



# **Password-Store Audit Report**

Version 1.0

*Ramprasad*

November 10, 2025

# Password-Store Audit Report

Ramprasad

March 16, 2025

Prepared by: Ramprasad

## Executive Summary

This audit was conducted to evaluate the security of the PasswordStore contract. The scope included a manual review of the code, static analysis, and fuzz testing to identify vulnerabilities. Key findings revealed critical security risks, including the exposure of sensitive data and inadequate access controls.

**Critical Security Risks Identified:** - On-chain password storage in plaintext, leading to potential data leakage. - Lack of access control in the `setPassword` function, allowing unauthorized users to modify passwords.

## Table of Contents

- Table of Contents
- Scope & Objectives
- Disclaimer
- Risk Classification
- Audit Details
- Findings
- Fix Review
- Conclusion

## Scope & Objectives

The audit focused on the following objectives: - **Testing Methods:** Manual code review, static analysis with tools like Slither, and fuzz testing. - **Contracts Audited:** - `PasswordStore.sol`: Responsible for storing and retrieving user passwords.

## Disclaimer

This audit was conducted by Ramprasad in an individual capacity. All efforts were made to identify vulnerabilities in the code within the given time constraints; however, no guarantees are provided regarding the completeness or security of the system. This audit is not an endorsement of the underlying business or product. The review focused solely on the security aspects of the Solidity implementation of the contracts and was conducted within a limited time frame.

## Risk Classification

	High	Medium	Low
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

**The findings described in this document corresponded the following commit hash:**

```
1 2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

## 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

### Issues found

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

## Findings

### High

#### [H-1] Passwords stored on-chain are visible to anyone, not matter 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

```
1 make anvil
```

- ## 2. Deploy the contract to the chain

```
1 make deploy
```

- ### 3. Run the storage tool

We use 1 because that's the storage slot of `s_password` in the contract.

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

You'll get an output that looks like this:

[illegible]

You can then parse that hex to a string with:

[illegible]

And get an output of:

```
1 myPassword
```

**Recommended Mitigation:** Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and then store the encrypted password on-chain. This would require the user to remember another password off-chain to decrypt the password. However, you'd also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password.

**[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.`

```
1     function setPassword(string memory newPassword) external {
2 @>         // @audit - There are no access controls here
3             s_password = newPassword;
4             emit SetNetPassword();
5     }
```

**Impact:** Anyone can set/change the password of the contract.

**Proof of Concept:**

Add the following to the `PasswordStore.t.sol` test suite.

```
1     function test_anyone_can_set_password(address randomAddress) public {
2         vm.prank(randomAddress);
3         string memory expectedPassword = "myNewPassword";
4         passwordStore.setPassword(expectedPassword);
5         vm.prank(owner);
6         string memory actualPassword = passwordStore.getPassword();
7         assertEq(actualPassword, expectedPassword);
8     }
```

**Recommended Mitigation:** Add an access control modifier to the `setPassword` function.

```
1     if (msg.sender != s_owner) {
2         revert PasswordStore__NotOwner();
3     }
```

## Low Risk Findings

### L-01. Initialization Timeframe Vulnerability

#### 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:

```
1      /*
2      * @notice This allows only the owner to retrieve the password.
3  @>   * @param newPassword The new password to set.
4      */
5      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.

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