UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 1

TRABAJO DE DIPLOMA EN OPCIÓN AL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS

Título: “ Herramienta para la detección de vulnerabilidades en el código fuente del repositorio de GNU/Linux Nova “

Autor:

Adrian Serafín Lamadrid Cuesta

Tutores:

Msc. Alaín Guerrero Enamorado

Ing. Edilberto Blez Deroncele

Ciudad de La Habana, Junio 2012

"Porque esta gran humanidad ha dicho basta y ha echado a andar. Y su marcha, de gigantes, ya no se detendrá hasta conquistar la verdadera independencia, por la que ya han muerto más de una vez inútilmente."

Ché

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas (UCI) los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste, firmamos la presente a los ____ días del mes de _____ del año _____.

Adrian Serafin Lamadrid Cuesta

_____

Firma del Autor

Msc. Alain Guerrero Enamorado

_____

Firma del Tutor

Ing. Edilberto Blez Deroncele

_____

Firma del Tutor

AGRADECIMIENTOS

- *Agradezco a la UCI por el aporte de conocimientos brindado durante estos 5 años de la carrera y por la oportunidad de convertirme en ingeniero en Ciencias Informáticas.*
- *Agradezco a la Revolución, a Fidel y a Raúl por brindarme la opción de estudiar en una universidad de excelencia.*
- *Agradezco a mis familiares por el apoyo incondicional brindado y la confianza depositada.*
- *Agradezco a mis amigos por la ayuda brindada.*
- *Agradezco a los tutores por el apoyo brindado durante todo el proceso de investigación y desarrollo de la tesis.*

DEDICATORIA

A mi familia en general, por ser los principales protagonistas de cada uno de los capítulos de mi vida, a ellos que con su apoyo y confianza me han hecho mejor persona, que con su magia me enseñaron a ver la vida de la mejor forma, que me desmostaron ser las personas más maravillosas que he conocido, esos que enfrentaban el día a día de frente sin temor a nada, con esa fuerza y entereza que nunca dejaban de sorprenderme, por enseñarme a levantarme y aprender de mis errores, por inculcarme valores que siempre llevaré conmigo.

A mi madre primera, mi madre segunda y mi novia, principales pilares de mi pensamiento y únicas culpables de que día tras día me esfuerce por ser mejor persona.

A mi padre fallecido, el mejor hombre que he conocido y conoceré jamás.

Resumen

Con la realización de este trabajo titulado “Herramienta para la detección de vulnerabilidades en el código fuente del repositorio de GNU/Linux Nova”, se pretende elevar la seguridad de cada uno de los procesos que intervienen en la construcción, actualización y configuración de los repositorios del sistema operativo GNU/Linux Nova, a partir de la detección de vulnerabilidades presentes en el código fuente de las aplicaciones que interactúan en dichos repositorios, logrando así que los usuarios finales puedan descargar dichas aplicaciones sin comprometer la seguridad de sus estaciones de trabajo.

En el escenario actual del desarrollo de la distribución de GNU/Linux Nova, se incluyen los procesos de construcción y actualización del repositorio de paquetes, este se construye partiendo del código fuente de las aplicaciones, las cuales son compiladas para obtener los binarios que finalmente conforman dicho repositorio. Un punto neurálgico en el proceso de desarrollo de la distribución es poder detectar de manera automática posibles vulnerabilidades existentes en dicho código fuente. Con vista a lograr dicho objetivo se identificaron las vulnerabilidades tipo: desbordamiento de buffer, desbordamiento de pila, conexiones de red abiertas y ataques de código remoto. Para ello se han desarrollado patrones de búsquedas léxicas, obteniéndose como resultado una herramienta para la detección de vulnerabilidades en el código fuente utilizado para la construcción y actualización de los repositorios de GNU/Linux Nova.

Palabras claves: ataques de código remoto, búsquedas léxicas, conexiones de red, desbordamiento de buffer, desbordamiento de pila, detección de vulnerabilidades, vulnerabilidades en el código fuente.

Índice

Introducción

La Seguridad Informática es una disciplina cuya importancia crece día a día. Aunque la seguridad es un concepto difícil de medir, su influencia afecta directamente a todas las actividades de cualquier entorno informatizado en los que interviene el Ingeniero en Informática, por lo que es considerada de vital importancia.

[Huerta2000] define la seguridad como “una característica de cualquier sistema (informático o no) que nos indica que ese sistema está libre de todo peligro, daño o riesgo y que es, en cierta manera, infalible”... “para el caso de sistemas informáticos, es muy difícil de conseguir (según la mayoría de los expertos, imposible) por lo que se pasa a hablar de confiabilidad[1]”.

[IRAM/ISO/IEC17799] sostiene que “la seguridad de la información protege a ésta de una amplia gama de amenazas, a fin de garantizar la continuidad comercial, minimizar el daño al negocio y maximizar el retorno sobre las inversiones y las oportunidades, protegiendo sus 3 propiedades fundamentales”.

1. **Integridad**: garantiza que se proteja la exactitud y totalidad de los datos y los métodos de procesamiento.

2. **Confidencialidad**: garantiza que la información sea accesible solamente a las personas autorizadas.

3. **Disponibilidad**: garantiza que los usuarios autorizados tengan acceso a la información y a los recursos cuando los necesiten.

[1]Que se puede estar seguro de que funcionará.

[IRAM/ISO/IEC17799] también agrega “La seguridad de la información se logra implementando un conjunto adecuado de controles, que abarca políticas, prácticas, procedimientos, estructuras organizacionales y funciones del software”.

Al analizar detenidamente las anteriores definiciones se puede entender que un sistema informático seguro, tal como un sistema operativo (SO), no es solo aquel que implementa políticas de seguridad durante la fase de desarrollo, sino que contempla durante todo el proceso de concepción del sistema informático políticas que permitirán minimizar posibles vulnerabilidades en el futuro.

En la facultad 1 de la Universidad de las Ciencias Informáticas (UCI) tiene lugar un Centro de Software Libre (CESOL), cuyo objetivo es dirigir y representar a los varios proyectos productivos de sus departamentos. Uno de estos proyectos, ha venido llevando a cabo el desarrollo del sistema operativo “GNU/Linux Nova”, como apoyo a la informatización de la sociedad cubana y al proceso de migración al software libre que experimenta nuestro país.

Nova es una distribución de GNU/Linux desarrollada por estudiantes y profesores de la Universidad de las Ciencias Informáticas, con la participación de miembros de otras instituciones, para apoyar la migración a tecnologías de software libre y código abierto que experimenta Cuba como parte del proceso de informatización de la sociedad.

Durante el desarrollo y mantenimiento del sistema operativo Nova tienen lugar importantes procesos y fases las cuales deben ser aseguradas por el impacto que estas tienen sobre el producto final, así como su incidencia en las características del mismo. Dichos procesos y fases se mantienen constantes en las variadas versiones de Nova, haciendo que cada uno de estos productos sean iguales de inseguros respecto a sus procesos.

Uno de estos procesos es el de construcción y actualización de los repositorios, el cual tiene gran importancia ya que a través de los mismos se le brinda la posibilidad al usuario de interactuar con los programas que desee y que se encuentren disponibles para las diferentes versiones de Nova.

- Nova Escritorio
- Nova Ligero
- Nova Servidores

En el proceso de construcción y actualización de los repositorios tiene lugar otro importante subproceso "El empaquetado[2] de los fuentes". Durante este subproceso el código fuente de las aplicaciones que se desean empaquetar es transformado en un instalador, todo ello a través de mecanismos que automatizan los procedimientos. Luego de estar empaquetadas, sus respectivos paquetes o instaladores son enviados a los repositorios correspondientes, de acuerdo a la categoría de la herramienta desarrollada.

Al apreciar la manera en que se llevan a cabo los procesos relacionados con la construcción de los repositorios, se puede entender que los repositorios son grandes contenedores muy eficientes de aplicaciones desarrolladas y empaquetadas en un formato específico llamado *deb*[3]*.* Sin embargo el proceso de construcción y actualización de repositorios puede ser inseguro siempre que las aplicaciones contenidas en ellos lo sean.

Al hablar de sistema "inseguro" se hace referencia a un sistema no confiable, no auditado, no controlado o mal administrado con respecto a la seguridad. Partiendo de esta situación surge el **problema científico**, el cual plantea: ¿Cómo detectar vulnerabilidades en el código fuente utilizado para la construcción y actualización de los repositorios de GNU/Linux Nova?

[2]Proceso de empaquetar.
[3]Tipo específico de empaquetado para instaladores de linux.

Para darle solución a este problema se ha trazado como **objetivo general** desarrollar una herramienta para la detección de vulnerabilidades en el código fuente utilizado para la construcción y actualización de los repositorios de GNU/Linux Nova. Para lograr darle cumplimiento a dicho objetivo general se ha establecido como **objeto de estudio** las vulnerabilidades en el código fuente y han sido enmarcadas como **campo de acción** las herramientas para la detección de vulnerabilidades en el código fuente utilizado para la construcción de los repositorios de GNU/Linux Nova.

Con este trabajo de diploma se **defiende la idea** de que al diseñar e implementar una herramienta capaz de detectar vulnerabilidades en el código fuente de las aplicaciones, se podrá reducir el número de las mismas.

