

Lab 1 - Simple Web Server: Security Goals and Principles

CSE 5359 - Software Security

Brandon Guerra

February 18, 2015

Objective

To work with a simple web server in order to understand the security goals and design principles in section 1 of this course.

1 Security Requirements for Simple Web Server

The web server shall only allow GET, HEAD, and POST requests and immediately disconnect from any web client that sends a malformed HTTP request to the server. Not implementing this leaves the server vulnerable to data loss, data theft, and server unavailability. The cost for an attacker to send a malformed HTTP request is almost none.

The web server shall handle internal errors gracefully and not just close the web server. By not doing this, you risk the web server being forced into a state not intended and or expected.

The web server shall not provide feedback to web clients that exhibits the existence of internal errors. Outputting the existence of internal errors can alert attackers to vulnerabilities of the web server.

The web server shall log web clients' sensitive actions. By not logging users actions', information about server usage is lost and it makes it harder to determine attacks occurring on the web server.

The web server shall use TLS to encrypt information output from web clients' requests. The cost of not implementing this is data theft. Without TLS, an attacker can easily eavesdrop on the communication link of a client and the server.

A GET request shall execute 99.99 percent of the time on the web server. Not ensuring the server remain available 99.99 percent of the time can cost clients, who will seek services elsewhere when the server has no functionality.

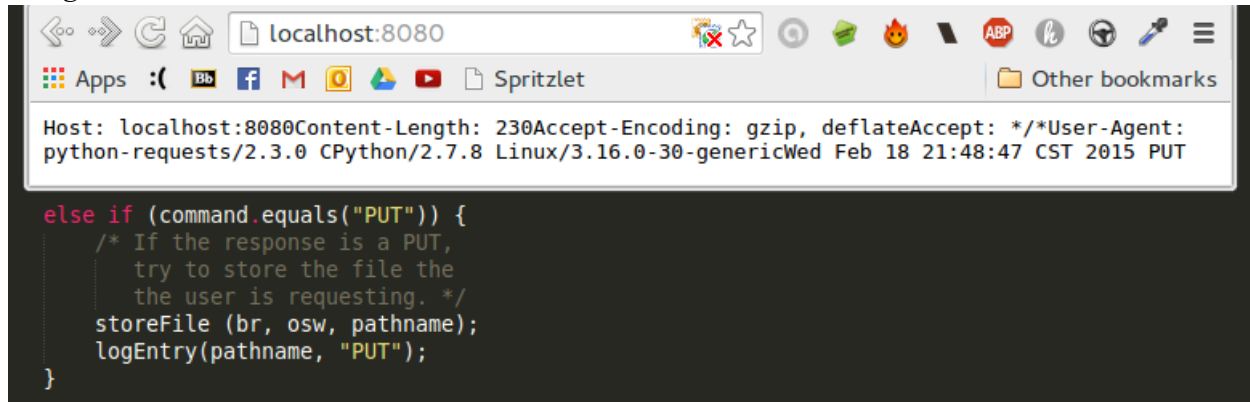
2 PUT Requests

- a. By allowing PUT requests to Simple Web Server, one needs to consider an attacker uploading a malicious file or set of files.
 - i. The server is vulnerable to a client uploading a file to replace legitimate web pages with illegitimate pages. This attack is known as defacement. A client could simply upload a file with a supplied URI that already exists on the server. The server would then overwrite the existing file on the server with the malicious file uploaded by the web client. Availability of the service that the web server provides may be broken, as entire files can be replaced with illegitimate files.
 - ii. The server is vulnerable to data theft and data loss. A web client may upload a php script that gives him root access to a database. All information in the database can then be removed or stolen. Even simpler, a web client can overwrite any file in the server as described above to remove data on the server by overwriting files. The authorization and confidentiality of the server is broken when an unauthorized user gains root access and or sensitive data is compromised.
 - iii. A web client may set up a phishing attack on the server. A file may be uploaded resembling the legitimate site but is malicious. This file may lure users to enter

sensitive information which is then stolen. This attack breaks the integrity of the web server.

- b. In order to mitigate this threat, one may just not provide PUT functionality for Simple Web Server. If PUT functionality is absolutely needed, an authentication method needs to be put in place in order to ensure the client sending a PUT request is a trusted web client by the server.
- c. Figure 2.1 shows the modified `processRequest()` code and the corresponding log data.

Figure 2.1



- d. Figure 2.2 shows the python script used to deface `index.html`. By sending a PUT request to `index.html`, the file can be overwritten.

