Protocol Audit Report

Prepared by: OSTE (Oudjani Seyyid taqy eddine)

Table of Contents

Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords. The protocol is designed to be used by a single user, and is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Disclaimer

The OSTE 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

The findings described in this document correspond the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

```
src/
--- PasswordStore.sol
```

Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords. The protocol is designed to be used by a single user, and is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	0
Gas Optimizations	0
Total	0

Findings

High

[H-1] `PasswordStore::setPassword()` has no access controle to change the password

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 is that 'this function allows only the owner to set a new password'

Impact: anyone can set/change the password of the contract , which break the contract functionality

Proof of Concept: Add the Following to the paswordstore.t.sol test file.

```
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(expectedpassword,actualpassword);
}
```

Recommended Mitigation: add some controle function to make sure that only the owner can set password

[H-2] Variable stored in storage on-chain are visible to anoyone,no metter the solidity meaning the password is not actually aprivate number OR Storing the password on-chain make it visible to anyome,and no longer privet (Root Cause + Impact)

Description: All date 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 shich is intended to be only called by the owner of the contract. we show one such method of reading any data off chain below

Impact: Anyone can read the private oassword, severly breaking the cunftionality of the protocol.

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

1. create a loccaly running chaing USing anvil

```
make anvil
```

2. deploy the contract to the chain

```
make deploy
```

3. Run the Storage tool

we use '1' because that's the storage slot of 's_password' variable, you can execute the command without the argument '1' and get all list of storage slots

```
cast storage <countract address> 1 --rpc-url http:127.0.0.1:8545
```

4. conver the hex result to humain readable format :

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

Result is "Mypassword"

Recommended Mitigation: Storing a password directly within a smart contract as storage is generally insecure due to the transparent nature of blockchain. However, if you must handle sensitive information, consider these mitigation steps:

1. Hash the Password: Store only a hashed version of the password (e.g., using keccak256 or SHA-3). This way, the actual password isn't visible, and only hashed data is stored on-chain.
2. Salting: Combine the password with a unique random salt before hashing. This helps protect against rainbow table attacks and ensures that identical passwords produce different hashes.
3. Off-Chain Storage: Store sensitive information like passwords off-chain, such as in a secure database. The smart contract can then interact with this storage by validating hashes or retrieving the password securely.
4. Zero-Knowledge Proofs (ZKP): Implement ZKP protocols, where the contract can verify knowledge of the password without revealing it. This is complex but provides a high level of security.
5. Encryption: Although complex and gas-expensive, consider encrypting the password with a public key. The contract can then store the encrypted version, while only the holder of the private key can decrypt it off-chain.

By applying these strategies, you can significantly improve password security within a blockchain context.