Lab - Explore the Evolution of Password Methods (Instructor Version)

Instructor Note: Red font color or gray highlights indicate text that appears in the instructor copy only.

Objectives

Part 1: Launch the DEVASC VM

Part 2: Explore Python Code Storing Passwords in Plain Text

Part 3: Explore Python Code Storing Passwords Using a Hash

Background / Scenario

In this lab, you will create an application that stores a username and password in plaintext in a database using python code. You will then test the server to ensure that not only were the credentials stored correctly, but that a user can use them to login. You will then perform the same actions, but with a hashed password so that the credentials cannot be read. It is important to securely store credentials and other data to prevent different servers and systems from being compromised.

Required Resources

- 1 PC with operating system of your choice
- Virtual Box or VMWare
- DEVASC Virtual Machine

Instructions

Part 1: Launch the DEVASC VM

If you have not already completed the **Lab - Install the Virtual Machine Lab Environment**, do so now. If you have already completed that lab, launch the DEVASC VM now.

Part 2: Demonstrate the Application

Instructor Note: You can use the script at the end of the lab to demonstrate the application.

Your instructor may demonstrate the Password Plain Text and Hashing Application. However, you will create this script step by step in this lab.

The application first stores a username and password in plaintext in a web service database. It then validates that the credentials were stored and work properly. The second method, hashing the password, also stores them and tests them in the web service database.

Part 3: Install Packages and Create a Web Service

In this Part, you will use Flask to create a simple web service that requires user authentication. User authentication requires a database which will be satisfied using SQLite.

Step 1: Open the security directory in VS Code and install Python packages.

a. Open VS code. Then click **File > Open Folder...** and navigate to the **devnet-src/security** directory. Click **OK**.

b. In the **EXPLORER** panel, click the **password-evolution.py** placeholder file to open it.

c. Open a terminal in VS Code. Click **Terminal > New Terminal**.

d. Use the following commands to install the packages needed in this lab. These packages may already be installed on your VM. If so, you will get a **Requirement already satisfied** message.

```
devasc@labvm:~/labs/devnet-src/security$ pip3 install pyotp
Defaulting to user installation because normal site-packages is not writeable
Collecting pyotp
  Using cached pyotp-2.3.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: pyotp
Successfully installed pyotp-2.3.0
devasc@labvm:~/labs/devnet-src/security$ pip3 install flask
Defaulting to user installation because normal site-packages is not writeable
Collecting flask
  Using cached Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Requirement already satisfied: click>=5.1 in /home/devasc/.local/lib/python3.8/site-
packages (from flask) (7.1.2)
Requirement already satisfied: Jinja2>=2.10.1 in /usr/lib/python3/dist-packages (from
flask) (2.10.1)
Requirement already satisfied: itsdangerous>=0.24 in
/home/devasc/.local/lib/python3.8/site-packages (from flask) (1.1.0)
Requirement already satisfied: Werkzeug>=0.15 in
/home/devasc/.local/lib/python3.8/site-packages (from flask) (1.0.1)
Installing collected packages: flask
Successfully installed flask-1.1.2
devasc@labvm:~/labs/devnet-src/security$
```

Step 2: Import libraries.

In the **password-evolution.py** file, add the following code. Notice the command, **db_name = 'test.db'**. This is an SQL database (sqlite3) that stores the usernames and passwords that you will be creating.

```
import pyotp    #generates one-time passwords
import sqlite3  #database for username/passwords
import hashlib  #secure hashes and message digests
import uuid     #for creating universally unique identifiers
from flask import Flask, request
app = Flask(__name__) #Be sure to use two underscores before and after "name"

db_name = 'test.db'
```

Step 3: Create a local web service.

