PasswordStore Initial Audit Report

Version 0.1

Special for my GitHub

October 19, 2025

PasswordStore Audit Report

Putfor

October 19, 2025

Prepared by: Putfor

Table of Contents

Protocol Summary

A smart contract application 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

Putfor 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 research by Putfor 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

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

Audit Details

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

```
be47a4de6d0ccb6621e075caee60409c3f7eac28
```

Scope

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

- Solc Version: 0.8.18
- Chain(s) to deploy contract to: Ethereum

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 security research was conducted using a combination of manual code review and automated testing with the Foundry framework. Manual analysis was performed to identify logical flaws, access control issues, and deviations from the intended design. Foundry was used to reproduce and validate potential vulnerabilities through local deployment and testing on a simulated blockchain environment.

This hybrid approach ensured comprehensive coverage of both code-level and behavioral security risks within the smart contract.

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] On-chain storage is publicly readable — private variables can be accessed off-chain

Description: All variables stored on the blockchain can be read one way or another. The `PasswordStore::s_password` variable is intended to store the user's password. However, despite being declared as private, its contents can still be read through off-chain methods.

Impact: Sensitive user data stored in the contract becomes publicly accessible, violating user privacy and the project's intended security model.

Proof of Concept: The issue can be demonstrated using Foundry tools and a local Anvil blockchain instance:

1. Start a local Anvil node (ensure you are in the project repository so the Makefile commands are available):

```
make anvil
```

2. Deploy the contract instance to the local blockchain:

```
make deploy
```

3. Using Foundry's `cast` tool, read the storage slot that contains the `s_password` variable (slot 1):

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

The command returns the raw bytes representation of the stored password:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

4. Decode the bytes to a human-readable string using `cast`:

```
cast parse-bytes32-string 0
    x6d7950617373776f726400000000000000000000000000000000000000000014
```

The output reveals the original password stored in the contract:

```
myPassword
```

Recommended Mitigation: Do not store plaintext passwords or other secrets on-chain. Store only non-reversible commitments (e.g. `keccak256(password || salt)`) or, preferably, perform password verification off-chain; alternatively use signature-based authentication so users prove ownership with cryptographic signatures instead of sending passwords.

Ensure clients apply a strong KDF (scrypt/argon2) and per-user salts before producing any on-chain commitment, and never include raw passwords in transaction calldata or contract storage.

[H-2] `PasswordStore::setPassword()` is callable by anyone — missing ownership check allows unauthorized password updates

Description: The `PasswordStore::setPassword(string memory newPassword)` function is documented as owner-only but contains no ownership modifier or internal msg.sender check. The function unconditionally writes `s_password = newPassword` and `emits SetNewPassword()`. As implemented, any external account can call `setPassword()` and overwrite the stored password.

Impact: Any external user can change the stored password, breaking the intended access control. This compromises the integrity of the contract's state and may allow attackers to overwrite or reset sensitive user data.

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

Code

```
function test_poc_non_owner_set_password() public {
    // initiate the transaction from the non-owner attacker address
    vm.prank(attacker);
    string memory newPassword = "attackerPassword";
    // attacker attempts to set the password
    passwordStore.setPassword(newPassword);
    console.log("The attacker successfully set the password:",
       newPassword);
}
```

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

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

Informational

[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 `PasswordStore::getPassword()` function signature is `getPassword()` while the natspec say is should be `getPassword(string)`

Impact: The natspec is incorrect

Recommended Mitigation: Remove the incorrect natspec line

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