string 0
    x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

```
myPassword
```

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and store the encrypted password on-chain. This would require the user to remember another password that is off-chain to decrypt the password. However, you'd also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password.

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

Description: The sole purpose of the function `PasswordStore::setPassword` was to make sure that the owner only sets the passwords, but the function is marked as external and no access control is provided anyone can set the password on behalf of the owner and get access to the new password

```
function setPassword(string memory newPassword) external {
@>    // @audit - There are no access controls
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Anyone can set or change the password, severly breaking the sanity and functionality of the contract

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 = "user@123";
        passwordStore.setPassword(expectedPassword);
        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();
        assertEq(expectedPassword, actualPassword);
    }
```

Recommended Mitigation: We can mitigate this problem by simply adding a access control conditional to the function, which checks that the address calling this function is that of the owner.

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

Informational

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter that dosen't exists, 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 `PasswordStore::getPassword` function signature is `getPassword()` while the natspec says it should be `getPassword(string)`.

Impact: The natspec is incorrect

Recommended Mitigation: Remove the incorrect natspec line

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

[I-2] Optimize Gas Efficiency by Making `PasswordStore::s_owner` Immutable

Description: The `PasswordStore::s_owner` variable should be declared as `immutable` to improve gas efficiency. Since the owner is set only once in the constructor, marking it as `immutable` is appropriate and will result in lower gas costs.

```
1 @>   address private s_owner;
```

Impact: Changing `PasswordStore::s_owner` to an immutable variable will reduce gas costs for transactions.

Proof of Concept: We compared the transaction costs using the Remix IDE:

1. Non-immutable transaction cost: 456,280 gas
2. Immutable transaction cost: 433,563 gas

This demonstrates a clear reduction in gas usage.

Recommended Mitigation: Declare the `PasswordStore::s_owner` variable as immutable to optimize gas efficiency.