a. Next, add the following Flask code into the file to create the first phrase of web content at the root path. When the user goes to URL (root directory), the output of the return statement will be displayed in the browser.

```
@app.route('/')
def index():
    return 'Welcome to the hands-on lab for an evolution of password
systems!'
```

b. Add the following code to the file to create a local web service on port 5000 with a self-signed TLS certificate. The parameter **ssl_context='adhoc'** allows you to run an application over HTTPS without

having to use certificates or when using a self-signed certificate. Be sure to use two underscores before and after **name** and **main**.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

c. Save and run the **password-evolution.py** file. The **nohup** (no hangup) command keeps the process running even after exiting the shell or terminal. The **&** makes the command run in the background.

```
devasc@labvm:~/labs/devnet-src/security$ nohup python3 password-evolution.py
&
[1] 26329
devasc@labvm:~/labs/devnet-src/security$ nohup: ignoring input and appending
output to 'nohup.out'


devasc@labvm:~/labs/devnet-src/security$
```

d. Press **Enter** to get a new command prompt.

e. Your Flask server is now running. In VS Code in the **/security** folder, you should see the **nohup.out** text file created by Flask. Click the file to read its output.

f. Verify that the web service has started. Be sure to use HTTPS and not HTTP. The **-k** option allows curl to perform "insecure" SSL connections and transfers. Without the **-k** option, you will receive an error message, "SSL certificate problem: self-signed certificate". The command will display the message from the **return** command you coded in your script.

```
devasc@labvm:~/labs/devnet-src/security$ curl -k https://0.0.0.0:5000/
Welcome to the hands-on lab for an evolution of password
systems!devasc@labvm:~/labs/devnet-src/security$
```

g. Press **Enter** to get a command prompt one a new line.

h. Before continuing, terminate the script. Use the following command to stop it:

```
devasc@labvm:~/labs/devnet-src/security$ pkill -f password-evolution.py
devasc@labvm:~/labs/devnet-src/security$
```

Part 4: Explore Python Code Storing Passwords in Plain Text

When passwords were first used, they were simply stored in a database as plaintext. When the user entered their credentials, the system looked up the password to see if it matched. The system was very easy to implement, but also very insecure. In this Part, you will modify the **password-evolution.py** python file to allow it to store user identity in the **test.db** database. You will then create a user and perform an authentication against these credentials. Finally, you will examine the **test.db** database to verify they were stored in plaintext.

Step 1: Remove the server configuration.

Remove the following lines from the **password-evolution.py** python file. You will add this code back later.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

Step 2: Configure the server to store credentials.

a. Append (copy) the following Flask code to configure the server to store a username and password for a user in plaintext. Using the HTTP POST method, this code allows a user to create ("signup") a new username and password that will be stored in the test.db SQL database file. Later when the user enters in a username and password, this code will return the message "signup success".

```
########################################## Plain Text
############################################################
@app.route('/signup/v1', methods=['POST'])
def signup_v1():
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS USER_PLAIN
           (USERNAME  TEXT    PRIMARY KEY NOT NULL,
            PASSWORD  TEXT    NOT NULL);''')
    conn.commit()
    try:
        c.execute("INSERT INTO USER_PLAIN (USERNAME,PASSWORD) "
                  "VALUES ('{0}', '{1}')".format(request.form['username'],
request.form['password']))
        conn.commit()
    except sqlite3.IntegrityError:
        return "username has been registered."
    print('username: ', request.form['username'], ' password: ',
request.form['password'])
    return "signup success"
```

Note: Be careful of word wrap in the above code. Be sure to indent properly or the code may not work correctly.

b. Append (copy) the following Flask code to your **password-evolution.py** file to verify the new account credentials.

```
def verify_plain(username, password):
    conn = sqlite3.connect('test.db')
    c = conn.cursor()
    query = "SELECT PASSWORD FROM USER_PLAIN WHERE USERNAME =
'{0}'".format(username)
    c.execute(query)
    records = c.fetchone()
    conn.close()
    if not records:
        return False
    return records[0] == password
```

c. Append (copy) the following Flask code to your **password-evolution.py** file. This code is used during each login attempt to read the parameters from an HTTP request and verify the account. If the login is successful, the message "login success" will be returned, otherwise the user will see the message "Invalid username/password".

```
@app.route('/login/v1', methods=['GET', 'POST'])
def login_v1():
    error = None
    if request.method == 'POST':
        if verify_plain(request.form['username'], request.form['password']):
            error = 'login success'
        else:
```

```
            error = 'Invalid username/password'
    else:
        error = 'Invalid Method'
    return error
```

