PasswordStore Initial Audit Report

Version 0.1

Feb 20, 2024

PasswordStore Audit Report

ARAVIND

Feb 20, 2024

PasswordStore Audit Report

Prepared by: ARAVIND Lead Auditors:

- ARAVIND

Assisting Auditors:

- None

Table of contents

See table

Risk Classification

	Impact		
	High	Medium	Low

	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

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

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

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.

Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract.

Executive Summary

Issues found

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

Findings

High

[H-1] Passwords stored on-chain are visable to anyone, not matter solidity variable visibility

Description: All data stored on-chain is visible to anyone, and can be read directly from the

blockchain.

ThePasswordStore::s_passwordvariableisintendedtobeaprivatevariable,andonlyaccessed through the PasswordStore::getPassword function, which is intended to be only called by the owner of the contract.

However, anyone can direclty read this using any number of off chain methodologies

Impact: The password is not private.

Proof of Concept: 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'll get an output that looks like this:

0x6d7950617373776f726400000000000000000000000000000000000000000014 You can then parse that hex to a string with:

```
cast parse-bytes32-string 0
    x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

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 on-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] PasswordStore::setPassword is callable by anyone

Description: The PasswordStore::setPassword function is set to be an external function, howeverthenatspecofthefunctionandoverallpurposeofthesmartcontractisthatThis function allows only the owner to set a **new** password.

```
1        function setPassword(string memory newPassword) external {
2  @>            // @audit - There are no access controls here
3                s_password = newPassword;
4                emit SetNetPassword();
5        }
```

Impact: Anyone can set/change the password of the contract.

Proof of Concept:

Add the following to the PasswordStore.t.sol test suite.

```
myPassword
```

```
function test_anyone_can_set_password(address randomAddress) public {
        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 modifier to the setPassword function.

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