CSE 5/7359 – Software Security

Lab 1 – Simple Web Server: Security Goals and Principles

Name: Nicole Sood

Objective: The Foundation's book contains a java-based sample web server called SimpleWebServer. This lab requires you to work with simple web server 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.**

(a) One of the first things that must go into designing security requirements for the Simple Web Server is making sure all developers are on the same page. This is because security should be designed from the beginning. As a result, security goals should be integrated into the software requirement documents which can be distributed amongst all developers. This allows for good communication amongst the team and means that it will be less likely that people will encounter vulnerabilities. The cost of bad communication could mean there are vulnerabilities left open, or work would have to be redone costing time and money.

(b) The Simple Webserver needs to obey all local/federal regulations regarding how data is stored. For example, making sure data is encrypted. Although the extent of these regulations can change based on location (i.e. country by country) implementing them correctly means the system will be constantly protected whether the server is running or not. This will prevent things such as data leaks. The cost of not following regulations means that the web server is vulnerable to things such as a buffer overflow. In addition, without encryption you are leaving data open and vulnerable to access. Ignoring local/federal regulations could also leave a company open to things such as lawsuits, costing time and money.

(c) The Simple Web Server must specify error handling. This will allow the system to identify errors early and allow the system to do something to handle them. This will protect the system from ill-formatted requests, and if something goes wrong the connection to the client should be closed. This will protect the server while it is running and closing the connection will prevent DoS attacks.

(d) The Simple Web server must be prepared to handle internal errors. This could be done by implementing unique session ids. Should an internal error occur, a new session id would be created, and the request should be continued on this new id. This protects the uniqueness of the session and maintains the integrity of transmissions while the server is running. An internal error could be used as for entry into the system, as a result, the cost of not handling internal errors could mean the system is exposed to things such as SQL injections, buffer overflows or other forms of exploitation.

(e) The Simple webserver must implement the principle of least privilege. This means separating the client side of the server from the development side. In addition, a user should be provided with the minimum number of permissions needed to complete a task. This protects unauthorized individuals from manipulating the behaviour of the system and limits what a user has access to. This will protect the webserver while it is running. Implementing this helps to protect data, it makes a system less vulnerable to a client editing and potentially deleting data.

(f) The Simple Web Server must keep a log of actions that a user takes. In addition, it must keep a log of requests that are being made to the system. Things which could be tracked include login attempts (this could be tracked using a sum check), GET/PUT/POST and data changes. This will help to protect the server while it is running and help maintain data integrity. Implementing a logging system will help to identify when an attacker might be trying to access the server. Implementing this will help prevent DoS attacks, makes the system less vulnerable to spoofing and protects the validity of any data within the server.

(g) The Simple Web Server should be designed to include a fail-safe stance. This can be done through the inclusion of things such as a firewall or putting a size cap on things such as a file input. This will protect the webserver from malicious users or infinite file length, preventing a crash of the server. If malicious traffic is suspected, then it will deny access by default or it will only read a part of a file. This will protect the system from things such as a buffer overflow. Not doing this could leave the system open to vulnerabilities.

2. **HTTP supports a mechanism that allows users to upload files in addition to retrieving them through a PUT command.**

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.

(a) What threats would you need to consider if a Simple Webserver also has functionality that could be used to upload files?

Three main areas of attack would need to be considered by having a file upload functionality.

- An attack on the infrastructure,
- An attack on users,
- And disruption of service.

Firstly, could the *file upload overwrite existing content (**defacement**)*? For example, if they have the same file name and extension an attacker could use them to bring the website down. Another thing to consider is if, during *the file transfer process, the file becomes corrupted*. How would it affect transportation to other aspects of the web server such as interacting databases? Could it expose vulnerabilities in the software?

What is the *file containing malware* ***(infiltration)***, could it affect users trying to access the simple web server? If the file does contain malware, could they embed a link to navigate to a different site causing damage to users? (***pharming***)
We also should consider a file of extremely large of infinite length ***(denial of service)****;* how would the system behave? Could it crash the server due to high consumption?

