Universidad de las Ciencias Informáticas
Facultad 1

# Módulo de JAVA para la herramienta Auditoría de Código Fuente.

Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas.

**Autora:**
Yanet Cabeza Chávez

**Tutores:**
Ing. Abel Alfonso Fírvida Donéstevez

Ing. Michel Evaristo Febles Parker

La Habana, junio 2015

¨Año 57 de la Revolución¨

## Declaración de autoría

Declaro ser la autora del presente Trabajo de Diploma y se reconoce a la Universidad de las Ciencias Informáticas, los derechos patrimoniales del mismo con carácter exclusivo. Para que así conste firmo la presente declaración jurada de autoría en La Habana a los _____ días del mes de _____ del año _____.

Yanet Cabeza Chávez

_____

Firma del autor

Ing. Michel Evaristo Febles Parker

_____

Firma del tutor

Ing. Abel Alfonso Fírvida Donéstevez

_____

Firma del tutor

# Módulo de JAVA para ACF.

*¨Y si alguna de las cosas que decimos las explota el enemigo y nos producen profunda vergüenza, ¡¡bienvenida sea la vergüenza!!... ¡¡bienvenida sea la pena!!, si sabemos convertir la vergüenza en fuerza, si sabemos convertir la vergüenza en espíritu de trabajo, si sabemos convertir la vergüenza en dignidad, si sabemos convertir la vergüenza en moral.¨*

Comandante Fidel Castro Ruz

## Dedicatoria

A mi familia por todo su apoyo y comprensión en los momentos más difíciles durante estos cinco años, especialmente a mi mamá porque es lo más grande que tengo en esta vida, aunque se ponga celosa de mi papá, que comparte el primer con mi mamá. Los amo a todos.

## Agradecimientos


- A mis tutores por su ayuda incondicional en este período, por haber sido mis mejores guías y apoyo.
- A mis amigos y compañeros de aula por aguantarme todos estos años sin protestar y por ser siempre sinceros conmigo.
- A mis profesores estos cinco años por haberme enseñado lo que hoy pongo en práctica. El esfuerzo, dedicación, conocimiento y amor que depositaron en mi enseñanza son los mejores recuerdos que tengo de esta etapa tan importante de mi vida.
- Al tribunal, por su ayuda y orientación este último mes.
- A mi prima Jenny y su esposo Landrian, por haberme ayudado estos cinco año de la carrera y no dejarme flaquear cuando algo no salía bien, porque han sido los que me han llevado de la mano, cuando mis padres no podían estar aquí en la escuela para mi.
- A mis suegros Lisset y Jorgito por haberme adoptado como su hija estos años y por velar por mi bienestar y educación como si fueran mis verdaderos padres, que en mi corazón ya lo son.
- A mi novio, que aunque da unos buenos dolores de cabeza, ha sido mi compañero inseparable estos tres últimos años de la carrera, que con su ayuda y dedicación he llegado hasta donde estoy hoy.

# Módulo de JAVA para ACF.


## Resumen

En el Centro de *Software* Libre, se ha desarrollado la herramienta Auditoría de Código Fuente (ACF), la cual se encarga de la revisión de todos los ficheros de código fuente de las aplicaciones o bibliotecas que estén o se agreguen al repositorio del sistema operativo Nova, en busca de fallas de seguridad. Actualmente ACF no soporta la revisión de código JAVA, lo que podría conllevar a la inclusión de vulnerabilidades en el repositorio. La presente investigación propone la realización de un módulo para la herramienta, con el objetivo de que se puedan auditar los ficheros con extensión .java y pueda disminuir la inclusión de vulnerabilidades dentro del repositorio de Nova. Para ello se realiza un estudio del arte de las herramientas que realizan auditoría de código fuente. Se define como metodología de desarrollo OpenUp, la cual guiará el proceso de desarrollo de *software*. Para la implementación de la solución propuesta se define como Entorno de Desarrollo Integrado (IDE) Eclipse, como herramienta de modelado Visual Paradigm y como lenguajes de programación C++ y XML. Finalmente se obtuvo como resultado práctico de la investigación, una biblioteca dinámica para la detección y auditoría de ficheros de código fuente JAVA, la cual posibilita la reducción de vulnerabilidades en el repositorio y una mayor seguridad.



**Palabras clave:** auditoría, código fuente, repositorio, vulnerabilidad

## Índice de contenido

## Índice de ilustraciones

## Índice de tablas

# Módulo de JAVA para ACF.

## Introducción

En la actualidad ha aumentado considerablemente el uso de las Tecnologías de la Información y las Comunicaciones (TIC's) ya que constituyen un avance considerable en el constante desarrollo del hombre, pues amplía sus capacidades físicas y mentales, contribuyendo a la vez al desarrollo de la propia sociedad. Insertados dentro del ámbito de las TIC's, se encuentran los lenguajes de programación definidos como un idioma artificial diseñado para establecer una comunicación entre el hombre y la computadora. Según el reporte de 2014 de la *Web Technology Surveys* (*W3 Techs*), JAVA es el tercer lenguaje de programación más usado con un 2,8 % de uso en el desarrollo de sitios web conocidos, este emergió después de decaer su uso en años anteriores, como se muestra en la siguiente imagen (1):


*Ilustración 1. Uso de JAVA por sitios web, 20 de Febrero de 2015 (1).*

JAVA fue creado por *Sun Microsystems*[1]. El cual fue pensado originalmente para utilizarse en cualquier tipo

1 *Sun Microsystems*: una empresa informática que se dedicaba a vender estaciones de trabajo, servidores, componentes informáticos, *software* (sistemas operativos) y servicios informáticos. Fue adquirida en el año 2010 por Oracle Corporation.

de electrodomésticos pero la idea fracasó y fue retomada al surgir la Web. Uno de los primeros triunfos de este lenguaje fue que se integró en el navegador Netscape y permitía ejecutar programas dentro de una página web, hasta entonces impensable con el HTML[2] (2).

En la Universidad de las Ciencias Informáticas (UCI) el lenguaje JAVA es utilizado en centros productivos y proyectos, como es el caso del Centro de Identificación y Seguridad Digital (CISED), Auditoría de Bases de Datos (AuditBD), Gestión Hospitalaria, el Centro de Telemática (TLM), el Centro Especializado en Informática Médica (CESIM) y el Centro de *Software* Libre (CESOL). Este último posee como objetivos fundamentales:

1. Desarrollar la Distribución GNU/Linux Nova para la migración en Cuba.
2. Definir las directrices, lineamientos y soluciones que guiarán la migración[3] nacional.

Según la Resolución 179/14 emitida por la Rectora de la UCI el centro CESOL, como miembro del Comité de Migración creado por la Dirección de Informatización, que es la que conducirá el proceso de migración en la Universidad, es el encargado del desarrollo y actualización de la Distribución GNU/Linux Nova, así como las acciones de capacitación y soporte que sean requeridas por los usuarios para el manejo de las aplicaciones o del propio sistema.

Los centros productivos que sean migrados incluirán aplicaciones al repositorio[4] de Nova, por tanto todos los que utilicen JAVA como lenguaje de programación, algunos de los cuales se mencionaron con anterioridad, podrían incluir vulnerabilidades en el repositorio si no se auditan con antelación.

Nova, la distribución cubana de GNU/Linux, cuenta con la herramienta Auditoría de Código Fuente (ACF), la cual se utiliza para encontrar fallas de seguridad en el código fuente de cualquier aplicación o biblioteca que esté o se agregue a su repositorio; pero hasta ahora, ACF no soporta la revisión de código fuente JAVA

2 HTML: *Hypertext Transfer Protocol* en inglés y en español Protocolo de Transferencia de Hipertexto**,** este es un estándar que sirve de referencia para la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes y videos.

3 La migración de un sistema de información tiene por finalidad su traslado a un nuevo ambiente operativo, conservando su funcionalidad y datos originales.

4 Repositorio de GNU/Linux: un repositorio de GNU/Linux es una colección de paquetes de programas de una distribución de Linux específica que generalmente contiene archivos binarios pre-compilados que pueden ser descargados e instalados por los usuarios de la distribución correspondiente. Es posible también encontrar paquetes de código fuente.

lo que conllevaría a la inclusión de vulnerabilidades en el repositorio al no poder dicha herramienta auditar las aplicaciones desarrolladas en JAVA que se pretendan incorporar en dicho repositorio.

Partiendo de esta situación se plantea el siguiente **problema de la investigación:** ¿Cómo detectar vulnerabilidades en código fuente JAVA en el repositorio de Nova haciendo uso de la herramienta ACF?

Se trazó como **objeto de estudio** el proceso de detección de vulnerabilidades en código fuente.

Se define como **objetivo general:** Desarrollar un módulo que permita a la herramienta ACF la revisión de código fuente JAVA en el repositorio de Nova.

Para darle cumplimiento al objetivo general planteado, se han proyectado los siguientes **objetivos específicos:**

1. Analizar las herramientas de auditoría de código JAVA existentes, enfocándose en las más utilizadas actualmente.
2. Diseñar un módulo para la herramienta ACF que permita la detección de vulnerabilidades en el código fuente JAVA.
3. Implementar las funcionalidades del módulo para la herramienta ACF.
4. Realizar pruebas funcionales al módulo implementado para la herramienta ACF.

Se define como **campo de acción** el código fuente JAVA y como **idea a defender** que un módulo para la detección de vulnerabilidades en el código fuente JAVA disminuirá la inclusión de vulnerabilidades en el repositorio de Nova.

Para dar cumplimiento al objetivo se definieron las siguientes **tareas de investigación:**

- Revisión de bibliografía asociada a las herramientas de auditoría de código JAVA.
- Identificación de los requisitos funcionales y no funcionales para el módulo a desarrollar.
- Análisis de la arquitectura de ACF.
- Diseño del módulo acoplado a la arquitectura de ACF.
- Codificación de las funcionalidades definidas.
- Diseño de casos de prueba a las funcionalidades implementadas.

- Ejecución de los casos de prueba a las funcionalidades implementadas.

Los **resultados esperados** son que la herramienta ACF sea capaz de analizar y detectar vulnerabilidades en el código fuente JAVA.

Los **métodos científicos** utilizados en el proceso investigativo fueron el **análisis histórico-lógico,** el **inductivo-deductivo** y la **modelación**. El análisis histórico-lógico es empleado con el objetivo de verificar cómo evolucionan teóricamente las aplicaciones que realizan análisis de código fuente y de este modo poder realizar una selección de las técnicas, las herramientas y los algoritmos que se van a utilizar para desarrollar el módulo para la herramienta ACF. Por otra parte, el método inductivo-deductivo permite llegar a proposiciones generales a partir de los hechos que conforman la teoría, o a partir de dichas teorías arribar a conclusiones sobre casos particulares que se llevaron a la práctica. En la investigación es usado este método para, a partir del análisis de las herramientas existentes que realizan este proceso de detección de vulnerabilidades, seleccionar las características que va a poseer el módulo para la herramienta ACF de forma tal que realice el análisis de código fuente JAVA correctamente. El método modelación es utilizado para confeccionar los diagramas correspondientes a la representación de la propuesta de solución.

Los métodos empíricos utilizados en el proceso investigativo fueron el **análisis documental** y la **tormenta de ideas**. El análisis documental es empleado para la revisión de la literatura necesaria durante la investigación sobre el proceso de detección de vulnerabilidades en el lenguaje JAVA. El método tormenta de ideas es utilizado para el levantamiento de requisitos, así como para la determinación de las restricciones de *software* e implementación que deben tenerse en cuenta en el proceso de desarrollo.

El presente documento se encuentra estructurado de la forma siguiente:

**Capítulo 1. Fundamentación teórica.** En este capítulo se define el concepto de herramientas Lint, se realiza un profundo análisis de las principales características de cada herramienta encontrada. Se define el concepto de vulnerabilidad, así como su clasificación. También se realiza un análisis de las principales vulnerabilidades que presenta el lenguaje JAVA, dentro de las que se destacan, denegación de servicio, obtención de información, ejecución de código arbitrario y obtención de permisos. Además se realiza un

estudio de las herramientas y tecnologías que van ser utilizadas para el desarrollo de la solución y sus principales características, así como la justificación del por qué fueron escogidas. Se selecciona la metodología de desarrollo que va a guiar todo el proceso de construcción de un módulo para la herramienta ACF que detecte vulnerabilidades en los ficheros de código fuente JAVA.

**Capítulo 2. Análisis y Diseño del Módulo de JAVA para ACF.** Se explica detalladamente la propuesta de solución mediante una imagen que muestra las relaciones entre los principales conceptos del sistema. Se realiza la especificación de requisitos funcionales y no funcionales, así como el diagrama de casos de uso y su descripción. Se explican los patrones de diseño y arquitectura que se utilizaron y el momento en que fueron empleados. Se presenta el diagrama de clases del diseño que describe la estructura de un sistema mostrando sus clases y sus relaciones con las demás. Se expone el diagrama de secuencia que muestra gráficamente las interacciones del actor y de las operaciones que dan origen.

**Capítulo 3. Implementación y Prueba.** Comprende la explicación de los estándares de codificación utilizados en el desarrollo de la solución propuesta, así como el diagrama de componentes que apoya el proceso de implementación. Se explican los distintos tipos de pruebas que se van a utilizar para comprobar el correcto funcionamiento de la solución. Se muestran los resultados de las pruebas y el impacto social de la investigación.

# Capítulo 1. Fundamentación Teórica

**Introducción**

En el presente capítulo se definen los principales conceptos asociados al tema investigativo dígase el concepto de vulnerabilidad, sus clasificaciones y las vulnerabilidades presentes en el lenguaje JAVA. También se realiza un análisis de las herramientas que existen y que se utilizan en la detección de vulnerabilidades en el código fuente JAVA. Se realizan las clasificaciones de las vulnerabilidades que se desean detectar a través del estándar *Common Vulnerability Scoring System* (CVSS). También se lleva a cabo un estudio que incluye las tecnologías actuales para el desarrollo de *software* y las principales herramientas que serán utilizadas para la implementación de la solución.

## 1.1 Definiciones asociadas al tema investigativo

### 1.1.1 Concepto de Vulnerabilidad

Las vulnerabilidades son puntos débiles del *software* que permiten que un atacante comprometa la integridad, disponibilidad o confidencialidad del mismo. Algunas de las vulnerabilidades más severas permiten que los atacantes ejecuten código arbitrario, denominadas vulnerabilidades de seguridad, en un sistema comprometido (3). Las vulnerabilidades de los Sistemas Informáticos (SI) se pueden agrupar en función de:

**Diseño**

- Debilidad en el diseño de protocolos utilizados en las redes.
- Políticas de seguridad, deficientes e inexistentes.

