

Due: Feb 9

Name Jason Stumbaugh

Objective: The Foundations book contains a java-based sample web server called SimpleWebServer. This lab requires you to work with simple web server in order to understand the security goals and design principles presented in section 1 of this course.

1. Develop a set of security requirements for the Simple Web Server example from the book. Be sure to include what needs to be protected, from whom, and for how long. Also, indicate the cost of not protecting the data or assets as a note to each requirement.

- The web server shall immediately disconnect from a client that sends a malformed HTTP request. SimpleWebServer's processRequest() method currently assumes that the request will always have at least three words in it. If there are not three words, an exception will be thrown. There is nothing to handle this exception, and would cause the application to crash. This requirement protects the server from potential DoS attack by protecting against a carriage return being sent. With this implemented, the server is protected from malicious requests indefinitely. The cost of not protecting this flaw is that the server could crash with only one malicious request.
- The web server shall only serve files from the current directory. SimpleWebServer currently serves any file on the file system. A user can send an HTTP request for any file and, if it exists on the server, it would be served. If only the files from the current working directory are served, the rest of the file system is protected against malicious users attempting to access it. If this requirement is not implemented, a user might request the /etc/shadow file, which hold a list of the users on the server and their encrypted passwords. This requirement protects the files outside of the current directory indefinitely.
- The web server shall impose a download limit for files being served. Currently, the server assumes that the file can be loaded into memory and served to the client. While this may be true for most instances, a user might request a file larger than the server can hold in memory. This would cause the server to crash. The server would be protected against users requesting extremely large files or infinite files. A download limit would allow the server to send the file byte by byte up to a specified level, without the fear of loading too much of the file into memory.
- The web server must use SSL to establish a connection with the client. The web server now does not use SSL, which means that any of the files served are not encrypted. An

attacker could be eavesdropping on the connection and able to see everything being sent in clear text. An SSL connection would allow the server to encrypt the bytes being sent, and thus protect against an eavesdropping attack.

2. Problem 6 on page 77-78 of the book asks you to consider threats against a web server with file upload capability. Complete all parts (a – e) of Problem 6 and...

Include the following considerations:

1. Reference the threat characters that we talked about, which of these threats are involved in the various attack scenarios in this problem?
2. Explain the steps that lead to each successful attack.
3. Explain which of the security goals is broken by the attacks.

Problem 6) HTTP supports a mechanism that allows users to upload files in addition to retrieving them through a PUT command.

- a. What threats would you need to consider if SimpleWebServer also had functionality that could be used to upload files?
 - i. The threats that need to be considered if SimpleWebServer had file upload capability are protecting against malicious files, malicious file types, and files larger than the server can hold.
- b. For each of the specific threats you just listed, what types of security mechanisms might you put in place to mitigate the threats?
 - i. To protect against malicious files being uploaded, one could use an antivirus scanner on the file before accepting it. One could also make a whitelist of acceptable file types and block disallowed files from being uploaded. One way to protect against files being too large for the server is to specify a maximum file size that is acceptable to the server.
- c. Consider the following code, which allows for text file storage and logging functionality. Modify the `processRequest()` method in SimpleWebServer to use the preceding file storage and logging code.
 - i. The `processRequest()` function was modified to accept a PUT request by modifying the if statement to also check for PUT requests. If the request

