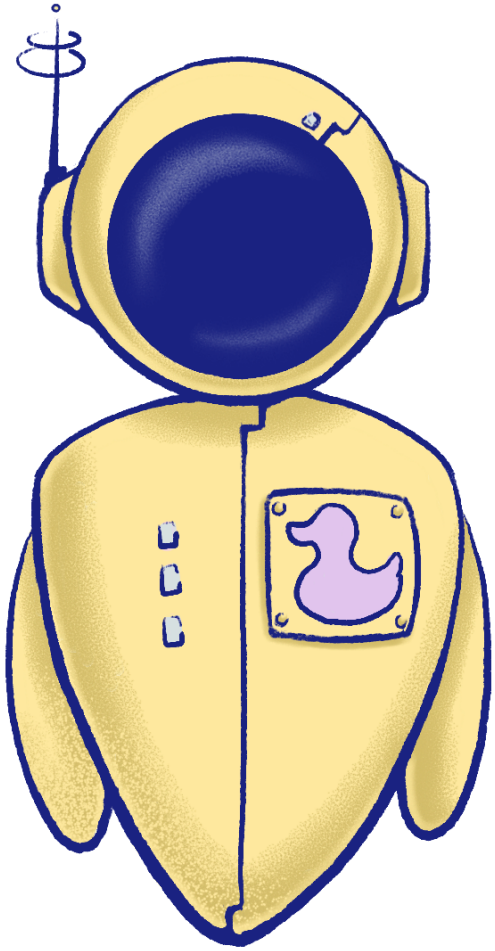

Protocol Security Review: Password Store

Prepared by: Cryptodant

Lead Reviewer: Anthony Spedaliere

Table of Contents

Protocol Summary

The PasswordStore protocol is a simple smart contract designed to provide secure password storage functionality on the blockchain. The contract implements a basic access control mechanism where only the contract owner can store and retrieve a private password.

Core Functionality

The PasswordStore contract should offer the following key features:

- **Password Storage**: Allows the contract owner to store a private password as a string variable
- **Password Retrieval**: Enables the owner to retrieve the stored password through a view function
- **Access Control**: Implements owner-only access control to ensure only the contract deployer can interact with the password
- **Password Updates**: Supports updating the stored password at any time by the owner

Disclaimer

The Cryptodant 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
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

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

```
Commit Hash: 7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

```
./src/
└── PasswordStore.sol
```

Roles

Owner: The user who can set the password and read the password. Outsiders: No one else should be able to set or read the password.

Executive Summary

This security audit of the PasswordStore smart contract was conducted by a single auditor over a 2-hour period. The audit focused on identifying potential security vulnerabilities, access control issues, and implementation flaws in the contract's core functionality.

Audit Process

The audit was performed using a systematic approach that included:

- **Code Review**: Thorough examination of the Solidity implementation
- **Static Analysis**: Analysis of access control mechanisms and state management
- **Functionality Testing**: Verification of intended behavior vs. actual implementation
- **Security Pattern Analysis**: Evaluation against common smart contract security best practices

Audit Scope

The audit covered the complete PasswordStore contract implementation, including:

- Access control mechanisms
- State variable management
- Function implementations
- Error handling
- Event emissions

Issues found

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

Findings

High

[H-1] Storing the password on-chain makes it visible to everyone, and no longer private

Description: All data 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, which is intended to be only called by the owner of the contract.

We show one such method of reading any data on-chain below.

Impact: Anyone can read the private password, severely breaking the functionality of the protocol.

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

▶ Code

```
function test_anyone_can_read_password() public {
    // Deploy the contract
    PasswordStore passwordStore = new PasswordStore();

    // Set a password as the owner
    string memory expectedPassword = "myPassword123";
    passwordStore.setPassword(expectedPassword);

    // Anyone can read the password directly from storage
```

```
    // This demonstrates that private variables are not actually private
    bytes32 passwordSlot = keccak256(abi.encodePacked(uint256(1))); // s_password
is at slot 1
    bytes32 passwordData = vm.load(address(passwordStore), passwordSlot);

    // The password can be decoded from storage
    string memory actualPassword = string(abi.encodePacked(passwordData));

    // This proves the password is readable by anyone
    assertTrue(bytes(actualPassword).length > 0);
}
```

Recommended Mitigation: Consider using off-chain storage solutions for sensitive data like passwords. Some options include:

1. **Use IPFS or similar decentralized storage** - Store only a hash or encrypted reference on-chain
2. **Implement client-side encryption** - Encrypt the password before storing, store only the encrypted version
3. **Use commit-reveal schemes** - Store only a commitment hash, reveal the actual password through a separate mechanism
4. **Move to off-chain solutions** - Use traditional databases or encrypted cloud storage for truly private data

If on-chain storage is absolutely necessary, consider:

- Using encryption libraries like `ethers.js` to encrypt data before storage
- Implementing access control mechanisms that require multiple signatures
- Using zero-knowledge proofs to verify password correctness without revealing the actual password

Note: This is a fundamental limitation of blockchain technology - all data stored on-chain is publicly readable. For truly private data, consider whether blockchain is the appropriate technology choice.

[H-2] `PasswordStore::setPassword` has no access controls, meaning a non-owner can 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.`

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

Impact: Anyone can set/change the password of the contract, severely breaking the contract's 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 != s_owner) {
            revert PasswordStore__NotOwner();
        }
```

Informational

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

Description: The `PasswordStore::getPassword` function has incorrect natspec documentation. The function documentation states:

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

However, the `getPassword` function takes no parameters and only returns the stored password. The `@param newPassword` documentation is incorrect and misleading.

Impact: Incorrect documentation can lead to:

- Developer confusion when integrating with the contract
- Misunderstanding of the function's purpose and parameters
- Potential bugs in frontend applications or other contracts that interact with this function
- Poor developer experience and reduced code maintainability

Proof of Concept: The function signature clearly shows no parameters:

```
function getPassword() external view returns (string memory)
```

But the natspec incorrectly documents a `newPassword` parameter that doesn't exist.

Recommended Mitigation: Fix the natspec documentation to accurately reflect the function's behavior:

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

Additional Recommendation: Consider using automated documentation generation tools or linters that can catch such inconsistencies between function signatures and natspec documentation.