**Implementación**

- Errores de programación.
- Existencia de “puertas traseras” en los sistemas informáticos.
- Descuido de los fabricantes.

**Uso**

- Configuración inadecuada de los sistemas informáticos.
- Desconocimiento y falta de sensibilización de los usuarios y de los responsables de informática.
- Disponibilidad de herramientas que facilitan los ataques.
- Limitación gubernamental de tecnologías de seguridad.

### 1.1.2 Clasificación de las vulnerabilidades

**Vulnerabilidades de condición de carrera**

La condición de carrera se da principalmente cuando varios procesos acceden al mismo tiempo a un recurso compartido, por ejemplo una variable, cambiando su estado y obteniendo de esta forma un valor no esperado de la misma (4).

**Vulnerabilidades de error de formato de cadena**

La principal causa de los errores de cadena de formato es aceptar sin validar la entrada de datos proporcionada por el usuario. Es un error de programación y el lenguaje más afectado es C/C++. Un ataque puede conducir de manera inmediata a la ejecución de código arbitrario y a revelación de información (5).

**Vulnerabilidades de Inyección SQL**[5]

Una inyección SQL se produce cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos (6).

**Vulnerabilidades de denegación del servicio**

La denegación de servicio provoca que un servicio o recurso sea inaccesible a los usuarios legítimos. Normalmente provoca la pérdida de la conectividad de la red por el consumo del ancho de banda de la víctima o sobrecarga de los recursos informáticos del sistema (7). A continuación se presenta una tabla con seis ejemplos de vulnerabilidades de JAVA que provocan denegación de servicios.

| Identificación | Lugar donde radica | Clasificación | Denegación de Servicio | Forma de |
|---|---|---|---|---|

5 SQL: el lenguaje de consulta estructurado o SQL (por sus siglas en inglés *Structured Query Language*) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en ellas.

| | | | | **Explotación** |
|---|---|---|---|---|
| CVE-2012-1719 | CORBA (*Common Object Request Broker Architecture*) | Moderada | Denegación parcial del servicio. | Forma local |
| CVE-2012-1713 | Administrador de fuentes. | Crítica | Denegación de servicio parcial en la máquina virtual de JAVA. | Explotable remotamente |
| CVE-2012-1716 | Componente *Swing* | Crítica | Denegación parcial del servicio o saltar las restricciones de la *sandbox*[6] de JAVA. | Explotable remotamente |
| CVE-2012-1718 | Lista de Certificados Revocados (CRL, *Certificate Revocation Lists*) | Moderada | Denegación de servicio parcial. | Forma local |
| CVE-2012-1723 y CVE-2012-1725 | JAVA *HotSpot Virtual Machine* | Crítica | Una denegación de servicio en la máquina virtual de JAVA, saltar las restricciones de su *sandbox* o potencialmente ejecutar código arbitrario a través de una aplicación JAVA especialmente manipulada. | Explotable remotamente |
| CVE-2012-1726 | Función java.lang.invoke.MethodHandles.Lookup | Importante | No concede correctamente los modos de acceso. Lo que podría permitir a una aplicación JAVA no verificada saltar las restricciones de seguridad impuestas por la *sandbox*. | Explotable Remotamente |

*Tabla 1. Ejemplo de vulnerabilidades de JAVA que conllevan a una denegación de servicios (7).*

6 *Sandbox*: el aislamiento de procesos (del inglés *sandbox*) es un mecanismo para ejecutar programas con seguridad y de manera separada. A menudo se utiliza para ejecutar código nuevo, o *software* de dudosa confiabilidad proveniente de terceros.

**Vulnerabilidades de ventanas engañosas (*Windows Spoofing*).**

Las ventanas engañosas son aquellas que dicen que eres el ganador de tal o cual evento, lo cual es mentira y lo único que quieren es que des información (8). Hay otro tipo de ventanas que, si las sigues, obtienen datos del ordenador para luego realizar un ataque.

**Exploit *0-day***

Esta vulnerabilidad afecta a todos aquellos usuarios que posean la versión 7 de JAVA. Ya existe en circulación un *malware*[7] que aprovecha esta vulnerabilidad y permite infectar a aquellos usuarios que ingresen a un sitio web con un *applet*[8] malicioso (9).

Desde el laboratorio de ESET Latinoamérica[9] se realizó un análisis sobre el mencionado *exploit* y a partir de este estudio es importante destacar algunos aspectos para entender su funcionamiento. Los *applets* generalmente se ejecutan en el entorno de un sitio web y proveen funcionalidades diferentes, agregando en muchos casos, dinamismo al sitio web. El código de un *applet* en un sitio web es como el siguiente:

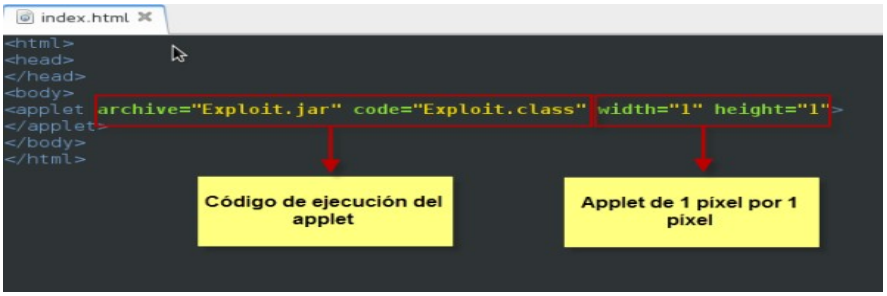


*Ilustración 2. Ejemplo de código de un applet.*

Como se observa en la imagen anterior, el código HTML del sitio web carga un archivo *jar*[10], que en el ejemplo responde al nombre de Exploit.jar, el cual se ejecuta en el entorno del *applet,* aquí es donde yace la vulnerabilidad. JAVA permite la ejecución de código del tipo *applet* pero lo hace con permisos

7 Malware: el término *malware* es muy utilizado por profesionales de la informática para referirse a una variedad de *software* hostil, intrusivo o molesto.

8 *Applet*: los *applets* son programas hechos en JAVA que se coloca en un servidor web y se asocia con una etiqueta <*applets*> en una página HTML.

9 ESET: es una compañía de seguridad informática establecida en Bratislava, Eslovaquia.

10 Un archivo jar (por sus siglas en inglés, ***J****ava* ***Ar****chive*): es un tipo de archivo que permite ejecutar aplicaciones escritas en el lenguaje JAVA.

restringidos. Esta vulnerabilidad, en términos generales, permite elevar el nivel de permisos y ejecutar código arbitrario sobre el sistema de la víctima. De esa forma, si el usuario ingresa a un sitio web que contiene un *applet* malicioso, su sistema puede ser comprometido. Vale aclarar que la carga del *applet* es totalmente legítima, es el propio archivo *jar* el que manipula la máquina virtual de JAVA (JVM) y salta las protecciones correspondientes para ejecutar el código arbitrario (10).

**JAVA/Exploit.Bytverify. Vulnerabilidad en JAVA**

Características generales del Exploit.Bytverify:

**Nombre:** JAVA/Exploit.Bytverify

**Nombre Nod32:** JAVA/Exploit.Bytverify

**Tipo:** Caballo de Troya[11]

**Alias:** Trojan.ByteVerify, Exploit-ByteVerify, Exploit.Java.Bytverify, JAVA_BYTVERIFY.A

**Fecha de actualización:** 2/septiembre/2004

**Plataforma:** Windows 32-bit

**Tamaño:** variable

Se trata de un caballo de Troya que explota una vulnerabilidad descrita por Microsoft en el boletín MS03-011 (abril de 2003) y que permite a un sitio malicioso ejecutar código en los equipos vulnerables. Un fallo en las versiones 3809 y anteriores de la JVM de Microsoft, permite, por un error en el componente ByteCode *Verifier*, que sitios o usuarios maliciosos, lo utilicen para ejecutar código en forma arbitraria en la máquina de la víctima, saltándose las restricciones impuestas a todo código JAVA (11).

El código está incluido en un *applet* de JAVA, o puede ser descargado por un troyano, de modo que la computadora del usuario es afectada cuando ello ocurre. Esto también puede explotarse a través de la apertura de un correo electrónico con formato HTML.

Cuando un *applet* malicioso se ejecuta, puede realizar las siguientes acciones:

1. Modificar los permisos de ejecución para su propio código.
2. Abrir determinadas páginas web para obtener datos de configuración, con los que puede modificar

11 Caballo de Troya: en informática, se denomina 'caballo de Troya' a un virus informático que se presenta al usuario como un programa aparentemente legítimo e inofensivo, pero que, al ejecutarlo, le brinda a un atacante acceso remoto al equipo infectado.

la página de inicio y las opciones de búsqueda del Internet Explorer, además del archivo HOST[12] de Windows, para redireccionar determinados sitios a otros proporcionados por el atacante.

3. Agregar enlaces a la lista de favoritos, generalmente relacionados con sitios indebidos.
4. Crear archivos HTML o scripts[13] locales, que permiten ejecutar otras acciones, incluso la descarga de otros códigos malignos.

La notificación de una infección con este troyano, no significa necesariamente que la máquina ya esté infectada, sino que el código capaz de hacer todo lo indicado antes está incluido en algún *applet* descargado (11).

Los *applets* suelen distribuirse en archivos comprimidos, los cuáles son abiertos en tiempo de ejecución. La eliminación del código significa borrar todo el archivo comprimido.

### 1.2 Análisis de las principales vulnerabilidades del lenguaje JAVA

Las vulnerabilidades en JAVA no son un tema nuevo puesto que han estado siempre presente pero con dos diferencias importantes respecto a las actuales: la plataforma recién comenzaba a popularizarse y *Sun Microsystems* tenía una capacidad de respuesta muy buena para corregir las vulnerabilidades reportadas, sin embargo es de esperarse que al volverse famoso sería blanco de inherentes ataques por lo tanto la motivación por encontrar nuevas vulnerabilidades aumenta.

Cada año se detectan nuevas vulnerabilidades, pero coincidentemente desde el año 2010 en que Oracle[14] compró a *Sun Microsystems* todo cambió. El número de vulnerabilidades en JAVA ha incrementado su búsqueda no sólo por parte de los investigadores sino también de aquellos que buscan una oportunidad con fines de lucro, utilizando una peligrosa combinación: Internet y la capacidad de JAVA de ejecutarse en

12 El término *HOST* ("anfitrión", en español): es usado en informática para referirse a las computadoras conectadas a una red, que proveen y utilizan servicios de ella.

13 En informática un script, archivo de órdenes, archivo de procesamiento por lotes o guión**:** es un programa usualmente simple, que por lo regular se almacena en un archivo de texto plano. El uso habitual de los *scripts* es realizar diversas tareas como combinar componentes, interactuar con el sistema operativo o con el usuario. Por este uso es frecuente que los *shells* sean a la vez intérpretes de este tipo de programas.

14 Oracle Database: es un sistema de gestión de base de datos objeto-relacional (u ORDBMS por el acrónimo en inglés *de Object-Relational Data Base Management System*), desarrollado por *Oracle Corporation*.

un navegador (12).

Hay quienes se han dado la tarea de informar cada vez que ocurre una nueva vulnerabilidad de día 0 (*zero-day*) o los detalles de las vulnerabilidades encontradas. A continuación se muestra en una gráfica las cantidades de vulnerabilidades que se han encontrado en la plataforma JAVA desde el 2001 hasta el 2013, donde se evidencia el aumento considerable en los últimos años, así como una segunda gráfica donde se puede observar las cantidades de vulnerabilidades reportadas por tipo en donde existen cinco tipos con cantidades mayores a quince, lo que demuestra la creciente incertidumbre a la que se enfrenta el lenguaje JAVA en la actualidad.

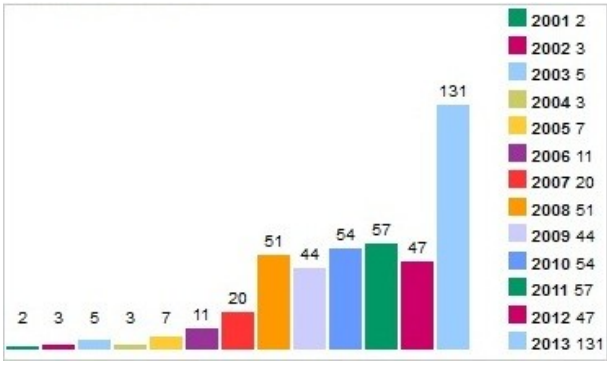


*Ilustración 3. Cantidad de vulnerabilidades de JAVA reportadas en los últimos años (13).*

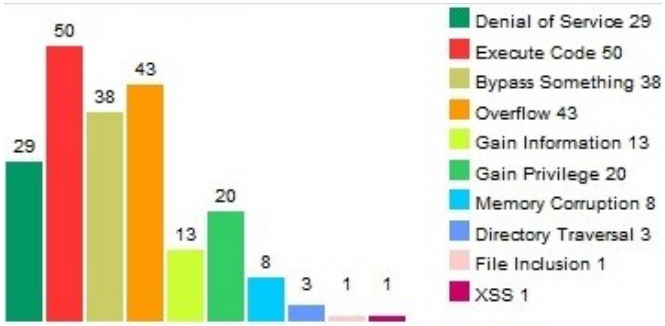


*Ilustración 4. Cantidad de vulnerabilidades de JAVA por tipo reportadas (14).*

### 1.2.1 *Applets* de JAVA

Cuando un usuario entra a una página con un *applet* asociado, el *applet* se descarga junto con la página y una vez descargado, en el navegador del usuario se ejecuta siempre y cuando el navegador tenga la configuración necesaria para permitirlo (15). Es así donde el *applet*, si fue programado para explotar una vulnerabilidad, se ejecuta y arruina la navegación en Internet.

En condiciones normales un *applet* sigue un estricto control de seguridad, pues no permite la ejecución de código nativo[15] en la computadora del usuario, ni puede conectarse a un servidor diferente a aquel de donde fue descargado. El problema radica cuando un programador de *applet* descubre la forma de burlar esos controles.

El navegador que carga y ejecuta el *applet* se conoce en términos genéricos como el "contenedor" de los *applets*. El kit de desarrollo de *software* para JAVA *Standard Edition* 7 (1.7.1 versión más actual, puesta en marcha el 18 de octubre de 2011) incluye un contenedor de *applets*, llamado *appletviewer*, para probar los *applets* antes de incrustarlos en una página web (16).

