Protocol Audit Report

Prepared by: Royal

Table of Contents

Protocol Summary

The `PasswordStore` smart contract is designed to allow users to store and retrieve passwords securely. The primary functionality includes:

- Storing a password on-chain.
- Retrieving the stored password only by the owner.

However, due to the nature of blockchain technology, storing sensitive data like passwords directly on-chain poses significant security risks.

Disclaimer

I made every effort to identify vulnerabilities within the given time frame but does not assume responsibility for any findings 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 focused solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

We use the CodeHawks severity matrix to determine the severity of findings. Below is the classification table:

Likelihood	Impact	High	Medium	Low
High	H	H/M	M	-
Medium	H/M	M	M/L	L
Low	M	M/L	L	-

Audit Details

Scope

- **Commit Hash**: `7d55682ddc4301a7b13ae9413095feffd9924566`
- **Files Audited**:
  - `./src/PasswordStore.sol`
- **Solidity Version**: `0.8.18`
- **Chain(s)**: Ethereum (Testnet deployment)

Roles

- **Owner**: The user who can set and retrieve the password.
- **Others**: No one else should be able to set or retrieve the password.

Executive Summary

The `PasswordStore` contract implements basic functionality for storing and retrieving passwords. However, during the audit, several critical issues were identified that compromise the security and integrity of the system.

Issues Found

1. **High Severity**:

   - Storing sensitive data (passwords) on-chain makes it publicly accessible.
   - Lack of access controls in the `setPassword` function allows unauthorized users to change the password.

2. **Informational**:

   - Incorrect natspec documentation in the `getPassword` function.

Findings

High Severity

[H-1] Storing Passwords on-Chain Makes Them Publicly Accessible

Description: In Solidity, all data stored on the blockchain is publicly accessible, regardless of the visibility keyword (`private`, `internal`, etc.). This means that sensitive information such as passwords can be accessed by anyone with access to the blockchain.

Impact: Exposure of sensitive data can lead to unauthorized access, data breaches, and potential exploitation of the system. This compromises user privacy and trust, resulting in financial and reputational damage.

Proof of Concept:

1. Deploy the contract locally using `make deploy`.
2. Use the `cast storage` command to read the password from storage:

   ```
   cast storage <contract-address> 1
   ```

3. Parse the hex output to retrieve the password:

   ```
   cast parse-bytes32-string <hex-output>
   ```

Recommended Mitigation: Avoid storing sensitive data on-chain. Instead:

- Store a hash of the password on-chain and verify it off-chain.
- Encrypt the password off-chain before storing it on-chain, ensuring the decryption key is managed securely.

Example:

```
pragma solidity ^0.8.0;

contract PasswordStore {
    bytes32 private passwordHash;

    constructor(bytes32 _passwordHash) {
        passwordHash = _passwordHash;
    }

    function verifyPassword(string memory _password) public view returns (bool) {
        return keccak256(abi.encodePacked(_password)) == passwordHash;
    }
}
```

[H-2] Lack of Access Controls in `setPassword`

Description: The `setPassword` function lacks access controls, allowing anyone to update the password stored in the contract.

Impact: Unauthorized users can change the password, leading to potential security risks and unauthorized access to the system.

Proof of Concept: Write a test case where a non-owner changes the password:

```
function test_non_owner_can_set_password() public {
    vm.assume(address(1) != owner);
    vm.prank(address(1));
    string memory expectedPassword = "myNewPassword";
    passwordStore.setPassword(expectedPassword);
    vm.prank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
}
```

Recommended Mitigation: Add access control to ensure only the owner can call the `setPassword` function:

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

Informational

[I-1] Incorrect Natspec Documentation in `getPassword`

Description: The natspec for the `getPassword` function incorrectly references a parameter (`newPassword`) that does not exist in the function signature.

Impact: The natspec is misleading and may confuse developers or users.

Recommended Mitigation: Remove the incorrect line from the natspec:

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

Conclusion

The `PasswordStore` contract demonstrates basic functionality but contains critical security vulnerabilities that must be addressed before deployment. Key recommendations include:

1. Avoid storing sensitive data on-chain.
2. Implement proper access controls.
3. Ensure accurate and meaningful documentation.

By addressing these issues, the contract can achieve a higher level of security and reliability.