is a PUT, then the name of the file is found from the path. The path is modified to remove everything up to the last backslash. The remaining string is the name of the file. The file is then opened and made into a `BufferedReader`. The reader and filename are then passed to the `storeFile` function to be stored on the server. A log entry is made into `log.txt` containing the command and the name of the file to be uploaded. The modified `processRequest()` function is shown in Excerpt 1.

```

/* Reads the HTTP request from the client, and
   responds with the file the user requested or
   a HTTP error code. */
public void processRequest(Socket s) throws Exception {
    /* used to read data from the client */
    BufferedReader br = new BufferedReader (new InputStreamReader
(s.getInputStream()));

    /* used to write data to the client */
    OutputStreamWriter osw = new OutputStreamWriter
(s.getOutputStream());

    /* read the HTTP request from the client */
    String request = br.readLine();
    String command = null;
    String pathname = null;

    /* parse the HTTP request */
    StringTokenizer st = new StringTokenizer (request, " ");
    command = st.nextToken();
    pathname = st.nextToken();

    if (command.equals("GET")) {
        /* if the request is a GET
           try to respond with the file
           the user is requesting */
        serveFile (osw,pathname);
    } else if (command.equals("PUT")) {
        /* if the request is a PUT, store the file */

        String file_to_write_to = pathname.replaceAll(".*\\/", "");

        FileReader input_fr = new FileReader(pathname);
        BufferedReader input_br = new BufferedReader(input_fr);

        storeFile(input_br, osw, file_to_write_to);
        logEntry("log.txt", command + " " + file_to_write_to);
    } else {
        /* if the request is a NOT a GET,
           return an error saying this server
           does not implement the requested command */
        osw.write ("HTTP/1.0 501 Not Implemented\n\n");
    }
}

```

```

    /* close the connection to the client */
    osw.close();
}

```

Excerpt 1: The modified processRequest() function accepting PUT requests, storing files, and logging them

- d. Run your web server and mount an attack that defaces the index.html home page.

After changing some of the code in storeFile() to respond with “HTTP/1.1 100 Continue” and adding “... throws Exception” to the end of logEntry()'s signature, I was successfully able to upload a file. I uploaded a file named index.html to the server, which would replace the existing index.html. The attack can be shown in Figure 1.