REFERENCE: https://www.opswat.com/blog/file-upload-protection-best-practices

(b) For each of the specific threats you just listed, what type of security mechanisms might you put in place to mitigate the threats?

You could prevent a file from overwriting (defacement) existing content by *checking file extensions*. For example, not allowing .exe or scripts to be uploaded. This would make it less likely to overwrite any existing development files. To prevent the upload of malware/pharming and corrupted files you can *scan for masked files, and if possible, scan for malware or links that could be embedded into the file.* Lastly, making use of a field size check*, and setting a maximum file length size* can help to prevent extremely large files from draining recourses from the servers.

REFERENCE: https://www.opswat.com/blog/file-upload-protection-best-practices

(c) Modify the processRequest() method in SimpleWebServer to use the preceding file storage and logging code.

```
if (command.equals("GET")) {
        serveFile (osw,pathname);
        logEntry("logFile.txt", command);
    }
    else if (command.equals("PUT")) {
        serveFile (osw,pathname);
        logEntry("logFile.txt", command);
    }
```

(d) Run your web server and mount an attack that defaces the index.html homepage.

To mount an attack that defaced my index.html homepage, I made use of curl commands. As per the beginning of this question, I used a PUT command to implement this.

Simply running:

curl -d test.txt -X PUT http://10.0.2.15:8080/index.html

which will update the contents of the index.html displaying the contents of the text file.

However, an issue with using this is that mounting an attachment this way will get logged into the logFile.txt and therefore is not recommended.

Instead, I would alter the about statement to run a:

curl -d test.txt http://10.0.2.15:8080/index.html

which defaults to a POST function. This will do virtually the same job as specifying a PUT command; however, it will not get logged into the logFile.txt, seeing as is it not accounted for in the logging algorithm provided by the textbook.

Reference 1: https://www.w3schools.com/tags/ref_httpmethods.asp (understanding the difference between a PUT and POST)
Reference 2: https://everything.curl.dev/http/put (curl command)

(e) Assume that the web server is run as a root on a Linux workstation. Mount an attack against SimpleWebServer in which you take ownership of the machine it is running on. Cover your tracks so that the web log does not indicate that you mounted an attack.

How I would mount this attack is first, I would work out the permissions for the root user. This could be through social engineering to gether person's user and password, or by brute forcing into the permissions file of the root user through a series of GET commands. Once I have accessed the permissions folder, I would make use of a POST command to overwrite the contents of the file with my information so that I can access the web server. Making use of GET and POST means that our tracks are covered because a logged GET can be caused by a user simply refreshing the webserver. Hence, it would not necessarily be an alarm or a red flag. POST commands are not logged in the file, therefore you can update/overwrite files without it getting tracked.

References: https://www.ibm.com/docs/en/urbancode-deploy/7.0.3?topic=reference-running-rest-commands

3. 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

```
public void serveFile (OutputStreamWriter osw, String pathname) throws Exception {
    FileReader fr=null;
    int c=-1;
    int bytesSent = 0;
    StringBuffer sb = new StringBuffer();
      if(Files.size(Paths.get(pathname)) <= downloadsize){
        osw.write ("HTTP/1.0 200 OK\n\n");
        while (c != -1 && bytesSent <= downloadsize) {
            bytesSent++;
            sb.append((char)c);
            c = fr.read();
        }
        osw.write (sb.toString());
    }
    else{
        osw.write("HTTP/1.0 403 Forbidden\n\n");
        logEntry("errorlog.txt", "File too large" + Inet4Address.getLocalHost());
    }
```
REFERENCE: Pages 70 – 71 of the textbook (Section 3.5.4 - Impose a download limit)

(a) **What happens if an attacker tries to download /dev/random after you have made your modification?**

If an attacker tries to download a file, they can only at most get 1000 bytes of data, where x is the maximum number of bytes which you allow for a download. The user will then receive an error 403.

