PasswordStore Audit Report

Title: PasswordStore Audit

Author: Lionel Legacy

Date: June 26th 2025

Protocol Summary

This document details a security audit of a smart contract application designed for storing a password. The application intends to allow a user to store a password and then retrieve it later, with the expectation that others should not be able to access the password.

Disclaimer

The LIONEL LEGACY 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

We use the CodeHawks severity matrix to determine severity:

Impact	**Likelihood:High**	**Likelihood:Medium**	**Likelihood: Low**
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium/Low
Low	Medium	Medium/Low	Low

Audit Details

Scope

The audit focused on the following file:

- `./src/PasswordStore.sol`
- **Commit Hash:** `7d55682ddc4301a7b13ae9413095feffd9924566`

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

The overall architecture of the protocol should be re-examined as the blockchain is not ideal for storing sensitive data. Furthermore, critical access control issues and incorrect NatSpec documentation were identified, which should be addressed immediately.

Findings

This report details security vulnerabilities and recommendations for the `PasswordStore` smart contract.

[S-1] Missing Access Control in `PasswordStore::setPassword`

Severity: High

Description: The `setPassword` function in the `PasswordStore` contract lacks proper access control. This means that any external account can call this function and arbitrarily change the stored password, not just the owner of the contract.

Impact: This vulnerability severely compromises the security and integrity of the protocol. A malicious actor could change the password stored in the contract at any time, leading to unauthorized modification of sensitive data and complete loss of control for the legitimate owner.

Proof of Concept: The following Solidity code from the `setPassword` function, without any sender validation, demonstrates the vulnerability:

```
function setPassword(string memory newPassword) external {
    s_password = newPassword;
    emit SetNetPassword();
}
```

The following Forge test demonstrates that a non-owner address can successfully call `setPassword` and override the existing password:

```
function testAnyoneCanSetPassword(address randomAddress) public {
    vm.assume(randomAddress != owner); // Ensure randomAddress is not the
owner
```

```
    vm.prank(randomAddress);          // Impersonate a random address
    string memory expectedPassword = "myNewPassword";
    passwordStore.setPassword(expectedPassword); // Non-owner sets the
password

    vm.prank(owner); // Switch back to owner to read (or any address can read
getPassword after fix)
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword); // Assert that the password
was changed by the non-owner
}
```

Recommended Mitigation: Add an access control conditional (e.g., an `onlyOwner` check, leveraging the `s_owner` state variable) to the `setPassword` function to ensure that only the contract owner can modify the password.

```
function setPassword(string memory newPassword) external {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    s_password = newPassword;
    emit SetNetPassword();
}
```

[S-2] Variables On-chain are Accessible to Everyone and Thus the Password is Not Safe

Severity: High

Description: All data stored on a public blockchain, regardless of its `private` or `internal` visibility specifiers in Solidity, is publicly accessible to anyone. The `private` keyword only restricts direct function calls from other *contracts*; it does **not** hide the data from external observers or tools that can query the blockchain state directly.

Impact: The `s_password` state variable, intended to be private, can be read by anyone by directly inspecting the contract's storage. This defeats the core purpose of a "PasswordStore" contract, as the password is not secure and can be compromised. Sensitive information like passwords should never be stored unencrypted on-chain.

Proof of Concept:

1. **Run local Anvil chain:**
2. `anvil`

3. **Compile contract:**
4. `forge build`

5. **Deploy to Anvil:** (Replace `<private-key>` with an Anvil account's private key)

```
forge create src/PasswordStore.sol:PasswordStore \
    --rpc-url http://127.0.0.1:8545 \
    --private-key <private-key> \
    --broadcast
```

   Note the deployed contract address (e.g., `0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512`).

10. **Set password:** (Replace `<contract-address>` and `<anvil-key>`)

```
cast send <contract-address> "setPassword(string)" "MyAwesomeSecret" \
    --rpc-url http://127.0.0.1:8545 \
    --private-key <anvil-key>
```

    Wait for the transaction to confirm.

14. **Get raw bytes from storage slot 1:** (This slot holds `s_password`. Assuming `s_owner` is in slot 0, `s_password` will be in slot 1).

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

    This will output a `bytes32` value (e.g., `0x4d79417765736f6d6553656372657400000000000000000000000000000020` for "MyAwesomeSecret", where `0x20` is `2 * length` for short strings).

16. **Read password:** Use `cast parse-bytes32-string` to decode the output from the previous step.

```
cast parse-bytes32-string <output-from-step-5>
```

    The output will be `"MyAwesomeSecret"` (or whatever password was set), proving direct readability of the "private" variable.

Recommended Mitigation: The core functionality of storing truly private passwords directly on a public blockchain is fundamentally incompatible with the transparent and immutable nature of blockchain technology. Sensitive information like passwords should never be stored unencrypted on-chain.

Instead, consider alternative approaches such as:

- Encrypting the password off-chain using a key known only to the user, and then storing only the encrypted ciphertext on-chain.
- Storing only a hash of the password (e.g., `keccak256(password)`) on-chain for verification purposes, never the password itself.
- Re-evaluating if a blockchain is the appropriate technology for storing such sensitive, private data.

[S-3] Incorrect Natspec Parameter in `PasswordStore::getPassword`

Severity: Informational

Description: The NatSpec documentation for the `getPassword` function incorrectly includes a `@param newPassword` tag. This function does not take any parameters.

Impact: Incorrect NatSpec can lead to confusion for developers interacting with the contract, misinterpretation of function parameters, and degraded documentation quality. While not a security vulnerability, it hinders code readability, maintainability, and accurate automatic documentation generation.

Location: `PasswordStore::getPassword` function definition.

Proof of Concept: The erroneous line is highlighted below in the contract's source code:

```
/*
 * @notice This allows only the owner to retrieve the password.
 * @param newPassword The new password to set. // <--- This line is incorrect
 */
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

Recommended Mitigation: Remove the incorrect `@param newPassword` tag from the `getPassword` function's NatSpec. The `getPassword` function does not take any parameters.

```
/*
 * @notice This allows only the owner to retrieve the password.
 */
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

Findings Summary

Severity	Count
High	2
Medium	0
Low	0

Informational	1
Gas	0