Figure 2.2

```
import requests  
payload = open('index.html', 'r')  
path = "http://localhost:8080//home/brandon/projects/Security/Project1/src/index.html"  
r = requests.put(path, data=payload)
```

3 serveFile()

The log entry in `error_log.txt` is shown on Figure 3.1. When the maximum file size is reached, an error is returned in the browser, shown in Figure 3.2. Note that for testing purposes, I made the maximum file size 350 bytes and tested `index.html`. This is due to the fact that `/dev/random` only outputs a few bytes every couple of seconds and it was also easier to show the behavior of `serveFile()` with an HTML file. The code for serve file is shown in Figure 3.3.

- a. If an attacker tries to download `/dev/random`, the attacker will only be able to download the number of bytes `MAX_DOWNLOAD_LIMIT` permits and then stops downloading and prints an error message in the browser as shown in Figure 3.2.
- b. An alternative to allowing the client to download up to `MAX_DOWNLOAD_LIMIT` number of bytes and leaving the client with truncated files, you could instead simply not allow

any downloading if the overall file size is greater than `MAX_DOWNLOAD_LIMIT`. This is done by removing `osw.write(c);` underneath the while loop.

Figure 3.1

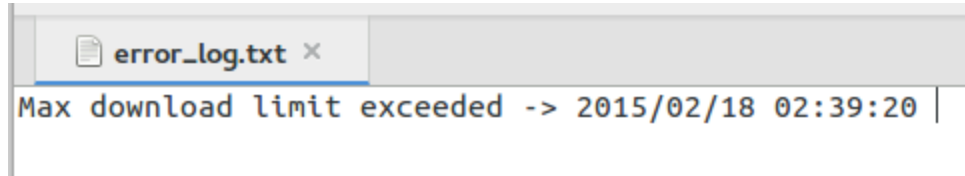


Figure 3.2

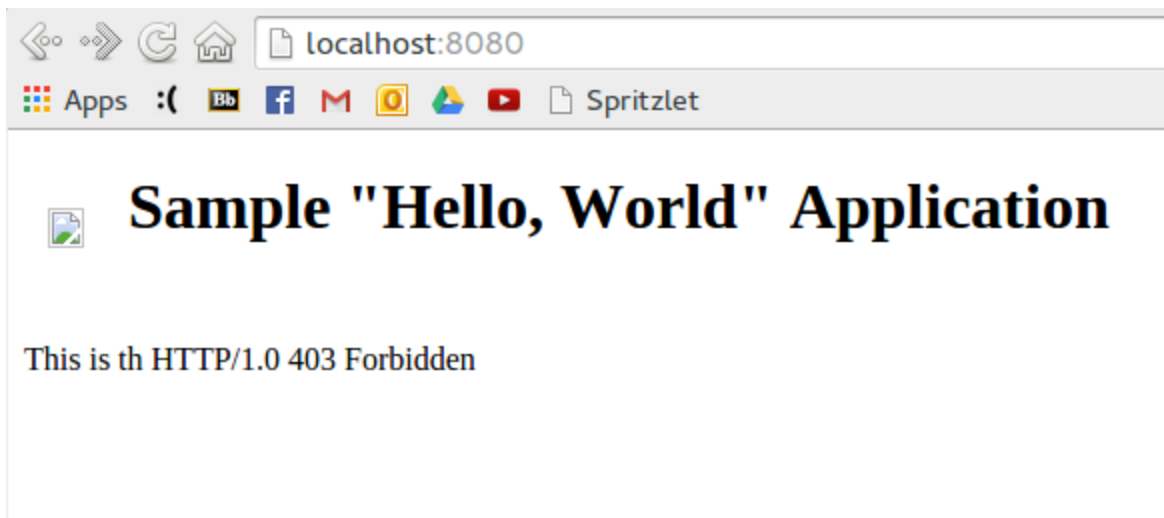


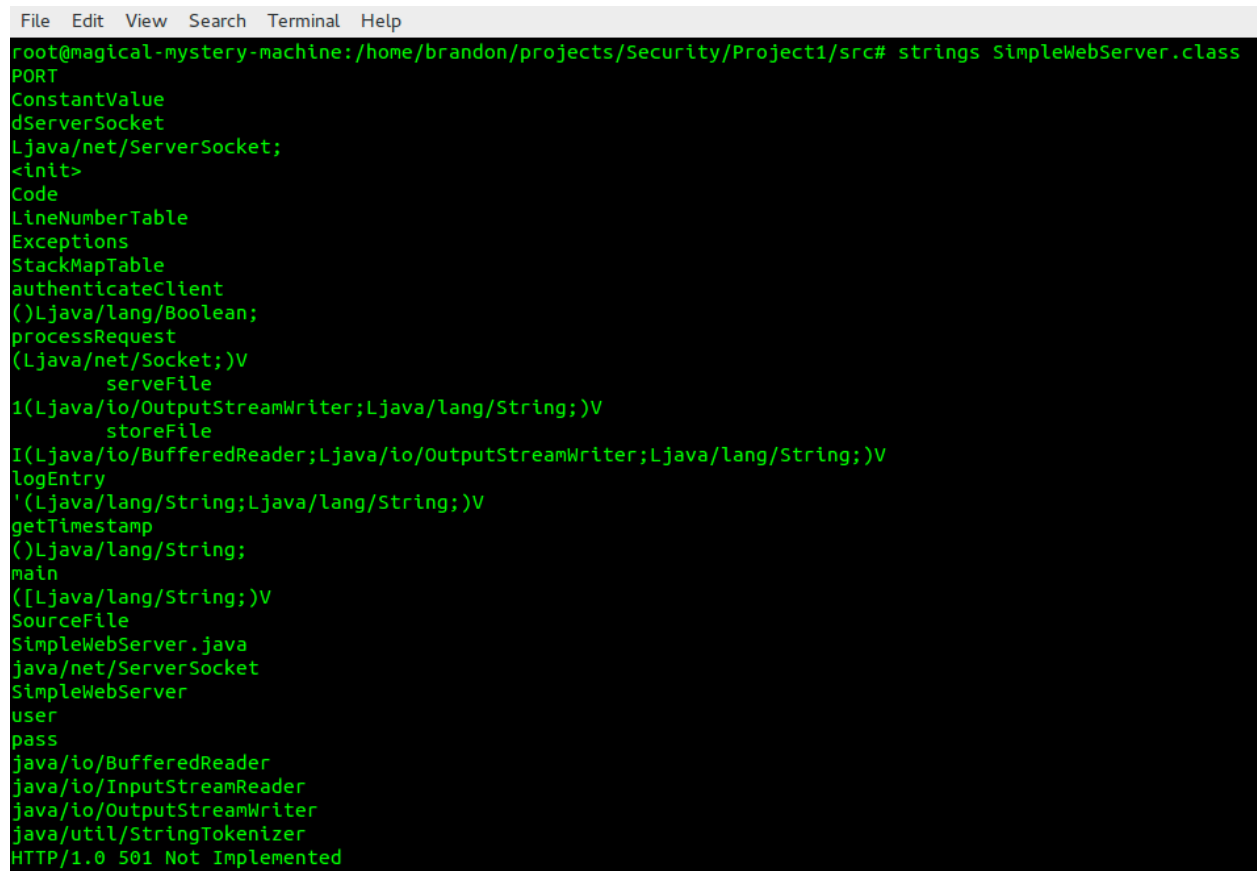
Figure 3.3

```
public void serveFile (OutputStreamWriter osw, String pathname) throws Exception {  
  
    FileReader fr = null;  
    int c = -1;  
    int sentBytes = 0;  
    int MAX_DOWNLOAD_LIMIT = 1000;  
  
    /* Remove the initial slash at the beginning  