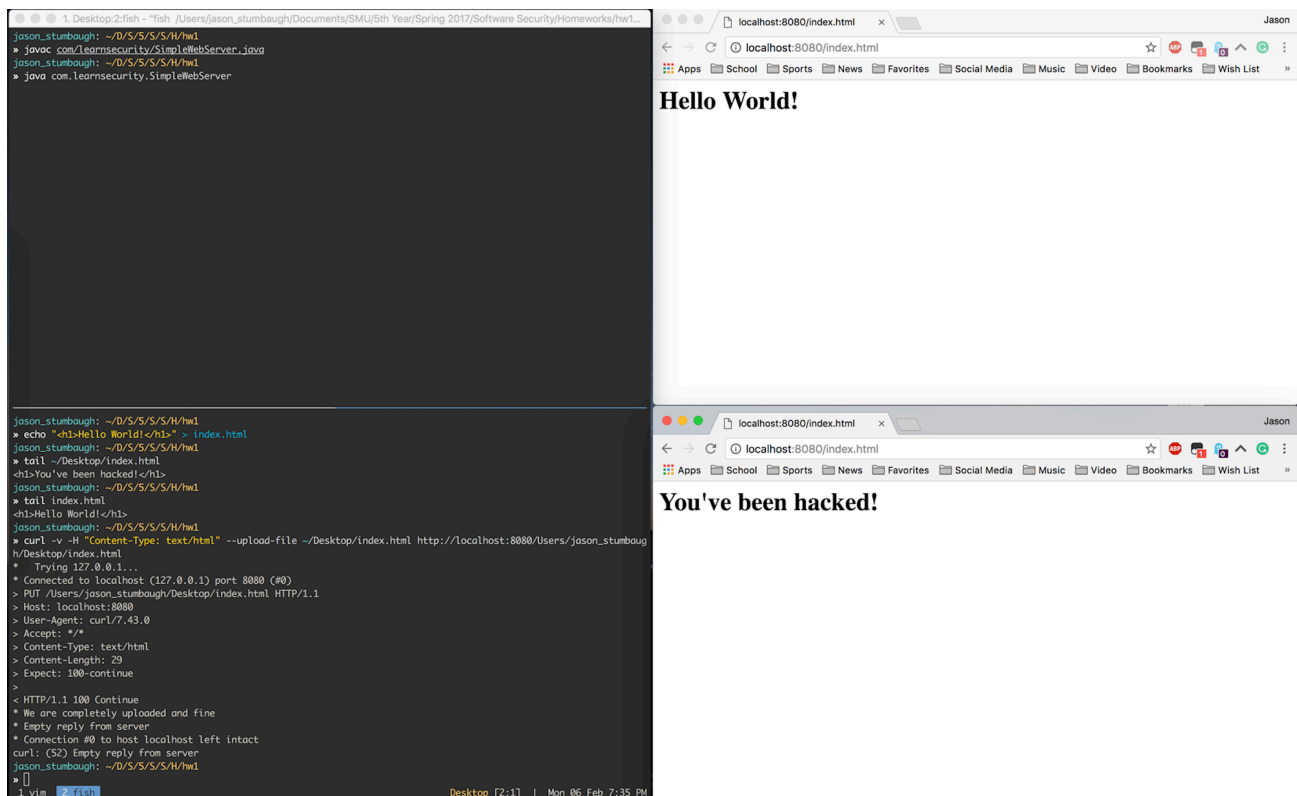


Figure 1: An attack to deface SimpleWebServer's home page.

- e. Assume that the web server is run as root on a Linux workstation. Mount an attack against SimpleWebServer in which you take ownership of the machine that it is running on. By taking ownership, we mean that you should be able to gain access to a root account, giving you unrestricted access to all the resources

on the system. Be sure to cover your tracks so that the web log does not indicate that you mounted an attack.

One way of gaining access to the machine that the server is running on is to request the `/etc/shadow` file, which contains the list of all users on the system and their encrypted passwords. One can run a dictionary attack against the passwords file to obtain the password of the root user. Since the IP address is already known, the attacker can ssh into the machine with the root user's credentials. This gives the attacker root access to the whole system. To cover your tracks, the attacker would have to go erase the logs that SimpleWebServer stores about them requesting the `/etc/shadow` file.

- 3.** Implement Problem 7 on page 78-79 of the book. Answer the questions in part *a* and *b*. Part *c* is optional, but you'll receive a 1337 point for implementing it.

Problem 7) Rewrite the `serveFile()` method such that it imposes a maximum file size limit. If a user attempts to download a file that is larger than the maximum allowed size, write a log entry to a file called `error_log` and return a "403 Forbidden" HTTP response code.

The `serveFile()` method originally serves any file requested. In order to make the server more secure, a maximum file size was implemented. When a file is requested, the server checks the length of the file in bytes. If the file is larger than the default maximum file size, chosen to be 1000 bytes, then it will not be served to the client. A log entry is made into `error_log.txt` saying that the request submitted was forbidden. Figure 2 shows the server refusing to serve a file larger than 1000 bytes.

```
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» javac com/learnsecurity/SimpleWebServer.java and java com.learnsecurity.SimpleWebServer
```

```
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» la
total 32
drwxr-xr-x  6 jason_stumbaugh  staff   204B Feb  7 18:12 .
drwxr-xr-x  5 jason_stumbaugh  staff   170B Feb  6 17:16 ..
-rw-r--r--@ 1 jason_stumbaugh  staff   6.0K Feb  7 18:12 .DS_Store
-rw-r--r--  1 jason_stumbaugh  staff   1.8K Feb  7 18:11 big_file.html
drwxr-xr-x  4 jason_stumbaugh  staff   136B Feb  6 16:59 com
-rw-r--r--  1 jason_stumbaugh  staff    22B Feb  7 18:08 index.html
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» echo -n "GET /big_file.html HTTP/1.0\n\n" | nc localhost 8080
HTTP/1.0 403 Forbidden
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» tail error_log.txt
Tue Feb 07 18:12:35 CST 2017 HTTP/1.0 403 Forbidden

jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» █
```

Figure 2: Attempting to access a file larger than the 1000 byte limit.

- What happens if an attacker tries to download `/dev/random` after you have made your modification?

When the `/dev/random` file is requested, the server crashes because it runs out of space. Since the file is zero bytes, it passes the file size check and attempts to serve it to the client. However, the server crashes because it attempts to load the file, which is randomly populated, into memory. The server eventually runs out of space and crashes due to too much of the file being loaded into memory.

