Protocol Audit Report

Prepared by: Maxmilla, Security researcher

Protocol Summary

The following section provides an overview of the protocol under audit. It describes the purpose and scope of the system being analyzed and sets the context for the findings.

Disclaimer

This audit report is provided for informational purposes only. It does not constitute legal or financial advice. The findings are based on our analysis at the time of the audit and do not guarantee future security.

Risk Classification

Issues identified during the audit are categorized by severity. The following table summarizes the risk levels:

Risk Level	Description
High (H)	Critical issues that require immediate action.
Informational (I)	Minor findings or recommendations.

Audit Details

Scope

The audit scope included a thorough review of the `PasswordStore` contract and associated components. The analysis focused on authentication mechanisms, access controls, and storage of sensitive data.

Roles

The audit was conducted by a team of security professionals with the following roles:

- **Security Researcher (Maxmilla):** Performed code review, threat modeling, and identified security issues.
- **Reviewer:** Provided independent assessment and validation of the findings and report.

Executive Summary

The audit revealed multiple issues with varying levels of severity. Below is an overview of the main findings.

Issues Found

- [**H-1**] High severity: The password is stored on-chain in plain text, making it visible to anyone.
- [**H-2**] High severity: The `setPassword` function lacks access control, allowing any user to change the password.
- [**I-1**] Informational: The NatSpec comment for `getPassword` mentions a non-existent parameter.

Findings

High

[H-1] Storing the password on-chain makes it visible and no longer private Description:

All data stored on-chain is publicly visible. Anyone can read it directly from the blockchain. The `PasswordStore::s_password` variable is meant to be private and accessed only through the `PasswordStore::getPassword` function, which is intended to be called only by the contract owner.

Here's one method of reading such data off-chain:

Impact:
Anyone can read the private password, which breaks the intended functionality of the protocol.

Proof of Concept:
The following test case demonstrates how the password can be read directly from the blockchain:

1. Start a local chain:

   ```
   make anvil
   ```

2. Deploy the contract:

   ```
   make deploy
   ```

3. Read the password using the storage slot.
   We use slot `1`, assuming that's where `s_password` is stored:

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

   This returns:

   ```
   0x6d7950617373776f726400000000000000000000000000000000000000000014
   ```

   Decode it using:

   ```
   cast parse-bytes32-string 0x6d7950617373776f726400000000000000000000000000000000000000000000
   ```

   Output:

   ```
   myPassword
   ```

Recommended Mitigation:
The contract architecture needs a redesign. One option is to encrypt the password off-chain and store only the encrypted result on-chain. This would require users to manage a decryption key off-chain. Additionally, remove the `view` function to prevent users from accidentally submitting transactions containing the decryption key.

[H-2] `PasswordStore::setPassword` lacks access control—non-owners can change the password Description:
The `PasswordStore::setPassword` function is marked `external`, but its NatSpec comment implies that only the owner should call it. There is no actual access control in place.

```
function setPassword(string memory newPassword) external {
    //@audit - no access controls present
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact:
Anyone can change the contract's password, which completely breaks the intended behavior.

Proof of Concept:
Add the following to `passwordStore.t.sol`:

Code

```
function test_anyone_can_set_password() public {
    address randomAddress = vm.addr(1);
    vm.assume(randomAddress != owner);
    vm.prank(randomAddress);

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

    assertEq(actualPassword, expectedPassword);
}
```

Recommended Mitigation:
Add an access control check to `setPassword`:

```
if (msg.sender != s_owner) {
    revert PasswordStore_NotOwner();
}
```

Informational

[I-1] Incorrect NatSpec for `PasswordStore::getPassword` showing that a parameter doesn't exist The NatSpec comment for `getPassword` refers to a parameter that is not present in the function signature. This mismatch could cause confusion for developers reviewing the code. For example:

```
//**
    * @notice This allows only the owner to retrieve the password.
    * @param newPassword The new password doesn't exist.
    */
function getPassword() external view returns (string memory) {}
```

The function does not take any parameters, but the NatSpec includes a `@param` line for a nonexistent `newPassword`.

Impact:
The NatSpec is incorrect and may mislead users or automated documentation tools.

Recommended Mitigation:
Remove the incorrect NatSpec line:

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