    of the pathname in the request. */  
    if (pathname.charAt(0) == '/')  
        pathname = pathname.substring(1);  
  
    /* If there was no filename specified by the  
    client, serve the "index.html" file. */  
    if (pathname.equals(""))  
        pathname = "index.html";  
  
    /* Try to open file specified by the pathname. */  
    try {  
        fr = new FileReader (pathname);  
        c = fr.read();  
    }  
    catch (Exception e) {  
        /* If the file is not found, return the  
        appropriate HTTP response code. */  
        osw.write ("HTTP/1.0 404 Not Found\n\n");  
        return;  
    }  
  
    /* If the requested file can be successfully opened  
    and read, then return an OK response code and  
    send the contents of the file. */  
    osw.write ("HTTP/1.0 200 OK\n\n");  
    while ((c != -1) && (sentBytes < MAX_DOWNLOAD_LIMIT)) {  
        osw.write(c);  
        sentBytes++;  
        c = fr.read();  
    }  
  
    /* If the file exceeds the maximum file  
    size limit, write a log entry to  
    error_log and return a 403  
    Forbidden HTTP response code. */  
    if (sentBytes == MAX_DOWNLOAD_LIMIT) {  
        osw.write (" HTTP/1.0 403 Forbidden");  
        DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");  
        //get current date time with Calendar()  
        Calendar cal = Calendar.getInstance();  
        PrintWriter writer = new PrintWriter("error_log.txt", "UTF-8");  
        String time = dateFormat.format(cal.getTime());  
        writer.printf("Max download limit exceeded -> %s\n", time);  
        writer.close();  
    }  
  
    fr.close();  
}
```