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

1. Asimilar las herramientas de auditoría de código existentes enfocándose en las más utilizadas actualmente.
2. Diseñar una herramienta que permita la detección de vulnerabilidades en las aplicaciones partiendo de su código fuente.
3. Implementar y probar todas las funcionalidades de la herramienta para la detección de vulnerabilidades en las aplicaciones partiendo de su código fuente.

Partiendo de estos objetivos específicos se han desglosado las siguientes **tareas para la investigación científica:**

1. Análisis de los sistemas que se utilizan para auditar código fuente.
2. Levantamiento de requerimientos funcionales y no funcionales del sistema.

3. Diseño de los módulos necesarios para desarrollar las funcionalidades que den cumplimiento a los requerimientos.
4. Implementación de las funcionalidades utilizando como base las principales vulnerabilidades encontradas en las aplicaciones en los repositorios de Nova.
5. Diseño de pruebas a la herramienta.
6. Realización de pruebas a la herramienta.

El **principal resultado** esperado es la obtención de una herramienta para la detección de vulnerabilidades en el código fuente utilizado para la construcción y actualización de los repositorios de GNU/Linux Nova.

En el proceso investigativo se emplean como **métodos teóricos** el **análisis histórico-lógico** y el **inductivo-deductivo**. El análisis histórico-lógico plantea que los fenómenos no suceden al azar, sino que son producto de un proceso que los origina, motiva o simplemente da lugar a su existencia. Este método fue empleado para la profundización del estudio de la evolución de las aplicaciones que forman parte del objeto de estudio, lo que facilitó la comprensión de soluciones al problema planteado.

El método **inductivo-deductivo** permite llegar a proposiciones generales a partir de hechos aislados que confirman la teoría o a partir de estas teorías arribar a conclusiones sobre casos particulares que se verifican en la práctica. Este método se utiliza con el fin de inferir a partir de las aplicaciones dedicadas a la auditoría de código fuente, la manera en que se efectúan estos chequeos, para llegar a una mejor concepción de la solución que se propone. Se utilizó como **técnica de recogida de información la entrevista**. “La entrevista es una conversación planificada entre el investigador y el entrevistado para obtener información. Su uso constituye un medio para el conocimiento cualitativo de los fenómenos o sobre características personales del entrevistado y puede influir en determinados aspectos de la conducta humana. Por lo que es importante una buena comunicación en vista a lograr resultados positivos”.

Introducción

Se realizaron entrevistas al cliente con el fin de comprender a fondo los procesos que se abordan, lo que contribuyó al perfeccionamiento de la herramienta realizada.

El documento está estructurado en tres capítulos donde se recogen todo el proceso de investigación y desarrollo del sistema propuesto.

- **Capítulo 1. Fundamentación teórica**: Estudio del estado actual de las herramientas y sistemas de auditoría de código fuente. Argumentación sobre las tecnologías a utilizar en el desarrollo.
- **Capítulo 2. Análisis y diseño**: Identificación de las funcionalidades del sistema. Descripción de los componentes de la solución y su estrategia de integración.
- **Capítulo3. Implementación y pruebas**: Implementación de la solución propuesta y diseño y realización de los casos de prueba.

La presente investigación aportará a la distribución GNU/Linux Nova la posibilidad de realizar las muy necesarias auditorías al código fuente de las aplicaciones que se incorporan a este producto y se encuentran en sus repositorios.

Capítulo 1
Fundamentación teórica.

En el presente capítulo se abordan los principales conceptos que son utilizados en el transcurso de la investigación. Se realiza el análisis sobre la actualidad del problema y de las principales herramientas existentes que entran en el entorno del tema a investigar, así como un estudio que incluye varias de las tecnologías actuales para el desarrollo de software y las principales herramientas que serán usadas en la implementación de la solución que se va a desarrollar.

1.1. Conceptos asociados al dominio del problema.

Para una completa noción del alcance de este trabajo científico es necesario el conocimiento y comprensión de algunos conceptos básicos y terminologías relacionadas con el tema. La auditoría de código tiene su origen en la necesidad de detectar malas prácticas de programación en las cuales frecuentemente los programadores de aplicaciones incurren, en ocasiones por desconocimiento y en otras por comodidad y aparente simpleza. La búsqueda de vulnerabilidades a partir del código fuente corresponde a una de las técnicas que se pueden aplicar para lograr una programación segura, la cual a su vez está enfocada en lograr una alta seguridad informática en las aplicaciones.

1.1.1 Seguridad informática.

La seguridad informática es el área de la informática destinada a proteger los recursos informáticos de una entidad o institución. Dichos recursos informáticos se refieren a todo el hardware, software, información,

bases de datos, metadatos[4], archivos y todo lo que la organización valore (activo) y signifique un riesgo si estos llegan a manos de otras personas. Este tipo de información se conoce como confidencial.

1.1.2 Programación segura.

La programación segura es una rama de la programación que estudia la seguridad del código fuente de un software. Su objetivo es encontrar y solucionar los errores de software, esto incluye: utilización de funciones seguras para evitar desbordamientos de pila o desbordamiento de buffer[5] para lograr una shellcode[6], declaración segura de estructuras de datos, control del trabajo con el flujo de datos, análisis profundo de otros errores de software mediante testeos[7] del software en ejecución y creación de parches para los mismos, diseño de parches heurísticos[8] y metaheurísticos para proveer cierto grado de seguridad proactiva[9], utilización de criptografía y otros métodos para evitar que el software sea crakeado[10].

1.1.3 Desbordamiento de pila.

En informática un desbordamiento de pila es un problema aritmético que hace referencia al exceso de flujo de datos almacenados en la pila de una función, esto permite que la dirección de retorno de la pila pueda ser modificada por parte de un atacante para obtener un beneficio propio, que generalmente es malicioso.

[4]Término que se refiere a datos sobre los propios datos.
[5]En informática un buffer es una ubicación de memoria en un disco.
[6]Término en ingles para describir un ataque de código remoto.
[7]Término en ingles para referirse a hacer pruebas.
[8]Viene de heurística: Técnicas que se emplean para reconocer códigos maliciosos.
[9]Término para referirse a la seguridad en profundidad.
[10]Viene de crakear: Crear un parche cuya finalidad es la de modificar el comportamiento del software original y creado sin autorización del desarrollador del programa.

1.1.4 Desbordamiento de buffer.

En seguridad informática y programación un desbordamiento de buffer es un error de software que se produce cuando un programa no controla correctamente la cantidad de datos que se copian sobre un área de memoria reservada. De forma que si dicha cantidad es superior a la capacidad preasignada, los bytes sobrantes se almacenan en zonas de memoria adyacentes, sobrescribiendo su contenido original. Esto constituye un fallo de programación. En las arquitecturas comunes de computadoras no existe separación entre las zonas de memoria dedicadas a datos y las dedicadas a programa, por lo que los bytes que desbordan el buffer podrían grabarse donde antes había instrucciones, lo que implicaría la posibilidad de alterar el flujo del programa, llevándole a realizar operaciones imprevistas por el programador original. Esto es lo que se conoce como una vulnerabilidad. Una vulnerabilidad puede ser aprovechada por un usuario malintencionado para influir en el funcionamiento del sistema. En algunos casos el resultado es la capacidad de conseguir cierto nivel de control saltándose las limitaciones de seguridad habituales. Si el programa con el error en cuestión tiene privilegios especiales constituye en un fallo grave de seguridad.

1.1.5 Ataques de código remoto (shellcode).

Se denomina shellcode al código ejecutable especialmente preparado por el atacante que se copia al host objeto del ataque para obtener privilegios del programa vulnerable. El término shellcode deriva de su propósito general, esto era una porción de un exploit[11] utilizada para obtener una shell. Este es actualmente el propósito más común con que se utilizan. Para crear una shellcode generalmente suele utilizarse un lenguaje de más alto nivel, como es el caso del lenguaje C,C++ o Java, para luego, al ser compilado, generar el código de máquina correspondiente, que es denominado opcode[12].

[11]Es una pieza de software, o una secuencia de comandos con el fin de causar un error o un fallo en alguna aplicación.
[12]Término en ingles para referirse al código de operación, es la porción de una instrucción en lenguaje máquina que especifica la operación a ser realizada.

1.2 Análisis de las principales herramientas existentes.

En los siguientes epígrafes se analizarán detalladamente las herramientas correspondientes al campo de acción, que a su vez constituyen las más utilizadas en el área de la búsqueda de vulnerabilidades en las aplicaciones a partir del código fuente de las mismas.

1.2.1 Herramienta: Flawfinder.

Su primera versión fue la 0.12 y la más actual es la 1.26, diseñadas para las plataformas: GNU/Linux, UNIX, Windows, OS/2 y Mac. Está orientada a analizar código fuente escrito en los lenguajes C y C++, teniendo como principal objetivo la detección de potenciales debilidades de seguridad que se llevan a cabo durante la escritura del código fuente. El programa funciona escaneando el código y buscando el uso de las funciones que tiene en su base de datos de funciones que normalmente se usan mal. Cuando se hace funcionar sobre un directorio que contiene código fuente, se obtiene un informe de los problemas potenciales que ha detectado, ordenados por riesgo (riesgo es un entero de 1 a 5). Para obviar los riesgos menores, es posible decirle al programa que no informe sobre debilidades menores de un nivel de riesgo particular. De forma predefinida, la salida aparecerá en texto plano, pero también hay disponible un informe en HTML. Para facilitar la lectura del informe, se puede indicar que contenga la línea en la que se usa la función, lo que puede ser útil para detectar un problema inmediatamente, ya que de otra forma podría ser imposible. Examina el código fuente buscando fallos de seguridad (de ahí el nombre). Busca funciones con problemas conocidos. Como desbordamientos de buffer (strcpy(), scanf() y otras por el estilo), errores de formato (printf(), syslog()) condiciones de carrera como son (access(), chmod(), entre otros), posibles problemas de uso de meta caracteres en la shell (exec(), system()) y generadores de números aleatorios (random()), la aplicación brindará un informe con todos los posibles errores.

