UD05: Secure coding

1. Introduction
2. Fundamentals of computer security
   2. 1. Security/reliability concepts
   2. 2. Types of security
   2. 3. Vulnerabilities and types of threats
   2. 4. Protection and standards related to computer security
3. Vulnerabilities in the software.
   3. 1. Defensive Programming
      3. 1. 1. Password check
      3. 1. 2. Error information
   3. 2. Exceptions
      3. 2. 1. Throwing exceptions
      3. 2. 2. Exception handling
      3. 2. 3. Create our own exceptions
      3. 2. 4. Advice
   3. 3. Records or Logs (`Log4j2`)
      3. 3. 1. Automatic configuration
      3. 3. 2. Log Levels
      3. 3. 3. Appenders
      3. 3. 4. Example code
      3. 3. 5. Manual configuration
   3. 4. Input validation.
   3. 5. Security politics.
   3. 6. Forbidden
4. Information sources

1. Introduction

Information and Communication Technologies (ICT), computer systems and computing are found in practically all areas of our day to day. All these technologies help us to carry out our work or enjoy our leisure time in a faster and more decentralized way. At the same time, this means that something as transversal as computer security must accompany us at all times. We must always keep computer security in mind in all the actions and decisions we make in our computer systems. This is the only way to guarantee that they are available for as long as possible, that they correctly perform the task for which they were designed and that they are used only by those users who have been authorized.

2. Fundamentals of computer security

Computer security is a discipline that requires a lot of dedication. This is due to the constant evolution of protection systems. For their part, criminals continually improve the techniques and mechanisms to obtain information or access to systems for which they should not have it.

In addition, in computer security there is a very contradictory situation in which three characteristics come together: functionality, security and usability. As we can see in the figure they are interrelated, the more secure a system is, it will be at the cost of its functionality and usability that will decrease. The same thing if what we want is to have a very usable system, we will penalize its functionality and security in some way.

Therefore, it is about reaching a consensus among all the elements involved in the security of our computer system.

> From the beginning you have to understand that absolute and/or guaranteed security does not exist, so our goal should be to achieve security levels as high as our objectives set. According to our needs, requirements and costs.

2.1. Security/reliability concepts

There are three basic principles called the CIA triad, for its acronym in English, regarding computer security:

- Confidentiality: It is the characteristic that a data, information or message has that allows it to be understandable only by the system or person who is authorized to understand and use its content.
- Integrity: It is the property that has a data, information or message that makes it possible to ensure that it has not undergone a voluntary or accidental modification of the original content. You must preserve its accuracy and completeness.
- Availability: It is the possibility offered by the data, information, messages or systems that allows them to be accessible at all times by any user or system that is authorized to do so.

In addition to these, the services provided to us are important:

- Authentication: It is the mechanism that allows verifying that a user or system is who it claims to be. A combination of at least two elements of the following factors is usually used:
    - Something you know, that we have chosen (for example: our user's password, our credit card pin, etc.)
    - Something that you have, that someone has given us (for example: a credit card, digital certificate, mobile device, etc.)
    - Something that you are, from our body (for example: our fingerprints, our iris, our face, etc.)
- Access control: ensures that access to information or systems is restricted based on security and business needs.
- No repudiation: serves to ensure that a system/user has participated in a communication, the preparation of information or access to it.
    - at origin: the sender cannot deny that it is he who sent the message.
    - at destination: the receiver cannot deny that he received the message.
- Traceability: records the information on the use or accesses that have occurred on the systems or the information object of observation. It stores something similar to a history with everything that has happened in the system and its information.

The three basic principles, together with the four services that we have just named, form the 7 pillars of computer security:

2.2. Types of security

Depending on when security measures are applied, we can find:

- Active security: These are measures that are applied before the security problem occurs, and therefore preventive measures.
- Passive security: These are corrective measures or that are applied after the problem occurs and are aimed at reestablishing the service in order to reduce the impact of the information security problem.

2.3. Vulnerabilities and types of threats

We call vulnerability to the possible weaknesses that a system or information presents and that can be exploited by a threat. That is, they are the weak points of our system.