4 HTTP Authorization

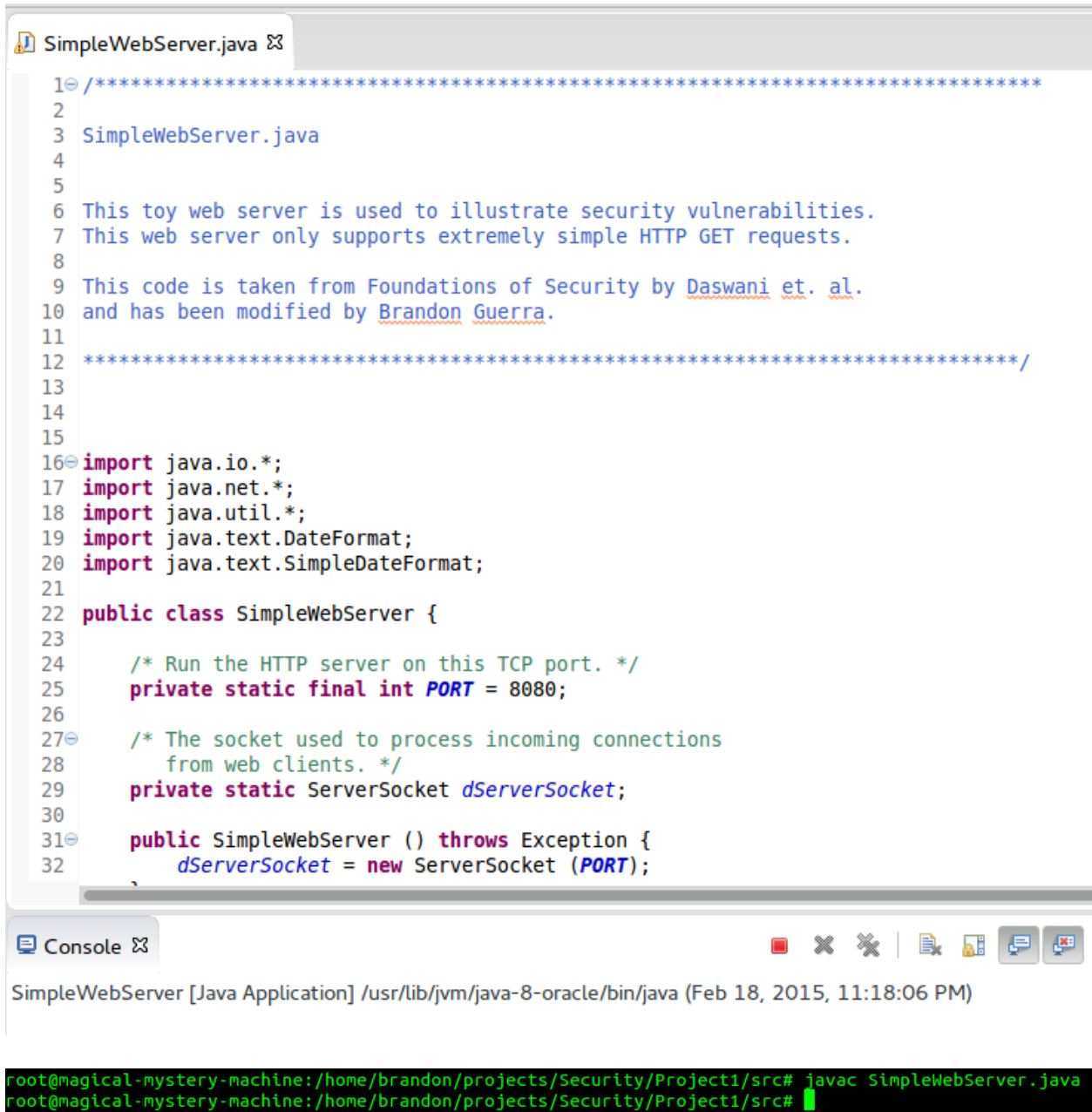
The linux utility *strings* prints the printable character sequences from object files. By running this utility on a Java class file, you are able to reveal printable character sequences. Because the username and password of the server was hardcoded, *strings* reveals both character sequences (user and pass) as shown in figure 4.1.

