# Security Lab – Secure File Storage Service

## VMware

This lab can be done with the **Ubuntu image**. The remainder of this document assumes you are working with this image.

The lab could basically also be done on any Linux system with an installed IDE, Java and nmap. However, it is possible to render a system non-usable by carrying out the attacks in this lab (especially attack 1 in section 6). It's therefore strongly recommended you solve this lab using the Ubuntu image.

## 1 Introduction

In this lab, you get the server-side component of a simple secure file storage service that is implemented in Java as a basis. Basically, the service allows different users to register, to login, to store and fetch files, and to execute system commands. The server component 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 «sharpen your awareness» in the sense that you should understand how much can go wrong with respect to security even in an apparently simple program and how easily serious vulnerabilities can find their way into a program. Usually the corresponding mistakes are associated with a single or at most a few lines of code and typically happen because the developer lacks a profound understanding of software security and, as a result of this, does not take into account the wide range of attacks that may happen during runtime. Such developers also often assume the user (or attacker) uses a standard client software (here: the client component of the secure file storage service, but you won't use this in this lab) that follows the protocol correctly – but in reality, attackers use specialized tools to arbitrarily interact with a server application.

## 2 Basis for this Lab

- Download *SecureFileStorage.zip* from OLAT.
- Move the file to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. The resulting directory *SecureFileStorage* 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 *SecureFileStorage* in the *Projects* tab and select *Build*.
    - Whenever you do some changes, the project is rebuilt automatically, so it's usually not necessary to use *Build* again. However, if «something doesn't work as it should», do again a *Build*.
    - The executable code is placed in directory *SecureFileStorage/target/classes*.
- The project contains two programs:
    - *SecureFileStorageServer.java*: The secure file storage server component.
    - *SecureFileStorageServerTester.java*: A test program to perform different attacks against the server. You can look at the code, but please don't change it! The program uses two command line parameters: The host name or the IP address of the server and a number (0-14) of the attack you want to execute (see sections 4 and 6). Entering no number at all runs all attacks.

  - Both programs are located in package *ch.zhaw.securitylab.securefilestorage*, which means the class files are placed in directory *ch/zhaw/securitylab/securefilestorage* below *SecureFileStorage/target/classes.*
  - In addition, a file *Common.java* is included. This includes constants and methods that are used by both programs.
- The server uses some additional directories and files. They are located in a directory *data*, which can be found below *SecureFileStorage/target/classes*:
  - Files *users.org* and *users* in directory *data/system*: File *users.org* contains usernames and passwords of three users of the secure file storage service. The usernames are *user1*, *user2* and *user3* with passwords *password1*, *password2* and *password3*. Each row in the file contains one username/password pair and they are stored in the form *user1:password1*. When the server is started, file *users.org* is copied to file *users* and file *users* will be used during the actual program execution. This guarantees that you start with the original and unmodified version of the *users* file whenever the program is restarted. Of course, these passwords are totally insecure, but we consider them to be acceptable for test purposes.
  - Directory *data/files* contains the stored files of the users. The files of a specific user are stored in a subdirectory that corresponds to the username, e.g., in directory *data/files/user1* in the case of *user1*. As there are three users, there are three directories *data/files/user1*, *data/files/user2*, and *data/files/user3*.
  - In each user-specific directory, there's one file belonging to the corresponding user: *file1.txt* in *data/files/user1*, *file2.txt* in *data/files/user2*, and *file3.txt* in *data/files/user3*.

## 3 Functionality and Protocol

Communication between client and server uses a simple, ASCII-based protocol. Messages from client to server are called *requests*, messages from server to client are called *responses*. Overall, five different request types are supported: *REGISTER*, *LOGIN*, *GET*, *PUT* and *SYSTEM*. In the following, the request types and how they are used in the communication protocol are described.

### 3.1 REGISTER

*REGISTER* is used to register a new user. To register a user with username *user4* and password *password4*, the following request is sent to the server:

```
REGISTER user4:password4
-----DONE-----
```

When the server receives this and if *user4* is not yet existing, it creates a new line *user4:password4* in file *users* and responds with:

```
OK
-----DONE-----
```

Otherwise, the following response is sent back:

```
NOK
-----DONE-----
```

This also shows some general properties of the protocol used by the secure file storage service:

- Every request always starts with one of the five request types (*REGISTER*, *LOGIN* etc.), followed by one or more arguments (here one argument: *user4:password4*).
- Every response always starts with a response status *OK* or *NOK*, depending on whether the request could successfully be handled or not.

- Every request and every response is always terminated with a marker -----*DONE*-----.
- Every part of the request or response uses a separate line. Lines are always terminated with a newline character (\n). For instance, the *REGISTER* request to the server contains two lines – *REGISTER user4:password4* and -----*DONE*----- – and both lines are terminated with a \n character.
- As an additional feature (although not shown in the massages above), the arguments in the first line of a request may be URL encoded (this is also supported by some other ASCII-based protocols, e.g., HTTP). URL encoding uses a *%* character followed by the hex value of the ASCII code of the character. So for instance, the URL encoded value of *A* is *%41*. As a result of supporting URL encoding, the arguments are URL decoded by the server before they are processed further.

