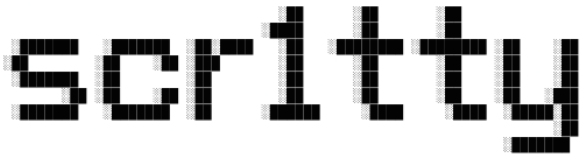

PasswordStore Audit Report

Version 1.0

scr1tty.io

December 31, 2025

PasswordStore Audit Report

scr1tty

December 31, 2025

Prepared by: scr1tty.io

Lead Auditors:

- scr1tty

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

scr1tty 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: 7d55682ddc4301a7b13ae9413095feffd9924566
- In 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.
- Outsides: No one else should be able to set or read the password.
```

Executive Summary

scr1tty were able to find vulnerabilities that leads to the full compromise of the protocol. These includes password disclosure and broken access control. Tests are done using multiple static analyzers (e.g., Slither, Aderyn, etc) to supplement the vulnerability research process. Manual code review were also done to ensure the protocol code were checked for robustness and maturity.

Issues found

Findings

High

[H-1] Password stored on-chain

Description: The value stored in PasswordStore::s_password is visible on-chain allowing any actor to retrieve the password.

Impact: A malicious actor can retrieve any users password.

Proof-of-Concept:

Steps

1. Deploy local instance using the Makefile provided.

```
$ make anvil   # Starts a local instance of the Ethereum Virtual
    Machine (EVM)
$ make deploy  # Deploys the PasswordStore contract
```

2. Copy the PasswordStore contract address. In this case: 0x5FbDB2315678afecb367f032d93F642f64

3. Retrieve the stored password using the Foundry cast tool

```
$ cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1 --rpc-url
    http://localhost:8545
0x6d7950617373776f726400000000000000000000000000000000000000000014 #
    Above command should return this value.
```

4. To convert the raw bytes into a readable string, use the following command,

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

5. This will output the password: `myPassword`

Recommended Mitigation: Instead of storing the password in plain text, consider storing its hashed version. Use a cryptographically secure hashing function like SHA-256, or better yet, a password-specific hashing algorithm like bcrypt or Argon2. This ensures that even if the contract is compromised, the password cannot be easily recovered.

[H-2] Missing Access Control on `PasswordStore::setPassword` allows non-owner accounts to modify password

Description: `PasswordStore::setPassword` lacks access control checks allowing anyone to modify the password stored in the contract.

Impact: Unauthorized actors may modify any user's password, allowing them to potentially impersonate the user or cause other security issues like data theft or unauthorized access to protected features.

Proof of Concept:

Steps

1. Place the provided `JavaScript` code snippet below on the test suite of the target contract.

```
  function test_non_owner_setting_password() public {
    // Added attacker address
    address attacker = address(0xdeadbeef);
    // Check if the expected password is same as the actual password
       set.
    vm.startPrank(owner);
    string memory expectedPassword = "myPassword";
    passwordStore.setPassword(expectedPassword); // set the password as
        the owner of the contract
    assertEq(passwordStore.getPassword(), expectedPassword);
    console.log("====== DEBUG ======");
    console.log("[i] Interacting as owner...");
```

```
11      console.log("[i] Password set by owner: ", passwordStore.
            getPassword());
12      vm.stopPrank();
13      // Try to set the password as an attacker
14      vm.startPrank(attacker);
15      string memory maliciousPassword = "hehehackedandpwned";
16      console.log("====== DEBUG ======");
17      console.log("[+] Interacting as attacker...");
18      console.log("[+] Modifying the password set from:",
            expectedPassword, "to:", maliciousPassword);
19      passwordStore.setPassword(maliciousPassword); // Modify the
            password
20      vm.stopPrank();
21      // Retrieve the password as the owner
22      vm.startPrank(owner);
23      console.log("====== DEBUG ======");
24      console.log("[i] Interacting as owner...");
25      console.log("[i] Password set: ", passwordStore.getPassword());
26      assertEq(passwordStore.getPassword(), maliciousPassword);
27      vm.stopPrank();
28    }
```

2. Run the test function using the Foundry tool, `forge` using this command:

- `forge test --mt test_non_owner_setting_password -vv`