Therefore, we must have them identified, know their importance and define security measures that help us control or avoid them. Vulnerabilities, with examples, can be:

- physical: inadequate installations (temperature, humidity, location, wiring, fire protection, etc.)
- natural: earthquakes, hurricanes, rains, floods, etc.
- hardware: incorrect equipment maintenance, lack of backup, etc.
- software: incorrect configuration, lack of updates, etc.
- about comunications: use unencrypted communications, allow access from outside without control, and so on.
- storage: manufacturing defect, inappropriate place to store digital media, etc.
- Human: weak passwords, sharing usernames and passwords, lack of training, etc.

Threats are any type of event or action that can cause damage to the system or information. They can cause a security incident, which in turn causes damage and whose risk of occurring is significant.

- security incident: unexpected and unwanted event that can compromise security.
- damage: the damage that occurs when a threat occurs.
- risk: the product of the damage and the probability that this threat occurs.
- Regarding the types of threats, we are going to differentiate between:
    - how they are produced:
        - Natural: Caused by nature.
        - Involuntary: Unconscious, accidental actions or mistakes.
        - Intentional: Deliberate actions such as: vandalism, sabotage, theft, etc.
    - how they behave:
        - Active: The attacker's objective is to modify the information or create a new one that will pass off as original.
        - Passive: The attacker gains access to the information, but does not modify it.

2.4. Protection and standards related to computer security

One way to secure the computer system in our charge is to carry out a security audit, which will analyze the threats and/or risks that we may suffer and calculate a degree of importance or probability that these situations occur.

> Computer security audit: To carry out a security audit we must obtain express authorization from the owner of the computer system to be analyzed. It consists of several phases: enumeration, detection and evaluation of vulnerabilities, corrective measures and implementation of preventive measures.

Regarding the standards related to computer security, we must refer to the ISO/iec 270000 family of standards as well as UNE/ISO 17799.

We must also consider these laws and regulations: LOPD-GDD, LSSICE, RGPD. We have support services such as INCIBE, CCN-CERT and CSIRT.

3. Vulnerabilities in the software.

Due to the rise of the Internet, software vulnerabilities are one of the biggest problems in computing today. Current applications increasingly deal with more data (personal, banking, etc.) and for this reason, it is increasingly necessary for applications to be secure.

A software vulnerability can be defined as a flaw or security hole detected in a program or computer system that can be used to enter systems in an unauthorized way. In a simple way, it is a design flaw in a program, that you may have installed on your computer, and that allows attackers to perform a malicious action. If we focus on an application, it is possible that this security breach allows a user to access or manipulate information that is not authorized.

But the big problem with security holes lies in the way they are detected. When an application is of a certain size, the job of detecting security flaws is entrusted to external users to use the application normally and/or to security experts to thoroughly analyze the security of the application.

Despite the fact that before launching an application on the market, an attempt is made to discover and solve all its security flaws, on many occasions they are discovered later and, therefore, it is necessary to inform users and allow them to update to the latest version that is released. find fault free.

In order to guarantee the security of a Java application, we are going to focus on two pillars:

- Internal application security. When creating the application, it must be programmed in a robust way so that it behaves as we expect of it. Some of the techniques that we can implement are exception management, data entry validations and the use of log files.
- Access policies. Once we have a "safe" It is important to define the access policies to determine the actions that the application can perform on our computer. For example, we can indicate that the application can read the files in a certain folder, send data over the Internet to a certain computer, etc. In this way, even if a malicious user misuses the application, the impact of their attack is limited. Thus, if we have indicated that it can only read the files in the c:/data folder, the attacker will never be able to modify that data or read the files in another directory.

3.1. Defensive Programming

Error situations occur from different causes, such as *incorrect implementation*, when the code does not conform to the specifications, an *inappropriate object request*, such as an invalid index (into an array) or a null reference, or that the state of an object is inappropriate or non-existent.

But it is not always due to a programming error. Errors may occur in the environment, such as the wrong input of data or an interruption in communications; Also, during file processing, such as the fact that the file we want to access does not exist or that the access permissions are inadequate.

These events mean that, in order to guarantee the proper functioning of the applications, the technique of defensive programming must be used, which entails the incorporation of a large number of checks , with the consequent decrease in performance.

But what things need to be checked? Should the fault be dealt with at the point at which it occurs, or are errors reported to the code that invoked the code where it occurred? how should the error be handled? If the error is not addressed, how do you report it? Can the code anticipate the failure? And if it does, when?

