Protocol Audit Report

Will Stansill

August 8th, 2024

Protocol Audit Report

Version 1.0

Will Stansill

August 8th, 2024

Protocol Audit Report

Will Stansill

August 8th, 2024

Protocol Audit Report

Prepared by: Will Stansill
Lead Auditors:

- Will Stansill

Table of Contents

Protocol Summary

The `PasswordStore` protocol is designed to securely store and manage a user's password on the Ethereum blockchain. The protocol is intended for single-user

interaction, where only the contract owner can set or retrieve the stored password. The key functionalities of the protocol include:

1. **Password Storage**: The contract allows the owner to store a password on-chain. However, due to the transparent nature of blockchain data, this password, despite being marked as private within the contract, can be accessed by anyone using off-chain methods.
2. **Password Retrieval**: Only the owner of the contract can retrieve the stored password through a dedicated function. This retrieval is protected by an access control mechanism that restricts access to the owner.
3. **Password Updates**: The protocol supports updating the stored password. This operation should be restricted to the owner, but in the current implementation, it lacks the necessary access controls, allowing any user to modify the password.

The protocol is simplistic in its design, focusing on core functionality without advanced security measures, which leads to significant vulnerabilities as highlighted in the audit findings.

Disclaimer

Will Stansill 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 Will Stansill 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

Scope

The findings described in this document correspond to the following commit hash: `2e8f81e263b3a9d18fab4fb5c46805ffc10a9990`.

Files in Scope:

- `src/PasswordStore.sol`

Roles

- **Owner**: Is the only one who should be able to set and access the password.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	1
Info	1
Gas Optimizations	0
Total	4

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.

We show one such method of reading any data off-chain below.

Impact: Anyone can read from the private password, severely 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 with:

   ```
   anvil
   ```

2. Deploy the contract to the chain:

   ```
   make deploy
   ```

3. Run the storage tool:

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

   You'll get a big string like:

   ```
   0x6d7950617373776f726400000000000000000000000000000000000000000014
   ```

   This string can be parsed with:

   ```
   cast parse-bytes32-string 0x6d7950617373776f726400000000000000000000000000000000000000000000
   ```

   Finally, you will get the password:

   ```
   myPassword
   ```

Recommended Mitigation: Encryption needs to be done off-chain and then proceed by storing the encrypted password back on-chain.

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

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 is that this function allows only the owner to set a new password.

Impact: Anyone can set/change the password of the contract, compromising the intended functionality.

Proof of Concept:

Add the following to the test file:

```
function test_anyone_can_set_password(address randomAddress) public {
    vm.assume(randomAddress != owner);
    vm.prank(randomAddress);
    string memory expectedPassword = "myPassword";
    passwordStore.setPassword(expectedPassword);

    vm.prank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
```

```
}
```

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

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

[L-1] Initialization Timeframe Vulnerability

Description: 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.

Impact During this initialization timeframe, the contract's password is effectively empty and can be considered a security gap.

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

[I-1] The getPassword natspec indicates a parameter that doesn't exist, causing the natspec to be incorrect

Description: The `getPassword` function signature is getPassword() but the NatSpec comments indicate a parameter that doesn't exist.

Impact: The NatSpec is incorrect.

Recommended Mitigation: Remove the incorrect NatSpec line:

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