Figure 4.1

A terminal window with a black background and green text. The title bar shows 'File Edit View Search Terminal Help'. The command prompt is 'root@magical-mystery-machine:/home/brandon/projects/Security/Project1/src#'. The command executed is 'strings SimpleWebServer.class'. The output lists various strings found in the class file, including 'PORT', 'ConstantValue', 'dServerSocket', 'Ljava/net/ServerSocket;', '<init>', 'Code', 'LineNumberTable', 'Exceptions', 'StackMapTable', 'authenticateClient', '()Ljava/lang/Boolean;', 'processRequest', '(Ljava/net/Socket;)V', 'serveFile', 'I(Ljava/io/OutputStreamWriter;Ljava/lang/String;)V', 'storeFile', 'I(Ljava/io/BufferedReader;Ljava/io/OutputStreamWriter;Ljava/lang/String;)V', 'logEntry', '(Ljava/lang/String;Ljava/lang/String;)V', 'getTimestamp', '()Ljava/lang/String;', 'main', '([Ljava/lang/String;)V', 'SourceFile', 'SimpleWebServer.java', 'java/net/ServerSocket', 'SimpleWebServer', 'user', 'pass', 'java/io/BufferedReader', 'java/io/InputStreamReader', 'java/io/OutputStreamWriter', 'java/util/StringTokenizer', and 'HTTP/1.0 501 Not Implemented'.

```
File Edit View Search Terminal Help
root@magical-mystery-machine:/home/brandon/projects/Security/Project1/src# strings SimpleWebServer.class
PORT
ConstantValue
dServerSocket
Ljava/net/ServerSocket;
<init>
Code
LineNumberTable
Exceptions
StackMapTable
authenticateClient
()Ljava/lang/Boolean;
processRequest
(Ljava/net/Socket;)V
    serveFile
I(Ljava/io/OutputStreamWriter;Ljava/lang/String;)V
    storeFile
I(Ljava/io/BufferedReader;Ljava/io/OutputStreamWriter;Ljava/lang/String;)V
logEntry
'(Ljava/lang/String;Ljava/lang/String;)V
getTimestamp
()Ljava/lang/String;
main
([Ljava/lang/String;)V
SourceFile
SimpleWebServer.java
java/net/ServerSocket
SimpleWebServer
user
pass
java/io/BufferedReader
java/io/InputStreamReader
java/io/OutputStreamWriter
java/util/StringTokenizer
HTTP/1.0 501 Not Implemented
```

Below shows successful building and running of `SimpleWebServer.java` in Eclipse. It also shows successful compilation on the terminal.



The screenshot displays the Eclipse IDE interface. The top editor window shows the `SimpleWebServer.java` file. The code includes a multi-line comment at the top, followed by imports for `java.io.*`, `java.net.*`, `java.util.*`, `java.text.DateFormat`, and `java.text.SimpleDateFormat`. The `SimpleWebServer` class is defined with a static final `PORT` of 8080, a static `ServerSocket` named `dServerSocket`, and a constructor that throws an `Exception` and initializes `dServerSocket` with `new ServerSocket (PORT)`.

Below the editor is the `Console` window, which shows the output of the compilation command: `SimpleWebServer [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Feb 18, 2015, 11:18:06 PM)`. At the bottom, a terminal window shows the command `javac SimpleWebServer.java` being executed successfully in the directory `/home/brandon/projects/Security/Project1/src`.

```
1 1/*****
2
3 SimpleWebServer.java
4
5
6 This toy web server is used to illustrate security vulnerabilities.
7 This web server only supports extremely simple HTTP GET requests.
8
9 This code is taken from Foundations of Security by Daswani et. al.
10 and has been modified by Brandon Guerra.
11
12 *****/
13
14
15
16 import java.io.*;
17 import java.net.*;
18 import java.util.*;
19 import java.text.DateFormat;
20 import java.text.SimpleDateFormat;
21
22 public class SimpleWebServer {
23
24     /* Run the HTTP server on this TCP port. */
25     private static final int PORT = 8080;
26
27     /* The socket used to process incoming connections
28        from web clients. */
29     private static ServerSocket dServerSocket;
30
31     public SimpleWebServer () throws Exception {
32         dServerSocket = new ServerSocket (PORT);
33     }
34 }
```

```
SimpleWebServer [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Feb 18, 2015, 11:18:06 PM)

root@magical-mystery-machine:/home/brandon/projects/Security/Project1/src# javac SimpleWebServer.java
root@magical-mystery-machine:/home/brandon/projects/Security/Project1/src#
```