Step 3: Run the server and test it.

a. Add back the server configuration code you deleted earlier.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

b. Save and run the script to start the updated web service.

```
devasc@labvm:~/labs/devnet-src/security$ nohup python3 password-evolution.py
&
[1] 27826
devasc@labvm:~/labs/devnet-src/security$ nohup: ignoring input and appending output to
'nohup.out'
```

c. Use the following curl commands to create (signup) two user accounts, **alice** and **bob**, and send a POST to the web service. Each command includes the username, password, and the signup function being called that stores this information including the password as plaintext. You should see the "signup success" message from the **return** command that you included in the previous step.

Note: After each command, press **Enter** to get a command prompt on a new line.

```
devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=alice'
-F 'password=myalicepassword'  'https://0.0.0.0:5000/signup/v1'
signup successdevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=bob' -F
'password=passwordforbob'  'https://0.0.0.0:5000/signup/v1'
signup successdevasc@labvm:~/labs/devnet-src/security$
devasc@labvm:~/labs/devnet-src/security$
```

Step 4: Verify your new users can login.

a. Use the following curl commands to verify that both users can login with their passwords that are stored in plaintext.

```
devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=alice'
-F 'password=myalicepassword' 'https://0.0.0.0:5000/login/v1'
login successdevasc@labvm:~/labs/devnet-src/security$
devasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=bob' -F
'password=passwordforbob' 'https://0.0.0.0:5000/login/v1'
login successdevasc@labvm:~/labs/devnet-src/security$
```

b. Terminate the server.

```
login successdevasc@labvm:~/labs/devnet-src/security$ pkill -f password-
evolution.py
[1]+  Terminated              nohup python3 password-evolution.py
devasc@labvm:~/labs/devnet-src/security$
```

Step 5: Verify the contents of test.db.

You may have noticed that SQLite created a **test.db** file in your **/security** folder. You can **cat** this file and see the username and passwords for **alice** and **bob**. However, in this step you will use an application for viewing SQLite database files.

a. Open the **DB Browser for SQLite** application
   - Select the **menu** icon on the lower-left of the VM.
   - Select: Applications > **All**
   - Select :**DB Browser for SQLite**

b. After the **DB Browser for SQLite** is running, open the **test.db** file:
   - Select: **File > Open database…**
   - Navigate to the **labs/devnet-src/security** directory and select **test.db**.
   - Click **Open**.

c. In the **Database Structure** tab, notice the USER_PLAIN table that coincides with the code you created earlier.

d. Expand the table to see the two fields: USERNAME and PASSWORD.

e. Select the **Browse Data** tab.

   The **Table: USER_PLAIN** is already selected. Here you can see usernames **bob** and **alice** along with their passwords in plaintext.

f. Close the **DB Browser for SQLite** application.

Part 5: Password Hashing in Python

Instead of storing passwords in plaintext, you can hash it when it is created. When the password is hashed, it is converted into an unreadable collection of characters. This prevents anyone from converting it back to its correct, plaintext version. Even if the database is stolen it cannot be used because the hash is not known. You will now modify the **password-evolution.py** file to create a web API that can accept a web request and save a new user's password in a hashed format.

Step 1: Remove the server configuration.

a. Remove the following two lines from the **password-evolution.py** file. These lines will be appended again later.

```
if __name__ == '__main__':
app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

Step 2: Configure the server to store credentials.

a. Add the following code to the bottom of the file to enable the server to hash the password using SHA256 hashing method. Notice that this code is similar to the code you included previously. This code allows a user to create ("signup") a new username and password that will be stored in the **test.db** SQL database file. The difference is that the passwords will be stored as hash values instead of being in plaintext. This routine uses sha256 but does not salt the hash. You will see the implications of using a hash without salt when you view the **test.db** database file.