Let's look at a simple example. Suppose we have an agenda; we identify the entries in it by some kind of key that we can use to locate them. We want to delete a record: we locate it through its key and then we delete it:

```
public void eliminar(String clave) {
    Contacto contacto = agenda.buscar(clave);
    agenda.borrar(contacto);
}
```

An error occurs: for example, that the record does not exist. Who is guilty? In general, it is better to anticipate than complain.

3.1.1. Password check

Arguments are a vulnerability:

- Initialize the state in a constructor
- Contribute to the behavior in a method

Argument checking is a defensive measure:

```
public void eliminar(String clave) {
    Contacto contacto;
    if (agenda.existe(clave)) {
        contacto = agenda.buscar(clave);
        agenda.borrar(contacto);
    }
}
```

In this case, we check if the record we want to delete exists before proceeding with the deletion.

3.1.2. Error information

The classic model for reporting that an error has occurred is to return a diagnostic. For example, the method can be implemented as a boolean function that returns true only if the operation is successful:

```
public boolean eliminar(String clave) {
    Contacto contacto;
    if (agenda.existe(clave)) {
        contacto = agenda.buscar(clave);
        agenda.borrar(contacto);
        return true;
    } else {
        return false;
    }
}
```

The output of the function must be checked by the method that called it:

```
if (!objeto.metodo(parametros)) {
    // tratar error
} else {
    // código normal
}
```

This is a simple method but it unnecessarily complicates the code due to nesting. In addition, other types of runtime errors can occur that are not trappable.

Modern languages, especially those that support object-oriented programming, incorporate an error management mechanism through the so-called exceptions.

The structure of the code in this case, as before, has two parts, the called method, in which the error is raised, and the calling method, in which the error can be handled. In the first, when the error occurs, either because some condition is not met or because there is some error in the environment, the exception is indicated and the execution of the method is finished. In the second, two differentiated parts are codified: the normal execution sequence, on the one hand, and the error treatment, on the other.

3.2. Exceptions

3.2.1. Throwing exceptions

When an error occurs and an exception is thrown, the exception is said to be thrown: *throw*. Any code is liable to throw an exception, be it the JVM framework, a library, or your own code. We have already commented that an exception will be an instance of some subclass of Throwable.

When the exception is thrown, the currently executing method stops prematurely and returns no value. Also, control does not return to the point from which the method was called: it is directed to the code that handles the exception or re-thrown to a previous method, which can handle the error. Of course, it's possible to *catch* it (have a code point for exception handling) and not handle it (handle the error); it would be appropriate, in any case, to provide some kind of message or notification.

But how do we indicate in our code that an exception has occurred?

1. Construct an exception object. An exception, as we have indicated, is a class; therefore, we will use the new operator.
2. The exception is thrown with the operator throw. Typically, both operations are performed in a single instruction.

```
throw new ExceptionType();
```

3. Document that the exception may occur and why. It is included in the Javadoc documentation for the method:

```
@throws ExceptionType causa_que_provoca_el_lanzamiento
```

An exception is an event that occurs during the execution of a program and interrupts the normal flow of instructions. For example, if you want to create an application that reads a file and sends it over the network, some of the possible exceptions are: the device where the file is stored is not available, it does not have read permissions on the file, the file does not exist, that the network is not available, etc.

3.2.2. Exception handling

It is important to manage the possible exceptions that may occur in the program to avoid any type of failure in its execution. In general, to include exception handling in a program, the following steps must be carried out:

1. Inside a try block insert the normal flow of statements.
2. Study the errors that can occur during the execution of the instructions and verify that each of them causes an exception.
3. Catch and handle exceptions in catch blocks.

When the programmer executes code that might cause an exception (for example: reading or writing a file), he must include this code snippet inside a try block:

```
try {
    // Possibly problematic code

}
```

But the important thing is how to control what to do with the possible exception that is thrown. To do this, the `catch` clauses are used to specify the action to be taken when a specific exception occurs.

```
try {
    // Possibly problematic code

} catch ( tipo_de_excepcion e) {
    // Code to solve the exception e
} catch ( tipo_de_excepcion_mas_general e) {
    // Code to solve the exception e

}
```

As you can see in the example, catch statements can be nested to handle different errors in a different way. It is convenient to do this by indicating the most general exceptions last (ie, those that are higher up in the exception inheritance tree), because the Java interpreter executes the catch code block whose parameter is the type of a thrown exception.