### 3.2 LOGIN

*LOGIN* is used to login. To login as user1, the following request is sent to the server:

```
LOGIN user1:password1
-----DONE-----
```

When the server receives this request, it checks the credentials, i.e., whether file *users* contains an entry *user1:password1*. It this is the case, the server first creates a random, 256-bit session ID (encoded as a 64 characters long hex string) and stores this session ID and associates it with *user1*. If it has issued a session ID to this user before, the previous one is overwritten with the new one. The server responds with the following message:

```
OK
-----CONTENT-----
user1:147FB6E880724B340A79F21D531E635FD48018BE3AA458BE340C9D865720AD80
-----DONE-----
```

Otherwise (if the credentials from the user are not valid), a *NOK* response is sent back:

```
NOK
-----DONE-----
```

This shows a further general property of the protocol used by the secure file storage service: Whenever a request or response includes content, a marker line -----*CONTENT*----- is used to separate the content from the first part of the request or the response. Here, the content is the session ID (with prepended *user1:*) that is included in the response.

The session ID is used to associate further requests by the user with his identity, similar as done with session IDs in web applications or with authentication tokens in the case of web service APIs.

### 3.3 GET

*GET* is used to read a file. It requires a valid session ID, i.e., a previous login. Of course, users should be able to only access their own files, e.g., *user1* should only be able to access files stored in directory *data/files/user1*. To get *file1.txt* as *user1*, the following request is sent to the server:

```
GET file1.txt
user1:147FB6E880724B340A79F21D531E635FD48018BE3AA458BE340C9D865720AD80
-----DONE-----
```

When the server receives this and if a valid session ID is included, the requested file is accessed (in this case *data/user1/file1.txt*, and its content is sent back to the client as follows:

```
OK
-----CONTENT-----
```

```
This is the content of file1.txt belonging to user1.
It even contains a second line.
And a third as well, unbelievable!
-----DONE-----
```

The file content is included line by line between -----*CONTENT*----- and -----*DONE*-----. Note that for simplicity, we assume the files are always text files as this makes it much simpler to print the file content in a terminal (see later). Security-wise, supporting binary files wouldn't change anything. If a problem happens while processing the request, e.g., if the received session ID is not valid or if the file does not exist, a *NOK* response is sent back:

```
NOK
-----DONE-----
```

### 3.4 PUT

*PUT* is used to store a file on the server. It requires a valid session ID, i.e., a previous login. Of course, users should be able to store files only in their own directory, e.g., *user1* should only be able to store files in directory *data/files/user1*. To create a file *testfile.txt* as *user1*, and with the specified content, the following request is sent to the server:

```
PUT testfile.txt
user1:147FB6E880724B340A79F21D531E635FD48018BE3AA458BE340C9D865720AD80
-----CONTENT-----
Test data: Terrific test file content!
Spread across two lines.
-----DONE-----
```

When the server receives this and if a valid session ID is included, the file is created (in this case *data/user1/testfile.txt* and written with the specified content. If the file already exists, it is overwritten. If the file can be created and (over-) written, the following response is sent back to the client:

```
OK
-----DONE-----
```

If a problem happens while processing the request, e.g., if the received session ID is not valid, a *NOK* response is sent back:

```
NOK
-----DONE-----
```

For simplicity, no subdirectories are supported, so – in the case of *user1* – files can only be written to (and read) from *files/user1/*, but not to/from, e.g., *files/user1/subdir/*.

### 3.5 SYSTEM

*SYSTEM* is used to issue a command in the underlying operating system of the server. The only command that is currently supported is getting the disk space that is consumed by the files of the user. It requires a valid session ID, i.e., a previous login. To get the disk space consumed by the files of *user1*, the following request is sent to the server:

```
SYSTEM USAGE *
user1:147FB6E880724B340A79F21D531E635FD48018BE3AA458BE340C9D865720AD80
-----DONE-----
```

When the server receives this and if a valid session ID is included, the server accesses the *du* command in the underlying operating system (using *du -h data/files/user1/**) and includes the output of the *du* command in the response as follows:

```
OK
-----CONTENT-----
4.0K      data/files/user1/file1.txt
4.0K      data/files/user1/testfile.txt
-----DONE-----
```

Note that *du* returns the disk usage (i.e., the size of the used disk blocks) and not the effective size of the files, which is why *4.0K* is returned for both files of *user1*. If a problem happens while processing the request, e.g., if the received session ID is not valid, a NOK response is sent back:

```
NOK
-----DONE-----
```

Besides *USAGE **, the request can also use *USAGE .* (with a dot instead of an asterisk), which returns the total disk usage of all the files of the user.

