Prepared by: [0xError] Lead Auditors: - 0xError (Hossein Rezaei Fard)

Table of Contents

Protocol Summary

A smart contract application for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Disclaimer

The 0xError 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 describe in this document correspond the following commit hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

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. ## Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0

Severity	Number of issues found
Info	1
Gas	0
Total	3

Findings

High

[H-1] Storing the `PasswordStore::s_password` on-chain makes it visible to anyone, and no longer private

Description: The contract uses the `private` keyword to store a password (`PasswordStore::s_password`) with the assumption that it cannot be accessed by others. However, in Solidity, the private keyword only restricts access from other contracts — not from external parties who can directly read contract storage using tools like Hardhat, Foundry, Ethers.js, or blockchain explorers like Etherscan.

Impact:

Stored password is publicly accessible despite being marked private.

Attackers can retrieve the password by directly reading storage slot data.

May lead to compromise of off-chain systems that rely on the secrecy of the stored password.

Proof of Concept: The below test case shows how anyone could read the password directly from the blockchain. We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1. Create a locally running chain

```
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` in the contract.

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

You'll get an output that looks like this:

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

You can then parse that hex to a string with:

```
cast parse-bytes32-string 0x6d7950617373776f72640000000000000000000000000000000000
```

And get an output of:

```
myPassword
```

Recommended Mitigation: Storing sensitive information like passwords directly on-chain — even in `private` variables — is fundamentally insecure, as all on-chain data is publicly accessible. We strongly recommend reconsidering the design of this contract:

- **Avoid storing raw passwords on-chain.** Instead, store only a hash (e.g., `keccak256`) of the password if necessary for verification.
- **Re-architect the contract logic** to avoid relying on secrecy of data. Smart contracts are transparent by nature.
- **If confidentiality is required**, move the password handling off-chain and use the contract only to verify proofs (e.g., hashed comparisons or zero-knowledge proofs).
- **Educate developers** on the limitations of visibility in Solidity to avoid similar misunderstandings in future implementations.

By shifting the architecture away from on-chain secrecy and towards verification-based logic, the contract will become both more secure and more aligned with best practices for decentralized applications.

[H-2] `PasswordStore::setPassword` has no access controls, meaning a non-owner could change the password

Description: The `setPassword` function lacks proper access control, allowing any external address to call it and overwrite the stored password. This violates the intended contract logic, where only the contract owner should have the ability to update the password. Although the variable `s_owner` is set to the deploying address in the constructor, it is never checked during the `setPassword` execution. As a result, unauthorized users can arbitrarily modify the contract's internal state, rendering the contract untrustworthy and vulnerable to misuse.

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

Impact: Anyone can set/change the password of the contract, severly breaking the contract 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 != owner) {
        revert PasswordStore__NotOwner();
    }
```

Informational

[I-1] `PasswordStore::getPassword` natspec indicates a paramater that doesn't exist, causing the natspec to be incorrect

Description:

```
    /*
     * @notice This allows only the owner to retrieve the password.
@>   * @param newPassword The new password to set.
     */
    function getPassword() external view returns (string memory) {
```

The natspec for the function PasswordStore::getPassword indicates it should have a parameter with the signature getPassword(string). However, the actual function signature is getPassword().

Impact: The natspec is incorrect.

Recommended Mitigation: Recommended Mitigation: Remove the incorrect natspec line.

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