**Ventajas de los *applets***

- Son multiplataforma (funcionan en Linux, Windows, OS X[16] y en cualquier sistema operativo para el cual exista una JVM).
- Es compatible con la mayoría de los navegadores web.
- Puede ser almacenado en la memoria caché de la mayoría de los navegadores web, de modo que se cargará rápidamente cuando se vuelva a cargar la página web, aunque puede quedar atascado en la caché, causando problemas cuando se publican nuevas versiones.
- Puede tener acceso completo a la máquina en la que se está ejecutando, si el usuario lo permite.
- Puede ejecutarse a velocidades comparables a las de otros lenguajes compilados, como C++ (dependiendo de la versión de la JVM).

**Desventajas de los *applets***

15 Código nativo: se usa como seudónimo de lenguaje de máquina. Este puede ser creado directamente para microcontroladores extremadamente sencillos o código fuente ya compilado, que puede ser interpretado por la máquina.

16 OS X, antes llamado Mac OS X, es un entorno operativo basado en Unix, desarrollado, comercializado y vendido por Apple Inc.

- Requiere el *plug-in*[17] de JAVA, que no está disponible por omisión en todos los navegadores web.
- No puede iniciar la ejecución hasta que la JVM esté en funcionamiento y esto puede tomar tiempo la primera vez que se ejecuta un *applet*.
- Si no está firmado como confiable, tiene un acceso limitado al sistema del usuario - en particular no tiene acceso directo al disco duro del cliente o al portapapeles.
- Puede tener vulnerabilidades que permitan ejecutar código malicioso.

### 1.2.2 Vulnerabilidades de JAVA.

A continuación se presenta un listado de vulnerabilidades encontradas en sitios web atacados en los últimos 6 meses del 2014 y así poder determinar cuáles son las vulnerabilidades más explotadas por los ciberdelincuentes[18].

| Vulnerabilidad | Tecnología |
|---|---|
| CVE-2013-2423 | JAVA |
| CVE-2012-1723 | JAVA |
| CVE-2012-0159 | Win32 |
| CVE-2012-0507 | JAVA |
| CVE-2011-3402 | Adobe |
| CVE-2013-0422 | JAVA |
| CVE-2012-0158 | Active X |
| CVE-2012-4681 | JAVA |
| CVE-2012-0003 | Win32 |
| CVE-2007-0071 | Adobe |

*Tabla 2. Listado de vulnerabilidades más explotadas y la tecnología donde se encuentran (17).*

Al analizar esta tabla sorprende que dentro de las 10 vulnerabilidades más explotadas se encuentra una que fue reportada en el año 2007 (CVE-2007-0071). Esta permite que sobre la máquina del usuario se

17 Plug-in: es un módulo de hardware o *software* que añade una característica o un servicio específico a un sistema más grande.

18 Ciberdelincuentes: son personas que realizan actividades delictivas en Internet como robar información, acceder a redes privadas, estafas y todo lo que tiene que ver con los delitos e ilegalidad.

ejecuten códigos maliciosos a través de la ejecución de un archivo tipo SWF[19] diseñado para generar un desbordamiento de *buffer* (17).

La vulnerabilidad CVE-2013-2423, es la más reciente de las vulnerabilidades descubiertas por lo cual tiene sentido que sea la más explotada. Aprovecha una vulnerabilidad en el JAVA *Runtime Environment* (JRE) asociado a Oracle JAVA SE 7 *Update* 17 y versiones anteriores (18). La actualización de seguridad de esta vulnerabilidad fue lanzada por Oracle en abril del 2013.

Otra de las vulnerabilidades más utilizadas y que también está asociada con JAVA es la CVE-2012-1723, que afecta el JRE que está asociado con la versión Oracle JAVA SE 7 *Update* 4 y las versiones anteriores (19). Particularmente esta vulnerabilidad puede afectar la confidencialidad, integridad y disponibilidad de los datos que se encuentren en la máquina afectada. Para esta vulnerabilidad está disponible la actualización de seguridad desde junio del año 2014.

En el caso de la vulnerabilidad CVE-2012-0159 está asociada a Sistemas Operativos Microsoft Windows XP SP2 y SP3, Windows Vista SP2 y algunas versiones de Windows Server y del paquete Office 2003, 2007 y 2010 (20). Para esta vulnerabilidad Microsoft en sus boletines de seguridad de mayo y junio del 2012 puso a disposición de los usuarios las actualizaciones necesarias para corregirla.

La vulnerabilidad CVE-2012-0507 también afecta Oracle JAVA SE 7 pero esta vez solamente a las asociadas a la actualización 2 o anteriores. Una característica de esta vulnerabilidad es que permite a los atacantes evitar las restricciones del *sandbox* de JAVA, además de colapsar la JVM (21). La actualización de seguridad está disponible desde febrero del 2012.

Dentro de las primeras 5 vulnerabilidades, se destaca la CVE-2011-3402, que está relacionada con el producto Adobe Photoshop y un *plug-in* para GIMP[20]. Esta vulnerabilidad puede ser aprovechada por un atacante para ejecutar código malicioso en la computadora de la víctima utilizando un archivo PSD[21] alterado (22).

---

19 Archivo SWF: es un formato de archivo de gráficos vectoriales creado por la empresa Macromedia (actualmente *Adobe Systems*).

20 GIMP (**G**NU ***I**mage **M**anipulation **P**rogram*): es un programa de edición de imágenes de fuente libre, licenciado bajo GNU GPL. Puede ser usado para editar imágenes de mapa de bits, como fotografías.

21 PSD, PDD: formato estándar de Photoshop con soporte de capas.

Las restantes cinco vulnerabilidades tienen comportamientos similares, que permiten a un atacante ejecutar código arbitrario en la máquina de la víctima, lo cual lo hace susceptible de ser infectado con algún tipo de código malicioso. Es importante destacar que para estas diez vulnerabilidades, actualmente existen actualizaciones de seguridad que las corrigen (23). Por lo cual es importante hacer énfasis en la importancia de mantener actualizadas las aplicaciones que se utilizan en la computadora, como medida de protección adicional al hecho de tener una solución de seguridad instalada.

### 1.3 Herramientas Lint

Originalmente **Lint** era el nombre de una herramienta de programación utilizada para detectar código sospechoso, confuso o incompatible entre distintas arquitecturas en programas escritos en C, es decir, errores de programación que escapan al habitual análisis sintáctico que hace el compilador (24). En la actualidad, se utiliza este término para designar a herramientas que realizan estas tareas de comprobación en cualquier lenguaje de programación. Las herramientas de tipo Lint generalmente funcionan realizando un análisis estático del código fuente.

Las construcciones sospechosas que se suelen buscar son usos de variables antes de ser inicializadas o creadas, condiciones que no varían bajo ninguna circunstancia (son siempre verdaderas o siempre falsas) y cálculos cuyos resultados probablemente caigan fuera del rango permitido por las variables utilizadas.

También se está empezando a incluir la detección de estas construcciones incorrectas en su lista de avisos (*Warnings*).

Las herramientas más avanzadas realizan cada vez más comprobaciones, como por ejemplo, que el código sea consistente entre distintos compiladores, o el soporte para incorporar anotaciones acerca del comportamiento esperado o las propiedades del código.

La primera versión de Lint (fuera de los laboratorios *Bell Labs*) apareció en la versión 7 (V7) del sistema operativo UNIX, en 1979. Formaba parte del compilador PCC (*Portable C Compiler*), que fue el segundo compilador de C incorporado a las máquinas PDP-11[22]. Su autor, Stephen C. Johnson, es también el

22 PDP-11: fue una computadora fabricada por la empresa *Digital Equipment Corp*. en las décadas de 1970 y 1980. Fue la primera minicomputadora en interconectar todos los elementos del sistema.

creador de yacc (*Yet another compiler compiler*) y del previamente mencionado PCC (25).

### 1.3.1 Análisis de las principales herramientas Lint

En este apartado se analizan las herramientas para el análisis de código fuente más extendidas. Haciendo una descripción inicial de su disponibilidad y los tipos de errores que detectan.

A continuación se muestra una tabla donde se realiza una comparación entre herramientas Lint que detectan errores en código JAVA, se tiene en cuenta como criterios de comparación sus características fundamentales y la plataforma en la que se ejecutan.

| **Nombre** | **Plataforma** | **Características** | **Detección de vulnerabilidades de tipo DS[23], ECA[24] o Proceso L/E[25].** |
|---|---|---|---|
| Ant | Multiplataforma | Motor de ejecución de scripts similar a "*make*" o "*gnu make*". Sus ficheros de script son ficheros XML[26] donde se definen los objetivos y el orden de los mismos. Se pueden establecer propiedades dentro del fichero de construcción, pasárselas como parámetros mediante la línea de comandos y usar las propiedades de sistema (26). | No |
| Jlint | Multiplataforma | Detecta los errores analizando ficheros compilados (.*class*) de JAVA. Puede detectar errores de sincronización de | No |

23 DS: denegación de servicio.

24 ECA: ejecución de código arbitrario.

25 Proceso L/E: procesos de lectura y escritura sobre recursos críticos del sistema.

26 XML, siglas en inglés de *e**X**tensible **M**arkup **L**anguage* ('lenguaje de marcas extensible'), es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.

| | | | |
|---|---|---|---|
| | | hilos[27], en la jerarquía de herencia y de valores o rangos de variables locales o campos (27). | |
| AntiC | Linux | Los errores que puede detectar son de prioridad de operadores, en el cuerpo de las declaraciones y los relacionados con los *tokens*[28] (27). | No |
| FindBugs | Linux | Se centra en el análisis de ficheros *.class*. Para ello hace uso de la librería BCEL[29], que sirve para manipular este tipo de código. Mediante dicho análisis se pretende encontrar patrones de error en las clases (28). | No |
| PMD | Linux | Intenta indagar posibles errores mediante la comprobación de reglas en los ficheros (29). | No |
| JTest | Linux | Incluye tecnología para la detección de errores de generación Unit, de análisis estático, de prueba de regresión. | No |

*Tabla 3. Comparación de las herramientas que realizan análisis de código JAVA.*

### 1.3.2 Características comunes de las herramientas presentadas

Las herramientas que se han presentado anteriormente, corresponden a las alternativas más utilizadas para analizar código fuente, asegurando gran efectividad de detección ante vulnerabilidades tales como:

27 Sincronización de hilos: la sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.

28 *Tokens*: es una cadena de caracteres que tiene un significado coherente en cierto lenguaje de programación.

29 Librería BCEL: *Byte Code Engineering Library*, es una librería para analizar, manipular y crear archivos Java *class*. En nuestro contexto es una ayuda para parsear los archivos *.class* y obtener toda la información que necesitamos para la ejecución del bytecode en nuestra máquina virtual de JAVA.

desbordamientos de *buffer*, condiciones de carrera y problemas asociados a funciones de impresión con formato, reconocen patrones los cuales son posibles errores que pueden aparecer en tiempo de ejecución, código que no se puede ejecutar nunca porque no hay manera de llegar a él, código que puede ser optimizado, expresiones lógicas que puedan ser simplificadas y malos usos del lenguaje.

Ninguna de estas herramientas realizan búsquedas de vulnerabilidades asociadas a denegación de servicio, ni evitan la ejecución de código arbitrario. Tampoco verifican los procesos de lectura y escritura sobre recursos críticos del sistema, tales como los ficheros donde son almacenados los usuarios del sistema y sus respectivas contraseñas, ya que al poder leer o escribir en estos ficheros pueden ser creados usuarios ilegítimos en el sistema o alterar los permisos de los ya existentes. Por ello, el uso de las mismas, aunque sea efectivo en otras áreas no evitará enfrentarse a estas vulnerabilidades sin ser detectadas. Por lo que no se escoge ninguna de ellas como propuesta de solución, a pesar de esto el estudio de las mismas permitió un mejor entendimiento de las prácticas utilizadas en el proceso de detección de vulnerabilidades.

### 1.4 Métricas y clasificaciones de las vulnerabilidades a detectar

CVSS es una iniciativa pública concebida por la *National Infraestructure Assurance Council* (NIAC) de Estados Unidos (EE.UU), un grupo que provee de recomendaciones al *Department of Homeland Security* de los EE.UU. Entre las organizaciones que lo adoptaron tempranamente destacan Cisco, *US National Institute of Standards and Technology* (NIST), *Qualys* y Oracle. En la actualidad, el CVSS está bajo la custodia del *Forum for International Response Teams* (FIRST) (49).

Entre los beneficios que ofrece el CVSS están:

- **Puntuación estándar de vulnerabilidades:** el CVSS es neutro desde el punto de vista de las aplicaciones, permitiendo que distintas organizaciones asignen una puntuación a sus vulnerabilidades a través de un único esquema.
- **Puntuación contextualizada:** la puntuación asignada por una organización corresponde al riesgo que la vulnerabilidad representa para dicha organización.

- **Sistema abierto:** el CVSS provee todos los detalles sobre los parámetros usados en la generación de cada puntuación permitiendo comprender tanto el razonamiento que sustenta una puntuación como el significado de diferencias entre puntuaciones.

Las puntuaciones asignadas por el CVSS se derivan de los tres grupos de métricas siguientes:

- **Base:** este grupo representa las propiedades de una vulnerabilidad que son inmutables en el tiempo, específicamente: complejidad de acceso, vector de acceso y grado en que compromete la confidencialidad, integridad y disponibilidad del sistema.
- **Temporal**: este grupo mide las propiedades de una vulnerabilidad que sí cambian en el tiempo, como por ejemplo la existencia de parches o código para su explotación.
- **Medio ambientales**: este grupo mide las propiedades de una vulnerabilidad que son representativas de los ambientes de uso de las TIC como por ejemplo la prevalencia de sistemas afectados y pérdidas potenciales.

CVSS usa fórmulas sencillas y a partir de los grupos de métrica arriba enumerados arroja la puntuación final asociada a la vulnerabilidad.

