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

Description: Variables stored in on-chain are visible to anyone, no matter the solidity visibility keyword, 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.

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.

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
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

```
myPassword
```

Recommended Mitigation: Due to these concerns, the overall architecture of the contract should be reconsidered. A more secure approach would involve encrypting the password off-chain and storing only the encrypted version on-chain. This would require the user to remember a separate decryption key (e.g., a

passphrase) off-chain to decrypt the password when needed. Additionally, the view function should be removed to prevent the risk of accidentally exposing the decryption key or sensitive data through on-chain transactions. This approach ensures that sensitive information remains private and secure, while still allowing the user to manage their password effectively.

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

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

Impact: Anyone can set/change the pasword of the contract, severely breaking the contract intended functionality.

Proof of Concept: Add the following to the `PasswordStore.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(actualPassword, expectedPassword);
    }
```

Recommended Mitigation: Add an access control conditional to the `setPassword` function.

```
    function setPassword(string memory newPassword) external {
+        if (msg.sender != s_owner) {
+            revert PasswordStore__NotOwner();
+        }
        s_password = newPassword;
        emit SetNewPassword();
    }
```

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter 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 `PasswordStore::getPassword` function signature is `getPassword()` which the natspec say it should be `getPassword(String)`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line.

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