Security Practical 1

Dr Chris G. Willcocks

Last Modified: October 2, 2019

Practical 1

Background

In this practical series you will be you will be developing, cracking, and securing a web server written in Python 3.7. The majority of computer security issues occur around client and server applications. The aim of this practical series is to raise awareness of such issues, and also raise awareness of how easy they are to exploit. Although not mandatory, you are encouraged to complete this practical series in Linux if it is available on your machine.

Non-Python Users

If you do not have any Python experience, do not worry as you will be editing an existing codebase without any requirement of Python or OOP. The concepts in these practicals extend to other server-side languages (such as Java or a C++ server).

Setup

Ensure you are in a Python 3.7 activated environment. For example, you can use miniconda `https://docs.conda.io/en/latest/miniconda.html`, then: `source miniconda3/bin/activate` and also ipython is helpful for debugging in an interactive session `pip install ipython`.

Tasks

1. Change directory to '1' and run the"hello world" server:

Listing 1: Bash

```
cd ~/practicals/1
python server.py
```

   (a) Navigate to "`http://127.0.0.1:8000`"
   (b) You should see a message: "The server says Hello World!"
   (c) Shutdown the server with Ctrl+C in the terminal.

2. Change directory to '2' and run the "basic" server with authentication:

Listing 2: Bash

```
cd ~/practicals/2
python server.py
```

   (a) Navigate again to "`http://127.0.0.1:8000`"
   (b) Try to login a few times from the browser.
   (c) Study the server code and login successfully.
   (d) Clear the browser cookies and other site data such that you are logged out.
   (e) We will now attempt to hack this password, so keep the server running.

3. Crack the basic authentication with a dictionary attack:

   (a) Open a **new terminal instance** and change directory to '3'. Run the "password cracker":

   Listing 3: Bash

   ```
   cd ~/practicals/3
   python cracker.py
   target URL: http://127.0.0.1:8000
   ```

   (b) You should now see a list of hacked users with their passwords.

   (c) You may wish to try logging in with one of these users details.

   (d) Discuss with the person next to you or the demonstrator what the problem is with all these passwords. What rules would guarantee better passwords? How would you implement these rules?

   (e) If interrested, review the code at "`https://github.com/Sanix-Darker/Brute-Force-Login/blob/master/main.py`" to see how you can access different login fields from webpages.

4. Now make the server resistant to dictionary attacks.

   (a) Write any code to prevent frequent login attempts

   (b) Perhaps the most intuitive approach is with the following parameterization:

   Listing 4: Python

   ```
   max_login_retries = 3
   retry_period = 300.0
   lockout_length = 6000.0
   ```

   (c) Although, for high-performance servers, can you think of some heat function which does not require any data structures or loop iterations? Consider this parameterisation:

   Listing 5: Python

   ```
   heat = 0.0
   last_heat = 0.0
   ```

   (d) Hint, in your solutions you may (or may not) wish to use the following:

   Listing 6: Python

   ```
   import time
   time.time()
   self.client_address[0]
   ```

   (e) Confirm your solution prevents hacking through the dictionary attack.

   (f) Confirm that you can still login normally through the browser (you may need to clear the browsing cache/cookies accordingly, and restart the server to remove any accumulated login heat).

   (g) Discuss your approach with the person next to you or the demonstrator.

      i. Can you think of any problems with it?

      ii. How efficient will it be with thousands of users?

      iii. Would one hacker be able to exploit your defences to prevent lots of users from logging in? How would you mitigate this?

   (h) For more information, you can read '`https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks`'

5. In the event that your server gets hacked, it is important that your user passwords are not explicitly stored, as you don't want hackers to access your users bank details, email, facebook etc. Hash the user passwords & ensure that login continues to work correctly. You may store the hashed passwords as either bytes data, or as strings with hexidecimal values.

   (a) You may use the following code to generate a SHA3-256 hash:

   Listing 7: Python

   ```
   hashlib.sha3_256(b'plain text password').hexdigest()
   # this outputs: '8761ca675f2be417e8b0a1ede5c67440498cc453ceda7377e98b177b1e07072b'
   ```

(b) Also you may wish to decode the authorization header, for example:

Listing 8: Python

```
def verify(self, data):
    raw_data = base64.b64decode(data[6:]).decode('UTF-8')
    username = raw_data.split(':')[0]
    password = raw_data.split(':')[1]
    ...
```

(c) Confirm that you can still login normally with the newly hashed passwords.

6. One of your employees was rightfully furious for not switching sooner to Arch Linux, however he leaked all the server data and code on wikileaks! The BBC is covering the story and everyone is now able to download the lists of usernames, emails, and hashed passwords.

   (a) Discuss (or draw a diagram) with the person next to you/demonstrator, how to use a rainbow table to recover the original plaintext passwords. Make sure that you both understand how rainbow tables actually work (reduction function, chains, storage/performance trade-off, etc). The explanation here is good: `http://kestas.kuliukas.com/RainbowTables/`. Also the top answer here is good: `https://security.stackexchange.com/questions/379/what-are-rainbow-tables-and-how-are-they-used`.

   (b) Another interesting read is the answer: `https://security.stackexchange.com/questions/3448/how-long-does-it-take-to-actually-generate-rainbow-tables`

7. Protect against rainbow tables by adding a 64-bit salt for each of your passwords. You can read about salt on wikipedia: `https://en.wikipedia.org/wiki/Salt_(cryptography)`.

   (a) You can generate a 64-bit salt using:

   Listing 9: Python

   ```
   import secrets
   secrets.token_hex(8)
   # this outputs n random bytes in hexidecimal such as: 'e7f655db5956a220'
   ```

   (b) Test that you can login with the updated password system that uses salt.

   (c) Ensure that your hashes for users 'chris' and 'test' (who have the same password) are different.

   (d) Discuss with the person next to you why salted password make rainbow tables or similar pre-computed hashed attacks infeasible.

   (e) Even using salt, can some of the passwords still be hacked? How would you prevent this?

8. Current secure approaches:

   (a) MongoDB (a very good NoSQL database) and many other open source software suggest using SCRAM for challenge-response authentication.

   (b) Read up about the SCRAM at the RFC: `https://tools.ietf.org/html/rfc5802` and in the white paper: `https://www.isode.com/whitepapers/scram.html`

   (c) See how you would use it (Python): `http://api.mongodb.com/python/current/examples/authentication.html`

9. This concludes the practical. If you finish early, you may wish to extend the server & cracker with more functionality and smarter attacking or protection methods. For example you may wish to improve the Cracker by combining the dictionary lookup with other knowledge of the user, such as their date-of-birth or other personal information. You may also wish to brute-force permutations of: single-random-mixed-case-letter, random-word, character-of-username, date-of-brith, random-number, and think about how large the search space becomes. Would it be feasible to write a smart cracker than can find passwords such as c080697kitten! where 'c' = first character of username, 080697 = date of birth, kitten=random word from 1000 popular password word list, '!' = common random symbol?