Protocol Audit Report

Harpreet Gill

Oct 2, 2024

Prepared by: Harpreet Gill Lead Auditors: - Myself

Table of Contents

Protocol Summary

The protocol is desgined as the only owner can be able to get and set the password not anyone are able to do so.

Disclaimer

The Harpreet Gill 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

Findings that are defined in this document is follow the following commit hash:
Commit Hash:

```
8ea93e895e70fa868ad30c3d6b2c54a18db9f1ca
```

Scope

```
./src/
└── PasswordStore.sol
```

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

As i mentioned that we spend a lot of time of it and make sure that our contract is not having any severity anymore as far the best of my knowleadge

Issues found

serverity	Number of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] On-Chain Storage Visibility – Password is Not Actually Private

Root Cause / Main Reason
All variables stored on-chain are publicly visible, regardless of Solidity's visibility keyword. This means that the password stored in `PasswordStore` is not private, even though it is marked as `private`.

Description
All data stored on-chain can be accessed by anyone with the right tools, even if marked `private`. In the `PasswordStore` contract, the variable `s_password` is intended to be accessible only via the `getPassword()` function, which is restricted to the contract's owner. However, anyone can directly read the stored password from the blockchain's storage.

We demonstrate how to retrieve on-chain data off-chain.

Impact
Anyone can read the private password, which severely compromises the security and intended functionality of the protocol.

Proof of Concept

The following steps demonstrate how to read the password directly from the blockchain:

1. Create a locally running chain:

   ```
   make anvil
   ```

2. Deploy the contract on the locally running chain:

   ```
   make deploy
   ```

3. Run the storage tool to read the stored password from the contract's storage.

   In this case, `s_password` is stored in slot `1`:

   ```
   cast storage <ADDRESS> 1 --rpc-url <RPC_URL>
   ```

   Example:

   ```
   cast storage 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512 1 --rpc-url http
   ```

   You will get the following output:

   ```
   0x0000000000000000000000000000000000000000000000000000000000000000
   ```

4. Parse the hexadecimal value into a string:

```
cast parse-bytes32-string 0x0000000000000000000000000000000000000000000000000
```

The result will be:

```
YOUR_PASSWORD
```

Recommended Mitigation
To mitigate this issue, never store sensitive data like passwords directly on-chain in plain text. Instead, store a hashed version of the password using cryptographic functions like `keccak256`. Password verification can then be done by comparing the hashed input with the stored hash. Alternatively, you can encrypt sensitive data off-chain and store the encrypted data on-chain, decrypting it off-chain when necessary. For enhanced security, consider using Zero-Knowledge Proofs (ZKP) to verify information without revealing sensitive data.

Example code for storing and verifying a hashed password:

```
bytes32 private s_passwordHash;

// Store the hash of the password
function setPassword(string memory _password) public onlyOwner {
    s_passwordHash = keccak256(abi.encodePacked(_password));
}

// Verify password by comparing hashes
function verifyPassword(string memory _password) public view onlyOwner
returns (bool) {
    return keccak256(abi.encodePacked(_password)) == s_passwordHash;
}
```

[H-2] No Access Control on `PasswordStore::setPassword`

Description
The `PasswordStore::setPassword` function is marked as `external`, but there is no access control in place. This allows anyone to change the password, even though the intended functionality of the contract is to allow only the owner to change the password.

```
//@audit there is no access control
function setPassword(string memory newPassword) external {
    s_password = newPassword;
    emit SetNetPassword();
}
```

Impact
Anyone can change the password, which significantly undermines the contract's intended functionality.

Proof of Concept
The following test case demonstrates how a non-owner can change the password:

▶ Test Code

Recommended Mitigation
Add access control to the `setPassword` function to ensure that only the owner can change the password:

```
if (msg.sender != owner) {
    revert PasswordStore__NotOwner();
}
```

Medium

Low

Informational

[I-3] Incorrect NatSpec in `PasswordStore::getPassword`

Description
The NatSpec documentation for the `getPassword()` function incorrectly includes a parameter description that does not exist in the function signature.

```
 /*
  * @notice This allows only the owner to retrieve the password.
  * @param newPassword The new password to set.
  */
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

The `getPassword()` function does not take any parameters, yet the NatSpec includes a`@param` entry for `newPassword`.

Impact
This makes the NatSpec documentation incorrect and potentially confusing for developers and auditors.

Recommended Mitigation
Remove the incorrect `@param` entry from the NatSpec documentation:

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

Gas