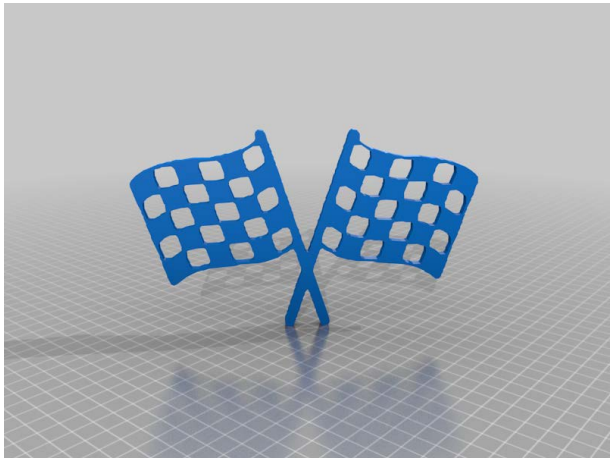

Protocol Audit Report

Version 1.0

Samuel Troy Dominguez

January 12, 2024

Protocol Audit Report

Samuel Troy Dominguez

January 12, 2024

Prepared by: [Samuel Troy Dominguez] (https://github.com/samtdomi) Lead Security Reseacher:

Table of Contents

Protocol Summary

As stated by the protocol, "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, Samuel Troy Dominguez, have genuinely given my greatest honest attempt at fidning as many vulnerabilities as possible in the code within the given time period, but hold no responsibility for the findings in this document and/or their implementations. A security audit completed by myself 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

The findings described in this documetn correspond to the following commit hash:

```
Commit Hash: 2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

- Owner: The address of the user who has the authority to - set, change, and read - the password.
- Outsiders: Everyone except the Owner. Only the owner should be able to set and read the password.
-

Executive Summary

- The smart contracts within the scope of the audit were examined methodically, beginning with the documentation of the project, to the code itself. With the overall purpose of the contract in mind, vulnerabilities regarding privacy were found, which if not fixed, could undermine the integrity of the entire project.
-
- ** I spent the max amount of time agreed upon for the audit, examining the code and attempting to find ways to exploit and attack the project. Upon finding a vulnerability, I showed a way of how the exploit would take place and the impact it would have. The remaining time was spent creating mitigation tactics and ways to fix the vulnerability to make the security of the project stronger.

Issues found

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

Severity	Number of issues found
Total	3

Findings

High

[H-1] Storing the password on-chain makes it visible to anyone, even though its marked private

Description: The `PasswordStore::s_password` variable is inteded to be private and unreadable and unretrievable by anyone but the Owner of the password, only after calling the `PasswordStore::getPassword` function - that is why the variable is marked with the solidity visibility keyword "private" as the way of ensuring the variable remains private in storage. This is does not make the variable unreadble and unretreivable though, all data stored on the blockchain can be read by anyone, it is public information. The "private" keyword only means that other contracts cannot read the variable.

Below is an example / method of reading any data off chain

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

Proof of Concept: (proof of code)

The below test case shows how anyone can read the password directly from the blockchain

1. Create a locally running chain

```
anvil
```

2. Deploy the contract to the local anvil chain, using the deploy script

```
make deploy
```

- Copy the returned address of the deployed `PasswordStore` contract

3. Run the storage tool using the power of Foundry's Cast to retreive the bytes version of `s_password`

- We use `1` because that is the storage slot of `s_password` in the contract

```
cast storage <PASSWORDSTORE_ADDRESS_HERE> 1 --rpc-url http://
    127.0.0.1:8545
```

- you will get an output that looks like this, which is the bytes version of `s_password`:

0x6d7950617373776f726400000000000000000000000000000000000000000014

4. Parse the bytes version of `s_password` to a string

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

5. Literally view the saved password

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, adn then store the encrypted password on-chain. This would require the user to remember another password off-chain to decrypt the password. However, youd also likely want to remove the view function as you wouldnt want the user to accidentally send a transaction with the password that decrypts your password.

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

Description: `PasswordStore::setPassword` function has its visibility set to `external` with the purpose of giving the owner the option to call the function and change their password. The natspec for the function as well as the overall purpose of the smart contract is to ensure: `This function allows only the owner to set a new password`. However, this function does not ahve any access controls preventing any non-owner from calling the function and changing the password - even if they are not the owner.

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

Impact: Anyone that calls the function will be able to input their own password and successfully change the current password. The owner will have their password changed by anyone that calls the function.

Proof of Concept: The following is a fuzz test added to `PasswordStore.t.sol` test file that will create a new address and have that new address (which is not the owner) - call `PasswordStore::setPassword` and change the password to `broken`. This process called `fuzzing` will run the

describe test scenario over and over again, craeting and using a different address as the caller for each iteration (compunding the validity of the exploit). The original password that is set upon deployment is `myPassword`.

```
    function testFuzzAnyoneCanCallSetPassword(
        address randomAddress
    ) public {
        vm.assume(randomAddress != owner);

        vm.startPrank(randomAddress);
        string memory expectedPassword = "broken";
        passwordStore.setPassword(expectedPassword);
        vm.stopPrank();

        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();

        console.log(
            "If the password was changed by a non-owner it will
               read 'broken', the new password is: ",
            actualPassword
        );
        assertEq(actualPassword, expectedPassword);
    }
```

1. to run the test, open the terminal and write:

```
forge test --match-test testFuzzAnyoneCanCallSetPassword
```

Recommended Mitigation: Add an access control condition to the `setPassword` function.

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

Informational

[N-1] `PasswordStore::getPassword` natspec documents a paramater in the function that is non-existent, making the natspec documentation for the function wrong

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()` - but the natspec says that it should be `getPassword(string)`.

Impact: The natspec and documentation is incorrect

Recommended Mitigation: Remove the incorrect natspec line, the function does not have a paramter.

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