Security Lab – Simple Web Server

VMware

- This lab can be solved with the **Ubuntu image**, which you should start in networking-mode **Nat**. The remainder of this document assumes you are working with the Ubuntu image.
- The lab could basically also be solved on any Unix/Linux system with an installed IDE and Java. However, it is possible to render a system non-usable by carrying out the tests in this lab (especially attack 2 in section 5). It's therefore strongly recommended you solve this lab using the Ubuntu image.

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 understand 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

- Download *SimpleWebServer.zip* from OLAT.
- Move the file to an appropriate location (e.g. in a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. The resulting directory *SimpleWebServer* contains a Java project based on Maven. This should be importable in any modern IDE. The remainder assumes you are using *NetBeans*, which is installed on the Ubuntu image.
- Start *NetBeans* and open the project.
- To build the project, right-click *SimpleWebProject* in the *Projects* tab and select *Clean and Build*.
    - Whenever you do some changes, the project is rebuilt automatically, so it's usually not necessary to use *Clean and Build* again. However, if «something doesn't work as it should», do again a *Clean and Build*.
    - The executable code is placed in directory *SimpleWebServer/target/classes*.
- The project contains two programs:
    - *SimpleWebServer.java*: The web server.
    - *SimpleWebServerTester.java*: The test program to perform the attacks. You can look at the code but don't 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.

- Both are located in package *ch.zhaw.securitylab.simplewebserver*, which means the class files are placed in directory *ch/zhaw/securitylab/simplewebserver* below *SimpleWebServer/target/classes.*

- The server uses some additional directories and files. They are created automatically when the server is started. They are located in a directory *data*, which can be found below *SimpleWebServer/target/classes*:
  - *index.html*: the index file with content *Hello*
  - *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 first creates the directories and files used by the web server, then 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.

In general, run the programs from the command line (not from within the IDE). Do this in a terminal as user *user* unless something else is specified in the tasks.

To start the web server, change to directory *SimpleWebServer/target/classes* and enter the following:

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

Start the test in another terminal:

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

As you can see from the command line options, the server runs on the local host on port 8080.

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.SimpleWebServer
```

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

```
java ch.zhaw.securitylab.simplewebserver.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?

the hash value of root's password has changed.

Why are the passwords not directly visible?

The passwords are hashed using a cryptographic hash function.

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).

1. We are not checking the submitted pathname - the attacker can break out of web root directory.
2. The web server is not running in its jail; the file system of the underlying operating system is accessible.

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

1. run the server with least right (not root)
2. Input validation - path traversal

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.SimpleWebServer
```

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

HTTP 404 error: file not found

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.simplewebserver.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.

unchecked access to array index 1 is attempted in the process request method of the web server. Since the request is empty or malformed, the process request method cannot extract the token containing the request text.

The result is a runtime exception (ArrayIndexOutOfBoundsException).

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.simplewebserver.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.

Works just fine. Code checks for tokens array length > 1

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.simplewebserver.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.

Attack 5: Execute long request test... done (connection broken after 76308470 Bytes sent)
Attack succeeded (DoS)
Server crashed
String request = br.readLine() fails to read the very long request. The internal data structure (array) cannot be expanded to store the full request text.
JVM fails to allocate heap space in virtual memory.

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?

There is no need to accept such large requests in the real word. Instead a maximum request length in bytes should be defined. Requests larger than xx bytes should then just be dropped (or return a '400 bad request' message).

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.simplewebserver.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.

The internal data structure (array) couldn't allocate enough space on the heap to store the data of the requested file.

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.

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

5.7 Attack 7: Directory traversal attack

Start the server and carry out attack 7:

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

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

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

Attacker can break out of web root

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* (which is below *SimpleWeb-Server/target/classes*).
- If the client requests a non-legitimate resource, the server should reply with HTTP status code „404 Not Found".

One way to solve this is as follows:

- Inspecting *serveFile()*, you can see that *data/* is prepended to the pathname requested by the client. Then the file is opened and served. As the pathname is not checked at all, the attacker can break out of the *data* directory to access basically any file on the system (with the rights of the running web server process) – which is exactly what happened in the attack above.
- Therefore, you have to check that the requested pathname (including the prepended *data/*) is indeed below the *data* directory and only serve the file if this is the case. To do so, write a method such as *checkPath(String pathname)* that receives the pathname as a parameter and that performs this check. To implement the method, use these additional hints:
  - First, you should get the absolute path of directory *data*. «Absolute path» means that it starts at the root directory, i.e. /. To do so, use *System.getProperty("user.dir")*, which returns the absolute path (as a String) of the current working directory of the program (the directory in which the server was started, i.e. *SimpleWebServer/target/classes*). As the directory *data* is located just below this directory, you can now easily build the absolute path of the directory *data*.
  - Second, get the absolute and normalized path of the resource requested by the client. Normalized means that any .. in the path are correctly resolved. To do this, use *File file = new File(pathname)* using the pathname (including prepended *data/*) requested by the client, which creates a File object that represents the specified pathname (relative to the current working directory). Then, use the method *getCanonicalPath()* of the file object, which returns the absolute and normalized path of the pathname, i.e. of the resource requested by the client.
  - Now you can easily check whether the second path String is below the first path String.

Verify whether the attack is prevented by your solution and that the web server works correctly.

[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.

Note that a side effect of the solution is that attack 6 now most likely behaves differently: It now results in a response with HTTP code „404 Not Found". The reason is that that the resource requested in attack 6 – *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*, which means the file is no longer served.

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.simplewebserver.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:

The web server responds to PUT requests. Furthermore the pathname isn't checked.

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 overwritten.
- 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.