## 4 Run the Server and Test whether it Works Correctly

First, run the server and test whether it works correctly. To run the server, open a terminal (it's best to always run the programs in a terminal), change to directory *SecureFileStorage/target/classes* and enter:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageServer
```

Per default, the server uses TCP port 4567. If you want to, you can change this in *Common,java*, but there's usually no reason to do so.

To test whether the server works correctly, use a second terminal, change again to directory *SecureFileStorage/target/classes* and enter the command below to run the test program. The command line parameters identify the hostname of the server (*localhost*) and the number of the test or attack that should be executed. Here, we are using test number 0. Numbers 1 and higher are «reserved» for the actual attacks:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 0
```

The test consists of several requests to test the various request types. The output you get in the terminal of *SecureFileStorageTester* should be self-explanatory. As an example, test 0d is described:

```
Test 0d: Check LOGIN (valid credentials)... done
Test **SUCCEEDED**
Status:  OK
Content: user1:091EF6086D2EDC4CF80DF0A19A99005BEBAE1F73F19F5195BAAD01A
         551226B55
```

The first line describes the test. Here, a *LOGIN* request is tested that uses valid credential. The second line *Test **SUCCEEDED** means that the test was successful, i.e., it worked as expected. A failure would be indicated with *Test **FAILED**. Next, there's the status of the received response, which is either *OK* or *NOK* – here it is *OK* because the *LOGIN* request could successfully be handled. Finally, there's the content of the response, which – in this case – contains the session ID (combined with the username) that was generated by the server.

Whenever you are doing some changes to the server code throughout the remainder of this lab, it's always a good idea to re-run this test to check whether the basic legitimate requests are still handled correctly.

## 5 Source Code Analysis of *SecureFileStorageServer.java*

Before you start solving the tasks in the next section, you should study the source code of the server in detail to get a good understanding about its functionality and to possibly already identify some vulnerabilities. Basically, the program works as follows:

- The *main* method creates a *SecureFileStorageServer* object and starts the server (*run* method).
- In the *run* method, a while loop is used to listen for connection requests from clients. When a client establishes a connection, the method *accept()* returns and delivers a *Socket* object, which can be used to communicate with the client. This object is passed to the *processRequest* method, which handles the specific request. Note that for simplicity, the server runs single-threaded, which means the next connection (from the same or from another client) will only be accepted once the previous request has been completely handled (including sending back the response).
- *processRequest* reads the first line of the request and checks the type of the request. Depending of this type (*REGISTER*, *LOGIN*, *GET*, *PUT* or *SYSTEM*), it calls a corresponding method (*register()*, *login()*, *serveFile()*, *storeFile()* or *executeSystemCommand()*), where the remainder of the request is handled and where the response to the client is generated. When *processRequest* is completed, program control goes back to the *run* method so the next connection request can be accepted.

Take your time to get a good understanding of the code – you'll need this understanding later in this lab to make the right program adaptations to fix the security vulnerabilities. Also, write down any potential security vulnerability that you detect while studying the code in the following box. At the end of the lab, you can check how many of the vulnerabilities that are present in the program you already managed to detect while studying the code. Note that communication with the server is currently in plaintext. This could easily be changed by using TLS and is not considered to be a vulnerability. Also, the usage of plaintext passwords in the file *users* should not be considered as a vulnerability; this is currently done for simplicity and could easily be fixed by storing salted and hashed passwords.

There is almost no input validation in the code.
Currently every user input is accepted by the server,
which can have fatal consequences.

# 6 Attacks and Countermeasures

In each of the following subsections, *SecureFileStorageTester* is used to carry out an attack against the running server. Your task is always the same: First, carry out the attack to see its effect. Then determine the reason why the attack was successful. Next, adapt the server code to fix the vulnerability (this is needed with all attacks except attack 1). Finally, check whether the attack is truly prevented by executing it again and also check whether test 0 (see section 4) can still be executed successfully.

It's important that you go through the attacks in the given order as in some cases, the attacks build upon attacks and countermeasures of previous subsections. Also, always make sure to completely solve a subsection before starting with the next one. When adapting the code, you should always focus on fixing the vulnerability that allows the current attack and then you should 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 vulnerability correctly (according to the specifications in the individual subsections).

## 6.1 Attack 1: Compromising the root account by setting a new root password

In this attack, you are exploiting vulnerabilities in the server to set a new root password in the underlying system. Before you execute the attack, you should create a copy of the shadow file (which contains the passwords of the system users). This requires root permission and can be done – assuming you are working in a terminal as *user* – with the *sudo* command, followed by providing the password of *user* (which is *user*):

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

Next, start the server so that it runs with root permissions. This is also done with *sudo*:

```
$ sudo java
  ch.zhaw.securitylab.securefilestorage.SecureFileStorageServer
```

Finally, execute the attack (which has number 1) by entering the following in another terminal:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 1
```

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

Now, try to get root access, e.g., with an SSH login from the physical host or by using *su* in a terminal. Use the password *test*. The login attempt should be successful, which means the root account was indeed compromised by setting a new root password *test* (the original password was *root*) in the shadow file.

This can be verified by comparing the original and modified shadow files, e.g., by using *diff*:

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

What has changed? What hasn't changed?

It looks like the password has been changed => it can be inferred because the password hashes are different for both files.

Why are the passwords not directly visible?

Because of security. If the attacker mange to get the password file, he will be able to get all password in a very short time. Through hashing of the passwords it takes more time for the attacker to find out the correct passwords. He has to find hash function first.

The fact that this attack was successful has multiple reasons – which ones? To provide a complete answer to this, you'll have to study the source code of the server and also the requests (especially the first line of them) that are received by the server (for the latter, there's already a code line in method *processRequest*, but it's currently commented). It may also be helpful using some further debugging messages in the server code (e.g., with *System.out.println()*).

One of reason ist that program was running in root mode.
The other reason is missing validation for HTTP Get Request. It is not checked if the request is valid at all.

To remove the effect of the attack, replace the modified shadow file again with the original one:

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

Next, stop server and restart it, but this time as *user*:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageServer
```

Execute the attack again. What happens? Explain the behavior.

