# Protocol Audit Report

Version 1.0

*FINISH.io*

December 17, 2025

# Protocol Audit Report

FINISH.io

December 17, 2025

Prepared by: Cyfrin Lead Auditors: - xxxxxxx

## Table of Contents

## Protocol Summary

**PasswordStore** is a simple smart contract designed to store and manage a password on-chain. The contract allows the owner to set a new password and retrieve the current password. The protocol's core functionality relies on the assumption that the stored password remains confidential and can only be modified by the contract owner.

## Disclaimer

The Finsh 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

| | | 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

The audit focused on the **PasswordStore** smart contract, which implements basic password storage functionality on the Ethereum blockchain.

### Roles

- **Owner**: The account that deploys the contract and has exclusive rights to set and retrieve the password
- **External Users**: Any Ethereum address that can interact with the contract

## Executive Summary

This audit identified **2 High severity** and **1 Informational** issues in the PasswordStore contract. The most critical findings relate to on-chain data visibility and missing access controls, which completely undermine the contract's security assumptions.

### Issues found

- **High Severity**: 2 issues
- **Medium Severity**: 0 issues

- **Low Severity**: 0 issues
- **Informational**: 1 issue
- **Gas Optimization**: 0 issues

## Findings

## High

```
### [H-1] Storing Passwords On-Chain Makes Them Publicly Readable

**Description:**
All data stored on-chain is inherently public and can be read by anyone.
↪  The variable `PasswordStore::s_password` was intended to remain private
↪  and only be accessible to the contract owner via the
↪  `PasswordStore::getPassword` function.
However, despite access control at the Solidity level, the raw storage
↪  value can still be read directly from the blockchain using off-chain
↪  tools.
```

```
Below, a method is demonstrated to read arbitrary contract storage data
↪  off-chain.

**Impact:**
Any user can read the private password directly from contract storage. This
↪  completely breaks the confidentiality assumption of the protocol and
↪  undermines its core functionality.

**Proof of Concept:**

1. Start a local Anvil node:
   ```bash
   make anvil
```

2. Deploy the contract:

   ```
   make deploy
   ```

   The `PasswordStore` contract is deployed at:

   ```
   0x5FbDB2315678afecb367f032d93F642f64180aa3
   ```

3. Read the password from storage slot 1:

   ```
   cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1
   ```

   Output (bytes32):

   ```
   0x6d7950617373776f726400000000000000000000000000000000000000000014
   ```

4. Decode the bytes32 value into a string:

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

   Result:

   ```
   myPassword
   ```

**Recommended Mitigation:** The overall contract architecture should be reconsidered. Sensitive data such as passwords should **never** be stored in plaintext on-chain.

Possible approaches include:

- Encrypting the password off-chain and only storing the encrypted value on-chain.
- Avoiding password-based designs entirely in favor of cryptographic primitives (e.g., signatures, hashes, or commit–reveal schemes).
```

Additionally, consider removing `view` functions that expose sensitive data, as users may accidentally send transactions containing decryption secrets, leading to further leakage.

### [H-2] Missing Access Control in `setPassword` (No Ownership Check + Unauthorized State Modification)

**Description:**

The `setPassword` function is declared as `external` but does not implement any access control mechanism. Although a code comment indicates that only the owner should be able to set the password, this restriction is not enforced in the implementation.

As a result, any external account can call `setPassword` and arbitrarily overwrite the stored password (`s_password`), violating the intended authorization model of the contract.

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

**Impact:**

- Any user can reset the password without permission
- Breaks authentication and authorization logic relying on `s_password`
- May lead to:
  - Unauthorized access
  - Contract takeover
  - Loss of user trust
  - Potential fund loss if the password gates critical functionality

This issue represents a **critical security vulnerability** if the password is used to protect sensitive or privileged operations.

**Proof of Concept:**

An attacker can directly call the function from an EOA or another contract:

```
function test_anyone_can_set_password(address randomAddress) public {
      vm.assume(randomAddress != owner);
      vm.startPrank(randomAddress);
```

```
        string memory expectedPassword = "myNewPassword";
        passwordStore.setPassword(expectedPassword);

        vm.startPrank(owner);
        string memory actualPassword = passwordStore.getPassword();
        assertEq(actualPassword, expectedPassword);
    }
```

Because no ownership or role verification exists, the call succeeds regardless of the caller.

**Recommended Mitigation:**

Add an access-control check to the setPassword function:

```
if (msg.sender != s_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
```

**Impact:** The natspec is incorrect

**Recommended Mitigation:** Remove the incorrect natspec line.

”’diff / @notice This allows only the owner to retrieve the password. - * @param newPassword The new password to set. */ ”’