Security Lab – Simple Web Server

VMware

- You will work with the **Ubuntu image**, which you should start in networking-mode **Nat**.

1 Introduction

In this lab, you get a very simple web server that is implemented in Java as a basis. The web server supports the HTTP methods GET and PUT to read and write resources. The web server contains several serious vulnerabilities. Your task is to find and understand the vulnerabilities and to correct the code such that the vulnerabilities are no longer present.

The goal of this lab is to show you how much can go wrong even in an apparently simple program and how easily serious vulnerabilities are introduced during programming. Often, this does not concern typical security functions such as access control or data encryption. Instead – as it is also the case in this lab – it is frequently associated with a general lack of robust programming, which means the developer simply does not take everything into account that may go wrong during runtime. In particular developers that focus primarily on functional aspects (and maybe some functional security aspects) and that lack a profound understanding of security tend towards making such mistakes. Such developers also often assume the user (or attacker) uses a standard client software (here a web browser) that follows the protocols correctly – but in reality, attackers use specialized tools to arbitrarily interact with a server application.

2 Basis for this Lab

On the Ubuntu image, there already exists an Eclipse project. As it is possible to render a system non-usable by carrying out the tests in this lab (especially attack 2 in section 5), we strongly recommend that you use only the image to work on this lab. In the following, you find some information about Eclipse and the project.

- To start Eclipse open a terminal, change to directory */home/user/eclipse* and start Eclipse by entering *./eclipse*.
- The project directory is */home/user/workspace/SimpleWebServer*. Two directories *src* and *bin* are used for source and byte code.
- Start the programs in the directory *bin* on the command line in a terminal as standard user (user) unless something else is specified in the tasks.
- There are two programs in package *ch.zhaw.securitylab*:
  - *SimpleWebServer.java*: the web server
  - *SimpleWebServerTester.java*: The test program to perform the attacks. You can analyze the code but must not change it. The program uses three command line parameters: The host name or the IP address of the web server, the port used by the server, and a number (1-8) of the test you want to carry out (see section 5). Entering the number 0 runs all tests.
- The project directory also contains a directory *data*. Below this directory are the resources of the web server. They are as follows:
  - *index.html*: the index file with content „Hello“
  - *test*: test data to check the correct functioning, don’t change its content
  - *endless_file*: a link to */dev/urandom* to simulate a very large file
  - *upload*: files can be uploaded (with PUT) to this directory

3 HTTP Protocol

You don't have to be an HTTP expert to successfully solve this lab[1]. The following explains the most important basics for this lab:

- A GET requests fetches a resource from the server. In its most simple form (no additional headers, as used by the test program) it looks as follows:

  ```
  GET index.html HTTP/1.0
  ```

  The request is terminated with an empty line.

- The response of the server consists of a status line followed by an empty line. If data are delivered (e.g. the content of index.html), the data follow after the empty line:

  ```
  HTTP/1.0 200 OK

  Hello
  ```

- The PUT method allows storing resources on the server. On public web servers, this is of course typically not supported. The data to store are included in the request, separated from the request line with an empty line. For instance, to store the data „test data" in a resource „testfile" (in directory *data/upload*), the following request must be sent:

  ```
  PUT testfile HTTP/1.0

  test data
  ```

4 SimpleWebServer.java

Before you start solving the tasks you should study the source code of the web server in order to get a good understanding about its functionality. Basically, the program works as follows:

- The main method creates a SimpleWebServer object and starts the actual server (run method).
- In the run method, a while loop is used to accept connection requests, which are then passed to the processRequest method.
- processRequest analyzes whether it's a GET or PUT request and calls the appropriate method: serveFile or storeFile.

[1] More details about HTTP can be found in the RFC: http://www.w3.org/Protocols/rfc2616/rfc2616.txt

5 Tasks

Solve the following tasks in the specified order. You should always complete a task before starting with the next. With each task, focus on solving the described vulnerability and verify whether the vulnerability has indeed been removed by your corrective measures.

The test program tries to „find out“ whether an attack can be carried out or whether your corrective measures prevent the attack – the output of the test program provides you with the necessary information. If the test program generates an exception, then this usually means that you haven't fixed the web server correctly (according to the specifications in the individual tasks).

5.1 Test 1: GET and PUT functionality

This task only serves to test whether the web server and the test program work correctly. You can always repeat this test to check whether the web server you modified still works.