1.2.2 Herramienta: RATS (Rouge Auditing Tool for Security).

RATS o ratas como también se le conoce es una herramienta de auditoría áspera para la búsqueda de ratas de seguridad. Fue escrita en lenguaje C por la empresa Software Seguro Inc, disponible para las plataformas: GNU/Linux, UNIX y Windows Commandline. Sus versiones (desde la primera 0.9, hasta la más actual 2.1), se encuentran bajo la licencia GNU GPL v2. RATS es muy cercana a Flawfinder, con la excepción de que admite un mayor abanico de lenguajes. En la actualidad, tiene soporte para C, C++, Perl, PHP y Python. Ambas herramientas estuvieron desarrolladas simultáneamente. Cuando los desarrolladores respectivos descubrieron la otra herramienta, estos decidieron liberar la primera versión de cada aplicación exactamente el mismo día. RATS es una de las pocas herramientas que soporta los lenguajes de programación Perl, PHP y Python. Para cada vulnerabilidad encontrada imprime una explicación del problema (si está disponible en la base de datos de vulnerabilidades). Finalmente RATS imprime el número de las líneas analizadas y el tiempo que utilizó. La herramienta usa un archivo XML sencillo para leer las vulnerabilidades que es capaz de detectar.

1.2.3 Herramienta: Pscan.

Se trata de un programa específico para ayudar a detectar errores de cadena de formato. La herramienta intentará encontrar incidencias en el uso de funciones variadic[13] en código fuente C y C++, como printf, fprintf y syslog. Los errores de cadena de formato son bastante sencillos de detectar y corregir. A pesar de que sean el tipo más reciente de ataques de software, la mayoría de ellos ya se pueden descubrir y reparar.

[13]Funciones con un número indefinido de argumentos.

1.2.4 Herramienta: ITS4.

ITS4 es una herramienta que pertenece a la sección non-free (no libre) del archivo de Debian y solamente está disponible para la antigua distribución estable. ITS4 se puede usar para buscar potenciales agujeros de seguridad en C y C++, al igual que flawfinder. La salida que produce pretende ser inteligente, ya que no tiene en cuenta algunos de los casos en los que las llamadas a las funciones peligrosas se han hecho con cuidado.

1.2.5 Inconvenientes de las principales herramientas existentes.

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: desbordamientos de buffer, condiciones de carrera y problemas asociados a funciones de impresión con formato. No obstante ninguna de estas herramientas realizan búsquedas de vulnerabilidades asociadas a protocolos de red, tales como la detección de puertos abiertos, creación de conexiones y chequeos de conexiones entrantes, evitando de esta forma ser víctimas de puertas traseras o escaneos[14] de puertos. 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. De igual forma las herramientas antes analizadas no realizan auditorías sobre ficheros comprimidos, lo que se convierte en un gran problema, debido que los paquetes de código fuente almacenados en los repositorios de GNU/Linux Nova se

[14]Término que se emplea para designar la acción de analizar por medio de un programa el estado de los puertos de una máquina conectada a una red de comunicaciones. Detecta si un puerto está abierto, cerrado, o protegido por un cortafuegos.

encuentran en su totalidad comprimidos y se deberían descomprimir manualmente para luego poder aplicarles auditorías de código con dichas herramientas. Por los argumentos antes expuestos y teniendo en cuenta el hecho que muchas de estas aplicaciones generan grandes cantidades de falsos positivos en presencia de vulnerabilidades de bajo impacto para la seguridad de las aplicaciones y que ninguna de las mismas contempla la totalidad de los lenguajes de programación utilizados para la construcción y actualización de los repositorios de GNU/Linux Nova, se puede decir sin temor a equívoco, que las herramientas analizadas no cumplen con las necesidades que dieron lugar a la presente investigación, debido a sus características generales.

tabla 1: Capacidad de detección de vulnerabilidades.

Herramienta.	**Conexiones-red.**	**Overflows.**	**Condición-carrera.**	**Acceso-recursos.**	**Ejecución-procesos.**
Rats	No	Si	Si	No	Si
Flawfinder	No	Si	Si	No	Si
Pscan	No	Si	No	No	No
ITS4	No	Si	No	No	No
Litium	**Si**	**Si**	**Si**	**Si**	**Si**

1.3. Tecnologías a utilizar en el desarrollo de la herramienta.

A continuación se presentan las diferentes tecnologías utilizadas para desarrollar la solución propuesta, dentro de las cuales se encuentran, el lenguaje de programación utilizado, las herramientas de desarrollo seleccionadas y la metodología de desarrollo.

1.3.1. Metodologías de desarrollo de software.

La utilización de metodologías de desarrollo, entiéndase por estas, al conjunto de pasos y procedimientos a seguir que permiten estructurar, planear y controlar el proceso de desarrollo del software, es la que nos permite entre otros elementos el éxito de un proyecto de software [2]. El uso de una metodología define qué hacer, cómo y cuándo durante el planeamiento, desarrollo y mantenimiento de un proyecto. Entre las metodologías más conocidas se pueden mencionar a Extreme Programming (XP), Microsoft Solution Framework (MSF), Open Unified Process en español Proceso Unificado Abierto (OpenUP) y Rational Unified Process (RUP).

1.3.2 OpenUp.

OpenUp/Basic es un framework 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 al programa en análisis con solamente proveer un conjunto simplificado de contenidos, fundamentalmente relacionados con orientación, productos de trabajo, roles y tareas. 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.

1.3.3 Fases de OpenUp

Seguidamente se explican las 4 fases de la metodología OpenUp, mediante las principales actividades que tienen lugar durante las mismas:

1. **Fase de Concepción:**

- Realizar los planteamientos del problema.
- Elaborar la justificación.
- Definir objetivo general y objetivos específicos.
- Obtener toda la información relacionada con el proyecto.
- Capturar las necesidades de los clientes en los objetivos del ciclo de vida para el proyecto.

2. **Fase de Elaboración:**

- Definir los riesgos significativos para la arquitectura.
- Establecer la base para la elaboración de la arquitectura del sistema.

3. **Fase de Construcción:**

- Diseñar, implementar y realizar las pruebas de las funcionalidades para realizar un sistema completo.
- Completar el desarrollo del sistema basado en la arquitectura definida.

4. **Fase de Transición:**

- Asegurar que el sistema sea entregado a los usuarios.
- Evaluar la funcionalidad y escena del último entregable de la fase de construcción.

1.3.4 Herramientas CASE

Las herramientas CASE[15] (Computer Aided Assisted Automated Software Systems Engineering) son utilizadas para automatizar o apoyar una o más fases del proceso de desarrollo de software. Entre las más utilizadas se encuentran Rational Rose y Visual Paradigm, siendo esta última la usada en el desarrollo de la herramienta propuesta.

Visual Parading

Visual Parading es una cómoda herramienta que genera documentación del sistema que se está desarrollando en formato PDF, HTML y Word, también permite la generación de código a partir de diagramas. Esta herramienta CASE puede ser utilizada para el modelado de procesos de desarrollo de software que sigan la filosofía de software libre, lo cual la hace aplicable a la presente investigación. La misma soporta el modelado de negocio, captura de requisitos, además de permitir el control de versiones y ser multiplataforma. Todas estas características conllevaron a que Visual Parading fuese seleccionada como herramienta CASE.

1.3.5 Lenguajes de programación.

Un lenguaje de programación es un idioma artificial diseñado para expresar computaciones que pueden ser llevadas a cabo por máquinas como las computadoras. Es la notación para la descripción precisa de algoritmos o programas informáticos. Son el conjunto de instrucciones que permiten al programador pensar claramente sobre la complejidad del problema a resolver, de manera que pueda ordenarlas convenientemente para la creación de un programa que de solución al problema en cuestión.

[15]Computer Aided Software Engineering / Ingeniería de Software Asistida por Computadora

- **C:** Es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

- **C++:** Es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido. El nombre C++ fue propuesto por Rick Mascitti en el año 1983, cuando el lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Antes se había usado el nombre "C con clases". En C++, la expresión "C++" significa "incremento de C".

1.3.6 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, poder utilizarse para varios. El entorno de desarrollo es imprescindible en la producción de un software. Es donde se definen el conjunto de herramientas, tecnologías y versiones a usar que intervienen en un proceso de desarrollo del software.

NetBeans.