Figure 3 shows the server in action when attempting to server /dev/random.

```
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» javac com/learnsecurity/SimpleWebServer.java and java com.learnsecurity.SimpleWebServer
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3332)
    at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:137)
    at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:121)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:622)
    at java.lang.StringBuffer.append(StringBuffer.java:383)
    at com.learnsecurity.SimpleWebServer.serveFile(SimpleWebServer.java:160)
    at com.learnsecurity.SimpleWebServer.processRequest(SimpleWebServer.java:66)
    at com.learnsecurity.SimpleWebServer.run(SimpleWebServer.java:38)
    at com.learnsecurity.SimpleWebServer.main(SimpleWebServer.java:183)
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» █

jason_stumbaugh: ~/D/S/S/S/S/H/hw1
» echo -n "GET //dev/random HTTP/1.0\n\n" | nc localhost 8080
jason_stumbaugh: ~/D/S/S/S/S/H/hw1
»
```

Figure 3: Serving /dev/random causes the server to run out of memory.

- b. What might be some alternative ways in which to implement the maximum file size limit?

Some alternative ways of implementing a maximum file size limit other than checking the size of the file are to not store the file in memory or impose a download limit. If the file requested is larger than the limit, the loading it into memory to check the size is a terrible idea. The alternative is to read in the file byte by byte up to a specified limit. This protects the server from loading a file larger than its memory. Another method is to impose a download limit. That is, send the file to the user byte by byte up to a specified limit.

4. Complete parts *a*, *b* and *c* of problem 9 on page 79 of the book.

Hint: For part *a* it might help to observe how real websites use HTTP authorization. Use an HTTP proxy like Paros to view the HTTP headers involved in the negotiation.

Hint: For part *c* Ethereal is now known as Wireshark. The user manuals are pretty helpful in getting started, but use of the tool is relatively straightforward. I don't mind questions and discussion on the use of the tool on the BlackBoard discussion forum (I created one called Security Tools Discussion).

Problem 9) Implement basic HTTP authorization for SimpleWebServer. Read the HTTP 1.0 specification for more details (www.w3.org/Protocols/rfc2616/rfc2616.html) on how basic HTTP authorization works.

- a. Instrument SimpleWebServer to store a username and password as data members. Require that any HTTP request to the web server be authorized by checking for an authorization HTTP header with a base64-encoded username and password. Requests that do not contain an authorization header should receive a WWW-Authentication challenge. Requests that do contain an authorization header should be authenticated against the username and password hard-coded in the SimpleWebServer class. (In Chapter 9, you will learn to build a proper password manager for SimpleWebServer so that the username and password do not need to be hard-coded.)

To implement authentication with SimpleWebServer, a default username, "default_username", and password, "default_password", were chosen. To parse the username and password, it needed to first be decoded from Base64. Once it was decoded, it could be compared to the default username and password. If authorization was successful, the server could process the rest of the request. If not, a 401 Unauthorized was returned with the "WWW-Authentication" challenge. Figure 4 shows the authorization process.


```

jason_stumbaugh: ~/D/S/5/S/S/s/h/c/learnsecurity (master ? :2)
» curl -v -u default_username http://localhost:8080
Enter host password for user 'default_username':
* Rebuilt URL to: http://localhost:8080/
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'default_username'
> GET / HTTP/1.1
> Host: localhost:8080
> Authorization: Basic ZGVmYXVsdF91c2VybmFtZTpKZWZhdWx0X3Bhc3N3b3Jk
> User-Agent: curl/7.43.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
<
<h1>Hello World!</h1>
* Closing connection 0
jason_stumbaugh: ~/D/S/5/S/S/s/h/c/learnsecurity (master ? :2)
» curl -v -u default_username http://localhost:8080
Enter host password for user 'default_username':
* Rebuilt URL to: http://localhost:8080/
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'default_username'
> GET / HTTP/1.1
> Host: localhost:8080
> Authorization: Basic ZGVmYXVsdF91c2VybmFtZTp0ZXN0
> User-Agent: curl/7.43.0
> Accept: */*
>
WWW-Authenticate: Basic realm=SimpleWebServer
* Connection #0 to host localhost left intact
HTTP/1.1 401 Unauthorized
jason_stumbaugh: ~/D/S/5/S/S/s/h/c/learnsecurity (master ? :2)
» █

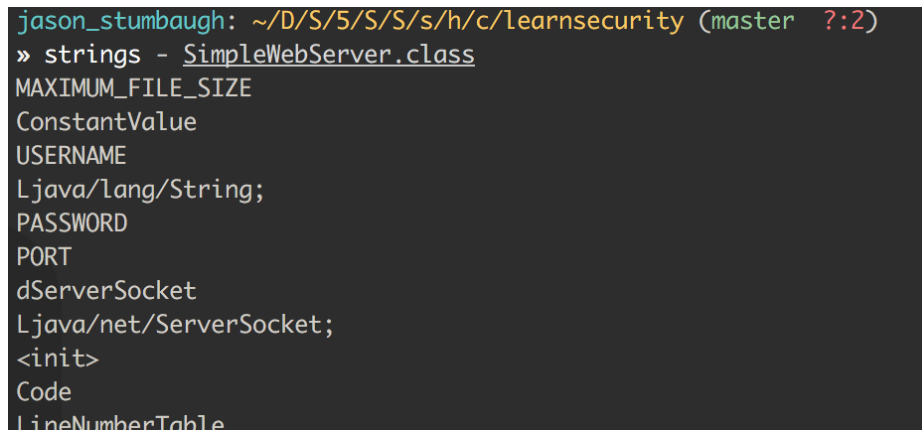
```

