PasswordStore Audit Report

Version 1.0

PandaBear30

January 4, 2025

PasswordStore

PandaBear30

January 4, 2025

Prepared by: PandaBear30

Table of Contents

Protocol Summary

A protocol 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

The PandaBear30 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

A smart contract applicatoin 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.

The findings of this audit is based on the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Roles

- Owner: The user who can set the password and read the password.
- Outsiders: No one else should be able to set or read the password.

Executive Summary

We spent 3 hours auditing this protocol.

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 the password on-chain makes it visible to everyone

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 `PasswordStore::getPassword` funciton, which is intended to only be called by the owner.

We show one method of reading the data from the blockchain below.

Impact: The private password is not actually 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:

```
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 your password.

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

Description: The `PasswordStore::setPassword` function is set to be an `external` function without any access control, however, the natspec and overall purpose of the smart contract is that `This function allows only the owner to set a new password.`

```
1
2       function setPassword(string memory newPassword) external {
3           // @audit - There are no access control
4           s_password = newPassword;
5           emit SetNetPassword();
6       }
```

Impact: Anyone can/set change the password of the contract, severely breaking the contract's functionality

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

Code

```
1
2       function test_anyone_can_set_password(address randomAddress) public
            {
3           vm.assume(randomAddress != owner);
4           vm.prank(randomAddress);
5           string memory expectedPassword = "myNewPassword";
6           passwordStore.setPassword(expectedPassword);
7
8           vm.prank(owner);
9           string memory actualPassword = passwordStore.getPassword();
10          assertEq(actualPassword, expectedPassword);
11
12      }
```

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

```
1
2       if(msg.sender != s_owner){
3           revert PasswordStore__NotOwner();
4       }
```

Informational

[I-1] newPassword parameter as indicated by the natspec does not exist in the `PasswordStore::getPassword` function

Description:

```
1
2       /*
3        * @notice This allows only the owner to retrieve the password.
4    @>  * @param newPassword The new password to set.
5       */
```

```
6       function getPassword() external view returns (string memory) {
7           if (msg.sender != s_owner) {
8               revert PasswordStore__NotOwner();
9           }
10          return s_password;
11      }
```

The `PasswordStore::getPassword` function's natspec includes `newPassword` parameter which does not match the actual function which takes no parameters.

Impact: The natspec is incorrect

Recommended Mitigation: Remove the incorrect natspec line.

```
1
2   - * @param newPassword The new password to set.
```