The attack in not working anymore.
The attacker has no permission to read shadow file, because the server is running in normal user mode.

With this, the attack is prevented, because one of the reasons that made it possible has been removed (as the server no longer runs with root permissions). However, the other reasons that made the attack possible (which are vulnerabilities in the code and which you likely identified above) are still there. This will be fixed later.

The important take away message of this attack is that you should never run a program with root privileges (unless this is really necessary), because if there exists a vulnerability, the impact is usually higher when the program runs with root permissions.

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

### 6.2 Attack 2: DoS attack with empty request

Make sure the server is running and execute attack 2:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 2
```

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

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

Reason for the server crush was the following exception: StringIndexOutOfBoundsException
Reason for this exception was missing validation for the request, it was assumed that the request is always formatted correctly and it consists of any not empty content.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

This problem could be fixed by checking if request is:
1. not empty
2. formatted correctly and consists of all required request elements as defined in the parts 3.1-3.5

Before adapting the code, let's discuss the vulnerability in more detail. As you probably found out above, the problem is that the server tries to process the request, which consists of an empty first line. As a result of this, a *StringIndexOutOfBoundsException* is thrown. As the exception is not caught, the program is terminated. This is a typical example of a vulnerability related to a lack of input validation. Input validation issues are very prevalent in software and more of them will follow in subsequent attacks. Generally speaking, a lack of input validation means that the software (here the server) blindly accepts and processes input data without validating the data first, i.e., without making sure that it conforms to an expected and legitimate data format before processing it. This can then lead to unexpected and unforeseen situations, which can have serious security implications. Here, it «only» resulted in terminating the server and thereby affects availability, but – as you'll see later – it can also have an impact on confidentiality and integrity.

The solution is to first validate input data before processing it. One reasonable way to do this in the current case is to implement a method with name *validateFirstLineOfRequest(String input)*, which gets the first line of the request as a parameter. The method then checks if the first line consists of a valid format. Looking at the protocol definition in section 3, you can see that a valid first request line consists of the request type (*REGISTER*, *LOGIN* etc.), followed by one space character, followed by one or more arguments, whereas each argument consists of printable characters (except space characters) and where the arguments are separated by space characters. This format can best be tested using a regular expression and in general, regular expressions play an important role when implementing input validation rules. Using a regular expression, *validateFirstLineOfRequest* can be implemented as follows:

```
private boolean validateFirstLineOfRequest(String input) {
  if (input != null && input.matches(
      "^(REGISTER|LOGIN|GET|PUT|SYSTEM) [\\x21-\\x7E][\\x20-\\x7E]+$")) {
    return true;
  }
  return false;
}
```

Study this method, especially the regular expression (in bold). In the following box, explain in detail why the regular expression is suited to validate the format of the first line of a request by describing

the purpose of the different parts of the regular expression. Note that the actual regular expression contains only single slashes (\), but in a Java string, it's necessary to escape a slash character (using \\) so that it is interpreted as a literal slash character. This means that the substring *[\\x21-\\x7E][\\x20-\\x7E]* corresponds to *[\x21-\x7E][\x20-\x7E]* in the actual regular expression.

Hint: To play around with regular expressions, *https://regex101.com* is an excellent resource.

```
^(REGISTER I LOGIN I GET I PUT I SYSTEM) new line starts with one of the string in the braces
follows be space than
[\x21-\x7E] any ASCII character in the range 21-7E, in decimal 33-126: contains all small
and capital letters, numbers and standard special characters. Space is excluded.
[\x20-\x7E] dito above, but here space is included
+ quantifier: matches one or more times
$ defines position at the end of a line

In short it is checked if the input matches the request format defined in the section 3
```

Next, call the method above in method *processRequest* to validate the first line of each request before processing it. If validation fails, respond with *NOK*. The easiest way to implement this is as follows (new code in bold):

```
if (line == null) {
  // Apparently, the client disconnected without sending anything, do
     nothing
} else {

  // Validate the first line of the request
  if (!validateFirstLineOfRequest(line)) {
    writeNOKNoContent(toClient);
    return;
  }

  // System.out.println("First line of request: " + line);
  // Get request type and argument from the first line of the request.
  int indexSpace = line.indexOf(' ');
  ...
```

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.3 Attack 3: DoS attack with malformed GET request

Make sure the server is running and execute attack 3:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 3
```

The server should crash, so just like with attack 2, this can be identified as a DoS attack.

