## Lab 14: Exploring Password Storage Techniques

### Case Study

SecureLogic Technologies, a cybersecurity training provider, is committed to preparing future professionals to build and defend secure applications. As part of its internal DevSecOps training program, the company introduced a hands-on lab focused on credential storage techniques. With increasing cybersecurity threats targeting user authentication systems, SecureLogic aimed to educate developers on the importance of secure password handling practices. The lab explores how passwords are stored in plaintext versus using hashing methods and demonstrates the real-world risks and mitigations associated with each.

### Business Challenge

SecureLogic noticed that many junior developers entering the industry were still unaware of secure password storage practices. Applications were being built with credentials stored in plaintext within databases, making them easy targets for attackers during data breaches. This lack of awareness posed a serious threat to client systems, compliance standards, and user trust.

The organization needed a practical training scenario to show how insecure storage methods expose sensitive data and how password hashing can be implemented to mitigate these risks. The objective was to shift developer mindsets from functionality-first coding to security-by-design approaches

### Solution

In this lab, SecureLogic developed a lab-based exercise focused on the evolution of password storage methods. The solution followed a simple progression:

1. Install Packages and Create a Web Service
2. Explore Python Code Storing Passwords in Plain Text
3. Explore Python Code Storing Passwords Using a Hash

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **// Install Packages and Create a Web Service**  1. 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.  2. Open VS code. Then click **File > Open** .. and navigate to the **devnet-src/security** directory. Click **OK**. In the **EXPLORER** panel, click the **password-evolution.py** placeholder file to open it.    3. Open a terminal in VS Code. Click **Terminal** and then **New Terminal**.    4. Use the **pip3 install pyotp** command 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.**    5. 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' |   6. 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!' |     7. 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') |     8. Save and run the password-evolution.py 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] 5725  devasc@labvm:~/labs/devnet-src/security$ nohup: ignoring input and appending output to 'nohup.out' |     9. Press **Enter** to get a new command prompt. Your Flask server is now running. In VS Code in the **/security** folder, you should see the **out** text file created by Flask. Click the file to read its output.  10. 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.    11. Press **Enter** to get a command prompt one a new line. Before continuing, terminate the script. Use thecommand **pkill -f password-evolution.py** to stop it:  **// Explore Python Code Storing Passwords in Plain Text**  1. Initially, passwords were stored in plaintext within databases and directly compared during login—simple but insecure. In this part, you will update **password-evolution.py** to store user credentials in **test.db**, create a user, authenticate them, and verify that the password is saved in plaintext.  2. 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') |   3. 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.  4. 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 |   5. 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') |   6. Add back the server configuration code you deleted earlier. 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' |   7. 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$ |   8. Verify your new users can login. Use the following **curl** commands to verify that both users can login with their passwords that are stored in plaintext. Then terminate the server.   |  | | --- | | 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'  sigup success |     9. You can now check the **test.db** file created in the /security folder. Instead of using **cat**, you will use a graphical tool to inspect the data.  10. First, open the DB Browser for SQLite application by clicking the menu icon in the lower-left corner of the VM. From there, navigate to Applications, then All, and select DB Browser for SQLite. Once the application is running, go to File and choose Open Database. Navigate to the **labs/devnet-src/security** directory, select the **test.db** file, and click Open.  11. In the Database Structure tab, you will see the **USER\_PLAIN** table that matches the structure defined in your code. Expand the table to confirm it contains the **USERNAME** and PASSWORD fields. Next, select the Browse Data tab. With the USER\_PLAIN table already selected, you should see the usernames **bob** and **alice**, along with their corresponding passwords stored in plaintext. Lastly, close the DB Browser for SQLite application once you’ve verified the data.  **// Explore Python Code Storing Passwords Using a Hash**  1. 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.  2. Remove the following two lines from the password-evolution.py These lines will be appended again later.   |  | | --- | | if \_\_name\_\_ == '\_\_main\_\_':  app.run(host='0.0.0.0', port=5000, ssl\_context='adhoc') |   3. 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 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" |   4. 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() |   5. Append (copy) the following code to your password-evolution.py 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 |   6. Add back the server configuration code you deleted earlier.   |  | | --- | | if \_\_name\_\_ == '\_\_main\_\_':  app.run(host='0.0.0.0', port=5000, ssl\_context='adhoc') |   7. 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' |   8. 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$ |   9. 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$ |   10. This confirms that the hashed password is safely stored, and the passwords of users are protected should they become compromised. Terminate the server usind command **pkill -f password-evolution.py.**  11. Open the DB Browser for SQLite    12. Open the db file.    13. Select the tab, **Database Structure.** You will notice two structures that coincide with the code you included earlier: **USER\_PLAIN** and **USER HASH**.    14. Select the **Browse Data.** 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. |