Start the web server in a terminal of user *user*:

```
java ch.zhaw.securitylab.SimpleWebServer
```

Start the test in another terminal:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 1
```

The GET test accesses the file *test* and checks its content (so you should never change its content). The PUT test creates a file *test* that contains the current timestamp in the upload directory. By inspecting the output of the test program you can check whether the test was successful.

5.2 Attack 2: Compromising the root account

First, create a copy of the shadow file. This requires root permissions and can be done – assuming you are working in a terminal as *user* – using the sudo command, followed by providing the password of *user* (which is *user*):

```
sudo cp /etc/shadow /etc/shadow.org
```

In a terminal, start the web server with root permissions, again using sudo:

```
sudo java ch.zhaw.securitylab.SimpleWebServer
```

Carry out the attack by entering the following in another terminal:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 2
```

Inspecting the output of the test program should inform you whether the attack was successful.

Now, try to get root access, e.g. with a ssh login from the physical host. Use the password *test*. The login attempt should be successful, which means the root account was indeed compromised by setting a new password for root (the original password was *root*).

Compare the original and new shadow file, e.g. by using diff::

```
sudo diff /etc/shadow /etc/shadow.org
```

What has changed? What hasn't changed? Why are the passwords not directly visible?

root:6 and the suffix ':15156:0:99999:7:::' did not change but the rest did.
The passwords are not visible because they're encrypted.

The fact that this attack was successful has two main reasons – which ones? To answer this, you'll have to study the web server code. It may also be helpful using some debugging messages (e.g. with System.out.println()) or using the debugger to inspect the request received by the server (this will also be helpful in all subsequent tasks).

```
if (pathname.charAt(0)=='/') {
pathname=pathname.substring(1);
}
pathname = WEBROOT + "/" + pathname;
pathname = WEBROOT + "/" + UPLOAD_DIR + "/" + pathname;
Previously mentioned lines of code do not prevent an access outside the WEBROOT. There is no check
that the resulting file is withing the boundaires.
```

Here, an important fundamental security principal was violated. What could this be?

Missing check that the value is within the boundaires.

Replace the manipulated shadow file again with the original one:

```
sudo mv /etc/shadow.org /etc/shadow
```

Stop the web server and restart it, but as *user*:

```
java ch.zhaw.securitylab.SimpleWebServer
```

Carry out the attack again. What happens? Explain the behavior.

The attack was unsuccessful since the webserver didn't have any access to the directory.

In the following, always run the web server as *user*.

5.3 Attack 3: DoS attack with empty request

Start the server and run attack 3:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 3
```

The web server should crash. As the server is no longer available for legitimate users, we identify this as a DoS (Denial of Service) attack.

Analyze why the server crashed by studying the code. The generated exception provides you with hints where in the code you should look. In the following box, explain why the web server crashed.

The server expects a request with some data.

Modify the code such that the attack is no longer possible. Modify it such that the server replies with HTTP status code „400 Bad Request“ whenever it receives an invalid request. Note that the source code contains constants for several HTTP responses including status codes, which you should use (STATUS_400 etc.).

5.4 Attack 4: DoS attack with malformed request

To verify how general your fix in the previous task is, this attack uses another malformed request. In detail, an HTTP method (GET) but no resource is sent. Carry out the attack as follows:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 4
```

Ideally, the server shouldn’t crash and should reply with HTTP status code „400 Bad Request“. Is this the case with your code? Write down whether the web server reacts correctly or not and explain why it worked correctly in your case or not.

It checks if the request is not empty and if not it checks the array length. When both command and pathname are not empty then the request will be processed.

If it didn’t work, adapt the code such the web server reacts correctly to any malformed request.

5.5 Attack 5: DoS attack with long request

Start the server and carry out attack 5:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 5
```

The server should crash. Analyze the exception and the source code of the web server and explain why the server crashed.

The server is enlarging the string without stopping.

The client sends a request with a length of up to 1 GB. The attack could be prevented by assigning the web server more memory (in Java, this is possible with the option -Xmx, e.g. -Xmx2G). Why is this a poor solution for the identified problem?

Chaning the memory parameters won't help since the attacker can just send a bigger request.

Adapt the source code such that successfully carrying out the attack is really prevented. The server should behave as follows;

- We define an upper limit for the length of the first line of the request: 8'192 Bytes[2]. You can use the already defined constant MAX_REQUEST_LENGTH.
- Make sure that the server does not read additional bytes of the request when this limit is reached.
- If the client sends more than 8'192 Bytes, additional bytes are ignored but the request should nevertheless be processed. In the case of a GET request, the server will most likely reply with HTTP code „404 Not Found" as the specified resource most likely won't exist.

Verify whether the attack is prevented and that the web server behaves correctly.

5.6 Attack 6: DoS attack by requesting an overly large resource

Start the server and carry out attack 6:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 6
```

The web server should crash (it takes a few seconds). Analyze the exception and the source code of the web server and explain why the server crashed.