(b) **What might be some alternative ways in which you implement the maximum file size limit?**

An alternative way to implement this is by using java servlet technology. The @MultipartConfig Annotation allows you to declare the location of the file, how much data should temporarily be stored on disk, the maximum file size as well as the maximum request size that someone can make from the website.

Another way I might implement a maximum file size limit is by making use of the jQuery validate plugin. This would allow a you to check and validate that a file is within than a certain size before posting to a website, it would also allow the system to check and validate file extensions.

REFERENCES: https://docs.oracle.com/javaee/7/tutorial/servlets011.html
https://stackoverflow.com/questions/33096591/validate-file-extension-and-size-before-submitting-form

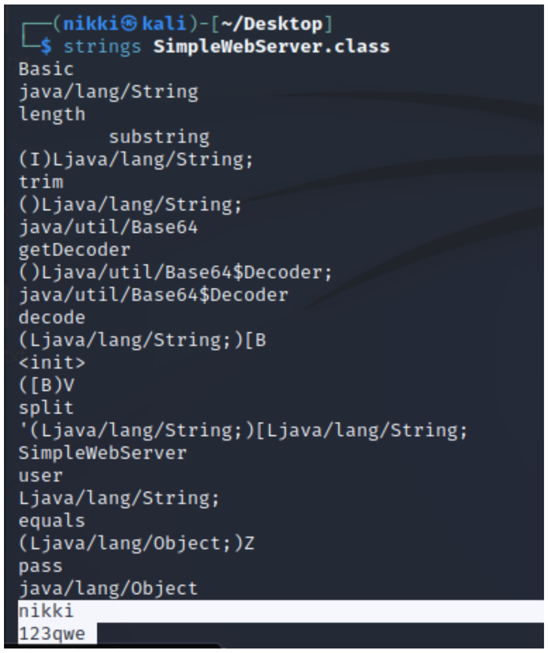
4. Complete parts *a*, *b* and *c* of problem 9 on page 79 of the book.

(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**

```
public boolean checkUser(String authHeader) {
        String encodedUP = authHeader.substring("Basic ".length());
        byte[] decodedBytes = Base64.getDecoder().decode(encodedUP);
        String decodedString = new String(decodedBytes);
        String userN = decodedString.split(":")[0];
        String passW = decodedString.split(":")[1];
        if(userN.equals(this.user) && passW.equals(this.pass)){
            return true;
        }
        else{
          return false;
        }
    }
```

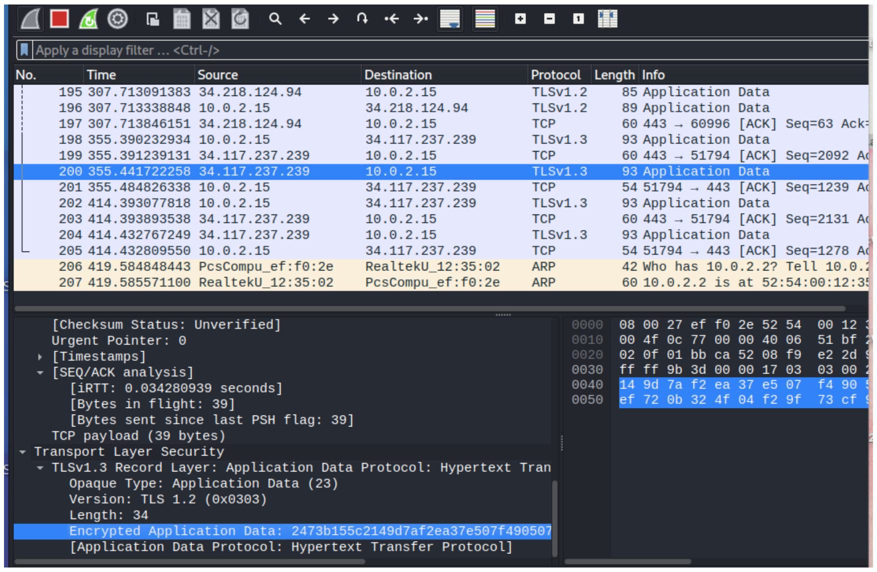
REFERENCES: https://mixedanalytics.com/tools/basic-authentication-generator/
REFERENCES: https://gist.github.com/eddiezane/5a1fd04383a497a63157

