GINTOKI

PasswordStore Audit Report

Version 1.0

Gintoki Sakata

July 28, 2025

PasswordStore Audit Report

Gintoki Sakata

Jully 29, 2025

Prepared by: Gintoki Sakata Lead Auditors: - Gintoki Sakata

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 Gintoki Sakata 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 him 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:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

- Owner: Is the only one who should be able to set and access the password.
- Outsiders: No one else should be able to set or read 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	1
Info	1
Gas Optimizations	0
Total	0

Findings

High

[H-1] storing `s_password` on-chain makes it visible to anyone

Description:

All data stored on-chain is visible to anyone, hence storing the varibles onchain is not so secure . `PasswordStore::s_password` is intended to be a private varible - meaning only authorized entity should be able to view and only accessed through `PasswordStore::getPassword` function, which is intended to be only called by owner of the contract.

Impact:

Anyone can read the private password , severly breaking the functionality of the contract

Proof of Concept:

Below Test cases shows anyone can read password directly from blockchain

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 `PasswordStore::s_password`

```
cast storage <CONTRACT_ADDRESS> 1 --rpc-url <LOCAL_ANVIL_URL>
```

You'll get an output like : `0x6d7950617373776f726400000000000000000000000000000000000000000000`

4. Parse Hex output to string

```
cast parse-bytes32-string 0
    x6d7950617373776f72640000000000000000000000000000000000000000000014
```

and We get output as :

```
myPassword
```

Recommended Mitigation:

- Don't store plain passwords on-chain — even if marked private, anyone can still read them.
- What you should do instead is store a hashed version of the password using something like keccak256, and then compare hashes during verification.
- If the password is truly sensitive, it's best to keep it off-chain entirely and use secure methods like signed messages or zero-knowledge proofs for authentication

Likelihood & Impact:

- Impact: High
- Likelihood: High
- Severity: High

[H-2] `PasswordStore::setPassword` has no access control , resulting in a non-owner can change the password

Description: The `PasswordStore::setPassword` is set to an `external` function , however , the natspec notation clearly states that `This function allows only the owner to set a new password`. But there is no condition for checking if the user changing or setting the password is actually `Owner` or not . Resulting in Access Control exploit.

```
    function setPassword(string memory newPassword) external {
>>      // @audit -> There is no check for Owner verification
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Anyone can change / Set the Password , severly breaking the protocol functionality.

Proof of Concept: Add following test in `PasswordStore.t.sol` test Suite

Code

```
    function test_anyone_can_set_password(address randomAddress) public {
        vm.assume(randomAddress != owner);
        vm.prank(randomAddress); // calling via randomeAddress / non-owner
        string memory expectedPassword = "myNewPassword";
        passwordStore.setPassword(expectedPassword); // setting a new password

        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword(); // retreiving password via owner address

        assertEq(expectedPassword , actualPassword); // results in randomAddress (non-owner) can set password
    }
```

Recommended Mitigation: Add Access Control to `PasswordStore::setPassword` as below :

```
    ...
    function setPassword(string memory newPassword) external {
        // add access control here
        if (msg.sender != s_owner) {
            revert PasswordStore_NotOwner();
        }
        s_password = newPassword;
        emit SetNetPassword();
    }
    ...
```

Likelihood & Impact:

- Impact: High
- Likelihood: High
- Severity: High

Informational

[I-1] `PasswordStorage::getPassword` indicates a natspec that doesn't exist.

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` function signature is `getPassword()` while natspec notation say it should be `getPassword(string)`

Impact: natspec notation is incorrectb

Recommended Mitigation: Remove incorrect natspec line

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

Likelihood & Impact:

- Impact: High
- Likelihood: None
- Severity: Informational / Gas.