De las métricas de base se deriva una puntuación de 0.0 a 10.0 basado en las respuestas a las siguientes preguntas:

**Métrica de base:**

1. **Métricas de Explotabilidad**

- **Vector de acceso**

  Local

  Remoto

- **Complejidad de ataque**

  Baja

  Alta

- **Nivel de autenticación requerida**

  No requerida

Requerida

2. **Métricas de Impacto**

- **Impacto en la confidencialidad:**

  Ninguna

  Parcial

  Completa

- **Impacto en la integridad:**

  Ninguna

  Parcial

  Completa

- **Impacto en la disponibilidad:**

  Ninguna

  Parcial

  Completa

- **Mayor impacto en:**

  Confidencialidad

  Integridad

  Disponibilidad

Las métricas temporales modifican la puntuación base, reduciéndola hasta en un tercio dependiendo de las respuestas a las siguientes preguntas:

**Métricas Temporales:**

- **Disponibilidad del exploit**

  No probada

  Prototipo de ataque

  Existencia de exploit

  Alta

- **Tipo de solución disponible**

  Solución oficial

  Solución temporal

  Solución de contingencia

  No disponible

Finalmente, las métricas medio ambientales modifican la puntuación obtenida y generan un valor final dependiendo de las respuestas a las siguientes preguntas:

**Métricas medio ambientales:**

- **Daño colateral potencial**

  Ninguno

  Bajo

  Medio

  Alto

- **Porcentaje de sistemas vulnerables**

  Ninguno

  Bajo

  Medio

  Alto

### 1.4.1 Resultado de la aplicación del estándar CVSS

En el Anexo 1 se muestra una tabla con las puntuaciones alcanzadas por las vulnerabilidades que se desean detectar. En la determinación de estas se tuvo en cuenta sólo las métricas de Base, ya que según la Guía de CVSS versión 2.0, con esta métrica es suficiente para una correcta identificación de las clasificaciones necesarias, se plantea que las demás son opcionales como se muestra en la siguiente imagen.

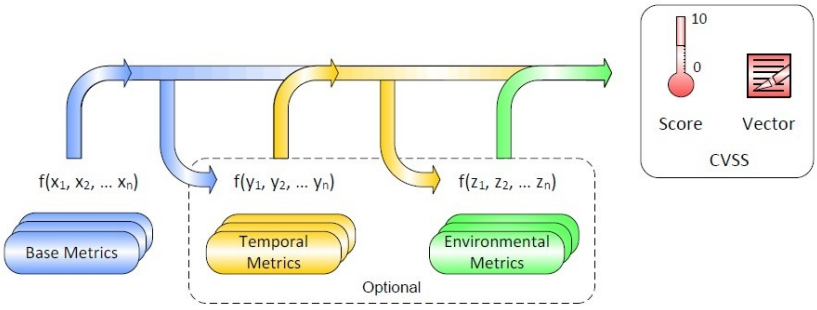


*Ilustración 5. Métricas y fórmulas de CVSS.*

Para el cálculo de los puntajes se hizo uso de la siguiente ecuación definida por la CVSS:

*BaseScore* = *round_to_1_decimal*( ( ( 0.6**Impact* ) +( 0.4 * *Exploitability* ) – 1.5 ) * f ( *Impact* ) )

*Impact* = 10.41 * ( 1 - ( 1-*ConfImpact* ) * ( 1 – *IntegImpact* ) * ( 1 – *AvailImpact* ) )

*Exploitability* = 20 * *AccessVector* **AccessComplexity* **Authentication*

Como resultado de la aplicación de este estándar, se obtuvo las siguientes clasificaciones:

- *Warning* para las vulnerabilidades de tipo denegación de servicio y obtención de información.
- *Alert* para las vulnerabilidades de tipo desbordamiento de *buffer* y obtención de privilegios.
- *Information* para las vulnerabilidades de tipo Inyección SQL.

**1.5 Metodología de desarrollo**

El proceso de desarrollo de la herramienta ACF fue guiado por la metodología OpenUp por lo que se hace uso de esta para guiar el proceso de desarrollo. Además es la idónea para el trabajo de equipos pequeños con poco tiempo para la culminación del sistema, razones por las cuales se escoge esta metodología para el desarrollo del módulo.

OpenUp/Basic es un *framework* (marco de trabajo) de procesos de desarrollo de programas de código abierto, que con el tiempo espera cubrir un amplio conjunto de necesidades en el campo del desarrollo de programas. Permite un abordaje ágil en el análisis con solamente proveer un conjunto simplificado de contenidos, fundamentalmente relacionados con orientación, productos de trabajo, roles y tareas (31). Es

un proceso interactivo de desarrollo de *software* simplificado, completo y extensible; para pequeños equipos de desarrollo que valoran los beneficios de la colaboración y los involucrados, con el resultado del proyecto por encima de formalidades innecesarias.

**Fases de OpenUp:**

**Inicio**

La fase de Inicio se trata de comprender el alcance y los objetivos del proyecto y obtener suficiente información para confirmar que el proyecto debe continuar. Se definen para el proyecto: el ámbito, los límites, el criterio de aceptación, los casos de uso críticos, una estimación inicial del coste y un boceto de la planificación.

**Elaboración**

En esta fase se realizan tareas de análisis del dominio y definición de la arquitectura del sistema. Se debe elaborar un plan de proyecto, estableciendo unos requisitos y una arquitectura estables. Por otro lado, el proceso de desarrollo, las herramientas, la infraestructura a utilizar y el entorno de desarrollo también se especifican en detalle en esta fase. Al final de la fase se debe tener una definición clara y precisa de los casos de uso, los actores, la arquitectura del sistema y un prototipo ejecutable de la misma.

La metodología propone como vistas de la arquitectura las siguientes:

- **Lógica**: describe la estructura y el comportamiento de porciones arquitectónicamente significativos del sistema. Esto podría incluir la estructura de paquetes, interfaces críticas, clases y subsistemas importantes y las relaciones entre estos elementos. También incluye vistas físicas y lógicas de datos persistentes, si la persistencia se construirá en el sistema. Este es un subconjunto documentado del diseño.
- **Operacional**: describe los nodos físicos del sistema y de los procesos, hilos y los componentes que se ejecutan en los nodos físicos. Este punto de vista no es necesario si el sistema se ejecuta en un proceso único.
- **Caso de uso**: una lista o diagrama de los casos de uso que contienen.

**Construcción**

Esta fase está enfocada al diseño, implementación y prueba de las funcionalidades del sistema. El propósito de esta fase es completar el desarrollo del sistema basado en la arquitectura definida.

**Transición**

Su propósito es asegurar que el sistema es entregado a los usuarios y evalúan las funcionalidades y rendimiento.

El siguiente diagrama muestra la relación existente entre las distintas fases y el ciclo de vida de la metodología.

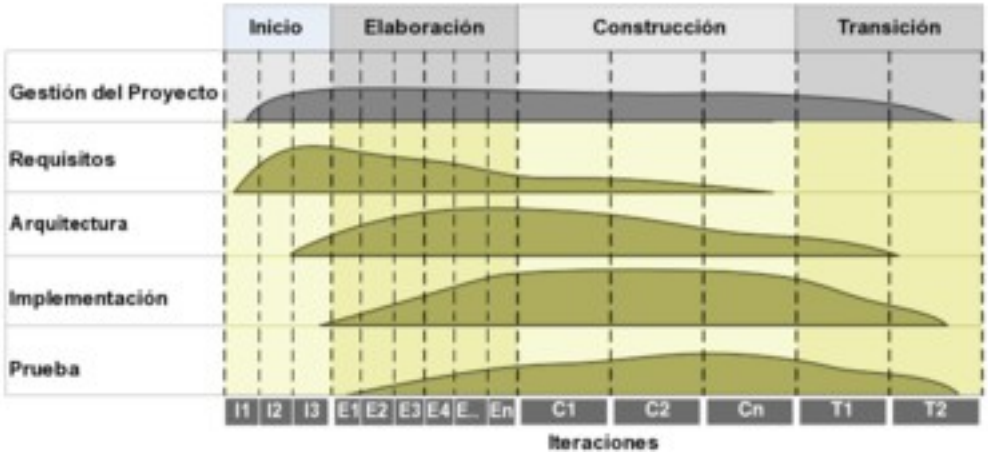


*Ilustración 6. Relación entre las fases y el ciclo de vida de OpenUp.*

### 1.6 Lenguajes de Programación

#### 1.6.1 Lenguaje C++

Se escoge como lenguaje de programación C++ ya que es el lenguaje que se utilizó para el desarrollo de la herramienta ACF, así como para el desarrollo de los módulos ya existentes, además se desea seguir los mismos estándares de codificación que posee ACF. Las principales características del Lenguaje C++ son:

1. Tiene un conjunto completo de instrucciones de control.
2. Permite la agrupación de instrucciones.
3. Incluye el concepto de puntero (variable que contiene la dirección de otra variable).

4. Los argumentos de las funciones se transfieren por su valor.
5. Los procesos de Entrada y Salida no forman parte del lenguaje, sino que se proporciona a través de una biblioteca de funciones.
6. Permite la separación de un programa en módulos que admiten compilación independiente.

### 1.6.2 Lenguaje XML

Se escoge el lenguaje XML para la creación de un diccionario que va a contener las vulnerabilidades, su tipo y el mensaje con la descripción, esto en forma de árbol, ya que este lenguaje se considera un estándar para el intercambio de información estructurada, además permite definir etiquetas personalizadas para descripción y organización de datos. La utilización de este permitió centralizar las vulnerabilidades en una solo fichero, de forma tal que para añadir una nueva vulnerabilidad no sea necesario modificar el código fuente del módulo a desarrollar, sino que con modificar el XML será suficiente para que el módulo realice la detección de las nuevas vulnerabilidades.

Una de las principales características de este lenguaje lo constitiye su cualidad de ser abierto, derivado del SGML[30], optimizado para su uso en la Web y que permite describir el sentido o la semántica de los datos. XML a diferencia del HTML, separa el contenido de la presentación. Es un Meta-Lenguaje[31], que permite la definición de lenguajes concretos de representación de documentos (50).

## 1.7 Entorno de Desarrollo Integrado (IDE)

Un entorno de desarrollo integrado (IDE) es un programa compuesto por un conjunto de herramientas para que el programador las utilice. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien, puede utilizarse para varios. El entorno de desarrollo es imprescindible en la producción de un *software*.

Para el desarrollo de la solución se propone la utilización del IDE Eclipse en su versión 3.8. Se escoge este ya que es el entorno de desarrollo integrado que más domina la autora, además que es el IDE idóneo para el trabajo con el lenguaje C++, definido como lenguaje a utilizar en el desarrollo de la propuesta de

30 SGML: **S**tandar **G**eneralizer **M**arkup **L**enguage en inglés y Estándar de Lenguaje de Etiquetado Generalizado en español. Es una norma que se aplica a todos los lenguajes de marcas, donde los más reconocidos son HTML y RTF.

31

solución. Mayoritariamente se utiliza para desarrollar lo que se conoce como "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Es una potente y completa plataforma de programación, desarrollo y compilación de elementos tan variados como sitios web, programas en C++ o aplicaciones JAVA. Integra herramientas y funciones necesarias para el trabajo, recogidas además en una atractiva interfaz que lo hace fácil y agradable de usar.

### 1.8 Herramienta Case[32] Visual Paradigm.

Visual Paradigm es una herramienta UML (*Unified Modeling Language*) profesional que soporta el ciclo de vida completo del desarrollo de *software*: análisis y diseño Orientados a Objetos, construcción, pruebas y despliegue (33). El *software* de modelado UML ayuda a una más rápida construcción de aplicaciones. Permite dibujar todos los tipos de diagramas de clases, código inverso y generar documentación.

Se selecciona esta herramienta ya que todos los diagramas del proceso de desarrollo de ACF en su versión actual están realizados en la misma.

### 1.9 Herramienta de documentación de código

Doxygen es un programa que analiza el código en busca de directivas en forma de comentarios y las transforma en documentación, ya sea HTML, LaTeX[33] e incluso se puede crear una página de manual de Linux para visualizar con el comando "man"[34]. Esta acepta multitud de lenguajes como C/C++ o JAVA**.** La principal ventaja de esta herramienta es que las directivas no son más que comentarios de forma tal que no es necesario crear una documentación aparte, sino, simplemente, comentar el código.

32 Las herramientas CASE (***C****omputer* ***A****ided* ***S****oftware* ***E****ngineering*, ingeniería asistida por computadora) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de *software* reduciendo el costo de las mismas en términos de tiempo y de dinero.

33 LaTeX**:** es un sistema de composición de textos, orientado a la creación de documentos escritos que presenten una alta calidad tipográfica. Por sus características y posibilidades, es usado de forma especialmente intensa en la generación de artículos y libros científicos que incluyen, entre otros elementos, expresiones matemáticas.

34 Man**:** muestra el manual del comando que se le indique.

```
void append (string path, const char* format, ...);
/**
* @brief Esta función comprueba que el archivo que recibe como
* parámetro es un script y en el caso que lo sea devuelve el nombre del programa que lo
* ejecuta. Si no es un script devuelve una cadena vacía.
* @return Devuelve una cadena de texto que representa el programa que ejecuta el script.
*/
string getScriptExtension(string path);
```

*Ilustración 7. Ejemplo de documentación de código usando Doxygen.*

Se escoge la misma en su versión 1.7.8 ya que esta es la herramienta de documentación de código que se utilizó para documentar el código de ACF.

**Conclusiones Parciales**

A lo largo de este capítulo han sido expuestos los principales puntos de interés relacionados con el concepto de vulnerabilidad y sus clasificaciones y el análisis de código fuente lo que permitió realizar un profundo análisis de las principales vulnerabilidades presentes en el lenguaje JAVA donde resaltan la denegación de servicio, ejecución de código arbitrario, obtención de información y el desbordamiento de *buffer* como las más utilizadas para explotar sistemas informáticos. Se realizó un estudio de las herramientas que realizan auditoría de código fuente y se concluyó que ninguna de ellas realiza una búsqueda completa de los tipos de vulnerabilidades presentes en el lenguaje JAVA.

El estudio del arte realizado permitió identificar la metodología de desarrollo de *software* y herramientas que se utilizarán en el desarrollo de la solución que se propone, teniendo como resultado las siguientes: como metodología de desarrollo de *software* OpenUp, como lenguajes de programación C++ y XML, entorno de desarrollo Eclipse y como herramienta Case: Visual Paradigm.

# Capítulo 2. Análisis y Diseño del Módulo de JAVA para ACF

**Introducción**

En el presente capítulo se establecen las principales características y funcionalidades con que contará la solución ya sean requisitos funcionales o no funcionales. Se generan los diagramas correspondientes a las etapas de planificación y diseño que propone la metodología de desarrollo OpenUp. Además se describen los patrones de diseño y arquitectura que se utilizarán, así como el modelo de dominio del sistema y el diagrama de clases del diseño.

## 2.1 Propuesta de Solución

Actualmente ACF se encuentra en su versión 1.0. Esta realiza análisis de código fuente de los lenguajes Bash, C++, Perl y Python, además cuenta con un módulo llamado Cuba el cual se encarga de prevenir ataques políticos a través de ficheros.

Para la creación de la solución propuesta se tuvo en cuenta que ya existen cinco módulos, por lo que es necesario que el que se cree cumpla con los mismos estándares de codificación que los demás. Deberá realizar un análisis estático de código fuente, donde detecte funciones o clases que podrían conllevar a una de las vulnerabilidades descritas en el capítulo anterior. En el Anexo 5 se muestra una tabla con dichas funciones o clases y las vulnerabilidades a las que corresponden.