```
1  # Output for the command above
2  Ran 1 test for test/PasswordStore.t.sol:PasswordStoreTest
3  [PASS] test_non_owner_setting_password() (gas: 49473)
4  Logs:
5  ====== DEBUG ======
6  [i] Interacting as owner...
7  [i] Password set by owner:  myPassword
8  ====== DEBUG ======
9  [+] Interacting as attacker...
10 [+] Modifying the password set from: myPassword to: hehehackedandpwned
11 ====== DEBUG ======
12 [i] Interacting as owner...
13 [i] Password set:  hehehackedandpwned
14
15 Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 349.54µs
       (111.69µs CPU time)
16 Ran 1 test suite in 3.27ms (349.54µs CPU time): 1 tests passed, 0
       failed, 0 skipped (1 total tests)
```

Recommended Mitigation: To fix this, inherit from OpenZeppelin's `Ownable` contract and apply the `onlyOwner` modifier to the `setPassword()` function:

```
1  pragma solidity 0.8.30;
2
```

```
3  // Import OpenZeppelin's Ownable contract
4  +    import "@openzeppelin/contracts/access/Ownable.sol";
5
6  // Contract inherits from OpenZeppelin's Ownable.sol
7  +    contract PasswordStore is Ownable {
8  ...
9
10 // Notice the onlyOwner modifier in the function.
11 -        function setPassword(string memory newPassword) external
12 +        function setPassword(string memory newPassword) external
       onlyOwner {
13             s_password = newPassword;
14             emit SetNewPassword();
15         }
16
17 // rest of the code
18       }
```

Informational

[I-1] Incorrect natspec for `PasswordStore::getPassword`

Description: The comment in `PasswordStore::getPassword (line 38)` contains misleading content which may reduce code readability for other developers / auditors.

Impact: Misleading documentation can result in incorrect assumptions about how the function behaves, which may lead to misunderstandings during audits, maintenance, or future development.

Proof of Concept: N/A

Recommended Mitigation: Update or remove the incorrect NatSpec comment to accurately reflect the function's purpose and behavior. Ensure that the description is clear, consistent with the implementation, and aligned with the project's documentation standards.

Gas

[G-1] `PasswordStore::s_owner` not declared as immutable increasing gas costs

Description: The `s_owner` state variable in `PasswordStore` is not declared as `immutable`, resulting in increased gas costs for both contract deployment and interaction. Immutable variables save gas because they are only assigned once during contract creation and do not require additional storage writes or reads.

Impact: The failure to declare `s_owner` as `immutable` increases gas costs associated with contract deployment and function execution. This inefficiency can result in **higher deployment costs**, **increased transaction fees**, and **slower contract interactions**, especially for contracts that have frequent calls to `s_owner` (e.g., in setter or getter functions). By using `immutable`, gas costs can be reduced, improving scalability and reducing the burden on users interacting with the contract.

Proof of Concept:

To illustrate the gas difference, the following tests were conducted using the `forge` tool with the `--gas-report` flag. Here's the comparison of gas usage before and after declaring `s_owner` as `immutable`:

- `forge test --gas-report test/PasswordStore.t.sol`

```
$  forge test --gas-report test/PasswordStore.t.sol
...

# Gas costs for tests with `s_owner` not set to immutable:
Ran 3 tests for test/PasswordStore.t.sol:PasswordStoreTest
[PASS] test_non_owner_reading_password_reverts() (gas: 13422)
[PASS] test_non_owner_setting_password() (gas: 141253)
[PASS] test_owner_can_set_password() (gas: 47241)

# Gas costs for tests with `s_owner` set to immutable:
Ran 3 tests for test/PasswordStore.t.sol:PasswordStoreTest
[PASS] test_non_owner_reading_password_reverts() (gas: 11288)
[PASS] test_non_owner_setting_password() (gas: 132711)
[PASS] test_owner_can_set_password() (gas: 45105)
```

Recommended Mitigation: Declare s_owner as `immutable` to reduce gas costs. This change will optimize the contract by storing the variable directly in the bytecode, thereby eliminating the need for additional storage operations during deployment and contract interaction.

```
// previous code here
contract PasswordStore {
  error PasswordStore__NotOwner();

- address private s_owner;
+ address private immutable s_owner; // gas-saving change
  ...
  // rest of the code here
}
```