Protocol Audit Report

February 21, 2025

Executive Summary

Prepared by: Baloo.X
Lead Auditors:
- xxxxxxx

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The YOUR_NAME_HERE 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
	High	H	H/M	M
Likelihood	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

Roles

Executive Summary

Issues found

This audit identified a total of 3 issues:
- **High Severity:** 2
- **Medium Severity:** 0
- **Low Severity:** 0

- **Informational:** 1
- **Gas Optimizations:** 0

The critical issues involve improper handling of sensitive data on-chain and missing access controls, both of which severely compromise the intended functionality of the protocol.

Findings

The following sections detail the vulnerabilities uncovered during the audit of the PasswordStore protocol.

High

[H-1] Storing the Password On-Chain Makes It Visible to Anyone and No Longer Private

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.

Impact: Anyone can read the private password, severely breaking the functionality of the protocol.

Proof of Concept: The below test case shows how anyone can read the password directly from the blockchain.

1. Create a locally running chain:

```
make anvil
```

2. Deploy the contract to the chain with deploy script:

```
forge script script/DeployPasswordStore.s.sol:DeployPasswordStore \
  --rpc-url http://127.0.0.1:8545 \
  --private-key <PRIVATE_KEY> \
  --broadcast
```

3. Run the storage tool:

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

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

You'll get an output (bytes) hex like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

You can then parse that hex to a string with:

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

And get an output of:

```
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 transactions with the password that decrypts your password.

Likelihood & Impact:

- Impact: HIGH
- Likelihood: HIGH
- Severity: HIGH

[H-2] `PasswordStore::setPassword` Has No Access Controls

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 state that "This function allows only the owner to set a new password."

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

Impact: Anyone can set/change the password of the contract, severely breaking the contract's intended functionality.

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

CODE

```
function test_anyone_can_set_password(address randomAddress) public {
        vm.assume(randomAddress != owner);
        vm.prank(randomAddress);
        string memory expectedPassword = "myNewPassword";
        passwordStore.setPassword(expectedPassword);

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

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

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

Likelihood & Impact:

- Impact: HIGH
- Likelihood: HIGH
- Severity: HIGH

[I-1] `PasswordStore::getPassword` Natspec Indicates a Non-Existent Parameter

Description:

```
/* @notice This allows only the owner to retrieve the password.
 * @param newPassword The new password to set.
 * @audit - there is no newPassword parameter
 */
function getPassword() external view returns (string memory) {
```

The `PasswordStore::getPassword` function signature is `getPassword()` while the natspec says it should be `getPassword(string)`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line:

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

Likelihood & Impact:

- Impact: HIGH
- Likelihood: NONE
- Severity: Informational