If you want to perform an action common to all exceptions, use the finally statement. This code is executed whether it is a catch or not.

```
try {
    // Código posiblemente problemático
} catch ( Exception e ) {
    // Código para solucionar la excepción e
} finally {
    // Se ejecuta tras try o catch
}
```

Example:

```
try {
    int[] a = new int[5];
    a[5] = 30 / 0;
} catch (ArithmeticException e) {
    System.out.println("Ha ocurrido un error aritmético");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Ha ocurrido un error de índice fuera de rango");
} finally {
    System.out.println("Este mensaje siempre lo veremos");
}
System.out.println("Esto se verá solo si el bloque completo sale sin excepción.");
```

3.2.3. Create our own exceptions

Although Java provides a large number of exceptions, there are times when you will need to create your own exceptions.

Proper exceptions must be **subclasses of the Exception** class.

Normally we will create our own exceptions when we want to handle exceptions not covered by the standard Java library. For example, let's create an exception type called InvalidValue that will be thrown when the value used in a certain operation is not correct.

The class could be the following:

```
public class ValorNoValido extends Exception{

    public ValorNoValido(){
    }

    public ValorNoValido(String cadena){
        super(cadena); //Llama al constructor de Exception y le pasa el contenido de cadena
    }
}
```

A program to test the exception created could be this:

```
public static void main(String[] args) {
    try {
        double x = leerValor();
        System.out.println("Raiz cuadrada de " + x + " = " + Math.sqrt(x));
    } catch (ValorNoValido e) {
```

```
6           System.out.println(e.getMessage());
7       }
8   }
9
10  public static double leerValor() throws ValorNoValido {
11      Scanner sc = new Scanner(System.in);
12      System.out.print("Introduce número > 0 ");
13      double n = sc.nextDouble();
14      if (n <= 0) {
15          throw new ValorNoValido("El número debe ser positivo");
16      }
17      return n;
18  }
```

3.2.4. Advice

- Do not use exceptions for control flow

  The technique attempts to clearly separate normal code execution from error handling.

- Exceptions are exceptional

  It refers to what happens infrequently, when some data is outside its domain, for example, and the error occurs.

- Do not throw or catch generic exceptions

  The more concretely we can determine the cause of the error, the more appropriate it will be to control it.

- Do not use empty catch clauses

  If it is not possible to fix it, it is better to propagate it.

- Use different try blocks for statements that throw the same exceptions

  This makes it possible to better define the origin of the error.

- Use standard exceptions

  The Java library defines exceptions for most errors that can occur

- Use exceptions when the constructor fails

  If the object cannot be created it is impossible to work with it

- Document exceptions

  Always add the pertinent information in the documentation of the method that launches it. And if it is a user exception, clearly explain its purpose.

3.3. Records or Logs (`Log4j2`)

Software with sufficient logging and monitoring will allow you to detect potential incidents when your code is deployed in a production environment. `Log4j` is used for logging. Logging is the process of writing log message in any file, db, console etc.

If we use SOP `System.out.print()` statement to print the log message then we can have some disadvantages:

1. We can print log message on console only. So when console is closed , we will lose all logs.

2. We can't store log message at any permanent place. These message will print one by one on console because it is single threaded environment.

To overcome these problems `Log4j` framework came into the picture. `Log4j` is an open source framework provided by Apache for only java projects.

To use Log4j2 in your framework, you just need to add the below libraries:

```
log4j-api-<version>.jar
log4j-core-<version>.jar
```

3.3.1. Automatic configuration

We can configure Log4j2 with our application using a configuration file written in XML, JSON, YAML, or properties format. We can do it programmatically as well but let's focus on configuring using configuration files as of now. Log4j has the ability to automatically configure itself during initialization. It has an order to look for the configuration file in the application. Log4j will provide a default configuration if it cannot locate a configuration file.

3.3.2. Log Levels

Log levels are a mechanism to categorise logs. Levels used for identifying the severity of an event. We can easily configure levels to specify which log details we want to see. Log4j provides below levels:

1. ALL – To log all events.
2. DEBUG – A general debugging event.
3. ERROR – An error in the application, possibly recoverable.
4. FATAL – A severe error that will prevent the application from continuing.
5. INFO – An event for informational purposes.
6. TRACE – A fine-grained debug message, typically capturing the flow through the application.
7. WARN – An event that might possibly lead to an error.
8. OFF – No events will be logged.

