PasswordStore Initial Audit Report

Version: Initial Audit

Hash Horizon

August 14, 2025

PasswordStore Audit Report

What Fate

August 14, 2025

PasswordStore Audit Report

Prepared by: What Fate Lead Auditors:

- WhatFate

Assisting Auditors:

- None

Table of contents

See table

About WhatFate

Hi, my name is Danila. I'm a beginner smart contract auditor with hands-on experience in smart contract development using Solidity and Foundry. I've been involved in the blockchain space for over two years, continuously learning and building secure, efficient solutions.

Disclaimer

The Hash Horizon team makes every 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 Solidity implementation of the contracts.

Risk Classification

Likelihood	Impact	High	Medium	Low
High		H	H/M	M
Medium		H/M	M	M/L
Low		M	M/L	L

The following risk classification matrix is used throughout this report to evaluate findings.

Audit Details

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

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

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

Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords. The protocol is designed to be used by a single user, and is not designed to be used by 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

The PasswordStore contract has severe architectural flaws that compromise its primary purpose — secure password storage. Two high-severity issues were identified, both of which allow unauthorized access and modification of stored passwords. Without fundamental design changes, the contract cannot securely fulfill its intended functionality.

Issues found

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

Findings

High

[H-1] Passwords stored on-chain are visible to anyone, regardless of Solidity variable 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, and only accessed 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 any number of off-chain methodologies

Impact: The password is not 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: In Solidity, the **`private`** keyword only restricts access to the variable *within the contract code* and does **not** make the variable's value invisible on-chain. All contract storage is publicly readable through node RPC calls or direct blockchain inspection (Solidity Docs – Visibility and Getter Functions).

To mitigate this, store only an *encrypted hash* of the password on-chain. Encryption should be performed `off-chain`, and the key to decrypt must be managed securely by the user. Additionally, remove public getter functions that could reveal the password or its hash, and avoid sending sensitive data in transactions.

[H-2] `PasswordStore::setPassword` is callable by anyone

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.`

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

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: Implement proper access control to restrict password updates to the contract owner. Instead of manually checking `msg.sender`, it is recommended to use a battle-tested library like OpenZeppelin's `Ownable` or `AccessControl` for scalable and maintainable permission management.

Example using `Ownable`:

```
    import "@openzeppelin/contracts/access/Ownable.sol";

    contract PasswordStore is Ownable {
        function setPassword(string memory newPassword) external
            onlyOwner {
            s_password = newPassword;
            emit SetNewPassword();
        }
    }
```

This ensures only the deployer (or a transferred owner) can set the password, preventing unauthorized changes.

Low Risk Findings

L-01. Initialization Timeframe Vulnerability

Submitted by dianivanov.

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-PasswordStore/blob/main/src/PasswordStore.sol

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.

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

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.

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

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.
```