Security Practical 1

Dr Chris G. Willcocks

Last Modified: October 20, 2017

Practical 1

Background

In this practical series you will be you will be developing, cracking, and securing a web server written in Java. 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-Java users

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

Windows users

If you are using Windows, please add Java 1.8 to the path before compiling any code:

Listing 1: Windows CMD

```
set PATH="C:\Program Files (x86)\Java\jdk1.8.0_111\bin";%PATH%
```

Tasks

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

   Listing 2: Bash

   ```
   cd ~/practicals/1
   javac Server.java && java Server
   ```

   (a) Navigate to "http://127.0.0.1:8000/test"

   (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 3: Bash

   ```
   cd ~/practicals/2
   javac Server.java && java Server
   ```

   (a) Navigate to "http://127.0.0.1:8000/login"

   (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 4: Bash

   ```
   cd ~/practicals/3
   javac Cracker.java && java Cracker
   ```

   (b) You should see the private key in the terminal after a successful login.
   (c) You may wish to try commenting out the break statement to hack the other user data.
   (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?

4. Now make the server resistant to dictionary attacks.
   (a) Write any code to prevent frequent login attempts (to save time, this does not to be for each user or IP address)
   (b) You may wish to parameterize the function with the following:

   Listing 5: Java

   ```
   private int maxLoginRetries = 3;
   private double retryPeriod = 300.0;
   private double lockoutLength = 6000.0;
   ```

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

   Listing 6: Java

   ```
   private double heat = 0;
   private long lastHeat = 0;
   ```

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

   Listing 7: Java

   ```
   System.currentTimeMillis()
   private List<Long> something = new ArrayList<Long>();
   something.add(...);
   ```

   (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. Can you think of any problems with it? How efficient will it be with thousands of users?

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 arrays, or as strings with hexidecimal values.
   (a) You may use the following code to generate a SHA-256 hash:

   Listing 8: Java

   ```
     import java.security.MessageDigest;
     import javax.xml.bind.DatatypeConverter;

     ...

     byte[] messageBytes = pass.getBytes("UTF-8");
     MessageDigest md = MessageDigest.getInstance("SHA-256");
     byte[] digest = md.digest(messageBytes);
     String hex = DatatypeConverter.printHexBinary(digest);
   ```

   (b) 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! 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 top answer here is quite 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: 'https://en.wikipedia.org/wiki/Salt_(cryptography)'.

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

   Listing 9: Java

   ```java
   import java.util.Random;

   ...

   byte[] r = new byte[8];
   new Random().nextBytes(r);
   String hex = DatatypeConverter.printHexBinary(r);
   ```

   (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 (Java): `http://mongodb.github.io/mongo-java-driver/3.6/driver/tutorials/authentication/`

   (d) 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?