El módulo debe ser capaz, además de detectar dichas vulnerabilidades, de generar un reporte detallado con la dirección local donde se encuentran los ficheros que presenten vulnerabilidades, así como el daño que pueden ocasionar, estas serán clasificadas en *Warning, Alert* o *Information*.

En la siguiente imagen se muestra cómo será el proceso de detectar vulnerabilidades. Este comienza cuando el usuario inserta en los campos de la interfaz de la herramienta los ficheros de código que desea auditar y la dirección donde se generará el reporte. Después de insertados los datos ACF verifica la extensión de los ficheros, si son .java entonces activa el módulo de JAVA, el cual analiza los ficheros y genera el reporte. ACF obtiene el reporte generado y lo muestra al usuario, culminando de esta forma el

proceso de detección.


*Ilustración 8. Ejemplo de cómo sería el funcionamiento de la solución propuesta.*

### 2.3 Requisitos funcionales

Un requisito funcional define una función del *software* o sus componentes; pueden ser cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas que se supone, un sistema debe cumplir (34).

**Requisitos funcionales:**

**RF1** Definir extensiones de los archivos del código fuente JAVA para conformar la información que debe brindar el módulo.

**RF2** Detectar vulnerabilidades que conlleven a una denegación de servicio.

**RF3** Detectar vulnerabilidades de ejecución de código.

**RF4** Detectar vulnerabilidades correspondiente a *bypass something*[35].

**RF5** Detectar vulnerabilidades en el uso de funciones de manejo de *buffer*.

**RF6** Detectar vulnerabilidades que impliquen la obtención de información.

**RF7** Detectar vulnerabilidades que impliquen la obtención de privilegios.

35 *Bypass something:* se refiere a la omisión de parámetros y objetos.

**RF8** Detectar vulnerabilidades que impliquen la corrupción de memoria.

**RF9** Generar reporte de vulnerabilidades encontradas.

### 2.4 Requisitos no funcionales.

Un requisito no funcional especifica criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que éstos corresponden a los requisitos funcionales. Sirven de apoyo a los requisitos funcionales (36).

**Usabilidad**

El módulo deberá presentar facilidades al usuario para el manejo de la información, se pretende acoplar a la vista descriptiva, sencilla y fácil de usar que posee ACF para la realización de todas las operaciones, con el objetivo de propiciar un buen entendimiento a los usuarios finales.

**Restricciones de implementación:**

1. Utilizar como lenguaje de programación C++.
2. El estilo de programación debe ser el estándar de codificación del lenguaje C++.
3. Uso de la herramienta de documentación de código Doxygen en su versión 1.7.8.

**Restricciones de *software*:**

1. Es necesario instalar el sistema operativo Nova 2013 ya la herramienta ACF está diseñada para que funcione correctamente sobre este sistema operativo.

**Compatibilidad**

El módulo a desarrollar deberá integrarse con el resto de la aplicación y analizar todos los ficheros de extensión .java que le proporcione la herramienta, además deberá ser capaz de generar un reporte de las vulnerabilidades encontradas.

### 2.5 Definición de actores y casos de uso del sistema.

Un caso de uso es una descripción de los pasos o las actividades que deberán realizarse para llevar a cabo algún proceso. Los personajes o entidades que participarán en un caso de uso se denominan actores (37).

Los diagramas de casos de uso sirven para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas (38). En resumen es un diagrama que muestra la relación entre los actores y los casos de uso en un sistema.

**Actores**

Se le llama actor a toda entidad externa al sistema que guarda una relación con éste y que le demanda una funcionalidad (39). Esto incluye a los operadores humanos pero también incluye a todos los sistemas externos, además de entidades abstractas, como el tiempo.

### 2.5.1 Definición de actores del sistema

En la siguiente tabla se describen los actores del sistema y las actividades que realizan.

| Actores | Justificación |
|---|---|
| Mantenedor de repositorio | Es la persona con necesidad de revisar el código fuente de una aplicación en busca de vulnerabilidades. |

*Tabla 4. Actores del sistema.*

### 2.5.2 Diagrama de Casos de Uso del Sistema.

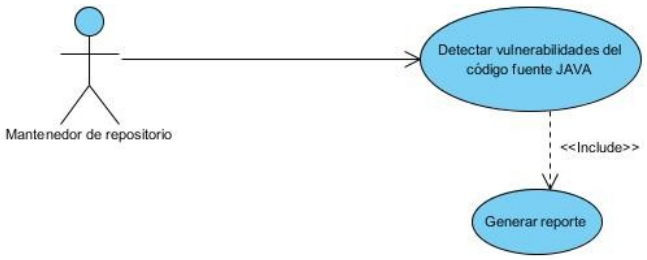


*Ilustración 9. Diagrama de Casos de Uso del sistema.*

**Descripción textual de los casos de uso del sistema.**

Las descripciones textuales, ayudan a los usuarios a entender y comprender las actividades que se llevan a cabo dentro de los casos de uso, sirviéndoles para evaluar la complejidad de los mismos y saber de qué forma se les ha dado solución a los requisitos que se definieron como funcionales. De igual forma indican a

los usuarios como interactuar con el sistema desarrollado, diciéndoles a cada momento que deben hacer y que deben esperar como respuesta. En las tablas siguientes se muestran las descripciones textuales de los casos de uso del sistema.

| **Caso de Uso # 1** | **Detectar vulnerabilidades del código fuente JAVA.** |
|---|---|
| **Actores** | Mantenedor de repositorio |
| **Resumen** | El caso de uso inicia cuando el usuario decide detectar las vulnerabilidades que existen en el código fuente de uno o varios ficheros y finaliza cuando estas son detectadas e informadas correctamente, o se muestran las razones por las que la detección de las mismas ha sido insatisfactoria. |
| **Precondiciones** | Debe haber un directorio o archivo de código fuente JAVA. |
| **Referencias** | RF1-RF9 |
| **Complejidad** | Alta |
| **Prioridad** | Alta |
| **Poscondiciones** | |

| **Flujo Normal de Eventos** | | |
|---|---|---|
| **#** | **Acción del Actor** | **Respuesta del Sistema** |
| **1** | 1. El caso de uso se inicia cuando el usuario introduce el archivo a auditar, especifica el directorio donde se van a almacenar los reportes y después oprime el botón “Buscar vulnerabilidades”. | 2. ACF toma los datos introducidos por el usuario, comprueba que el archivo corresponde con un archivo de código fuente JAVA y después es que se ejecuta el módulo. |
| | | 3. Se activa el módulo de JAVA, con el directorio donde están los ficheros a auditar y la dirección donde se desea guardar el reporte. |
| | | 4. Se procede a detectar las vulnerabilidades en el fichero. Si no se detectan vulnerabilidades ver flujo alterno 1. |

| | | |
|---|---|---|
| | | 4.1. Detecta vulnerabilidades de condición de carrera. |
| | | 4.2. Detecta vulnerabilidades de denegación de servicio. |
| | | 4.3. Detectada vulnerabilidades de obtención de información. |
| | | 4.4. Detectada vulnerabilidades de obtención de privilegios. |
| | | 4.5. Detectada vulnerabilidades de desbordamiento de *buffer*. |
| | | 4.6. Detecta vulnerabilidades de ejecución de código arbitrario. |
| | | 4.7. Detecta vulnerabilidades de corrupción de memoria. |
| | | 5. Se guarda el reporte en la dirección especificada. |
| **2** | El usuario visualiza el reporte. | |
| **Flujo Alterno 1** | | |
| | **Acción de los Actores** | **Respuesta del Sistema** |
| | | 1. Se muestra el mensaje: “No se detectaron vulnerabilidades”. |

*Tabla 5. Descripción textual del caso de uso del sistema ¨Detectar vulnerabilidades en el código fuente JAVA¨.*

### 2.6 Diagrama de clases del diseño

Los diagramas de clases del diseño son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema y los componentes que se encargarán del funcionamiento y la relación entre uno y otro. Describen gráficamente las especificaciones de las clases del *software* y de las interfaces en una aplicación. En su diseño contiene

como información (clases, métodos, información sobre los tipos de los atributos, dependencias) (46).

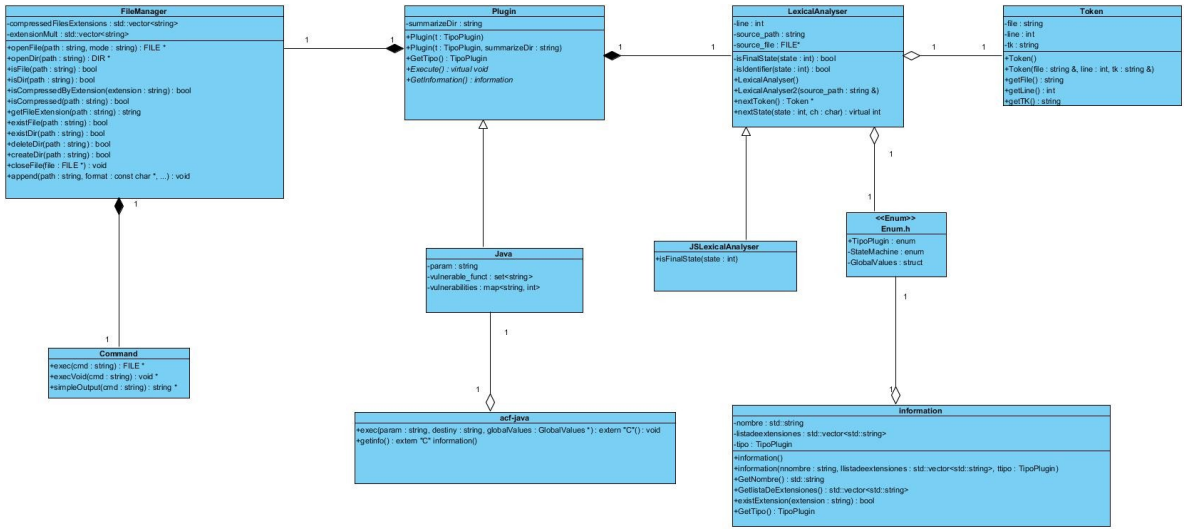


*Ilustración 10. Diagrama de clases del diseño.*

### 2.6.1 Descripción de las clases fundamentales del diseño

***Plugin***:

Esta es la clase padre del módulo Java. También tiene una relación de composición con la clase *LexicalAnalyser* lo que le permite el manejo de los *tokens*, que posteriormente serán analizados por el módulo. Realiza la acción de crear el directorio de reportes.

**Java**:

Esta clase hereda de *Plugin* y posee una relación de agregación con la clase acf-java. Se encarga del manejo de ficheros para la dirección donde se encuentran los ficheros a auditar y el directorio donde se va a guardar el resumen de las vulnerabilidades encontradas. Se encarga de analizar los ficheros de código fuente en busca de posibles vulnerabilidades. Después de obtener de la clase *Plugin* la dirección donde se encuentran los ficheros de código fuente, la dirección donde se generará el resumen y el tipo de *plugin*, este realizará una búsqueda *token* a *token* para identificar las funciones, métodos o clases que podrían conllevar a una vulnerabilidad. Después de analizar el fichero genera dos tipos de reportes, uno detallado,

con la dirección del fichero analizado, las vulnerabilidades detectadas, así como el mensaje generado; y otro reporte resumen con la dirección del fichero y las cantidades de advertencias e informaciones correspondientes a ese fichero.

**acf-java:**

Esta interfaz es la extensión que conecta al módulo con ACF. Además es la que se utiliza como fachada del módulo ya que es la que contiene funciones públicas y genéricas que permiten acceder a las funciones que este brinda.

En el Anexo 2 se podrán ver segmentos del código fuente de las clases descritas anteriormente.

### 2.7 Diagramas de Secuencia

Los diagramas de secuencia muestran gráficamente las interacciones del actor y de las operaciones a que dan origen. El diagrama de secuencia muestra un determinado escenario de un caso de uso, los eventos generados por actores externos, su orden y los eventos internos del sistema. Estos destacan la ordenación temporal de los mensajes. En el análisis y diseño se definió un diagrama de secuencia para el caso de uso planteado. En el Anexo 3 se muestra el diagrama de secuencia del caso de uso Detectar Vulnerabilidades.

### 2.8 Estilos Arquitectónicos y patrones de diseño

Categoría de los patrones según la escala o nivel de abstracción (40):

- **Patrones de arquitectura**: aquellos que expresan un esquema organizativo para sistemas de *software*.
- **Patrones de diseño**: aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de *software*.
- **Dialectos**: patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

**Clasificación de los patrones**

• Según su propósito:

**De creación**: conciernen al proceso de creación de objetos.

**De estructura**: tratan la composición de clases u objetos.

**De comportamiento**: caracterizan las formas en las que interactúan y reparten responsabilidades las distintas clases u objetos.

### 2.8.1 Arquitectura de la solución propuesta

La herramienta ACF está basada en la arquitectura n-capas, la cual divide un *software* en varias partes lógicas, ya sean módulos, paquetes o capas y ofrece la posibilidad de comprender fácilmente su filosofía y distribuir las tareas que ejecuta.

Las capas de la arquitectura de ACF son las siguientes:

- Capa *View*, contiene el paquete *Facade_Subsystem*.
- Capa *Business Logic*, contiene el paquete *Heart_Subsystem*.
- Capa *Data Access*, contiene el paquete *DBConexion_Subsystem*.

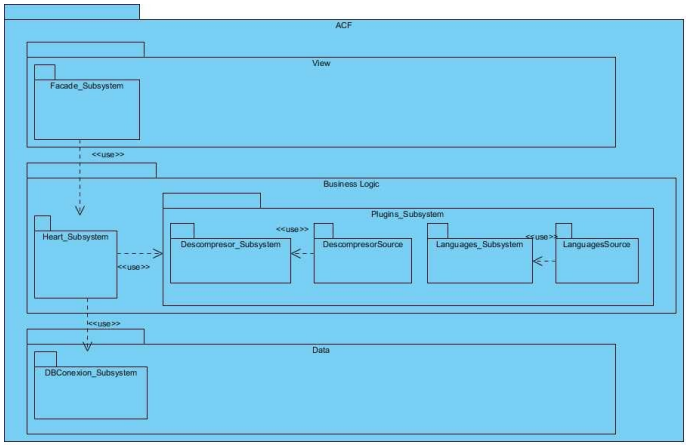


*Ilustración 11. Diagrama de Paquetes.*

Dentro de la capa *Business Logic* se encuentra el paquete *Plugin_Subsystem*, el cual hace uso de una arquitectura basada en componentes. Esta capa contiene los paquetes *Descompresor_Subsystem*, *DescompresorSource*, *Language_Subsystem* y *LanguageSource*. El módulo a programar se encuentra dentro del paquete *LanguageSource* por lo que se acoge a la arquitectura basada en componentes que posee el paquete principal *Plugin_Subsystem*.