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

It is one more time StringIndexOutOfBoundsException
Reason for this exception is missing validation for the second line of get request, which contains username and the session ID

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

This problem could be fixed by checking if second line of request:
1. contains usernames, which is followed by „:“ (colon) and session ID

Most likely, you came to the conclusion that this is another input validation problem. Therefore, implement another validation method *validateSessionID*, that checks it the line that follows right after the first line in some requests has a valid format, i.e., that it consists of a username and a session ID. The rules for this line to be legitimate are as follows:

- It starts with the username. For now, we assume a username has a length between 5 and 20 characters (including 5 and 20) and consists of printable ASCII characters except space characters (*[\x21-\x7E]*). Specifying the number of characters in a regular expression can be done with curly brackets that follow a character set. For instance, *[a-z]{5,10}* matches any sequence of lowercase characters of length between 5 and 10 (including 5 and 10).
- Between the username and session ID, there is exactly one colon (:).
- The session ID consists of exactly 64 hex characters (which corresponds to 256 bits), where hex characters include the following characters: *0123456789ABCDEF*.

This can again be implemented using a regular expression. Once you have implemented the method, use it in *serveFile* right after reading the second line from the client. If validation fails, respond with *NOK*:

```
// Get the next line from the request and extract username and
   sessionID
String line = fromClient.readLine();

// Validate the session ID
if (!validateSessionID(line)) {
  writeNOKNoContent(toClient);
  return;
}
```

Fix Regex: ^[\\x21-\\x7E]{5,20}:[(\\x30-\\x39)|(\\x41-\\x46)]{64}

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

Before you go on, use this validation method also in all other places in the server code where the session ID from the client is received and processed. Hint: The method is needed in two further places. Use the method in exactly the same way as above.

This last step is very important: If you have detected a vulnerability – as with the malformed GET request in this case – then always make sure to fix the issue throughout the entire software, and not just where it occurred to truly get rid of the vulnerability. If you don't do this, the attacker will usually still be able to successfully execute the attack simply by targeting another program component.

### 6.4 Attack 4: DoS attack with long request

Make sure the server is running and execute attack 4:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 4
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

Server crushed because of the following exceptions: OutOfMemoryError: Java heap space
The request is too large and fits not in the java memory

The client sends a request where the first line of the request has a length of 5 GB. The attack could basically be prevented by assigning the server more memory (in Java, this is possible with the option -Xmx, e.g., -Xmx10G). Why is this a poor solution for the identified problem?

If the server enables to set so large requests, the possible attack of DoS could be to set a lot of huge request within a short time. As result the server wouldn't crush, but it will be blocked for any other short requests.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

Possible solution is restriction of the request size.

To fix this problem, the best approach is to implement a variant of *readLine* that only reads a line up to a certain maximum number of characters. Do this by implementing a method *readLineMaxChar* that uses the following interface:

```
private String readLineMaxChar(Reader reader, int max)
    throws IOException {...}
```

The method gets an object *reader* of type *Reader* (e.g., a *BufferedReader* to read data from a socket) and a maximum number of characters to read (*max*). It then reads a line from *reader* (terminated with \n or end-of-file) and returns it (without the \n). If the line is longer than *max* characters (not counting

the \n), an *IOException* is thrown. To prevent an *OutOfMemoryError*, it's important that you count the read characters while reading them character-by-character from *reader* and that you throw an *IOException* as soon as more than *max* characters have been read (the remaining characters will be ignored in this case). If no data can be read from *reader* at all (i.e., if the first read operation returns end-of-file), *readLineMaxChar* should return *null*.

Then, use this method instead of the standard method *readLine* in *processRequest*, i.e., replace

```
// Read first line of request from the client
String line = fromClient.readLine();
```

with

```
// Read first line of request from the client
String line;
try {
  line = readLineMaxChar(fromClient, 1000);
} catch (IOException e) {
  writeNOKNoContent(toClient);
  return;
}
```

As you can see, the maximum line length is limited to 1'000 (it's best to use a constant for this). This is certainly good enough to process the first line on any reasonable request. If *readLineMaxChar* throws an exception (which means the line is longer than 1'000 characters), respond with *NOK*, as illustrated in the code above.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.5 Attack 5: DoS attack with PUT request with a long file line

Make sure the server is running and execute attack 5:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 5
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

The same problem as in 6.4: OutOfMemoryError , but this time PUT request and storeFile() - Method are affected.

Having already prevented attack 4, it should be obvious how to prevent attack 5: Make sure that *readLineMaxChar* is used instead of *readLine* for *all* the lines read from the client and not just for the first one. Therefore, adapt the code so that all *readLine* calls where data is read from the client are replaced with *readLineMaxChar* (do this throughout the entire code to make sure that all attacks that use too long lines are prevented). For simplicity, we restrict every single line to 1'000 characters, just like above. If *readLineMaxChar* throws an exception, always respond with *NOK*. Once you have done these adaptations, all code lines of this form:

```
line = fromClient.readLine();
```

should have been replaced with code lines of this form (including the necessary exception handling):

```
line = readLineMaxChar(fromClient, 1000);
```

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.6 Attack 6: DoS attack with PUT request with many lines

Make sure the server is running and execute attack 6:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 6
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

Server crushed one more time because of the out of memory problem.