NetBeans IDE es un reconocido entorno de desarrollo integrado disponible para Windows, Mac, Linux y Solaris. El proyecto de NetBeans está formado por un IDE de código abierto y una plataforma de aplicación que permite a los desarrolladores crear con rapidez aplicaciones web, empresariales, de escritorio y móviles utilizando la plataforma Java, así como JavaFX, PHP, JavaScript y Ajax, Ruby y Ruby on Rails, Groovy and Grails y C/C++. El proyecto de NetBeans está apoyado por una comunidad de desarrolladores dinámica y ofrece documentación y recursos de formación exhaustivos, así como una selección diversa de complementos de terceros. NetBeans IDE 6.9 es el segundo IDE después de la aparición de la versión 6.8 en ofrecer compatibilidad para todas las especificaciones de Java EE 6, con compatibilidad mejorada para JSF 2.0/Facelets, Java Persistence 2.0, EJB 3.1 (incluido el uso de EJB en aplicaciones web), servicios web RESTful y GlassFish.

Por las razones anteriormente mencionadas se ha decidido la utilización del IDE Netbeans en su versión 6.9 para el desarrollo de la herramienta propuesta, por las facilidades y comodidades que el mismo brinda.

1.4. Conclusiones parciales.

Como resultado de la investigación y el análisis bibliográfico realizado, a lo largo de este capítulo, han sido expuestos los principales puntos de interés relacionados con la seguridad informática y el análisis de código fuente. Ha sido elaborado el marco teórico que permitirá dar paso al desarrollo de la herramienta y fueron presentadas las principales herramientas actuales utilizadas en este entorno, así como los inconvenientes que conllevaría el uso de las mismas. De igual manera fueron tratados los aspectos relacionados con el objeto de la investigación y sus principales tendencias.

Fundamentación teórica.

De modo general se definió 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:

1. Metodología de desarrollo de software: OpenUp.
2. Lenguaje de programación: C.
3. Entorno de desarrollo: NetBeans 6.9.

Capítulo 2

Análisis y diseño de la herramienta: Novascan.

2.1. Propuesta de solución.

Como solución al problema existente en la detección de vulnerabilidades en el código utilizado para la construcción y actualización de los repositorios de GNU/Linux Nova, se dispone el desarrollo de una herramienta que sea capaz de detectar dichas vulnerabilidades. Como nombre para la misma se ha elegido: Novascan**,** ya que la función principal que desarrollará dicha herramienta será la de escanear todos los ficheros de código fuente de las aplicaciones que integrarán los repositorios de las diferentes distribuciones de Nova, en busca de potenciales debilidades y para ello deberá fragmentar el código fuente de dichos ficheros hasta llegar a estructuras básicas o tokens, tal como hace un analizador léxico o scanner.

En el siguiente epígrafe se exponen las principales características que poseería dicha herramienta, en vista a lograr de esta manera una mejor comprensión de la solución propuesta.

2.2. Requisitos funcionales.

Un requisito funcional define el comportamiento interno del software: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas. Los requisitos[16] funcionales de la herramienta propuesta son los siguientes:

[16]Capacidades o condiciones que el sistema debe poseer.

tabla 2: Requisitos funcionales.

Asignado a	Prioridad	Descripción	Estimación	Estimado por
Adrian Lamadrid	Alta	**R.1** -Detectar vulnerabilidades en el código fuente.	20 días	Adrian Lamadrid
Adrian Lamadrid	Alta	**R.2** -Seleccionar ficheros a revisar.	2 horas	Adrian Lamadrid
Adrian Lamadrid	Alta	**R.3** -Especificar directorio de salidas.	2 horas	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.1**- Determinar lenguaje de programación por la extensión de los archivos.	2 horas	Adrian Lamadrid
Adrian Lamadrid	Media	**R1.2-** Detectar vulnerabilidades en el uso de funciones de impresión con formato.	4 días	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.3-** Detectar vulnerabilidades en el uso de funciones de manejo de buffers.	4 días	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.4-** Detectar vulnerabilidades en la apertura	5 días	Adrian Lamadrid

		de sockets.		
Adrian Lamadrid	Alta	**R1.5-** Detectar vulnerabilidades en la creación de conexiones.	4 días	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.6-** Detectar vulnerabilidades en el uso (lectura y escritura) de recursos críticos del sistema.	2 días	Adrian Lamadrid
Adrian Lamadrid	Baja	**R1.7 –** Detectar vulnerabilidad por la existencia de condiciones de carrera y funciones de redirección.	1 día	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.8-** Detectar Vulnerabilidad en el uso de funciones para llamadas al sistema.	1 día	Adrian Lamadrid
Adrian Lamadrid	Alta	**R1.9-** Descomprimir ficheros.	1 día	Adrian Lamadrid

2.3. Requisitos no funcionales.

Los requisitos no funcionales especifican propiedades o cualidades que el software debe tener. Estos de una forma u otra restringen el entorno del sistema o de la implementación como por ejemplo, interfaz de usuario, rendimiento, entre otros.

2.3.1 Usabilidad.

La herramienta deberá funcionar correctamente ante cualquier usuario que desee revisar el código fuente de una o varias aplicaciones.

2.3.2 Eficiencia.

Las revisiones de cada fichero fuente no deberán exceder los 5 segundos, para de esta manera garantizar una respuesta rápida y eficiente de la herramienta. De igual manera las vulnerabilidades encontradas deben ser informadas en forma de reporte.

2.3.4 Restricciones de diseño e implementación:

1. Utilizar como lenguaje de programación C.
2. El estilo de programación debe ser el especificado en la Guía de estilo para lenguaje C.

2.3.5 Apariencia o interfaz externa:

La herramienta no deberá poseer una interfaz externa, sino que ejecutará las funcionalidades en modo consola, logrando así menos consumo por parte de la misma y manteniendo el aspecto tradicional de la herramientas de auditoría de código fuente, tales como flawfinder, rats, pscan e ITS4.

2.4. Definición de actores y casos de uso del sistema.

Los casos de uso del sistema son parte del análisis, ayudan a describir qué es lo que el sistema debe hacer desde el punto de vista del usuario.

2.4.1 Definición de actores del sistema.

En la tabla 3 se describen los actores del sistema y las actividades que los mismos realizan.

tabla 3: Actores del sistema.

Actores	Justificación.
Usuario	Es la persona con necesidad de revisar el código fuente de una aplicación en busca de vulnerabilidades.

2.4.2 Diagrama de Casos de Uso del Sistema.

El diagrama de casos de uso del sistema, muestra las funcionalidades que brinda la herramienta desarrollada y la forma en que el usuario se relaciona con las mismas. 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. El diagrama de casos de uso del sistema, se puede observar en la ilustración 1 que se muestra seguidamente.

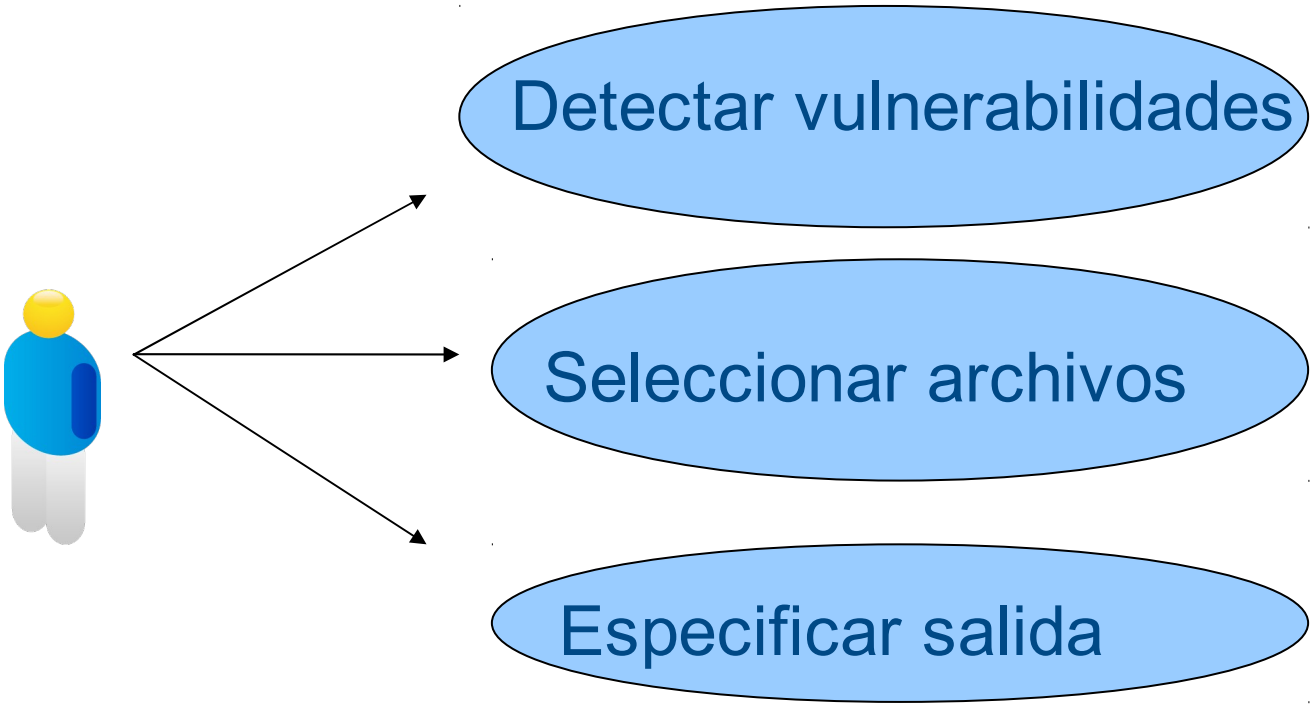


ilustración 1: Diagrama de casos de usos del sistema.