### 2.8.2 Patrones de diseño GRASP

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de *software* y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias (43).

Entre ellos se encuentran los Patrones Generales de *Software* para Asignar Responsabilidades llamados comúnmente GRASP, del acrónimo de *General Responsibility Assignment Software Patterns*. Los usados en el módulo fueron:

- **Alta cohesión:** se aplica para realizar un diseño que evite contener clases con un alto grado de abstracción, delegando las responsabilidades muy complejas a otros objetos.

En la imagen que se presenta a continuación se pone de manifiesto el uso de este patrón ya que la clase *FileManager* es capaz de cumplir con las funciones para las que fue creada, no posee un alto grado de abstracción y cuenta con funciones moderadas en un área funcional.

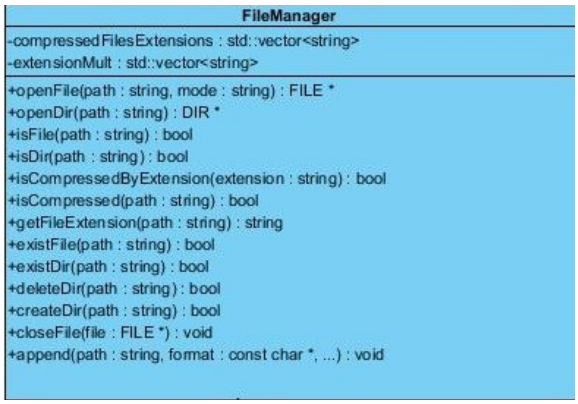


*Ilustración 12. Ejemplo donde se pone de manifiesto el uso del patrón Alta cohesión.*

- **Bajo acoplamiento:** "El patrón Bajo acoplamiento es una medida de la fuerza con que una clase se relaciona con otras, porque las conoce y recurre a ellas; una clase con bajo acoplamiento no

depende de muchas otras, mientras que otra con alto acoplamiento presenta varios inconvenientes: es difícil entender cuando está aislada, es ardua de reutilizar porque requiere la presencia de otras clases con las que esté conectada y es cambiante a nivel local cuando se modifican las clases afines". Este patrón es un principio que asigna la responsabilidad de controlar el flujo de eventos del sistema a clases específicas. El uso de este patrón permite que las clases no se afecten por cambios de otros componentes, haciendo posible que sean fáciles de entender y de reutilizar.

En la imagen que se muestra a continuación se pone de manifiesto el uso de este patrón ya que estas clases están relacionadas solamente con otras clases que le son indispensable para la colaboración a la hora de realizar funciones más complicadas. Otro ejemplo donde se puede observar el uso de este patrón es en la relación entre la clase Java con las clases *LexicalAnalyser* y *FileManager*.

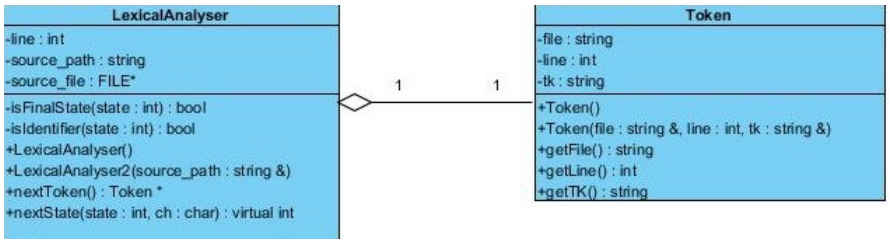


*Ilustración 13. Ejemplo donde se pone de manifiesto el uso del patrón Bajo Acoplamiento.*

- **Creador:** se aplica para la asignación de responsabilidades a las clases relacionadas con la creación de objetos, de forma tal que una instancia de un objeto sólo pueda ser creada por el objeto que contiene la información necesaria para ello. De esta forma una clase controladora puede crear objetos, pero es el modelo quien sabe cómo crearse.

```
Java* java;
java = new Java(TipoPlugin(1), destiny, param);
java->Execute(globalValues);

delete java;
```

*Ilustración 14. Ejemplo donde se pone de manifiesto el uso del patrón Creador.*

### 2.8.3 Patrones de diseño GOF

El catálogo de patrones más famoso es el contenido en el libro —*Design Patterns: Elements of Reusable Object-Oriented Software*, el de la banda de los 4, también conocido como el libro GOF (*Gang Of Four Book*). Según el libro GOF estos patrones se clasifican según su propósito en creacionales, estructurales y de composición, mientras que respecto a su ámbito se clasifican en clases y objetos:

Respecto a su ámbito:

1. **Clases**: Relaciones estáticas entre clases.
2. **Objetos**: Relaciones dinámicas entre objetos.

**Patrón Estructural *Facade* (Fachada)**

El patrón fachada viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos.

Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada y los clientes pueden permanecer ajenos a ello. Además, como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.

Como inconveniente, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

En el Anexo 2 se puede ver una imagen del uso de este patrón. El patrón fachada plantea que se debe definir una clase que publique las funcionalidades públicas de un subsistema módulo, de forma tal que se pueda acceder a sus funciones sin importar la lógica de implementación por detrás. El componente acf-java.cpp contiene funciones públicas y genéricas que permiten acceder a las funciones que brinda el

módulo, por lo que esta funciona como una interfaz. En esta es donde se pone de manifiesto el uso de este patrón.

**Conclusiones Parciales**

En este capítulo se identificaron los requerimientos funcionales y no funcionales, lo cual permitió obtener una descripción en lenguaje natural de las necesidades de los clientes. Se definió el actor y el diagrama de casos de uso, lo que deja bien definido como será el proceso de detección que tendrá que realizar el módulo. Se expuso el diagrama de clases del diseño así como una descripción de sus principales clases, lo que contribuirá a un mejor entendimiento de la solución. Se utilizaron los patrones Bajo Acoplamiento, Alta cohesión, Creador y *Facade*. Se hizo uso de la arquitectura Basada en componentes que es la misma que posee el paquete LanguageSource, donde se encuentran los componentes de los módulos incluyendo el que se desea desarrollar en la presente investigación.

# Capítulo 3. Implementación y Prueba

## Introducción

En este capítulo se documentarán las fases de construcción y transición con los artefactos que las mismas generan según la metodología empleada. Se definirán las métricas para la clasificación de las vulnerabilidades a detectar por la herramienta. Además se determina el funcionamiento y objetividad de la propuesta desarrollada con el objetivo de verificar su calidad mediante pruebas; para determinar si los requisitos identificados realmente definen el sistema que se debe construir.

### 3.1 Diagrama de Componentes

Un **componente** representa una parte de un sistema modular, desplegable y reemplazable, que encapsula la implementación y expone un conjunto de interfaces (45). Podría ser, por ejemplo, código fuente, binario o ejecutable.

En el siguiente diagrama se muestra la relación que posee el componente *Plugin*.h con el resto de los componentes que le ayudan a completar el proceso de análisis de código fuente y la generación de los reportes que debe mostrar la herramienta al culminar dicho proceso, como es el caso del componente *FileManager*.h, *LexicalAnalyser*.h, Información.h y Java.h; este último es el componente encargado del análisis del código fuente JAVA, el cual hereda del componente *Plugin*.h todos los atributos y métodos que le hacen falta para completar el análisis y generación de reportes.

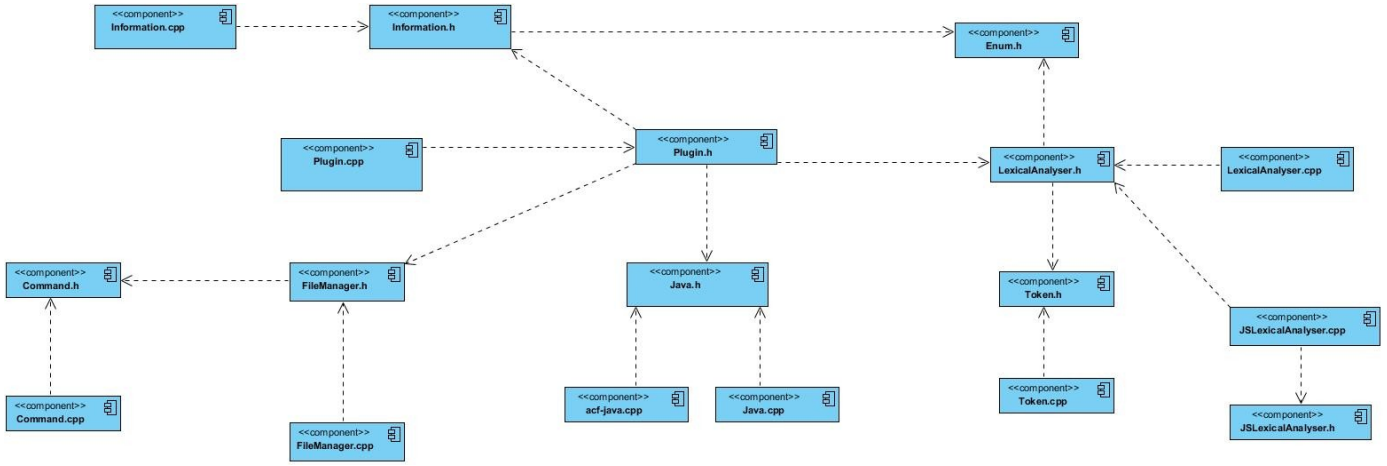


*Ilustración 15. Diagrama de Componentes.*

### 3.3 Código fuente

El código fuente de un sistema informático es un conjunto de líneas de texto que son las instrucciones que debe seguir la computadora para así poder ejecutar las funcionalidades requeridas por el *software*. El código fuente varía según el lenguaje de programación y debe ser traducido al lenguaje máquina para que sea ejecutado.

#### 3.3.1 Estándar de codificación

Para garantizar la uniformidad del código en el desarrollo del módulo, se definió un estándar de codificación, lo cual posibilita cometer menos errores durante la implementación y que la lectura y comprensión del mismo sea mucho más fácil para otros desarrolladores. A continuación se muestran varios ejemplos del estándar utilizado:

**Identación**

**Longitud de la línea:**

Las líneas tienen siempre no más de 80 caracteres.

**Rompiendo líneas:**

Cuando una expresión no entra en una línea, se rompe de acuerdo con estos principios:

- Romper después de una coma.
- Romper antes de un operador.

**Ejemplo:**

```
extern "C" void exec(string param,  string destiny, GlobalValues* globalValues) {

    Java* java;
    java = new Java(TipoPlugin(1), destiny, param);
    java->Execute(globalValues);

    delete java;
}
```

*Ilustración 16. Ejemplo del Estándar de Codificación Rompiendo líneas.*

**Declaraciones**

- Para declarar una clase se comienza con letra mayúscula y el resto con minúscula, siguiendo la línea de declaraciones que poseen el resto de los módulos ya implementados.

  Deben ser nombres cortos pero con significado.

  **Ejemplo:**

```
class Java: public Plugin {
```

*Ilustración 17. Ejemplo de Declaraciones de clases.*

- Los métodos y variables se declaran con minúscula.

**Ejemplo:**

```
private:

    string param;
    set<string> vulnerable_funct;
    map<string, int> vulnerabilities;
```

*Ilustración 18. Ejemplo de Declaraciones de métodos y variables.*

**Saltos de línea.**

- Añadir un salto de línea después de un punto y coma, cuando termina la sentencia.

**Ejemplo en el código:**

```
    if(info > 0 || warning > 0) {
        fileManager->append(sumarizedir, "%s\n", param.c_str());
        fileManager->append(sumarizedir, "->  ADVERTENCIAS = %d\n", warning);
        fileManager->append(sumarizedir, "-> INFORMACIONES = %d\n\n", info);
    }
}
```

*Ilustración 19. Salto de línea después de punto y coma.*

- Añadir un salto de línea después de abrir una llave.

  **Ejemplo en el código:**

```
#include "../Plugin.h"
class Java: public Plugin {

private:
```

*Ilustración 20. Salto de línea después de abrir una llave.*

### 3.4 Pruebas de *software*

Las pruebas de *software* son un conjunto de herramientas, técnicas y métodos que evalúan la excelencia y el desempeño de un *software*, involucra las operaciones del sistema bajo condiciones controladas y

evaluando los resultados. Las técnicas para encontrar problemas en un programa son variadas y van desde el uso del ingenio por parte del personal de prueba hasta herramientas automatizadas que ayudan a aliviar el peso y el costo de tiempo de esta actividad (47).

### 3.4.1 Niveles de Prueba

- **Pruebas de Sistema:**

Se escoge este nivel de prueba ya que lo que se desea es la verificación de las funcionalidades del sistema a través de las interfaces y su relación con los demás módulos que presenta la herramienta, específicamente con el módulo Java.

Las pruebas de sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica (47).

En lugar de caer en este absurdo, el ingeniero del *software* debe de anticiparse a posibles problemas de la interfaz:

- ✔ Diseñar ruta de manejo de errores que prueben toda la información proveniente de otros elementos del sistema.
- ✔ Aplicar una serie de pruebas que simulen datos incorrectos u otros posibles errores en la interfaz de *software*.
- ✔ Registrar los resultados de la prueba como evidencia en el caso de que se culpe.

- **Pruebas de Aceptación:**

Se escoge la prueba de aceptación ya que es necesario, para la validación de la solución, que el cliente esté de acuerdo con el funcionamiento del módulo desarrollado y así pueda emitir la carta de aceptación que demuestra la conformidad del cliente con la solución. Como técnica se escoge las pruebas alfa esta se lleva a cabo, por un cliente, en el lugar de desarrollo. Se usa el *software* de forma natural con el desarrollador como observador del usuario. Las pruebas alfa se llevan a cabo en un entorno controlado.

Para que tengan validez, se debe primero crear un ambiente con las mismas condiciones que se encontrarán en las instalaciones del cliente. Una vez logrado esto, se procede a realizar las pruebas y a documentar los resultados.

### 3.4.2 Método de Prueba

Se escoge el método de caja negra ya que los niveles de pruebas escogidos pretenden probar el funcionamiento de la interfaz y de las funcionalidades del sistema, para esto es necesario este método ya que nos permite comprobar el comportamiento del *software* a través de los requisitos funcionales (48), obviando el funcionamiento interno y la estructura del programa.