```
######################################## Password Hashing
############################################################


@app.route('/signup/v2', methods=['GET', 'POST'])
def signup_v2():
```

```
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS USER_HASH
           (USERNAME  TEXT    PRIMARY KEY NOT NULL,
            HASH      TEXT    NOT NULL);''')
    conn.commit()
    try:
        hash_value =
hashlib.sha256(request.form['password'].encode()).hexdigest()
        c.execute("INSERT INTO USER_HASH (USERNAME, HASH) "
                  "VALUES ('{0}', '{1}')".format(request.form['username'],
hash_value))
        conn.commit()
    except sqlite3.IntegrityError:
        return "username has been registered."
    print('username: ', request.form['username'], ' password: ',
request.form['password'], ' hash: ', hash_value)
    return "signup success"
```

b. Append (copy) the following code to your **password-evolution.py** file to verify that the password has been stored only in hashed format. The code defines the function **verify_hash** which compares the username and the password in hash format. When the comparison is true, the password has been stored only in its hash format.

```
def verify_hash(username, password):
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    query = "SELECT HASH FROM USER_HASH WHERE USERNAME =
'{0}'".format(username)
    c.execute(query)
    records = c.fetchone()
    conn.close()
    if not records:
        return False
    return records[0] == hashlib.sha256(password.encode()).hexdigest()
```

c. Append (copy) the following code to your **password-evolution.py** file. The following code reads the parameters from an HTTP POST request and verifies that the user has provided the correct password during login.

```
@app.route('/login/v2', methods=['GET', 'POST'])
def login_v2():
    error = None
    if request.method == 'POST':
        if verify_hash(request.form['username'], request.form['password']):
            error = 'login success'
        else:
            error = 'Invalid username/password'
    else:
        error = 'Invalid Method'
```

```
    return error
```

Step 3: Run the server and test it.

a. Add back the server configuration code you deleted earlier.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

b. Save and then run the script to start the updated web service.

```
devasc@labvm:~/labs/devnet-src/security$ nohup python3 password-evolution.py
&
[1] 28411
devasc@labvm:~/labs/devnet-src/security$ nohup: ignoring input and appending output to
'nohup.out'
```

c. Use the following curl commands to create three new user accounts with a hashed password. Notice that two of the users, **rick** and **allan**, are using the same password.

```
devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=rick' -
F 'password=samepassword' 'https://0.0.0.0:5000/signup/v2'
signup successdevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=allan'
-F 'password=samepassword' 'https://0.0.0.0:5000/signup/v2'
signup successdevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=dave' -
F 'password=differentpassword' 'https://0.0.0.0:5000/signup/v2'
signup successdevasc@labvm:~/labs/devnet-src/security$
```

d. Use curl commands to verify the login of all three users with their hash-stored passwords. The user **allan** is entered in twice, the first time with the wrong password. Notice the "Invalid username/password" that coincides with the code for this function that you added in a previous step.

```
devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=rick' -
F 'password=samepassword' 'https://0.0.0.0:5000/login/v2'
login successdevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=allan'
-F 'password=wrongpassword' 'https://0.0.0.0:5000/login/v2'
Invalid username/passworddevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=allan'
-F 'password=samepassword' 'https://0.0.0.0:5000/login/v2'
login successdevasc@labvm:~/labs/devnet-src/security$

devasc@labvm:~/labs/devnet-src/security$ curl -k -X POST -F 'username=dave' -
F 'password=differentpassword' 'https://0.0.0.0:5000/login/v2'
login successdevasc@labvm:~/labs/devnet-src/security$
```

This confirms that the hashed password is safely stored, and the passwords of users are protected should they become compromised.

e. Terminate the server.