(b) Pretend 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.



```
┌──(nikki㉿kali)-[~/Desktop]
└─$ strings SimpleWebServer.class
Basic
java/lang/String
length
        substring
(I)Ljava/lang/String;
trim
()Ljava/lang/String;
java/util/Base64
getDecoder
()Ljava/util/Base64$Decoder;
java/util/Base64$Decoder
decode
(Ljava/lang/String;)[B
<init>
([B)V
split
'(Ljava/lang/String;)[Ljava/lang/String;
SimpleWebServer
user
Ljava/lang/String;
equals
(Ljava/lang/Object;)Z
pass
java/lang/Object
nikki
123qwe
```

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



```
Apply a display filter ... <Ctrl-/>
No.     Time          Source             Destination        Protocol  Length Info
    195 307.713091383 34.218.124.94      10.0.2.15          TLSv1.2     85 Application Data
    196 307.713338848 10.0.2.15          34.218.124.94      TLSv1.2     89 Application Data
    197 307.713846151 34.218.124.94      10.0.2.15          TCP         60 443 → 60996 [ACK] Seq=63 Ack=
    198 355.390232934 10.0.2.15          34.117.237.239     TLSv1.3     93 Application Data
    199 355.391239131 34.117.237.239     10.0.2.15          TCP         60 443 → 51794 [ACK] Seq=2092 Ac
    200 355.441722258 34.117.237.239     10.0.2.15          TLSv1.3     93 Application Data
    201 355.484826338 10.0.2.15          34.117.237.239     TCP         54 51794 → 443 [ACK] Seq=1239 Ac
    202 414.393077818 10.0.2.15          34.117.237.239     TLSv1.3     93 Application Data
    203 414.393893538 34.117.237.239     10.0.2.15          TCP         60 443 → 51794 [ACK] Seq=2131 Ac
    204 414.432767249 34.117.237.239     10.0.2.15          TLSv1.3     93 Application Data
    205 414.432809550 10.0.2.15          34.117.237.239     TCP         54 51794 → 443 [ACK] Seq=1278 Ac
    206 419.584848443 PcsCompu_ef:f0:2e  RealtekU_12:35:02  ARP         42 Who has 10.0.2.2? Tell 10.0.
    207 419.585571100 RealtekU_12:35:02  PcsCompu_ef:f0:2e  ARP         60 10.0.2.2 is at 52:54:00:12:3

    [Checksum Status: Unverified]                                  0000  08 00 27 ef f0 2e 52 54  00 12
    Urgent Pointer: 0                                              0010  00 4f 0c 77 00 00 40 06  51 bf
  ▸ [Timestamps]                                                   0020  02 0f 01 bb ca 52 08 f9  e2 2d
  ▾ [SEQ/ACK analysis]                                             0030  ff ff 9b 3d 00 00 17 03  03 00
      [iRTT: 0.034280939 seconds]                                  0040  14 9d 7a f2 ea 37 e5 07  f4 90
      [Bytes in flight: 39]                                        0050  ef 72 0b 32 4f 04 f2 9f  73 cf
      [Bytes sent since last PSH flag: 39]
    TCP payload (39 bytes)
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Application Data Protocol: Hypertext Tran
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 34
      Encrypted Application Data: 2473b155c2149d7af2ea37e507f490507
      [Application Data Protocol: Hypertext Transfer Protocol]
```

Code:

To view the most recent code for this project, you can view my github repo for this class:

https://github.com/nsood1/SoftwareSecurity