2.5. Descripciones textuales 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 que 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.

tabla 4: Descripción textual del caso de uso "Detectar vulnerabilidades".

Caso de uso #1	Detectar vulnerabilidades.	
Actores	Usuario	
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		
Referencias	R1	
Complejidad	Alta	
Prioridad	Alta	
Poscondiciones		
Flujo Normal de eventos		
#	**Acción del Actor**	**Respuesta del Sistema**
1	El usuario solicita detectar las vulnerabilidades de uno o varios ficheros de códigos fuentes.	2. Para cada fichero la herramienta detecta si el lenguaje de programación en el que están escritos los códigos fuentes por la extensión de los archivos es (Perl,C,C++,Python o Bash), o si es un comprimido.
		3. Detecta vulnerabilidades en el uso de funciones de impresión con formato.
		4. Detecta vulnerabilidades en el uso de funciones de manejo de buffers.

		5. Detecta vulnerabilidades en la apertura de sockets.
		6. Detecta vulnerabilidades en la creación de conexiones.
		7. Detectar vulnerabilidades en el uso (lectura y escritura) de recursos críticos del sistema.
		8. Detecta vulnerabilidad por la existencia de una condición de carrera.
		9. Detecta Vulnerabilidad en el uso de funciones para llamadas al sistema.
		10. La herramienta muestra al usuario un resumen con toda la información sobre las verificaciones realizadas.
Flujo Alterno 1		
	Acción del Actor	Respuesta del Sistema
		2.a.1. Si el fichero no es un comprimido y el código fuente no pertenece a los lenguajes especificados, la herramienta no hace ninguna detección de vulnerabilidades sobre dicho archivo y continua con los demás.
		2.a.2. Si el fichero es un comprimido se activa el módulo de descompresión y luego se va al paso 3 del flujo normal de eventos.

tabla 5: Descripción textual del caso de uso "Seleccionar archivos".

Caso de uso #2	Seleccionar archivos	
Actores	Usuario	
Resumen	El caso de uso inicia cuando el usuario selecciona los ficheros o directorios a los cuales realizará las revisiones de código.	
Precondiciones		
Referencias	R2	
Complejidad	Alta	
Prioridad	Alta	
Poscondiciones		
Flujo Normal de eventos		
#	**Acción del Actor**	**Respuesta del Sistema**
1	El usuario selecciona los ficheros o directorios a los cuales realizará las revisiones de código.	2. La herramienta incluye los ficheros seleccionados en la lista de ficheros a revisar y si se ha seleccionado un directorio, se incluirán en la lista los ficheros contenidos en dicho directorio.

tabla 6: Descripción textual del caso de uso "Especificar salida".

Caso de uso #3	Especificar salida
Actores	Usuario
Resumen	El caso de uso inicia cuando el usuario elige el directorio de salida para los ficheros revisados.
Precondiciones	
Referencias	R3
Complejidad	Alta

Prioridad	Alta	
Poscondiciones		
Flujo Normal de eventos		
#	**Acción del Actor**	**Respuesta del Sistema**
1	El usuario elige el directorio de salida para los ficheros revisados.	2. La herramienta actualizará el directorio de salida para los ficheros revisados, que inicialmente es tmp.

2.6. Diagrama de estructuras de datos.

Este diagrama pretende ilustrar las dependencias y relaciones que tienen los módulos entre sí, por lo que pudiera ser visto como un diagrama similar al diagrama de clases. Partiendo de que el lenguaje de desarrollo seleccionado es el lenguaje C y que el mismo no posee el tipo de dato definido **clase**, cada fichero o cabecera pudiese verse como una estructura de datos similar a una **clase**. Esto es debido a que en estos ficheros, se definen comportamientos (métodos) y propiedades (atributos).

- El fichero **cabecera.h** contiene todas las declaraciones de variables y atributos globales.
- El fichero **auditoria.h** contiene la definición de las diferentes auditorías y búsquedas de vulnerabilidades que se realizarán sobre un fichero, dependiendo del lenguaje de programación en que se encuentre su código fuente. Dichas búsquedas se realizan siempre y cuando este lenguaje esté definido en los requerimientos.
- El fichero **signales.h** provee funcionalidades asociadas al manejo de señales y de errores que se puedan dar durante la ejecución de la herramienta.
- El fichero **descompresor.h** provee las rutinas de descompresión de aquellos ficheros comprimidos, dependiendo del tipo de formato de compresión.

- El fichero **novascan.c** puede ser visto como la clase o fichero principal donde son instanciados los otros ficheros, o incluidos como cabeceras. Este es el encargado de llamar a realizar cada unas de las funcionalidades definidas en los otros ficheros.

Posteriormente se muestra en la ilustración 2 el diagrama de estructuras utilizadas en el desarrollo de la herramienta, en el cual se encuentran representados los elementos antes analizados.

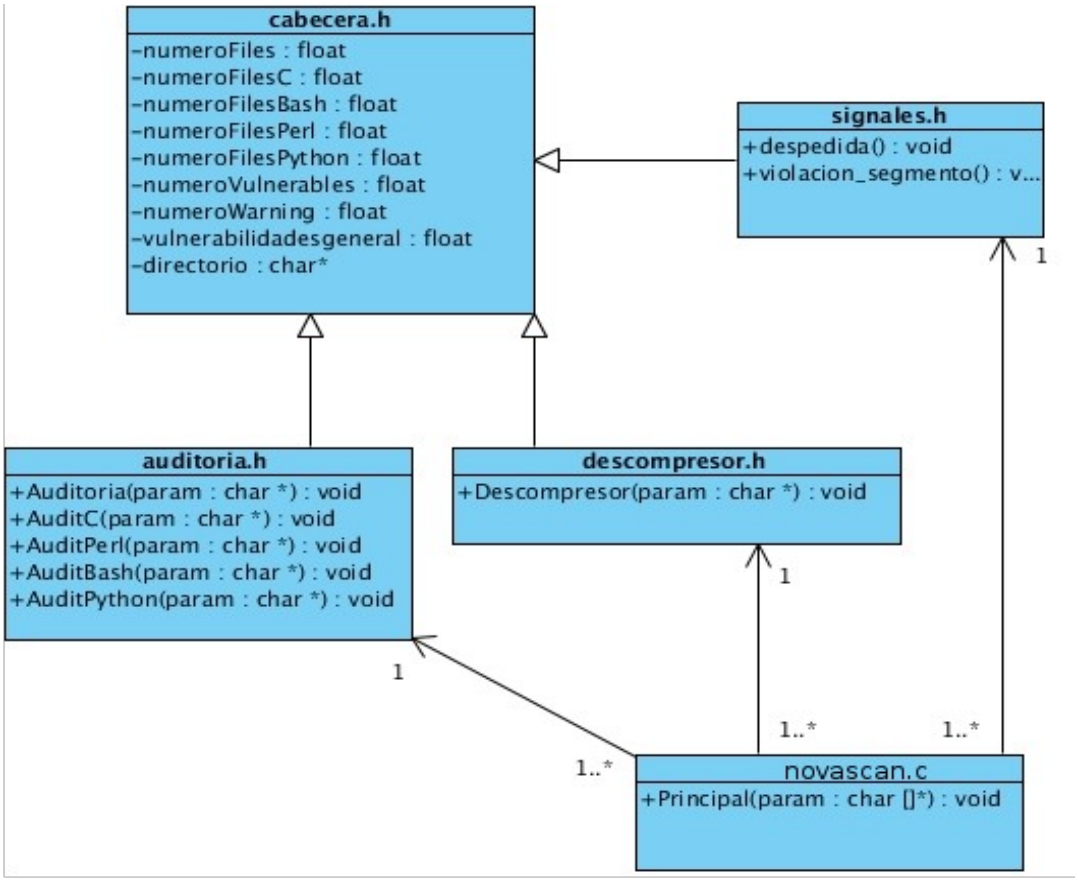


ilustración 2: Diagrama de estructuras de datos.

2.7. Diagramas de interacción.

Un diagrama de interacción consiste en un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar entre ellos. Los diagramas de interacción se utilizan para modelar los aspectos dinámicos de un sistema. Ambos diagramas (secuencia y colaboración) son semánticamente equivalentes. Se puede pasar de uno a otro sin pérdida de información y se utilizan para modelar los aspectos dinámicos de un sistema.

2.7.1 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 por cada uno de los casos de usos. Seguidamente se muestra en la ilustración 3 el diagrama de secuencia del caso de uso Seleccionar ficheros (Ver diagramas 1-3).

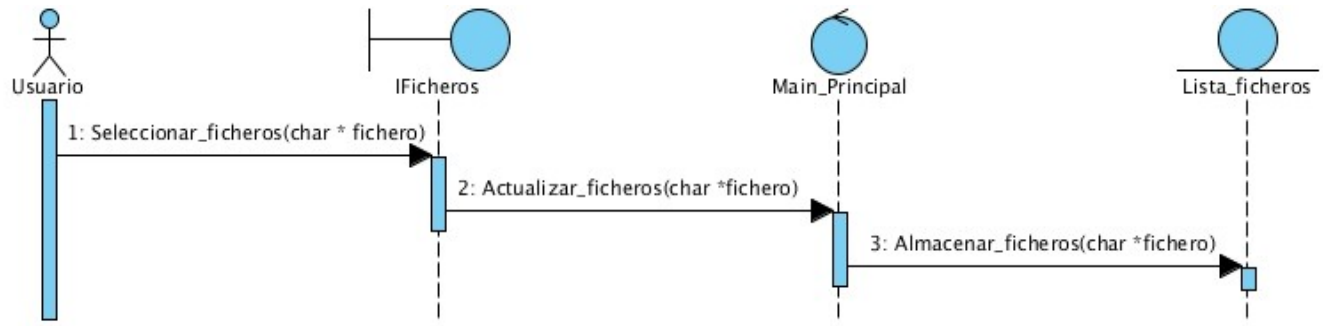


ilustración 3: Diagrama de secuencia del caso de uso "Seleccionar ficheros".

2.8 Integración de la herramienta al proceso de construcción de repositorios.

La herramienta que se propone tendrá un peso fundamental en el proceso de construcción y actualización de los repositorios del sistema operativo GNU/Linux Nova, específicamente en el proceso de compilación de paquetes, pues actualmente en este proceso el código fuente de las aplicaciones que se empaquetan para luego ser compiladas y enviadas a los repositorios, no se revisa, existiendo la posibilidad de enviar aplicaciones vulnerables e inseguras. Dicho proceso se puede observar en su totalidad en la ilustración 4.

Se propone hacer una integración de este proceso con la herramienta propuesta, en vista a lograr que las aplicaciones antes de ser enviadas a los repositorios sean previamente revisadas partiendo de su código fuente y detectar de esta forma las que no deben ser empaquetadas y compiladas hasta corregir las vulnerabilidades encontradas. Para ello es necesario incluir un nueva actividad, la cual se puede ejecutar en ambos lados del proceso. Dicha actividad es la de auditar el código fuente de las aplicaciones, utilizando la herramienta propuesta.

1. El mantenedor de paquetes la ejecutará antes de pasar a empaquetar las aplicaciones, logrando así que la herramienta detecte todas aquellas aplicaciones que no están listas para ser empaquetadas debido a que presentan vulnerabilidades.

2. El revisor de paquetes la ejecutará antes de compilar los paquetes y luego de haber verificado si se ha empaquetado correctamente. La herramienta entonces se encargará de descomprimir los paquetes comprimidos y revisar el código fuente de las aplicaciones, detectando así todas aquellas aplicaciones que no están listas para ser compiladas debido a que presentan vulnerabilidades. Se puede apreciar en el anexo como quedaría el proceso si se le incluye la nueva actividad antes explicada (Ver Anexo).

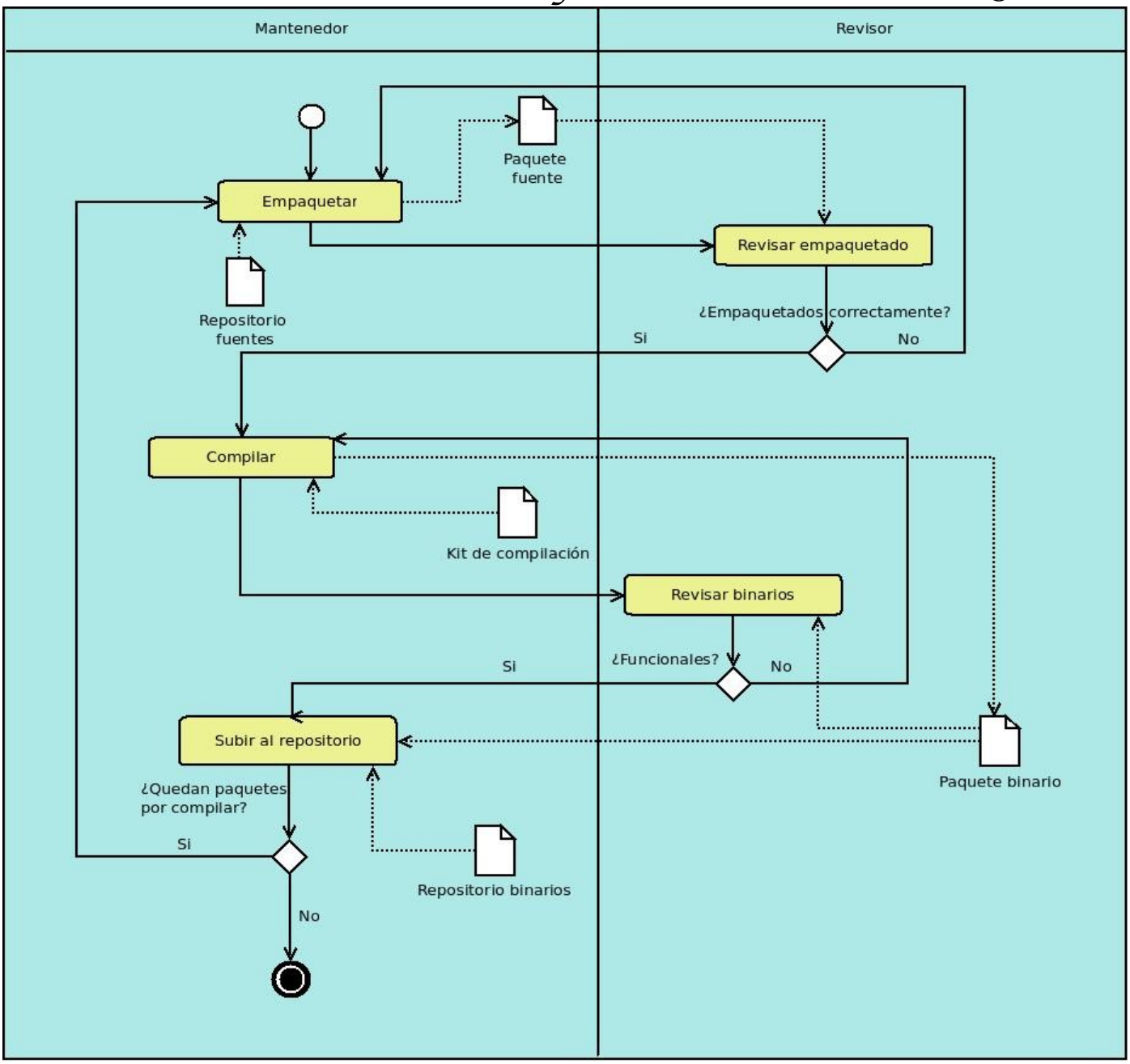


ilustración 4: Actual proceso de compilación de paquetes.

2.9. Conclusiones parciales.

En este capítulo se realizó un estudio de la propuesta de solución a partir del desglose de las principales características y funcionalidades que poseerá la misma. Fueron abordados y detallados los casos de uso del sistema, explicando paso a paso las acciones del actor y las respuestas del sistema, además de definir los requisitos funcionales y no funcionales. De igual forma fue explicada de que forma se podría integrar la herramienta propuesta al proceso de construcción y actualización de los repositorios del sistema operativo GNU/Linux Nova, tomando como escenario el subproceso de compilación de paquetes.

Capítulo 3
Implementación y pruebas.

3.1 Introducción

En este capítulo se realizará la implementación de la herramienta propuesta. Para ilustrar el proceso de desarrollo se muestra el código fuente de los principales métodos que fueron implementados en vista a dar solución a los requerimientos. De igual forma se realizan las pruebas necesarias para validar la solución y correcto funcionamiento de la herramienta.

3.2 Diagrama de componentes.

Los diagramas de componentes describen los elementos físicos del sistema y sus relaciones. Muestran las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan todos los tipos de elementos de software que entran en la fabricación de aplicaciones informáticas. Pueden ser simples archivos, paquetes, bibliotecas cargadas dinámicamente, entre otros.

Se puede usar un diagrama de componentes para describir un diseño que se implemente en cualquier lenguaje o estilo. Solo es necesario identificar los elementos del diseño que interactúan con otros elementos del diseño a través de un conjunto restringido de entradas y salidas. Los componentes pueden tener cualquier escala y pueden estar interconectados de cualquier manera.

En la ilustración 5 se puede observar el diagrama de componentes de la herramienta que se propone.

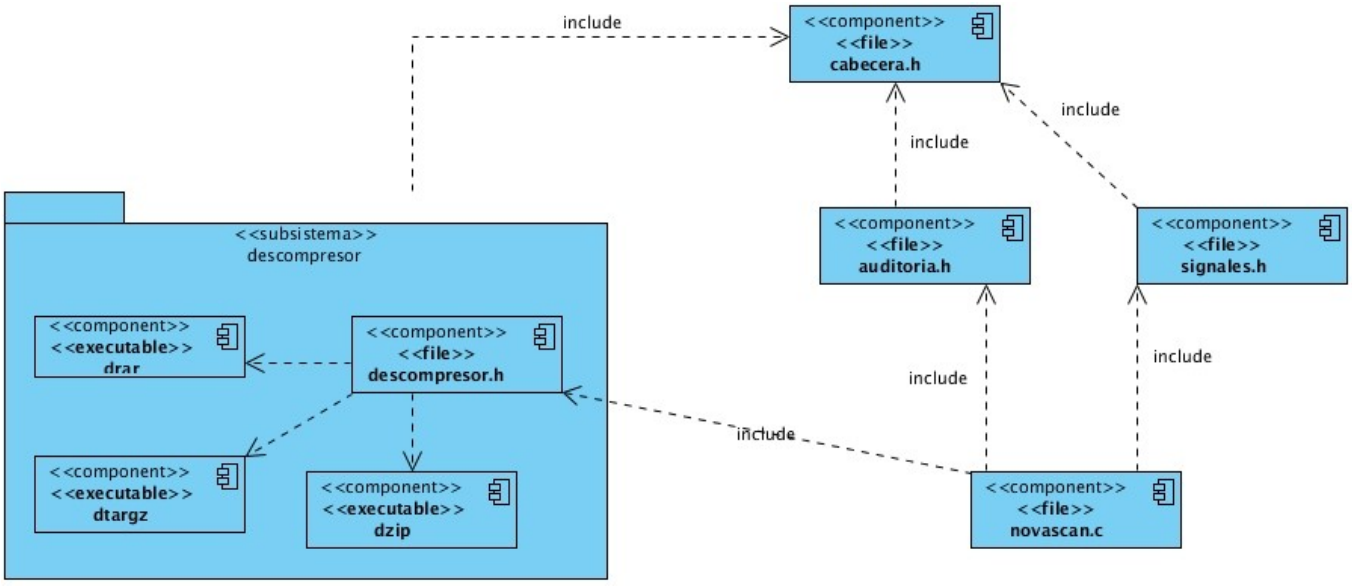


ilustración 5: Diagrama de componentes.

➢ Subsistema “descompresor”: contiene las funcionales necesarias para descomprimir los ficheros comprimidos que se deseen revisar. Para lograr la correcta descompresión, utiliza los ejecutables encargados de extraer los archivos según el tipo de comprimido. Cada ejecutable tendrá como nombre el tipo de comprimido que extrae antecedido por la letra *d,* ejemplo el ejecutable encargado de extraer los rar se llamará drar. La implementación de los restantes ejecutables para extraer los demás tipos de formatos de compresión, pueden ser implementados por el usuario, teniendo siempre en cuenta la regla para el nombre y que dichos ejecutables toman como parámetros, el fichero a descomprimir y la dirección de descompresión.

➢ Componente “drar”: ejecutable encargado de extraer los archivos comprimidos con rar.

➢ Componente “dtargz”: ejecutable encargado de extraer los archivos comprimidos con targz.

➢ Componente “dzip”: ejecutable encargado de extraer los archivos comprimidos con zip.

3.3 Resultados obtenidos.

Para desarrollar la herramienta deseada, se implementaron 4 métodos principales:

1. El método AuditPython, para detectar vulnerabilidades en los ficheros escritos en lenguaje de programación Python.
2. El método AuditC, para detectar vulnerabilidades en los ficheros escritos en los lenguajes de programación C y C++.
3. El método AuditPerl, para detectar vulnerabilidades en el los ficheros escritos en el lenguaje de programación Perl.
4. El método Descompresor, para descomprimir los ficheros comprimidos y colocarlos en el directorio de salida y allí pasar a la búsqueda de vulnerabilidades.

3.3.1 Método AuditPython.

Este método se encarga de recorrer el fichero que recibe como parámetro y que contiene código fuente escrito en lenguaje de programación Python, buscando de esta forma funciones que se usen mal, que se encuentren contraindicadas debido a que pueden generar fallos en casos especiales de ejecución o que pueden provocar desbordamientos de buffers. De igual manera localiza cualquier intento de acceso a recursos críticos del sistema, conexiones que se abran vía sockets, llamadas al sistema, uso de funciones de redirección de páginas web, las cuales pueden ser peligrosas en algunos casos, entre otras vulnerabilidades cuyas detecciones fueron definidas como requerimientos.

Al concluir su ejecución es generado un fichero con el mismo nombre, pero con la extensión bug[17], ubicado en el mismo directorio del fichero que se encuentra en revisión en caso de existir vulnerabilidades y escribe allí toda la información respecto a las mismas. También guarda en un fichero temporal la dirección del fuente vulnerable, para mostrarle al usuario al concluir todas las auditorías; de este modo el usuario podrá conocer cuales de todos los ficheros revisados fueron vulnerables. A continuación se muestra un fragmento del código fuente del método, solo se muestra una parte debido al tamaño del mismo, ya que por cada una de las vulnerabilidades chequeadas, se debe realizar un procedimiento similar al que se ilustra en la imagen siguiente, la cual corresponde a la rutina para determinar si se está haciendo una llamada al sistema a través de la función os.system() (Ver código).

3.3.2 Método AuditC.

Este método se encarga de recorrer el fichero que recibe como parámetro y que contiene código fuente escrito en lenguaje de programación C o C++, buscando de esta forma funciones que se usen mal, que se encuentren contraindicadas o que puedan generar desbordamientos de buffers. De igual manera localiza cualquier intento de acceso a recursos críticos del sistema, conexiones que se abran vía sockets, llamadas al sistema, mal uso de funciones de impresión y de copia a nivel de bytes, entre otras vulnerabilidades cuyas detecciones fueron definidas como requerimientos.

Al concluir su ejecución es generado un fichero con el mismo nombre, pero con la extensión bug, ubicado en el mismo directorio del fichero que se encuentra en revisión, en caso de existir vulnerabilidades y escribe allí toda la información respecto a las mismas. De igual forma guarda en un fichero temporal la dirección del fuente vulnerable, para mostrarle al usuario al concluir todas las auditorías; de este modo el usuario podrá conocer cuales de todos los ficheros revisados fueron vulnerables. A continuación se muestra un fragmento del código fuente del método, solo se muestra una parte debido al tamaño del

[17]Palabra en inglés que representa a un defecto de software.

mismo, ya que por cada una de las vulnerabilidades chequeadas, se debe realizar un procedimiento similar al que se ilustra en la imagen siguiente, la cual corresponde a la rutina para determinar si se está abriendo una conexión a través de la función de C socket() (Ver código).

3.3.3 Método AuditPerl.

Este método se encarga de recorrer el fichero que recibe como parámetro y que contiene código fuente escrito en lenguaje de programación Perl, buscando de esta forma funciones que se usen mal, que se encuentren contraindicadas o que puedan generar desbordamientos de buffers. De igual manera localiza cualquier intento de acceso a recursos críticos del sistema, conexiones que se abran vía sockets, llamadas al sistema, redirecciones de páginas web, condiciones de carreras en algunas funciones, entre otras vulnerabilidades cuyas detecciones fueron definidas como requerimientos.

Al concluir su ejecución es generado un fichero con el mismo nombre, pero con la extensión bug, ubicado en el mismo directorio del fichero que se encuentra en revisión en caso de existir vulnerabilidades y escribe allí toda la información respecto a las mismas. Luego guarda en un fichero temporal la dirección del fuente vulnerable, para mostrarle al usuario al concluir todas las auditorías; de este modo el usuario podrá conocer cuales de todos los ficheros revisados fueron vulnerables. A continuación se muestra un fragmento del código fuente del método, solo se muestra una parte debido al tamaño del mismo, ya que por cada una de las vulnerabilidades chequeadas, se debe realizar un procedimiento similar al que se ilustra en la imagen siguiente, la cual corresponde a la rutina para determinar si se está realizando una redirección de página web a través de la función de Perl, urllib.open() (Ver código).

3.3.4 Método Descompresor.

La función del método Descompresor es la de extraer todos aquellos ficheros que estén comprimidos en cualquiera de los siguientes formatos (rar, ar, zip, tar.gz, jar, tar.bz2). Para ello ejecuta el plugin encargado de descomprimir dicho fichero. El que estará situado en la carpeta plugins ubicada en el mismo directorio de la herramienta. Cada plugin se llamará con el nombre del formato que descomprime antecedido de la letra d, por ejemplo: el plugin que descomprime un rar, se llamará drar y estará ubicado en la carpeta plugins, de esta manera el método Descompresor sabrá como ejecutar cada uno. Estos plugins implementarán la forma en que cada uno de estos ficheros deben ser descomprimidos. Luego de haberlo hecho el método ejecutará las auditorías que correspondan a cada uno de los nuevos ficheros extraídos, que habrán sido creados en la ubicación especificada como directorio de salidas.

3.4 Validación funcional.

La calidad del software debe implementarse a lo largo de todo el ciclo de vida de un software, debe correr paralela desde la planificación del producto hasta la fase de producción de este como actividad de protección. El aspecto a considerar en el Control de la Calidad del Software es la “Prueba de Software”.

3.4.1 Pruebas de software.

Las pruebas de software es un concepto que a menudo, es conocido como verificación y validación. Integra las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del software. Entre algunas de las técnicas que se llevan a cabo para el proceso de prueba se encuentran las técnicas de caja negra y de caja blanca.

- **Prueba de caja negra.**

Cuando se habla de un software, la prueba de caja negra se refiere a las pruebas que se llevan a cabo sobre la interfaz del mismo. Los métodos de prueba de la caja negra se centran en los requisitos funcionales del mismo e intentan encontrar errores de las siguientes categorías: 1- Funciones incorrectas o ausentes. 2- Errores de interfaz. 3- Errores en estructuras de datos o en acceso a bases de datos externas. 4- Errores de rendimiento. 5- Errores de inicialización y terminación.

3.4.2 Validación funcional "Detectar vulnerabilidades".

En la tabla 7 se muestra el caso de prueba para el caso de uso del sistema Detectar vulnerabilidades y se describen las entradas y resultados esperados.

tabla 7: Caso de prueba "Detectar vulnerabilidades".