**Las pruebas de caja negra pretenden encontrar estos tipos de errores:**

- Funciones incorrectas o ausentes.
- Errores en la interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de comportamiento o desempeño.
- Errores de inicialización y de terminación.


*Ilustración 21. Esquema del Método de Caja Negra.*

### 3.4.3 Técnica de prueba Partición equivalente

**Características**

- Divide el dominio de entrada de un programa en clases de datos, a partir de las cuales pueden derivarse casos de prueba.
- Descubre clases de errores, que de otra manera, requeriría la ejecución de muchos casos antes de que se observe el error general.

- Reducir al máximo el total de casos de prueba que deben desarrollarse.

  **El diseño consiste:**

- Identificar clases de equivalencia.
- Crear los casos de prueba.

**Clase de equivalencia**

- Conjunto de estados válidos o inválidos para condiciones de entrada.

  Las clases de equivalencia se definen de acuerdo a las siguientes directrices:

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos no válidas.
2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos no válidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y otra no válida.
4. Si una condición de entrada es booleana, se define una clase de equivalencia válida y otra no válida.

### 3.4.4 Diseño de Casos de Prueba basado en casos de uso

**Caso de Prueba 1. Detectar vulnerabilidades**

| Escenario | Descripción | V1 Dirección de los ficheros a auditar | V2 Dirección donde se guarda el reporte | Respuesta del sistema | Flujo Central |
|---|---|---|---|---|---|
| EC 1.1 Análisis y detección de vulnerabilid | La herramienta analiza los ficheros y genera los reportes correspondientes. | / tmp/Asistente | /tmp | El sistema muestra el reporte detallado en un cuadro de texto en su interfaz y genera los reportes | 1. El usuario presiona el botón examinar del campo Dirección y selecciona el |

| | | | | | |
|---|---|---|---|---|---|
| ades. | | | | detallados y un reporte resumen. | directorio donde se encuentran los ficheros a auditar. |
| EC 1.2 Análisis y no detección de vulnerabilidades. | La herramienta analiza los ficheros y genera los reportes vacíos. | / tmp/Asistente | /tmp | El sistema muestra el mensaje “No se detectaron vulnerabilidades” y genera los reportes vacíos. | 2. El usuario presiona el botón examinar del campo Dirección de reporte y selecciona el directorio donde se desea guardar los reportes.<br>3. El usuario presiona el botón Auditar. |

*Tabla 6. Descripción de los escenarios del caso de prueba ¨Detectar vulnerabilidades¨.*

### 3.5 Resultados obtenidos de los casos de prueba

Se utilizó el método de caja negra para realizar las pruebas sobre la interfaz de la herramienta con el módulo del lenguaje JAVA ya incorporado. Donde se encontraron 10 no conformidades, en 3 iteraciones. De estas no conformidades 3 fueron de errores ortográficos y 7 de validación incorrecta. En la siguiente gráfica se muestra las iteraciones realizadas y las no conformidades detectadas, las resueltas y las pendientes.

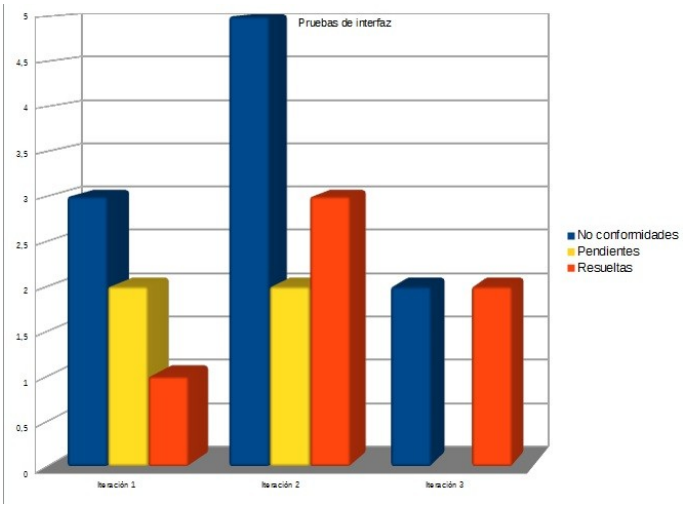


*Ilustración 22. Estadísticas de las pruebas.*

Se realizaron pruebas de aceptación a nivel de cliente con las que se pudo constatar que el sistema cumple puntualmente con todos los requisitos planteados. Al término de las pruebas se obtuvo un producto con calidad que cumple con todas las especificaciones inicialmente planteadas.

### 3.6 Análisis de proyectos JAVA

Para realizar el análisis de proyectos JAVA se realizaron pruebas de interfaz, estas están incluidas dentro de las prueba de sistemas, permitiendo comprobar el correcto funcionamiento de los requisitos funcionales planteados. Para esto se seleccionaron tres proyectos que contienen ficheros JAVA, a los cuales se les realizó una auditoría de código fuente. En el Anexo 4, se muestran imágenes de las pruebas realizadas.
En la siguiente tabla se muestran las cantidades de advertencias o informaciones que se esperan encontrar en cada directorio y las encontradas realmente.

| Nombre del proyecto JAVA | Cantidad de *Warning* esperadas | Cantidad de *Warning* obtenidas | Cantidad de Alert esperadas | Cantidad de *Alert* obtenidas | Cantidad de *Information* esperadas | Cantidad de *Information* obtenidas | |
|---|---|---|---|---|---|---|---|
| Asistente | 0 | Iteración 1: 0 | 2 | Iteración 1: 2 | 1 | Iteración 1: 0 | Iteración 2: 1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| InstaladorNew | 17 | Iteración 1: 16 | Iteración 2: 17 | 3 | Iteración 1: 2 | Iteración 2: 3 | 4 | Iteración 1: 4 |
| TesisGra | 10 | Iteración 1: 10 | | 0 | Iteración 1: 0 | | 2 | Iteración 1: 2 |

*Tabla 7. Análisis de proyectos JAVA.*

Como se pudo observar en la tabla el módulo no detectó varias vulnerabilidades en una sola iteración por lo que se obtuvo un total de 3 no conformidades, resueltas en una segunda iteración, demostrando el funcionamiento del módulo al analizar correctamente los proyectos JAVA propuestos.

**Conclusiones Parciales**

En el desarrollo del presente capítulo se realizó el modelo de implementación del módulo de JAVA para ACF con el propósito de mostrar los componentes del sistema y sus relaciones, a través del diagrama de componentes. Se especificó el uso de los estándares de codificación para lograr obtener claridad y organización en el código fuente de la solución. Se obtuvo una biblioteca dinámica[36] la cual le permite a ACF la revisión de código JAVA.

Se realizaron pruebas de sistema y pruebas de aceptación para comprobar el correcto funcionamiento del módulo, utilizando el método de caja negra basado en la técnica de partición de equivalencia. Se identificaron 10 no conformidades que fueron resueltas posteriormente.

36 Las bibliotecas dinámicas: son ficheros que contienen código objeto construido de forma independiente a su ubicación de tal modo que están preparadas para poder ser requeridas y cargadas en tiempo de ejecución por cualquier programa, en lugar de tener que ser enlazadas, previamente, en tiempo de compilación. Por tanto, han de estar disponibles como ficheros independientes al programa ejecutable (generalmente en directorios del sistema).

## Conclusiones Generales

Al desarrollar un módulo que detecte las vulnerabilidades de los ficheros de código fuente del lenguaje JAVA, haciendo uso de la herramienta ACF, se da cumplimiento al objetivo general planteado. Para llegar a este resultado se concluye lo siguiente:

1. Se realizó un estudio de las principales herramientas que realizan auditoría de código fuente JAVA lo cual posibilitó identificar las mejores prácticas en el proceso de detección de vulnerabilidades.
2. Se realizó el análisis, diseño e implementación del módulo de JAVA obteniendo una biblioteca dinámica que integrada a la herramienta ACF realiza análisis de ficheros de código fuente JAVA.
3. Se diseñaron y realizaron las pruebas de *software*, que permitieron la correcta comprobación del funcionamiento de la solución posibilitando realizar el proceso de entrega del *software* a consideración del cliente.

## Recomendaciones

1. Continuar con el estudio de los algoritmos que se utilizan para detectar otros tipos de vulnerabilidades en el código fuente JAVA y actualización de los módulos existentes.

## Referencias Bibliográficas

(1) w3techs.com [Online] [Citado febrero 2015, 20] http://w3techs.com/technologies/details/pl-java/all/all/.

(2) Belmonte Fernández, Oscar. Introducción al lenguaje de programación Java. Una guía básica, 2004.

(3) recursostic.educacion.es [Online] [Citado octubre 2014, 12] http://recursostic.educacion.es/observatorio/web/ca/software/software-general/1040-introduccion-a-la-seguridad-informatica?start=3.

(4) vivaolinux.com.br [Online] [Citado octubre 2014, 18] http://www.vivaolinux.com.br/artigo/Race-condition-vulnerabilidades-em-suids.

(5) hackplayers.com [Online] [Citado noviembre 2014, 8] http://www.hackplayers.com/2010/12/introduccion-format-string-attack-i-de.html.

(6) technet.microsoft.com [Online] [Citado noviembre 2014, 15] https://technet.microsoft.com/es-es/library/ms161953%28v=sql.105%29.aspx.

(7) oracle.com [Online] [Citado diciembre 2014, 5] http://www.oracle.com/technetwork/topics/security/javacpujun2012-1515912.html.

(8) Wen, Hengqing; Huang, Peter; Dyer, John; Archinal, Andy; Fagan, John (2004). «Countermeasures for GPS signal spoofing». University of Oklahoma. Consultado el 16 de diciembre de 2014.

(9) remote-exploit [Online] [Citado diciembre 2014, 10] http://www.remote-exploit.org.

(10) welivesecurity.com [Online] [Citado diciembre 2014, 16] http://www.welivesecurity.com/la-es/2012/08/30/como-funciona-explot-0-day-java/.

(11) vsantivirus.com [Online] [Citado enero 2015, 8] http://www.vsantivirus.com/java-exploit-bytverify.html/.

(12) abc.es [Online] [Citado enero 2015, 18] http://www.abc.es/tecnologia/20130115/abci-java-desactivar-problemas-201301151402.html.

(13) eromang.zataz.com [Online] [Citado enero 2015, 15] Oracle *Java Exploits and 0days Timeline.*

Recuperado de: http://eromang.zataz.com/uploads/oracle-java-exploits-0days-timeline.html 2013.

(14) cvedetails.com [Online] [Citado febrero 2015, 26] http://www.cvedetails.com/product/1526/SUN-JRE.html?vendor_id=5.

(15) «Artículo Security Restrictions sobre Applets de Sun Microsystem» (en inglés). Consultado el 15 de enero de 2015.

(16) java-0day.com [Online] [Citado enero 2015, 22] *Days since last known Java 0-day exploit*. Recuperado de: http://java-0day.com/.

(17) cisco.com [Online] [Citado diciembre 2014, 16] http://www.cisco.com/c/en/us/products/collateral/security/security-agent/product_bulletin_c25-484785.html.

(18) microsoft.com [Online] [Citado febrero 2015, 2] http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Exploit:Java/CVE-2013-2423.

(19) microsoft.com [Online] [Citado enero 2015, 27] http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Exploit:Java/CVE-2012-1723.

(20) microsoft.com [Online] [Citado marzo 2015, 2] http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Exploit%3AWin32%2FCVE-2012-0159.

(21) oracle.com [Online] [Citado enero 2015, 20] http://www.oracle.com/technetwork/topics/security/javacpufeb2012verbose-366319.html.

(22) microsoft.com [Online] [Citado enero 2015, 30] http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Exploit:Win32/CVE-2011-3402.

(23) welivesecurity.com [Online] [Citado marzo 2015, 4] http://www.welivesecurity.com/la-es/2013/05/22/vulnerabilidades-java-mas-explotadas-sitios-latinoamerica/.

(24) wikipedia.org [Online] [Citado diciembre 2014, 11] http://es.wikipedia.org/wiki/Lint.

(25) Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977. Consultado el 19 de diciembre de 2014.

(26) [Online] [Citado enero 2015, 20] Apache Ant, http://ant.apache.org/.

(27) [Online] [Citado enero 2015, 22] Jlint y Antic, http: //artho.com/jlint/.

(28) [Online] [Citado enero 2015, 22] FindBugs, http://findbugs.sourceforge.net/.

(29) [Online] [Citado enero 2015, 22] PMD, http://pmd.sourceforge.net/.

(30) ingenieriasoftware2011.wordpress/2011/07/08/herramientas-case-ejemplos-para-evaluarlas/. [Online] [Citado octubre 25, 2014].

(31) [Online] [Citado marzo 2015, 5] http://openup3.blogspot.com/2014/02/metodologia-open-up.html.

(32) [Online] [Citado marzo 2015, 2] https://eclipse.org.

(33) Macas Carrasco, Mayra Alexandra; Jácome Quintanilla, Ana Elizabeth. Análisis comparativo de bibliotecas multiplataforma para el desarrollo de aplicaciones de escritorio, aplicado a la escuela de diseño gráfico, Riobamba-Ecuador.

(34) Sommerville Ian (2006). *Software* Engineering, 8th ed. ISBN 0-321-31379-8. página 127 (en inglés).

(35) Pressman, Roger (2002). Ingeniería del *Software*, un enfoque práctico, Oc-Graw Hill página 145, consultado el 23 de abril de 2015.

(36) Barbosa Guerrero, Lugo Manuel. Arquitectura de *software* como eje temático de investigación. 2006.

(37) Jacobson, I., P. Jonsson, M. Christerson and G. Overgaard, Ingeniería de *Software* Orientada a Objetos - Un acercamiento a través de los casos de uso. Addison Wesley Longman, Upper Saddle River, N.J., 1992. Consultado el 4 de marzo de 2015.

(38) [Online] [Citado marzo 2015, 5] http://www.developer.com/design/article.php/2109801/Creating-Use-Case-Diagrams.html.

(39) El Proceso Unificado de Desarrollo de *Software*. 2000.2000. Consultado el 4 de marzo de 2015.

(40) Rutar, Almazan, Foster (2004), "A Comparison of Bug Finding Tools for Java". ISSRE '04 Proceedings of the 15th International Symposium on *Software* Reliability Engineering, IEEE, DOI:

10.1109/ISSRE.2004.1.

(41) Craig Larman, UML y Patrones, Epígrafe 38, página 505.

(42) Object Management Group, 2001. *OMG Unified Modeling Language Specification.* www.omg.org.

(43) Botero Tabares, R. Patrones grasp y anti-patrones: Un enfoque orientado a objetos desde la lógica de programación. Entre ciencia e ingeniería, (8):161–173, 2011. Página 22.