Again, it shouldn't be too difficult to identify the problem. Fix the vulnerability in *storeFile* by making sure that a file that should be stored cannot contain more than 1'000 lines. If there are more than 1'000 lines, respond with *NOK*. This means that in combination with the fix introduced in the previous subsection, a file can contain at most 1'000 lines, where each line cannot be longer than 1'000 characters. Note again that these limitations are chosen quite arbitrarily and in a real application, you'd probably choose other bounds; but what's important is that you realize that you have to specify and enforce some upper bound for the file size as otherwise, DoS conditions will likely occur.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

With this attack, we are stepping away from DoS attacks that target the availability. Maybe you found attacks 2-6 a bit repetitive as the attacks are all somewhat similar in the sense that they triggered different exceptions in the server that all resulted in terminating the running program. But exactly this is an important lesson that you should take away from attacks 2-6: In many cases, there are different ways to achieve an attack goal (here: five different ways to crash the server) and the attacker just needs to find one of these ways to be successful. So if you prevent four of the attacks by fixing the associated vulnerabilities but don't fix the fifth vulnerability, the attacker still wins if he finds the remaining vulnerability. Therefore, when thinking about possible attacks against the system you are developing (which is a very important activity during a secure software development process), you should always consider many different ways how a specific attack goal can be achieved and then you should make sure that all these ways are prevented in the code. In this sense, attacks 2-6 certainly gave you a taste what it means to prevent all different ways to achieve an attack goal and to truly achieve secure software.

### 6.7 Attack 7: Getting a file of another user (variant 1)

Make sure the server is running and execute attack 7:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 7
```

In this attack, a login is done as *user1* and then, *file2.txt* of *user2* is received (using GET). This is obviously bad and corresponds to an access control vulnerability, as users should only be able to access their own files.

Analyze the request and the source code of the server and explain why it is possible to get the file.

Missing validation for the filename. Server accepts filename, which contains e.g following characters: „/../../..“
This enables the attacker to travers the directory, so that he can access any file, he wants.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

Possible solution is validation of a filename to prevent using characters like above „/../../..“

The best way to fix this is again with input validation. Basically, you have to make sure that the filename specified in the GET request does not contain the slash (*/*) character, so it's not possible to escape «upwards» from the directory where the files of the current user are stored (here: *data/files/user1*).

Implement this by first providing a method *validateFilename* that gets the filename received in a GET request and that checks that the filename is legitimate. Do this again with a regular expression. In our case, a legitimate filename has a length between 1 and 50 characters and allowed characters are *a-z*, *A-Z*, *0-9*, and the following special characters: _+=$%?,.;: . Then, use *validateFilename* right at the beginning of *serveFile* to validate the received filename. If it is not valid, respond with *NOK*. Also, do the same check at the beginning of *storeFile*, so an attacker cannot (over-) write files of another user.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

The attack discussed in this subsection is also called directory traversal attack as the attacker traverses the directory tree to access areas that shouldn't be accessible by him. In addition, the validation of the filename you just added has two beneficial side effects:

- As the / character can no longer be used, it is enforced that a user cannot create or read files in subdirectories of *data/files/user1* (in the case of *user1*). This was specified Section 3, but so far, it was not enforced in the code.
- Attack 1 also makes use of a directory traversal attack. This means that with the newly added validation of the filename, attack 1 should not be possible any more, even if the server runs with root permissions. Verify this by repeating attack 1 – it should fail.

### 6.8 Attack 8: Getting the users file that contains the credentials (variant 1)

Make sure the server is running and execute attack 8:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 8
```

As a result of the attack, the content of the file *users* (which contains the usernames and passwords) is received. That's certainly a serious vulnerability.

Analyze the request and the source code of the server and explain why it is possible to get the file.

In the previous regex for filename validation, the ASCII characters are still allowed.

This attack shows that input validation is not always as easy as it seems because although we made sure in the previous subsection that filenames with slashes (*/*) are not accepted, directory traversal attacks are apparently still possible.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

One possible solution is extension of previous regex to handle also the ASCII characters.
But this is just a quick fix and doesn't solve the problem in general for other possible user inputs if URL Encoding is used.

For now, we just implement a «quick fix»: Remove URL decoding of the filename in *serveFile* and *storeFile* (simply remove the calls of the method *urlDecode* and directly use *filename* when building *filepath*). This is not a true fix as URL encoding is now no longer supported in *GET* and *PUT* requests, but for now we consider this to be good enough as it at least prevents the attack. We will fix this in a better way later.

Note: If you don't understand why this prevents the attack, then you didn't really understand why the attack was possible at all and in this case, you should spend some additional time to truly understand what happens during the attack.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.9 Attack 9: Getting a file of another user (variant 2)

Make sure the server is running and execute attack 9:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 9
```

In this attack, a login is done as *user1* and then, *file2.txt* of *user2* is received (using *GET*). Hmm... didn't we prevent this when discussing attack 7? So apparently, there's (at least) one additional vulnerability still present to achieve this attack goal.

Analyze the request and the source code of the server and explain why it is possible to get the file.

Access rights of the user still have to be validated. At the moment every user with the correct session id can get access to the files of the other users.

This is an example of quite a simple programming bug as it can easily happen during implementation, but which has major security consequences.

Adapt the code to fix the underlying problem. Make sure to solve the problem throughout the entire application, i.e., not only in the context of *GET* requests but also with the other requests if needed. In the following box, briefly describe what you did in principle to fix the vulnerability.

The method checkSessionID was extended. The method proofs now additional if the user has not only correct session id but also it checks if the user also has the access rights.
SESSION_IDS - HashMap was already implemented in the existing code and it stores and maps this both information: username + sessionID

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.10 Attack 10: Guessing a valid session ID

Make sure the server is running and execute attack 10:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 10
```

