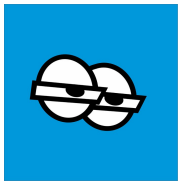

PasswordStore Audit Report

Version 1.0

Boris Kolev

April 14, 2024

Protocol Audit Report

Boris Kolev

April 14, 2024

Prepared by: Boris Kolev

Lead Auditors:

- Boris Kolev

Table of Contents

Protocol Summary

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.

Disclaimer

Boris Kolev 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
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

Scope

- Commit hash: 2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
- In scope:

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

- Solc version: 0.8.18
- Chain(s) to deploy contract to: Etherium

Roles

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

Executive Summary

The audit resulted in two `high` findings and one `informational` finding.

The audit took 1 day of reviwing, utilizing `manual review`, `fuzz testing`, etc.

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Informational	1
Gas	0

Findings

High

[H-1] No ownership check on setPassword, allows password to be overwritten

Description There is no ownership check in the `setPassword` method of the contract, which allows external actors to set new passwords, overwriting the owner's initial password.

```
1 function setPassword(string memory newPassword) external {
2 @>  // No access control
3     s_password = newPassword;
4     emit SetNetPassword();
5 }
```

Impact The owner will lose his stored passwords, leading to severe logic discrepency between what the `PasswordStore` must do, and actualy does.

Proof of Concept This data breach mechanism is shown in test `test_anyone_can_set_password` , where after the owner sets his password, an external address overwrites it.

Test proof

```
function test_anyone_can_set_password(address randomAddress) public {
    string memory expectedPassword = "myNewPassword";
    vm.prank(randomAddress);
    passwordStore.setPassword(expectedPassword);
    vm.prank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
}
```

Recommended Mitigation The same `owner` check which is added to the `getPassword` method should be added to the `setPassword` one as well. This way we will ensure that only the owner can set passwords.

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

[H-2] Password security is compromised due to the fact that storage variables in Solidity are visible to all, regardless of visibility

Description All data stored on-chain is visible to anyone, and can be directly read from the blockchain. The `s_password` variable is ment to be private, only visible to the owner, which is not the case now.

Impact Anyone can read the stored variable, severly breaking the idea of the `PasswordStore` functionality.

Proof of Concept

1. The first thing we do is deploy the contract using the deploy script `DeployPasswordStore.s.sol`, where we set the `s_password` to `test`.

```
function run() public returns (PasswordStore) {
    vm.startBroadcast();
    PasswordStore passwordStore = new PasswordStore();
    passwordStore.setPassword("test");
    vm.stopBroadcast();
    return passwordStore;
}
```

2. Then we use `cast` to get the `s_password` storage variable.

```
cast storage 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512 1 --rpc-url
   http://127.0.0.1:8545
```

3. This in-turn, returns the `bytes32` representation of the storage variable:

```
0x7465737400000000000000000000000000000000000000000000000000000008
```

4. We then use the `parse-bytes32-string` functionality of `cast` to retrieve the set password:

```
cast parse-bytes32-string 0
    x7465737400000000000000000000000000000000000000000000000000000008
> test
```

Recommended Mitigation Storing the raw versions of passwords in solidity contracts is a VERY bad idea. The easiest workaround to this would be to use an off-chain storage, which will hold the password and will be queried here in the contract. This, however, has its drawbacks, as the user will need to remember off-chain data in order to access the now external data. Another solution will be to store the `keccak256` version of the password here, and allow the user to check if the password he enters has the same hash as the one he initially stored. This changes the general idea of the protocol, and is up to the devs to decide if it is a feasible solution.

Informational

[I-1] Documentation for `getPassword` indicates a parameter that does not exist, causing the natspec to be incorrect

Description The `getPassword` method signature is `getPassword()` while the natspec says that it should be `getPassword(string)`.

```
    /*
     * @notice This allows only the owner to retrieve the password.
@>   * @param newPassword The new password to set.
     */
function getPassword() external view returns (string memory)
```

Impact The natspec is incorrect.

Recommended Mitigation Remove the wrong natspec.

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