Figure 4: SimpleWebServer's authentication process

- b. Pretend that you are an attacker who got ahold of the compiled SimpleWebServer.class file. Run the strings utility on the compiled SimpleWebServer.class file to reveal the username and password that your modified web server requires. (If you are running a UNIX-based system, the strings utility is most likely preinstalled on your system. If you are running Windows, you can obtain the strings utility from www.sysinternals.com/Utilities/Strings.html.)

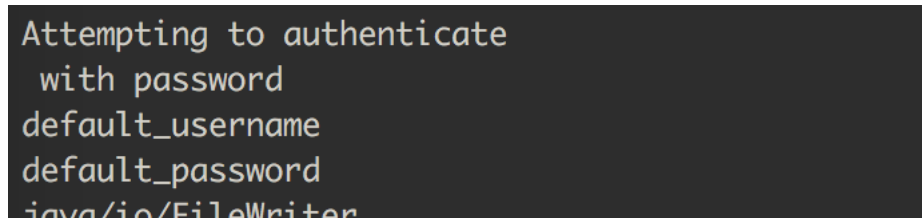
The strings command is very helpful in looking for strings in a compiled file. When run on the SimpleWebServer.class file, it returns all the strings in the file. There are a lot of results from the file, but when evaluating the results, one can see at the top the constant value of the username and password are declared, but their values are not explicitly stated. When looking further down the file, the

values of the default username and password are shown. Pictures of the output of “strings – SimpleWebServer.class” are shown in figures 5 and 6.



```
jason_stumbaugh: ~/D/S/5/S/S/s/h/c/learnsecurity (master ? :2)
» strings - SimpleWebServer.class
MAXIMUM_FILE_SIZE
ConstantValue
USERNAME
Ljava/lang/String;
PASSWORD
PORT
dServerSocket
Ljava/net/ServerSocket;
<init>
Code
LineNumberTable
```

Figure 5: Output of the strings command on a compiled SimpleWebServer



```
Attempting to authenticate
with password
default_username
default_password
java/io/FileWriter
```

Figure 6: A continuation of the output of the strings command

- c. Install Ethereal (www.ethereal.com) and a base64 decoder on your system. Make a few HTTP requests to your web server in which you use your username and password to authenticate. Use Ethereal to capture network traffic that is exchanged between your web client and server. Use the base64 decoder to convert the encoded username and password in the Ethereal logs to plain text.

Wireshark, known as Ethereal formally, is a very helpful tool for analyzing packets send on a network. When setup to capture packets on the loopback network, lo0, on which the SimpleWebServer is running, the initial GET request can be captured and analyzed. The HTTP header for the request is shown in the picture below. Thankfully, Wireshark already has a Base64 decoder built in, and it shows the decoded value of the credentials: “default_username: default_password”. Another request to the server was made using incorrect credentials just to make sure the decoding Wireshark does was correct. Figures 7 and 8 show Wireshark capturing and analyzing the GET requests.