In this attack, one tries to find out a valid session ID that has been issued by the server before. To do this, *SecureFileStorageTester* first logs in as *user1* to get a valid session ID. This step is only done to make sure the server has issued a valid session ID for *user1*, but *SecureFileStorageTester* does not use the knowledge about this session ID in the remainder of the attack. Then, *SecureFileStorageTester* creates several session IDs candidates and uses them to access a file of *user1*. If this works with any of the session ID candidates, the correct session ID has been guessed.

As you can see, the correct session ID can be guessed after several 1'000 attempts. Analyze the source code of the server and explain why it is possible to guess the session ID.

The current time (Instant.now()) is used as seed for the SHA-256. It is very bad practice, because knowing the Hash-Function and the possible seed value the attacker is able to find the correct ID in a very short time

Obviously, the session ID is currently created in a very insecure way. Adapt the code of method *createSessionID* to create a secure session ID that cannot so easily be guessed, even if the attacker knows the algorithm to create the session ID and has huge (but still realistic) computing resources. The generated session ID should still have a length of 256 bits (corresponding to 64 hex characters). Briefly

describe your adapted algorithm to create session IDs in the following box and explain why it creates non-guessable session IDs:

The code adaption is: using pseudo random number generator (PRNG) to provide a non-guessable seeds instead of current time. Java provides the class SecureRandom for this use case.

Assuming that you adapted the algorithm in any way (even in an insecure way), executing the same attack again using *SecureFileStorageTester* will obviously no longer work because the attack is based on the previously used algorithm to generate session ID candidates. If you want to, you can still run the attack again and it will give up after 50'000 attempts. However, make sure to run test 0 to make sure that the adapted mechanisms to create session IDs works when the client uses legitimate requests.

### 6.11 Attack 11: Registering as an existing user

Make sure the server is running and execute attack 11:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 11
```

In this attack, the attacker manages to register as *user2* (although *user2* is already registered and although method *register* uses method *userExists* to check if the user does not yet exist). This then allows the attacker to login as *user2* and access all the files of *user2*. You can also check the file *data/system/users* to verify that it now contains two entries for *user2*.

Analyze the request and the source code of the server and explain why it is possible to register as *user2*.

It is similar problem as in the section 8. Through possible using of html encoding, the attacker can introduce within get request body not allowed characters as e.g new line „%0A“

This attack, just like attack 8, shows that supporting encoding schemes (here: URL encoding) is potentially dangerous as it may allow attacks that are often overlooked during development. In the context of attack 8, we reacted to this by simply removing URL decoding of the filename in *serveFile* and *storeFile*. We could do this here as well, but let's assume supporting URL encoding is an important and mandatory feature in the secure file storage server. Therefore, simply removing it is not an option and instead, we are now going to implement it in a secure way.

Basically, the secure approach to deal with encoding schemes is to do the decoding first and do all further security checks on the decoded data. Therefore, first remove the calls of *urlDecode* in methods *register*, *login* and *executeSystemCommand*. Then, use *urlDecode* in *processRequest* right when extracting the argument from the first line of the request:

```
int indexSpace = line.indexOf(' ');
String requestType = line.substring(0, indexSpace);
String argument = urlDecode(line.substring(indexSpace + 1));
```

This guarantees that from now on, we are working with the raw decoded data and no longer have to worry about potentially encoded characters that may circumvent subsequent security checks. Note that attack 8 is still prevented although we re-introduced URL encoding as the decoding is now done *before* the filename is validated with *validateFilename*.

To prevent the current attack 11, the (decoded) credentials specified by the client must be validated, in particular that they do not contain the newline character – which was the main reason that the attack was successful. To do the validation, implement a method *validateCredentials* that checks whether credentials received during registration are legitimate. In our case, legitimate credentials start with a username with a length between 5 and 20 characters. This username consists of printable ASCII characters except space characters *([\x21-\x7E]*). The username is followed by exactly one colon (:), which is followed by the password. The password must have between 8 and 50 characters and allowed characters are *a-z*, *A-Z*, *0-9*, and the following special characters: _+=$%?,.;: . Then, use *validateCredentials* right at the beginning of *register* to validate the received credentials. If they are not valid, respond with *NOK*.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.12 Attack 12: Getting the users file that contains the credentials (variant 2)

Make sure the server is running and execute attack 12:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 12
```

As a result of the attack, the content of the file *users* (which contains the usernames and passwords) is received. We have already seen this in attack 8, but obviously there's another way to get this file.

Analyze the request and the source code of the server and explain why it is possible to get the file.

username still has to be validated. At the moment it is possible introduce char „/“ within the username.
It has to be prevented.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

Possible solution is regex expansion for the credentials, which proofs that the username doesn’t contain char - combination as „/“ or „../“ etc.

