# Smart Contract Security Audit Report

Philani A Dlamini

2025-07-18

## Contents

# 1 PasswordStore Security Audit

## 1.1 1. [High Severity] Sensitive Data Exposure: On-Chain Storage of Passwords

### 1.1.1 Description

The `PasswordStore` contract stores user passwords in the private variable `PasswordStore::s_password`. While declared private in Solidity, all on-chain data is inherently transparent and publicly accessible via node RPC, blockchain explorers, or direct storage inspection.

This directly violates the intended privacy and confidentiality of the stored password — anyone can extract the plaintext password from contract storage without needing to call the `PasswordStore::getPassword()` function, which is protected by an ownership check.

### 1.1.2 Impact

- **Privacy Violation:** Passwords stored on-chain are retrievable by anyone, compromising the protocol's functional promise of confidentiality.

- **Loss of Trust:** Storing sensitive data in plaintext on-chain can erode user trust, especially for privacy-preserving applications.
- **Data Leakage:** Sensitive data may remain permanently exposed on-chain, creating irreversibility in confidentiality breaches.

#### 1.1.3 Proof of Concept

The following steps demonstrate how any user can retrieve the password directly from blockchain storage, bypassing Solidity-level visibility restrictions.

##### 1.1.3.1 Steps

1. Start a local blockchain:

```
make anvil
```

2. Deploy the contract:

```
make deploy
```

3. Read storage directly:

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

Example output:

```
0x6d7950617373776f7264000000000000000000000000000000000000000000014
```

4. Parse the password:

```
cast parse-bytes32-string 0x6d7950617373776f7264000000000000000000000000000000000000000000014
```

Output:

```
myPassword
```

#### 1.1.4 Recommended Mitigation

1. **Off-chain Encryption:**
    - Encrypt passwords off-chain using user-controlled keys.
    - Store ciphertext on-chain instead of plaintext.
2. **Store Hashes Instead:**
    - Store a hash commitment (e.g., `keccak256(password)`) instead of the password.
    - Use hash comparison or zero-knowledge proofs to verify knowledge without revealing the password.
3. **Remove getPassword():**
    - Eliminate this retrieval function to prevent unintended data exposure.

---

## 1.2 2. [High Severity] Lack of Access Control on `setPassword()`

### 1.2.1 Description

The `PasswordStore::setPassword()` function is declared `external` but lacks any access control. Despite NatSpec implying that only the contract owner should update the password, no restriction is enforced.

```
function setPassword(string memory newPassword) external {
    s_password = newPassword;
    emit SetNetPassword();
}
```

### 1.2.2 Impact

- **Confidentiality Violation:** Any user can overwrite the password.
- **Integrity Risk:** Unauthorized users can hijack the password.
- **Loss of Control:** Owner loses exclusive control of sensitive data.

### 1.2.3 Proof of Concept

The following test verifies that any user can overwrite the password.

```

function test_anyone_can_set_password(address randomAddress) public {
    vm.assume(randomAddress != owner && randomAddress != address(0));
    vm.prank(randomAddress);
    string memory expectedPassword = "MyNewPassword";
    passwordStore.setPassword(expectedPassword);

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

#### 1.2.3.1 Code Example

### 1.2.4 Recommended Mitigation

1. **Introduce Access Control Modifier:**

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

function setPassword(string memory newPassword) external onlyOwner {
    s_password = newPassword;
    emit SetNetPassword();
}
```

2. **Alternative:**
   - Integrate OpenZeppelin's `Ownable` contract for standardized ownership control.

---

## 1.3 3. [Low Severity] Incorrect NatSpec on `getPassword()`

### 1.3.1 Description

The NatSpec for `getPassword()` incorrectly documents a parameter that does not exist:

```
/*
 * @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;
}
```

### 1.3.2 Impact

- **Documentation Inaccuracy:** Misleading for developers, auditors, and automated documentation tools.
- **Maintainability Risk:** Increases the chance of misinterpretation of the function's purpose.

### 1.3.3 Recommended Mitigation

1. **Correct NatSpec Documentation:**

```
/**
 * @notice Returns the current password. Only callable by the owner.
 * @return The stored password.
 */
function getPassword() external onlyOwner view returns (string memory) {
    return s_password;
}
```

2. **Consistency in Access Control:**
   - Apply the `onlyOwner` modifier to ensure consistent access patterns.

---

## 1.4 Conclusion

The `PasswordStore` contract demonstrates critical issues related to on-chain data privacy and access control. To align with best practices:

- Avoid storing sensitive data directly on-chain.
- Enforce strict access control on state-mutating functions.
- Ensure documentation accurately reflects function behavior.

Addressing these findings will significantly improve the contract's security posture and robustness against misuse or attacks.

---

**Report Prepared By:**
Philani A Dlamini
Date: 2025-07-18