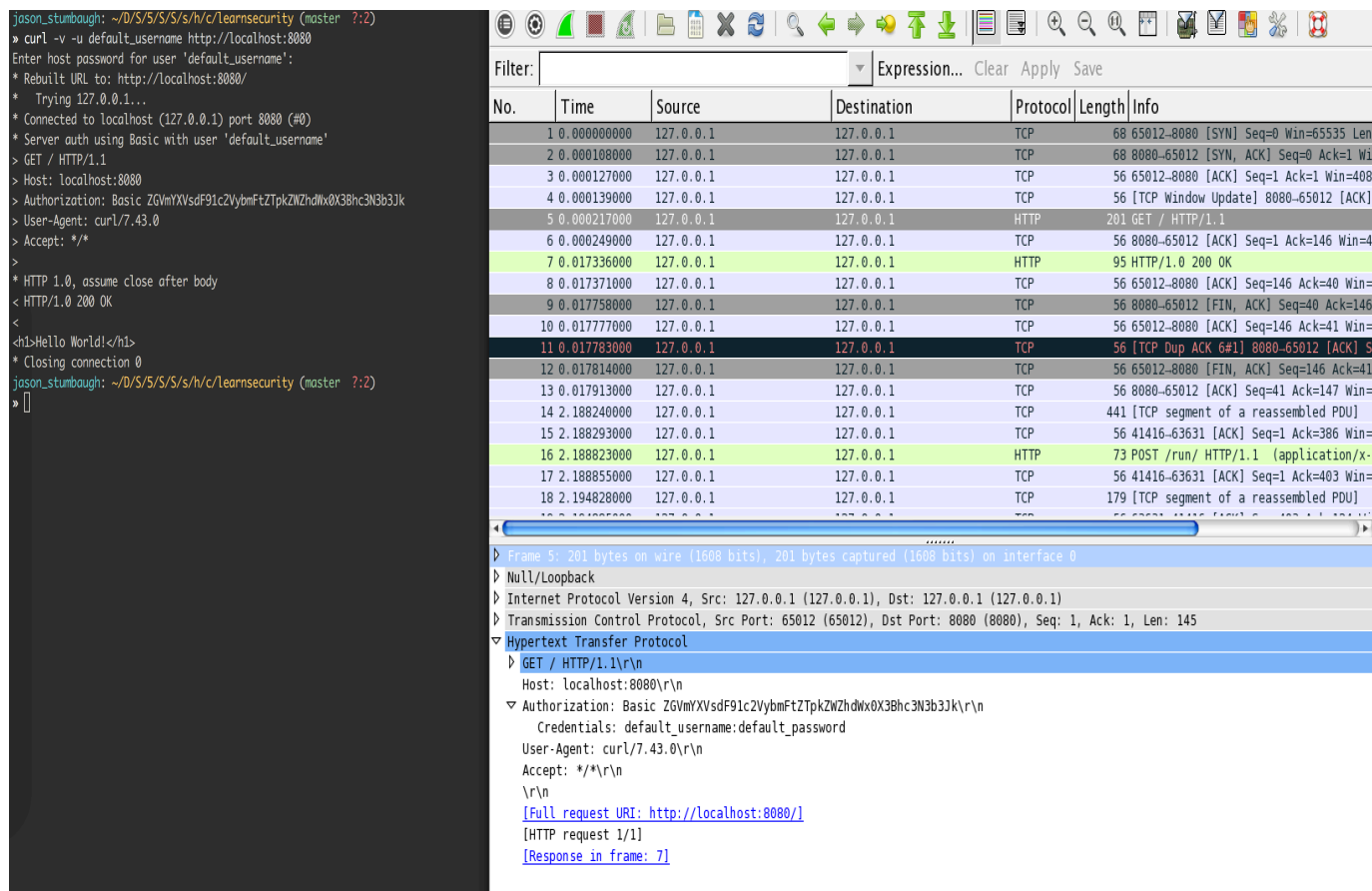


Figure 7: Wireshark capturing a GET request to SimpleWebServer with the correct credentials