Log4j follows order as below:

ALL<TRACE < DEBUG < INFO < WARN < ERROR < FATAL

If we mention log level as INFO then all INFO, WARN, ERROR and FATAL events will be logged. If we mention log level as WARN then all WARN, ERROR and FATAL events will be logged. In simple terms, all the levels below the specified level including the specified level will be considered.

3.3.3. Appenders

We can specify destinations to keep event logs. We may want to print those logs in the console or any external file. Appenders usually are only responsible for writing the event data to the target destination. We may use multiple appenders.

3.3.4. Example code

```
package UD05.Log4j2;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class ModuleA {

    private static final Logger logger = LogManager.getLogger();

    // Log messages
    public static void main(String[] args) {
        logger.debug("It is a debug logger.");
        logger.error("It is an error logger.");
        logger.fatal("It is a fatal logger.");
        logger.info("It is a info logger.");
        logger.trace("It is a trace logger.");
        logger.warn("It is a warn logger.");
    }
}
```

Output will be:

```
11:14:47.469 [main] ERROR UD05.Log4j2.ModuleA - It is an error logger.
11:14:47.471 [main] FATAL UD05.Log4j2.ModuleA - It is a fatal logger.
```

We have used all the levels but in the console, we are seeing only two levels. Actually when we do not provide any configuration file , by default `Log4j` uses a default configuration.

The default configuration, provided in the `DefaultConfiguration` class, will set up:

- A `ConsoleAppender` attached to the root logger i.e. logs will be printed on the console.
- A `PatternLayout` set to the pattern: "`%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n`" attached to the `ConsoleAppender`

Note that by default `Log4j` assigns the root logger to `Level.ERROR` and those logs will be printed on the standard console.

Let's understand the pattern format in which logs are printed. Since we have not passed any configuration file, it uses the default format as shown above. Let's visualize output with the default pattern.

%d{HH:mm:ss.SSS} is execution timestamp i.e. `18:07:15.984`. **[%t]** is thread name i.e. `[main]`. **%-5level** is level name i.e. `ERROR`. **%logger{36}** is logger name which we are creating as first step i.e. `UD05.Log4j2.ModuleA`. **%msg%n** is message i.e. "`It is an error logger`" followed by a new line character.

3.3.5. Manual configuration

Log4j has the ability to automatically configure itself during initialization. When Log4j starts it will locate all the ConfigurationFactory plugins and arrange them in weighted order from highest to lowest. As delivered, Log4j contains four ConfigurationFactory implementations: one for JSON, one for YAML, one for properties, and one for XML.

1. Log4j will inspect the "log4j2.configurationFile" system property and, if set, will attempt to load the configuration using the ConfigurationFactory that matches the file extension. Note that this is not restricted to a location on the local file system and may contain a URL.
2. If no system property is set the properties ConfigurationFactory will look for log4j2-test.properties in the classpath.
3. If no such file is found the YAML ConfigurationFactory will look for log4j2-test.yaml or log4j2-test.yml in the classpath.
4. If no such file is found the JSON ConfigurationFactory will look for log4j2-test.json or log4j2-test.jsn in the classpath.
5. If no such file is found the XML ConfigurationFactory will look for log4j2-test.xml in the classpath.
6. If a test file cannot be located the properties ConfigurationFactory will look for log4j2.properties on the classpath.
7. If a properties file cannot be located the YAML ConfigurationFactory will look for log4j2.yaml or log4j2.yml on the classpath.
8. If a YAML file cannot be located the JSON ConfigurationFactory will look for log4j2.json or log4j2.jsn on the classpath.
9. If a JSON file cannot be located the XML ConfigurationFactory will try to locate log4j2.xml on the classpath.
10. If no configuration file could be located the DefaultConfiguration will be used. This will cause logging output to go to the console.

This is our log4j2.xml configuration file sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="trace">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

We know now that Log4j2 will look for a configuration file in the classpath of projects. For this we should keep a configuration file under the resource folder. We will make changes in the above XML file to print all levels of logs in the console. Under the "Loggers" tag we have another tag called "Root". Root Logger is the topmost element in every Logger Hierarchy. This "Root" tag

contains a property called "level". We have changed level to "trace" so that all levels will be printed.