(44) Craig Larman, UML y Patrones, Epígrafe 27, página 337.

(45) Object Management Group, 2001. *OMG Unified Modeling Language Specification.* www.omg.org.

(46) Craig Larman, UML y Patrones, Epígrafe 27, página 164.

(47) Pressman, Roger (2002). Ingeniería del *Software*, un enfoque práctico, Oc-Graw Hill página 178, consultado 12 de mayo de 2015.

(48) Pressman, Roger (2002). Ingeniería del *Software*, un enfoque práctico, Oc-Graw Hill página 182, consultado 12 de mayo de 2015.

(49) Mell Peter, Scarfone Karen, Romanosky Sasha. (2007) A Complete Guide to the Common Vulnerability Scoring System Versión 2.0, página 6, consultado 1 de junio de 2015.

(50) Melián Montalvo Marlene, XML el nuevo lenguaje universal. CITMATEL. Consultado 10 de junio de 2015. http://www.bibliociencias.cu/gsdl/collect/eventos/index/assoc/HASH0104/f016d031.dir/doc.pdf.

# Anexo 1

Aplicación del estándar CVSS.

| Vulnerabilidad | *AccessValor* | | *AccessComplexity* | | *Authentication* | | Resultado de la ecuación *Exploitability* = 20* *AccessVector*AccessComplexity*Authentication* |
|---|---|---|---|---|---|---|---|
| | Tipo | Valor | Tipo | Valor | Tipo | Valor | |
| Desbordamiento de *buffer* | Local | 0,5 | Baja | 0,3 | No requerida | 0,0 | 0 |
| Inyección SQL | Local | 0,7 | Baja | 0,5 | No requerida | 0,0 | 0 |
| Denegación de servicio | Local | 0,8 | Alta | 0,6 | Requerida | 0,7 | 6,72 |
| Obtención de información | Local | 0,5 | Baja | 0,2 | Requerida | 0,5 | 1 |
| Obtención de permisos | Local | 0,5 | Baja | 0,2 | No requerida | 0,0 | 0 |

*Tabla 8. Métrica Explotabilidad.*

| Vulnerabilidad | *Confidentiality Impact* | | *Integrity Impact* | | *Availability Impact* | | Resultado de la ecuación *Impact* = 10.41 * ( 1-( 1-*ConfImpact* ) * ( 1-*IntegImpact* ) * (1-*AvailImpact* ) ) |
|---|---|---|---|---|---|---|---|
| | Tipo | Valor | Tipo | Valor | Tipo | Valor | |
| Desbordamiento de *buffer* | Ninguna | 0,0 | Parcial | 0,5 | Ninguna | 0,0 | 10,41 |
| Inyección SQL | Parcial | 0,5 | Parcial | 0,5 | Parcial | 0,5 | 9,10875 |
| Denegación de servicio | Ninguna | 0,0 | Ninguna | 0,0 | No requerida | 0,0 | 10,41 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Obtención de información | Completa | 1,00 | Ninguna | 0,0 | Ninguna | 0,0 | 10,41 |
| Obtención de permisos | Completa | 1,00 | Ninguna | 0,0 | Parcial | 0,5 | 10,41 |

*Tabla 9. Métrica Impacto.*

| Vulnerabilidad | *Impact* | *Exploitability* | Resultado de la fórmula BaseScore6 = round_to_1_decimal( ( ( 0.6*Impact)+ (0.4*Exploitability) –1.5) * f(Impact)) | Clasificación *Warning* >= 6 5<*Alert*<6 3<*Information*<5 |
|---|---|---|---|---|
| Desbordamiento de *buffer* | 10,41 | 0 | 5,581296 | *Alert* |
| Inyección SQL | 9,10875 | 0 | 4,663134 | *Information* |
| Denegación de servicio | 10,41 | 6,72 | 8,742384 | *Warning* |
| Obtención de información | 10,41 | 1 | 6,051696 | *Warning* |
| Obtención de permisos | 10,41 | 0 | 5,581296 | *Alert* |

*Tabla 10. Clasificación según la métrica base.*

# Anexo 2

La interfaz acf-java.cpp es extensión de la arquitectura basada en componente y ejemplo donde se muestra el uso del patrón *Facade* o Fachada.

```
#include <string>
#include <vector>


#include "Java.h"
#include "../Information.h"
#include "../../Enum.h"

extern "C" void exec(string param,  string destiny, GlobalValues* globalValues) {

    Java* java;
    java = new Java(TipoPlugin(1), destiny, param);
    java->Execute(globalValues);

    delete java;
}

extern "C" information getinfo() {

    std::vector<std::string> aux;
    aux.push_back("JAVA");
    aux.push_back("java");
    information info ("libacf-java.so", aux, TipoPlugin(1));

    return info;
}
```

*Ilustración 23. Imagen del código de la clase acf-java.*

**Clase *Plugin***. Padre de la clase Java.

```
/*
 * Plugin.cpp
 *
 *  Created on: 04/08/2013
 *      Author: michel febles parker
 */

#include "Plugin.h"

Plugin::Plugin(TipoPlugin t) {

    this->type = t;
    this->fileManager = new FileManager();

}

Plugin::Plugin(TipoPlugin t, string summarizeDir) {

    this->type = t;

    if(summarizeDir[summarizeDir.length()-1] == '\n')
        summarizeDir = summarizeDir.substr(0, summarizeDir.length()-1);

    this->summarizeDir = summarizeDir;

    this->fileManager = new FileManager();//cambio annareya
    fileManager->createDir(summarizeDir);
}


TipoPlugin Plugin::GetTipo() {
```

*Ilustración 24. Imagen del código de la clase Plugin.cpp.*

# Anexo 3

Diagrama de Secuencia ¨Detectar vulnerabilidades¨

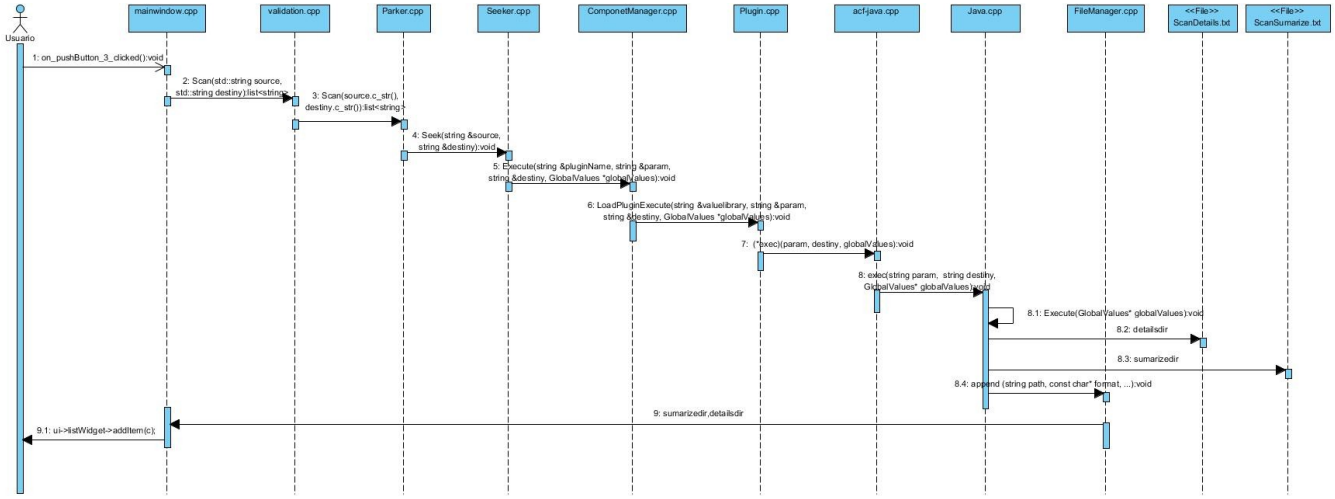


*Ilustración 25. Diagrama de secuencia ¨Detectar vulnerabilidades¨.*

# Anexo 4

Para el análisis del proyecto TesisGra se obtuvo la siguiente respuesta del sistema:

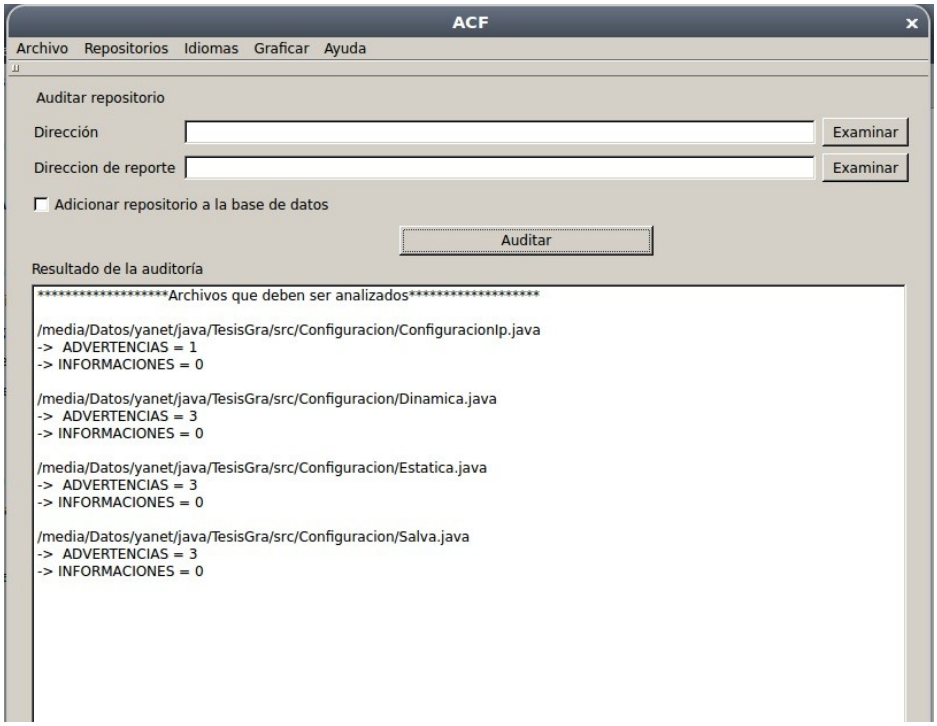


*Ilustración 26. Resultado de la auditoría del fichero TesisGra.*

Resumen detallado de la auditoría al proyecto TesisGra.

```
*----------------| /media/Datos/yanet/java/TesisGra/src/Configuracion/Dinamica.java | ----------------*
ADVERTENCIAS [line: 9] Función vulnerable 'BufferedReader', Esta clase dispone del método flush() para forzar la operación de entrada sal
ADVERTENCIAS [line: 48] Función vulnerable 'BufferedReader', Esta clase dispone del método flush() para forzar la operación de entrada sa
ADVERTENCIAS [line: 48] Función vulnerable 'BufferedReader', Esta clase dispone del método flush() para forzar la operación de entrada sa
->  ADVERTENCIAS = 3
-> INFORMACIONES = 0
*----------------| analizado | ----------------*

*----------------| /media/Datos/yanet/java/TesisGra/src/Configuracion/Dinamica.java | ----------------*
->  ADVERTENCIAS = 0
-> INFORMACIONES = 0
*----------------| analizado | ----------------*
```

*Ilustración 27. Imagen del resumen detallado de la auditoría del fichero TesisGra.*

## Anexo 5

| Nombre de la función | Tipo de Vulnerabilidad a la que pertenece | Clasificación dentro del módulo | Descripción de la función en la clase Java.cpp |
|---|---|---|---|
| JOptionPane.showInputDialog | Denegación de servicio. | *Warning* | Función vulnerable 'JOptionPane.showInputDialog', se debe validar la entrada de los datos por el usuario, podría provocar denegación de servicio o inyección SQL. |
| execute | Inyección SQL. | *Information* | Función vulnerable 'execute', el método execute debería usarse solamente cuando es posible que una sentencia nos devuelva más de un objeto Resultset, más de un *update count* o una combinación de ambos. |
| BufferedReader | *Bypass Something.* | *Warning* | Función vulnerable 'BufferedReader', Esta clase dispone del método *flush* () para forzar la operación de entrada y salida esté o no el *buffer* lleno, puede haber pérdida de información y desbordamiento de *buffer*. |
| java.io.FileInputStream | Desbordamiento de *buffer*. | *Alert* | Función vulnerable 'java.io.FileInputStream' puede conllevar a un desbordamiento de *buffer*. |
| getenv | Obtención de privilegios. Obtención de información. | *Warning* | Función vulnerable 'getenv', se utiliza para acceder a variables de entorno o propiedades del sistema. Cuando se invoca este método, se nos devuelve |

| | | | |
|---|---|---|---|
| | | | información sobre el sistema con que estamos trabajando, esto puede provocar obtención de información del usuario o privilegios. |
| javax.swing | Denegación de servicio. | *Warning* | Función vulnerable 'javax.swing' contiene la clase JTextField, si esta instrucción está establecida en *true*, permite que se pueda escribir sobre el JTextField y tiene el riesgo de ingresarse por este campo una inyección SQL o provocar una denegación parcial de servicio, recomendamos establecer la instrucción en *false* para que no se pueda escribir en el JTextField. |
| connect | Denegación de servicio. | *Warning* | Se esperan conexiones con la función 'connect', puede causar conexiones no deseadas. |
| SWFText | Desbordamiento de *buffer*. | *Alert* | Función vulnerable 'SWFText', puede conllevar a la ejecución de un archivo tipo SWF diseñado para generar un *buffer* overflow. |
| loadLibrary | *Bypass Something.* | *Warning* | Función vulnerable 'loadLibrary', cargar las bibliotecas dinámicas en el sistema cliente, de las clases de tiempo de ejecución o el sistema. |
| ThreadGroup | Denegación de servicio. Obtención de | *Warning* | Función vulnerable 'ThreadGroup', crea o manipular cualquier hilo que no es parte del mismo grupo, como el *applet*. |

| | | | |
|---|---|---|---|
| | información. | | |
| SetSecurityManager | Denegación de servicio.<br>Obtención de información.<br>Obtención de privilegios. | *Warning* | Función vulnerable 'SetSecurityManager', se utiliza para ejecutar un *applet* no firmado con privilegios de administrador. Esto puede provocar denegación de servicio u obtención de información o privilegios. |
| java.lang.invoke.MethodHandles.Lookup | Obtención de privilegios.<br>Denegación de servicio. | *Warning* | Función vulnerable 'java.lang.invoke.MethodHandles.Lookup', no concede correctamente los modos de acceso. Lo que podría permitir a una aplicación JAVA no verificada saltar las restricciones de seguridad impuestas por la *sandbox*. |

*Tabla 11. Funciones vulnerables a detectar.*