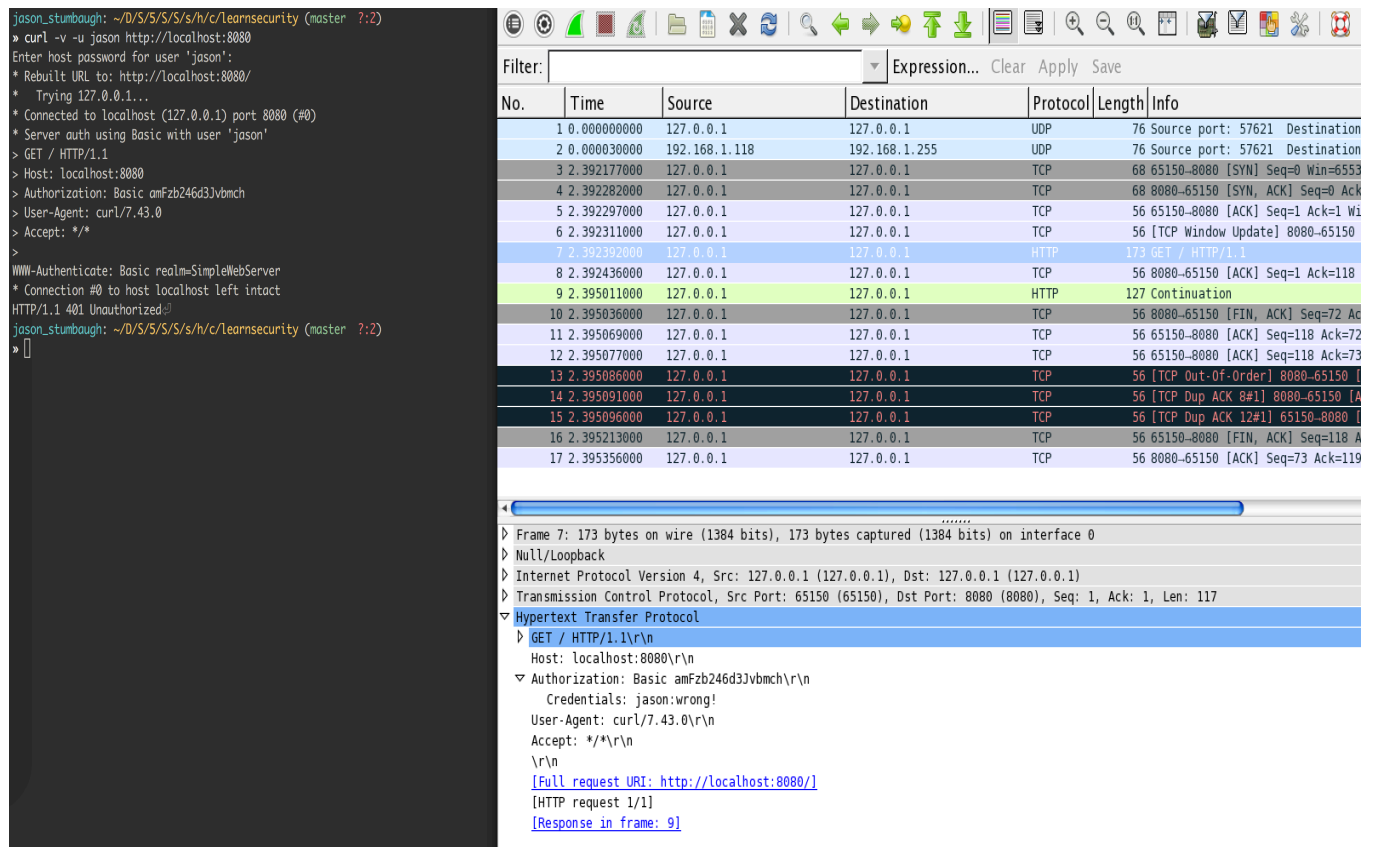


Figure 8: Wireshark capturing a GET request to SimpleWebServer with the incorrect credentials