If we add this file into our "src"/"resources" folder we will get this output:

```
11:29:06.786 [main] DEBUG UD05.Log4j2.ModuleA - It is a debug logger.
11:29:06.787 [main] ERROR UD05.Log4j2.ModuleA - It is an error logger.
11:29:06.787 [main] FATAL UD05.Log4j2.ModuleA - It is a fatal logger.
11:29:06.787 [main] INFO  UD05.Log4j2.ModuleA - It is a info logger.
11:29:06.787 [main] TRACE UD05.Log4j2.ModuleA - It is a trace logger.
11:29:06.787 [main] WARN  UD05.Log4j2.ModuleA - It is a warn logger.
```

We can also set configurationFile path with VM option:

```
-Dlog4j.configurationFille=src/UD05/Log4j2/log4j2.xml
```

We've followed this guide: http://makeseleniumeasy.com/2021/03/11/log4j2-tutorial-1-introduction-to-apache-log4j2/ and https://logging.apache.org/log4j/2.x/

3.4. Input validation.

A major pathway for security errors and data inconsistency within an application occurs through the data entered by users. A very frequent security flaw consists of errors based on buffer overflow, which occur when the size of a certain variable, vector, matrix, etc., overflows. and it is possible to access reserved memory areas. For example, if we allocate memory for the username which is 20 characters long and somehow the
If the user gets the application to store more data, a buffer overflow is taking place since the first 20 values are stored in a correct place, but the remaining values are stored in memory areas destined for other purposes.

Data validation allows:

- Maintain data consistency. For example, if we tell a user that he must enter her ID, it must always have the same format.
- Avoid buffer overflows. By checking the format and length of the field we prevent buffer overflows from occurring.

To carry out a rigorous control, on the input data of the users, it is necessary to take into account the validation of the format and the validation of the size of the input.

Java incorporates a powerful and useful library (`java.util.regex`) to use the `Pattern` class that allows us to define regular expressions. Regular expressions allow you to define exactly the format of the data input.

Example:

```
public class Validacion {

    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);

        Pattern pat = null;
        Matcher mat = null;

```

```
9           pat=Pattern.compile("[0-9]{8}-[a-zA-Z]");
10          System.out.print("Introduce un DNI con formato 00000000-X: ");
11          mat=pat.matcher(teclado.nextLine());
12
13          if (mat.find()){
14              System.out.println("El DNI cumple el formato");
15          } else {
16              System.out.println("El DNI NO cumple el formato");
17          }
18
19          //Otros patterns de ejemplo:
20          //pat=Pattern.compile("Almería","Granada","Jaén","Málaga",
21          //          "Sevilla","Cádiz","Córdoba","Huelva");
22          //pat=Pattern.compile("Verdadero","Falso","V","F",
23          //          "True","False","Córdoba","Huelva");
24
25      }
26  }
```

