~~~
### [H-1] Storing the password on-chain makes it visible to anyone

**Description:** All data stored on-chain is visible to anyone and can be read by
anyone, regardless of any Solidity visibility keywords applied to variables. The
`PasswordStore::s_password` variable is intended to securely and privately store a
password set by the user, and only read by the user through the
`PasswordStore::getPassword()` function which has an only owner check and revert.
However, the `PasswordStore::s_password` variable is not private because it is
stored on-chain.

We demonstrate one such method to read any data off the chain below.

**Impact:** Anybody can read the intended private password, severely breaking the
functionality and intended service of the protocol.

**Proof of Concept:** The below test case shows how we can read any data from the
blockchain.

1. Create a local chain with anvil
   ```bash
   $ make anvil
~~~

2. Deploy the contract

   ```
   $ make deploy
   ```

3. Get value from storage location

   We use `1` for storage slot because `s_password` is defined second in the contract, index starts at zero, and therefore occupies storage slot `1` in the contract.

   ```
   cast storage <address> <storage slot> --rpc-url <RPC URL>
   ```

4. Make value human readable

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

5. Private variable output

   ```
   myPassword
   ```

Recommended Mitigation: All data on the blockchain is public. To store sensitive information, additional encryption or off-chain solutions should be considered. Sensitive and personal data should never be stored on the blockchain in plaintext or weakly encrypted or encoded format.

[H-2] Missing access control for `PasswordStore::setPassword()`, allowing anyone to modify `PasswordStore::s_password`

Description: `PasswordStore::setPassword()` is an `external` function that can be called outside of the contract. This function is used to modify the `PasswordStore::s_password`. The intended use of this function is that only the owner of the contract can modify `PasswordStore::s_password` through this function, as detailed in the natspec of the contract - "`This function allows only the owner to set a new password`". However, the `PasswordStore::setPassword()` function does not carry out any checks if the caller of this function is the contract owner.

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

Impact: Anybody can call this function to modify the stored password, severely breaking the intended functionality of the contract.

FOR SPEED OF LEARNING, THE FOLLOWING SECTIONS OF THIS FINDING HAVE BEEN COPIED FROM THE GITHUB CODE BASE OF THIS LESSON

Proof of Concept:

▶ Details
Proof of Concept Steps and Code

Working Test Case

The `makeAddr` helper function is used to setup an `attacker` address to call the `setPasword()` function:

```
contract PasswordStoreTest is Test {
    PasswordStore public passwordStore;
    DeployPasswordStore public deployer;
    address public owner;
+   address public attacker;

    function setUp() public {
        deployer = new DeployPasswordStore();
        passwordStore = deployer.run();
        owner = msg.sender;
        // attacker address
+       attacker = makeAddr("attacker");
```

```
    }
}
```

The following test, sets the password to `"attackerPassword"` using the attacker address. When run, this test will pass, demonstrating that the attacker can set the password:

```
function test_poc_non_owner_set_password() public {
    // initiate the transaction from the non-owner attacker address
    vm.prank(attacker);
    string memory newPassword = "attackerPassword";
    // attacker attempts to set the password
    passwordStore.setPassword(newPassword);
    console.log("The attacker successfully set the password:" newPassword);
}
```

Run the test:

```
forge test --mt test_poc_non_owner_set_password -vvvv
```

Which yields the following output:

```
unning 1 test for test/PasswordStore.t.sol:PasswordStoreTest
[PASS] test_poc_non_owner_set_password() (gas: 20776)
Logs:
  The attacker successfully set the password: attackerPassword

Traces:
  [20776] PasswordStoreTest::test_poc_non_owner_set_password()
    ├─ [0] VM::prank(attacker: [0x9dF0C6b0066D5317aA5b38B36850548DaCCa6B4e])
    │   └─ ← ()
    ├─ [6686] PasswordStore::setPassword(attackerPassword)
    │   ├─ emit SetNetPassword()
    │   └─ ← ()
    ├─ [0] console::log(The attacker successfully set the password:
attackerPassword) [staticcall]
    │   └─ ← ()
    └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.36ms
```

Recommended Mitigation

Include access control to restrict who can call the `setPassword` function to be only the owner: `s_owner`. This can be achieved in two ways:

1. Using an `if` statement, as used in the `getPassword` function, and revert with the `PasswordStore__NotOwer()` custom error if the address calling the function is not the owner:

```
function setPassword(string memory newPassword) external {
    // @audit check that the function caller is the owner of the contract
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    s_password = newPassword;
    emit SetNetPassword();
}
```

2. Using an access modifier e.g. OpenZeppelin's `onlyOwner`. To use this modifier, the `PasswordStore` contract will need to inherit from OpenZeppelin's `Ownable` contract and call its constructor inside the constructor of `PasswordStore`:

```
// @audit import the ownable contract from OpenZeppelin
+ import "@openzeppelin/contracts/ownership/Ownable.sol";

// @audit inherit from the Ownable contract
+ contract PasswordStore is Ownable {
    error PasswordStore__NotOwner();

    address private s_owner;
    string private s_password;

    event SetNetPassword();

+    constructor() Ownable() {
        s_owner = msg.sender;
    }
}
```

As per the OpenZeppelin documentation, by default, the `owner` of an `Ownable` contract is the account that deployed it, meaning that the `s_owner` state variable can be removed.

Using `onlyOwner` modifier adds logic to check that the `msg.sender` is the `owner` of the contract before executing the function's

logic:

```
/*
* @notice This function allows only the owner to set a new password.
* @param newPassword The new password to set.
*/
```

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

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

Description: The `PasswordStore::getPassword()` function signature is `getPassword()` which the natspec says should actually be `getPassword(string)`.

```
/*
* @notice This allows only the owner to retrieve the password.
* @param newPassword The new password to set.
*/
//@audit-info - incorrect documentation for parameter inputs
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

Impact: The natspec of code is incorrect.

Recommended Mitigation: Remove incorrect natspec.

```
/*
* @notice This allows only the owner to retrieve the password.
- * @param newPassword The new password to set.
*/
```

```

```