```
devasc@labvm:~/labs/devnet-src/security$ pkill -f password-evolution.py
[1]+  Terminated              nohup python3 password-evolution.py
devasc@labvm:~/labs/devnet-src/security$
```

Step 4: Verify the contents of test.db.

a. Open the **DB Browser for SQLite** application.

b. Open the **test.db** file.

c. Select the tab, **Database Structure**.

You will notice two structures that coincide with the code you included earlier: USER_PLAIN and USER HASH.

d. Select the **Browse Data** tab.

e. The **Table: USER_HASH** should already be selected. You will now see the usernames **rick**, **allan**, and **dave** along with their hashed passwords. (You may need to adjust the table cells.) Notice that **rick** and **allan** have the same hashed passwords. This is because they had the same password and the hash function did not include a salt to make their hash unique. Salting the hash is the process of adding random data to a hash. To guarantee the uniqueness of the passwords, increase their complexity, and prevent password attacks even when the inputs are the same, a salt should be added to the input of a hash function.

Part 6: Review the Final Program

The following is the complete script you created in this lab.

```
import pyotp
import sqlite3
import hashlib
import uuid
from flask import Flask, request
app = Flask(__name__)

db_name = 'test.db'
@app.route('/')
def index():
    return 'Welcome to the hands-on lab for an evolution of password
systems!'
######################################## Plain Text
#########################################################
@app.route('/signup/v1', methods=['POST'])
def signup_v1():
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS USER_PLAIN
           (USERNAME  TEXT    PRIMARY KEY NOT NULL,
            PASSWORD  TEXT    NOT NULL);''')
    conn.commit()
    try:
        c.execute("INSERT INTO USER_PLAIN (USERNAME,PASSWORD) "
```

```
                    "VALUES ('{0}', '{1}')".format(request.form['username'],
request.form['password']))
        conn.commit()
    except sqlite3.IntegrityError:
        return "username has been registered."
    print('username: ', request.form['username'], ' password: ',
request.form['password'])
    return "signup success"
def verify_plain(username, password):
    conn = sqlite3.connect('test.db')
    c = conn.cursor()
    query = "SELECT PASSWORD FROM USER_PLAIN WHERE USERNAME =
'{0}'".format(username)
    c.execute(query)
    records = c.fetchone()
    conn.close()
    if not records:
        return False
    return records[0] == password
@app.route('/login/v1', methods=['GET', 'POST'])
def login_v1():
    error = None
    if request.method == 'POST':
        if verify_plain(request.form['username'], request.form['password']):
            error = 'login success'
        else:
            error = 'Invalid username/password'
    else:
        error = 'Invalid Method'
    return error
######################################## Password Hashing
############################################################

@app.route('/signup/v2', methods=['GET', 'POST'])
def signup_v2():
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS USER_HASH
           (USERNAME  TEXT    PRIMARY KEY NOT NULL,
            HASH      TEXT    NOT NULL);''')
    conn.commit()
    try:
        hash_value =
hashlib.sha256(request.form['password'].encode()).hexdigest()
        c.execute("INSERT INTO USER_HASH (USERNAME, HASH) "
```

```
                    "VALUES ('{0}', '{1}')".format(request.form['username'],
hash_value))
            conn.commit()
        except sqlite3.IntegrityError:
            return "username has been registered."
        print('username: ', request.form['username'], ' password: ',
request.form['password'], ' hash: ', hash_value)
        return "signup success"
def verify_hash(username, password):
        conn = sqlite3.connect(db_name)
        c = conn.cursor()
        query = "SELECT HASH FROM USER_HASH WHERE USERNAME =
'{0}'".format(username)
        c.execute(query)
        records = c.fetchone()
        conn.close()
        if not records:
            return False
        return records[0] == hashlib.sha256(password.encode()).hexdigest()
@app.route('/login/v2', methods=['GET', 'POST'])
def login_v2():
        error = None
        if request.method == 'POST':
            if verify_hash(request.form['username'], request.form['password']):
                error = 'login success'
            else:
                error = 'Invalid username/password'
        else:
            error = 'Invalid Method'
        return error
if __name__ == '__main__':
            app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```