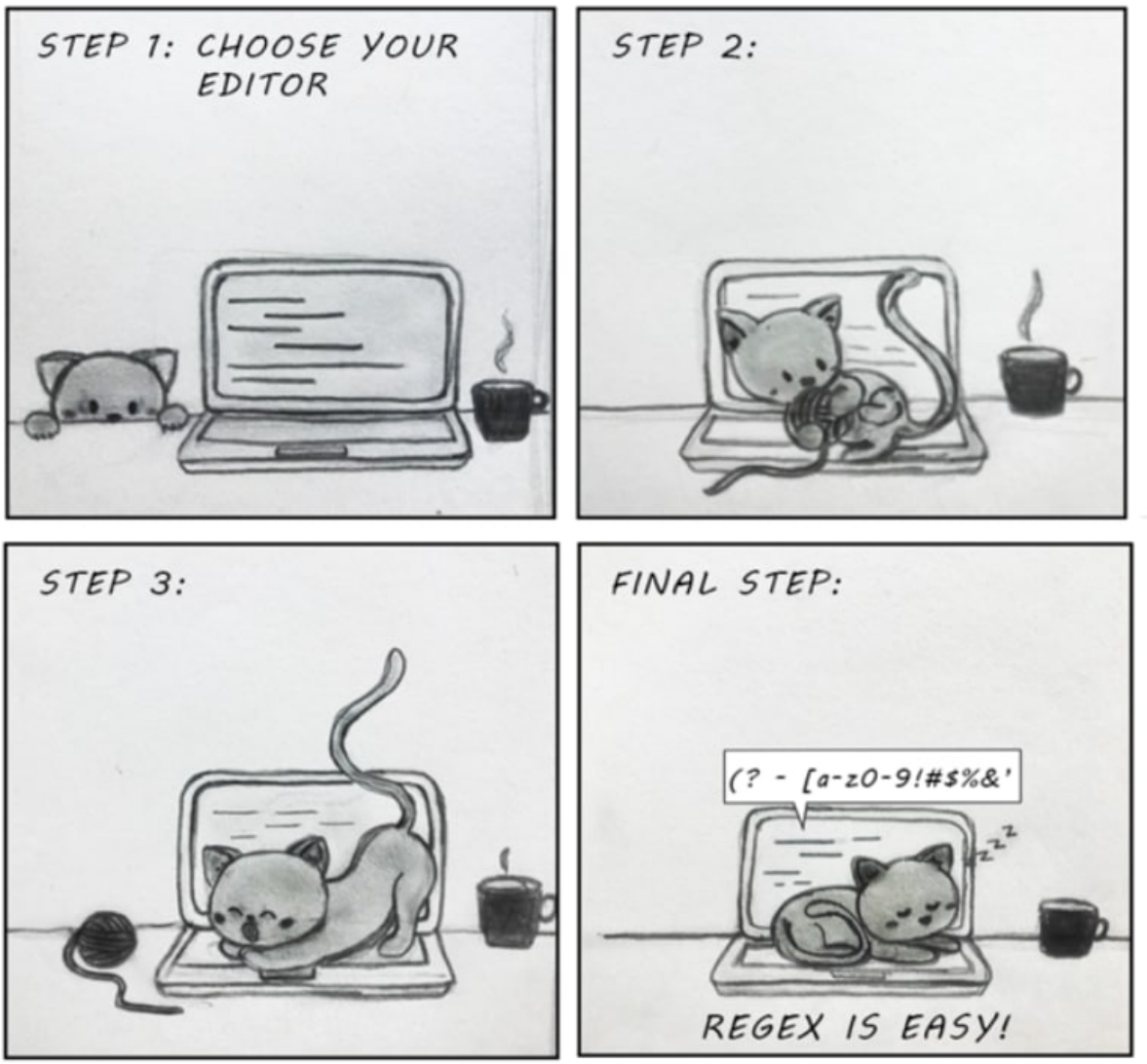

Figure 8.1 Letting your cat play over the keyboard is not the best solution for generating regular expressions (regex). To learn how to generate regexes you can use an online generator like **https://regexr.com/**.

3.5. Security politics.

System security is increased by allowing access policies to be established for both local and remote applications. Once the application is authorized, requests are sent to the Security Manager to gain access to system resources.

Security policies can be set using the following items:

- Origin (user or application path).
- Permissions (for example, you can indicate if you have read/write permissions on a file, if you can send information or not).
- Destination. Destination affected by the permit. For example, if we work with files, the destination is their location.
- Action. The actions that can be performed on the file. For example, in a file the permissions are read or write.

Note that **everything mentioned here doesn't work if the user has system access and can run the Java interpreter without restrictions**. That is, it takes the work of a system administrator to restrict the way in which the user executes the code.

Suppose then that we are in a secure environment where Java programs are executed using a security manager, that is, they are launched by executing `java -Djava.security.manager Class`. In principle, programs will not be able to do many things, such as connecting to the Internet or reading a file that is not in the same directory as the class.

3.6. Forbidden

Things not to do:

- Write code that uses relative filenames, full references must be used
- Refer twice in a program to the same file by its name
- Invoke unreliable and/or obsolete or unmaintained programs or services
- Assume that users are not malicious or cannot do bad things
- Do not perform exception handling and always assume success
- Invoke a shell or a command line from the application
- Authenticate with untrusted criteria
- Use storage areas with write permissions (be careful with this)
- Save sensitive data in a database without password or any encryption
- Echo passwords or show them in user views
- Issuing passwords or credentials by email (and forgetting the double factor, etc.)
- Distribute by program some confidential information by means of email
- Save passwords (or any other sensitive information) unencrypted
- Distribute unencrypted passwords between systems (or any sensitive information)
- Make access decisions based on variables or arguments at runtime
- Over-reliance on third-party software for critical operations

4. Information sources

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos