PasswordStore Initial Audit Report

Version 0.1

by Saniksin

November 6, 2024

PasswordStore Audit Report

Prepared by: Saniksin

Lead Auditors:

- Saniksin

Assisting Auditors:

- None

Table of Contents

See table

About Saniksin

I am Saniksin, a Solidity developer and security auditor. Currently, I am engaged in software engineering studies and have experience in smart contract auditing, particularly focusing on ensuring the safety of Solidity-based implementations.

Disclaimer

The Saniksin 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

Audit Details

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

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

Original audited code repository

This repository contains the original code of the audited contract, along with comprehensive documentation and test cases used during the audit process.

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

Protocol Summary

PasswordStore is a protocol dedicated to the storage and retrieval of a user's passwords. The protocol is designed to be used by a single user and is not intended for 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	1
Info	1
Gas Optimizations	0
Total	4

Findings

High-Risk Findings

H-1: Password Visibility

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, accessible only through the `PasswordStore::getPassword` function, which is intended to be only called by the owner of the contract.

However, anyone can directly read this using various off-chain methods.

Impact: The password is not private.

Proof of Concept: The below test case shows how anyone could read the password directly from the blockchain. I 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

I 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:

```
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 function as you wouldn't want the user to accidentally send a transaction with the password that decrypts their password.

H-2: Password Setting Access

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

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

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

Proof of Concept: Add the following to the `PasswordStore.t.sol` test suite.

```
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();
}
```

Low-Risk Findings

L-1: Initialization Timeframe Vulnerability

Summary The PasswordStore contract exhibits an initialization timeframe vulnerability. This means that there is a period between contract deployment and the explicit call to setPassword during which the password remains in its default state. It's essential to note that even after addressing this issue, the password's public visibility on the blockchain cannot be entirely mitigated, as blockchain data is inherently public, as already stated in the "Storing password in blockchain" vulnerability.

Vulnerability Details The contract does not set the password during its construction (in the constructor). As a result, when the contract is initially deployed, the password remains uninitialized, taking on the default value for a string, which is an empty string.

Impact The impact of this vulnerability is that during the initialization timeframe, the contract's password is left empty, potentially exposing the contract to unauthorized access or unintended behavior.

Tools Used No tools used. It was discovered through manual inspection of the contract.

Recommendations To mitigate the initialization timeframe vulnerability, consider setting a password value during the contract's deployment (in the constructor). This initial value can be passed in the constructor parameters.

Info Findings

I-1: Incorrect NatSpec for `getPassword`

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 NatSpec for the function `PasswordStore::getPassword` indicates it should have a parameter with the signature `getPassword(string)`. However, the actual function signature is `getPassword()`.

Impact: The NatSpec is incorrect.

Recommended Mitigation: Remove the incorrect NatSpec line.

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