There are different ways to fix this. One way is to make sure that during registrations, usernames cannot contain slashes (/) to prevent directory traversal during file access. Another way is to check that the file path, which is built in *serveFile* and *storeFile*, does not contain ../ substrings, again to prevent directory traversal. We are going to implement both, because security-wise, it's always good to have multiple safeguards to prevent an attack (this is called «defense in depth») as it may be that one of the safeguards is implemented faulty or it may be that one of them can be somehow circumvented by an attacker. For instance, in the current case, if we only prevent during registration that usernames con-

tain slashes and if the attacker finds another way to insert an entry with the username *../system* into the *users* file, then the attack would be successful. But by additionally checking the file path when it is built in *serveFile* and *storeFile*, the attack would still be prevented.

To fix the problem, adapt *validateCredentials* that you implemented in the previous subsection so that it only accepts characters *a-z*, *A-Z*, *0-9* and the special characters _ and . (dot) for the username. For the users, this certainly provides enough flexibility to pick a username. In addition, implement a method *validateFilepath* that checks if a file path does not contain the substring *../* (two dots followed by one slash).

Next, adapt *serveFile* and *storeFile* so that the file path that is built in the code is checked using *validateFilepath*. If the validation fails, respond with *NOK*. Add this check after the following code line, both in *serveFile* and *storeFile*:

```
String filepath = FILES_DIR + username + "/" + filename;
```

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.13 Attack 13: Login without knowing the password

Make sure the server is running and execute attack 13:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 13
```

In this attack, a successful login is done as *user3* without knowing the password. Analyze the request and the source code (focus on the method *checkCredentials*) of the server and explain why it is possible to log in.

The variable passwordCorrect is set by default to true. It means if there is some use case, that is not caught by exception, the server will terminate with accept the password.
In this case the password was set to empty string.

Fix method *checkCredentials* to remove the vulnerability. In the box below, briefly explain the adaptations you did to the method.

This problem can be prevented through setting the default value of passwordCorrect to false.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

This vulnerability is a typical example of «fail insecurely». This means that in case of a failure (or an unexpected situation, here the lack of a password in the credentials received from the client), the soft-

ware fails in an insecure way. In this case, this insecure failure has the effect that the login is accepted although the request didn't even include a password.

As a sidenote: To further increase security, method *validateCredentials* should additionally be used right at the beginning of the method *login*. This guarantees that credentials always have the expected format before they are processed further.

### 6.14 Attack 14: Do a portscan on the server

Make sure the server is running and execute attack 14:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost 14
```

As a result of this, you first get the disk space (output of the command *du*) used by the files of *user1*, followed by the output of a portscan (with *nmap*) that was done locally on the server system. A portscan is just used as an example here, other commands that are available on the server system could be used as well (e.g., to download and install malware).

Analyze the request and the source code of the server and explain why it is possible to do the portscan.

Command validation is missing. At the moment the server allows execution of all possible commands

Apparently, the attack makes it possible to execute basically arbitrary commands on the server system and that's definitely another major vulnerability. This kind of attack is often identified as command injection.

Obviously, this is another case for input validation. Therefore, implement a method *validateCommand* that makes sure only valid commands can be used by the clients. Up to now and according to section 3.5, the only valid command is *USAGE* and it either needs * or . as an option. This can easily be enforced by using a regular expression. Once implemented, use *validateCommand* right at the beginning of method *executeSystemCommand*. If validation fails, respond with *NOK*.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

### 6.15 Check that all tests work and that all attacks are prevented

As a final check, run *SecureFileStorageTester* without a test or attack number:

```
$ java ch.zhaw.securitylab.securefilestorage.SecureFileStorageTester
  localhost
```

Inspect the output in detail. All tests in the beginning (test 0) should succeed and all attacks (attacks 1-14) should fail. If that's not the case, correct the code until everything works as expected.

### 6.16 More attacks?

In section 5, you analyzed the source code to get a good understanding of the code and to identify some potential vulnerabilities. By comparing the vulnerabilities you identified there with the ones discussed in the context of attacks 1-14, you can assess «how well you did» when analyzing the source code.

In case you identified any vulnerabilities in section 5 that were not addressed when carrying out attacks 1-14 in section 6, briefly describe these vulnerabilities and also the corresponding attacks that would be possible by exploiting them in the following box:

## Lab Points

In this lab, you can get **4 Lab Points**. To get them, you must demonstrate to the instructor that you successfully fixed the security vulnerabilities in the server program. This means that when running *SecureFileStorageTester* without a test or attack number (as described in section 6.15), test 0 should still work and that attacks 1-14 should be prevented. Furthermore, you must show and explain your answers to the questions in sections 5 and 6 (which means the boxes must contain reasonable answers) and, if requested by the instructor, you have to describe the changes you did in the source code to fix the vulnerabilities.

***Plan B:*** In case you cannot demonstrate your solution in person: Make a video of a terminal that shows the correctly running *SecureFileStorageTester* program and send the video and a document with your answers to the questions in sections 5 and 6 (it's best to write the answers directly into this PDF document) by e-mail to the instructor. Use *SecLab - Secure File Storage - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big to include in an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.