Adapt the source code to prevent the attack. The web server should behave as follows:

- The web server should return at most 10'000'000 bytes of a requested resource. You can use the predefined constant MAX_DOWNLOAD_LENGTH.
- Make sure the server does not read additional bytes form the resource when this limit has been reached.
- Superfluous bytes of the resource should be ignored by the server, but the first 10'000'000 bytes should be returned to the client.
- Verify whether the attack is prevented and that the web server behaves correctly.

5.7 Attack 7: Directory traversal attack

Start the server and carry out attack 7:

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 7
```

The tester should read the file /etc/passwd from the server.

[2] This length is also used by some "real" web servers as the limit for requests.

Analyze the source code of the web server to identify the problem and describe it:

Prevent this attack by adapting the code[3]. The web server should behave as follows:

- The client is only allowed to access resources below directory *data*.
- If the client requests a non-legitimate resource, the server should reply with HTTP status code „404 Not Found“.

Some tips how this can be solved:

- Write a method (e.g. `checkPath(String pathname)`) that checks whether a resource requested by the client (pathname) is below *data*.
- `System.getProperty("user.dir")` returns the path (as a String) of the current working directory (the directory in which the server was started). The directory *data* is located „above“ this directory (*../data*).
- With `File file = new File(pathname)` you can create a File object that represents an arbitrary path.
- The class `File` contains the method `getCanonicalPath()`, which returns the absolute and normalized path (absolute means it starts at / and normalized means that any .. in the path are correctly resolved) of a File object.

Verify whether the attack is prevented and that the web server works correctly. Note that a side effect of the solution is that attack 6 behaves differently. The reason is that that the resource *endless_file* is a link to */dev/urandom*, which means the canonical path of *endless_file* (which is */dev/urandom*) is not located below *data*.

5.8 Attack 8: Website defacement

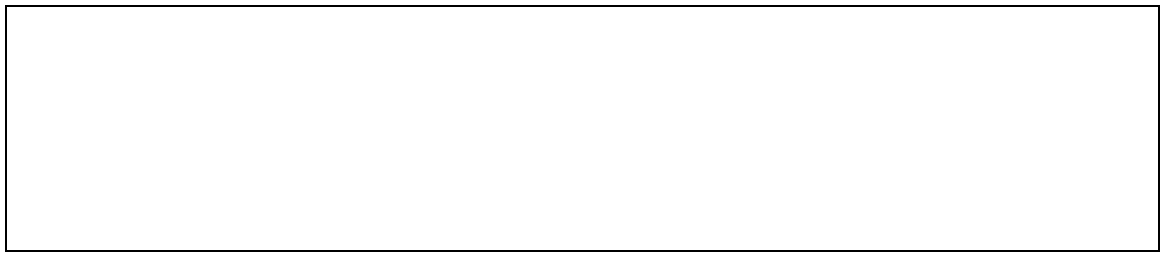
As a final attack we carry out a defacement of the website. Start the server and carry out attack 8

```
java ch.zhaw.securitylab.SimpleWebServerTester localhost 8080 8
```

The tester should have replaced *index.html* on the server with a new version (that contains the current timestamp). You can easily verify this by using the web browser.

Analyze once more the source code of the web server to identify the fundamental problem and describe it:

[3] There also exist other possibilities to prevent access to arbitrary directories, e.g. by running the web server in its „jail“ (with *nix, this is possible using the command chroot), where it can only access a subtree of the entire file system.



Prevent this attack by adapting the code. The rules are as follows:

- The client can write resources only below *data/upload*. If the resource already exists, it is over-written.
- If the client attempts to write a resource that is not below *data/upload*, the server should reply with HTTP status code „403 Forbidden“.
- Using (and maybe extending) the method checkPath from attack 7 is probably a good idea.

Verify whether the attack is prevented and that the web server behaves correctly.

Lab Points

For **2 Lab Points** you must show the filled-in sheet (one per group) to the instructor. In addition, you must demonstrate that test 1 still works, that attacks 2-8 are prevented, and that the web server reacts as specified in the tasks. Furthermore, you have to describe the changes you did in the source code to fix the vulnerabilities if requested by the instructor.