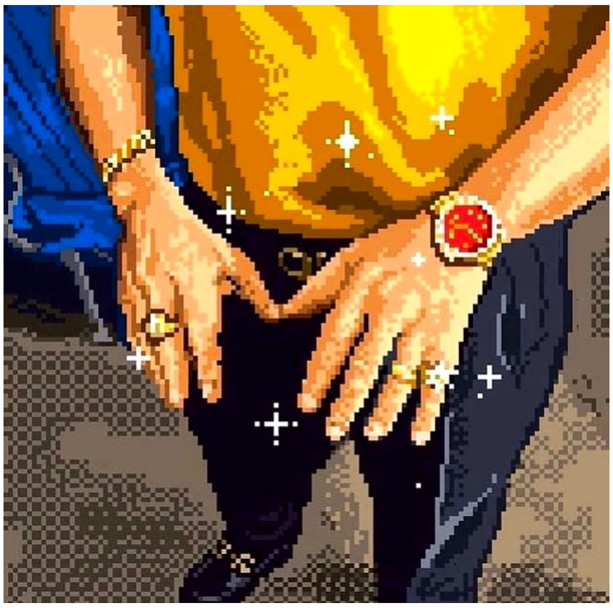

Protocol Audit Report

Version 1.0

Syber

April 15, 2025

Protocol Audit Report

0xsyber88

April 15, 2025

Prepared by: 0xsyber88

Table of Contents

Protocol Summary

A smart contract application for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Disclaimer

I make 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

Audit Details

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.

Executive Summary

The audit was conducted on a simple smart contract implementation designed for password storage. The codebase itself was straightforward, with no complex or obscure logic. However, two high-severity issues were identified that significantly undermine the intended functionality of the protocol.

`[H-1] Password Visibility On-Chain`: The contract stores passwords directly on-chain in plaintext, making them accessible to anyone who queries the blockchain. This fundamentally breaks the privacy guarantees expected from a password storage system.

`[H-2] Missing Access Controls`: The setPassword function lacks proper access restrictions, allowing any external user to overwrite the stored password. This opens the protocol to unauthorized modifications.

Additionally, one informational issue was found related to an incorrect NatSpec comment in the getPassword function, which does not affect functionality but may cause confusion during development or maintenance.

In conclusion, while the codebase was easy to navigate, critical security flaws exist that require rethinking the protocol's architecture to ensure data privacy and correct access management. Immediate mitigations are recommended to address these issues before deployment to a production environment.

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.

Impact: Anyone can read the pricate password, severly 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 a locally running chain

```
make anvil
```

2. Deploy the contract to the chain.

```
make deploy
```

3. Run the storage tool

We use `1` becuase 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:

```
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 password. However, you'd also likely want to remove the view functin as you would'nt want the user to accidenatally send a transactoin with the password that decrypts your password.

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

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

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

Impact: Anyone can set/change the password of the contrac, severly breaking the contract intended 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 coniditional to the `setPassowrd` function.

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

Informational

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't existJ

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()` which 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.
```