Caso de uso #1	Detectar vulnerabilidades.
Caso de prueba	Usuario
Entrada	Inicialmente el usuario solicita detectar las vulnerabilidades que se encuentran en el fichero seleccionado.
Resultado esperado	Se muestra un reporte que contiene el número de ficheros revisados, el número de ficheros vulnerables por lenguaje de programación y el promedio de efectividad de la herramienta.

Resultado de la prueba	En caso de que el fichero seleccionado sea un directorio se revisa cada fichero que se encuentra dentro del mismo que se ha seleccionado. En caso de ser un comprimido se descomprime y se coloca en el directorio especificado como salida y allí se le aplica la revisión y en caso de ser un fichero se le aplica la rutina de revisión sencilla. Al terminar de revisar cada uno de los ficheros, se obtiene que el resultado de la prueba fue esperado, mostrando el reporte correspondiente.

3.4.3 Descripción de variables.

En la tabla 8 se describen las variables involucradas en el caso de prueba Detectar vulnerabilidades, así como los valores, clasificación y descripciones de las mismas.

tabla 8: Variables del caso de prueba "Detectar vulnerabilidades".

No	Nombre del campo	Clasificación	Valor Nulo	Descripción
1	Path_fichero	Campo de texto	No	Dirección del fichero que se desea revisar.
2	Path_salida	Campo de texto	No	Dirección de salida para los ficheros revisados.

Seguidamente se muestra en la ilustración 6 un fragmento de la ejecución de la herramienta para el caso de prueba Detectar vulnerabilidades, en la cual se obtiene el reporte final.

```
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4--
dfsg/src/VBox/Runtime/common/string/RTStrNLen.cpp
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/memcmp_alias.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/strchr2_alias.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/memchr.cpp
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4--
dfsg/src/VBox/Runtime/common/string/RTStrNLen.cpp
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/memcmp_alias.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/strchr_alias.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-
dfsg/src/VBox/Runtime/common/string/memchr2.cpp
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//virtualbox-ose-4.0.4-

Numero de files revisadas: 2306
Perl:15 C/C++: 2236, Bash:17, Python:38
Numero de files vulnerables: 199
Perl:4 C/C++: 195, Bash:10, Python:14
Numero de files con warning: 0
Vulnerabilidades generales 852
Promedio de vulnerabilidades 8.629662


Desea ver los fuentes vulnerables ? s/n
```

ilustración 6: Caso de prueba “Detectar vulnerabilidades”.

3.4.4 Validación funcional “Seleccionar archivos”.

En la tabla 9 se muestra el caso de prueba para el caso de uso del sistema Seleccionar archivos y se describen las entradas y resultados esperados.

tabla 9: Caso de prueba "Selecciona archivos".

Caso de uso #2	Seleccionar archivos
Caso de prueba	Usuario
Entrada	El usuario introduce la dirección del fichero o directorio que desea revisar.
Resultado esperado	Se actualiza la la variable que contiene la dirección del fichero que debe ser revisado.
Resultado de la prueba	Se actualiza la la variable que contiene la dirección del fichero que debe ser revisado.

3.4.5 Descripción de variables.

En la tabla 10 se describen las variables involucradas en el caso de prueba Seleccionar archivos, así como los valores, clasificación y descripciones de las mismas.

tabla 10: Variables del caso de prueba "Seleccionar archivos"

No	Nombre del campo	Clasificación	Valor Nulo	Descripción
1	Path_fichero	Campo de texto	No	Dirección del fichero que se desea revisar.

Seguidamente se muestra en la ilustración 7 la ejecución de la herramienta para el caso de prueba, Seleccionar archivos, en la cual se muestra el error que se lanza cuando no se especifican los archivos que se desean revisar.

./novascan

Error en numero de para metros, especifique los ficheros.

….....

ilustración 7: Caso de prueba "Seleccionar archivos".

3.4.6 Validación funcional "Especificar salida".

En la tabla 11 el se muestra el caso de prueba para el caso de uso del sistema Especificar salida y se describen las entradas y resultados esperados.

tabla 11: Caso de prueba "Especificar salida".

Caso de uso #3	Especificar salida.
Caso de prueba	Usuario
Entrada	El usuario introduce la dirección directorio donde desea que se depositen los ficheros revisados.
Resultado esperado	Se actualiza la la variable que contiene la dirección del directorio de salida.
Resultado de la prueba	Se actualiza la la variable que contiene la dirección del directorio de salida.

3.4.7 Descripción de variables.

En la tabla 12 se describen las variables involucradas en el caso de prueba Especificar salida, así como los valores, clasificación y descripciones de las mismas.

tabla 12: Variables del caso de prueba "Especificar salida".

No	Nombre del campo	Clasificación	Valor Nulo	Descripción
1	Path_salida	Campo de texto	No	Dirección de salida para los ficheros descomprimidos.

En las ilustración 8 y 9 se muestran las ejecuciones de la herramienta para el caso de prueba: Especificar directorio de salidas.

Ejecución incorrecta con los parámetros incompletos.

```
./novascan /
Debe especificar directorio de salidas
….....................................................................
```

ilustración 8: Caso de prueba "Especificar directorio de salidas"

Ejecución correcta con los parámetros completos.

```
./novascan / --directory /media/pruebas
….....................................................
```

ilustración 9: Caso de prueba "Especificar directorio de salidas"

3.5 Pruebas reales sobre repositorios.

Las imágenes que se muestran seguidamente corresponden a fragmentos de pruebas realizadas sobre diferentes repositorios de fuentes. Los fuentes se encontraban comprimidos en formato tar.gz, de modo que la herramienta antes de realizar las rutinas de búsquedas de vulnerabilidades, realizó las rutinas correspondientes de descompresión.

Auditoría realizada sobre el repositorio de fuentes de zentyal (100 paquetes de fuentes).

/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/samba-source/winbindd/winbindd.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/samba-source/winbindd/winbindd_proto.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/samba-source/winbindd/idmap_hash/idmap_hash.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/samba-source/winbindd/idmap_adex/idmap_adex.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/include/vscan-fileaccesslog.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/include/vscan-message.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/include/vscan-quarantine.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/include/pstring.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/clamav/vscan-clamav_core.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/clamav/vscan-clamav.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/clamav/vscan-clamav.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/clamav/vscan-clamav_core.h

/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/global/vscan-quarantine.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/global/vscan-filetype.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-vscan-0.3.6cbeta5ebox4/global/vscan-fileaccesslog.c

Numero de files revisadas: 9814
Perl:131 C/C++: 9148, Bash:283, Python:252
Numero de files vulnerables: 678
Perl:9 C/C++: 653, Bash:10, Python:6
Numero de files con warning: 0
Vulnerabilidades generales 2250
Promedio de vulnerabilidades 6.908498

Desea ver los fuentes vulnerables ? s/n

ilustración 10: Prueba realizada.

Rutina de descompresión sobre los fuentes de zentyal.

/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/request.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/receive.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/srvtime.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/negprot.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/search.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/sesssetup.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/nttrans.c

/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/signing.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb/reply.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb_samba3.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/handle.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/session.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb_server.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/smb_server/smb_server.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/tests/bindings.py
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/sam.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/kerberos_heimdal.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/krb5_init_context.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/krb5_init_context.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/gssapi_parse.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/clikrb5.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/kerberos_pac.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/kerberos.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/kerberos/kerberos_util.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/session.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/auth_sam_reply.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/auth.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/samba_server_gensec.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-

3.4.9/source4/auth/credentials/tests/simple.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/tests/bindings.py
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/credentials_ntlm.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/credentials_krb5.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/credentials_files.c
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/credentials_krb5.h
/media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas//samba-3.4.9/source4/auth/credentials/credentials.c

ilustración 11: Prueba realizada.

Auditoría realizada sobre el repositorio de Nova 2010 (900 paquetes de fuentes).

/usr/src/linux-headers-2.6.32-34/sound/pci/ac97/.
/usr/src/linux-headers-2.6.32-34/sound/pci/ymfpci
/usr/src/linux-headers-2.6.32-34/sound/pci/ymfpci/Makefile
/usr/src/linux-headers-2.6.32-34/sound/pci/ymfpci/..
/usr/src/linux-headers-2.6.32-34/sound/pci/ymfpci/.
/usr/src/linux-headers-2.6.32-34/sound/pci/echoaudio
/usr/src/linux-headers-2.6.32-34/sound/pci/echoaudio/Makefile
/usr/src/linux-headers-2.6.32-34/sound/pci/echoaudio/..
/usr/src/linux-headers-2.6.32-34/sound/pci/echoaudio/.
/usr/src/linux-headers-2.6.32-34/sound/parisc
/usr/src/linux-headers-2.6.32-34/sound/parisc/Makefile
/usr/src/linux-headers-2.6.32-34/sound/parisc/..
/usr/src/linux-headers-2.6.32-34/sound/parisc/.
/usr/src/linux-headers-2.6.32-34/sound/parisc/Kconfig
/usr/src/linux-headers-2.6.32-34/sound/.
/usr/src/linux-headers-2.6.32-34/sound/Kconfig
/usr/src/linux-headers-2.6.32-34/sound/synth/emux
/usr/src/linux-headers-2.6.32-34/sound/synth/emux/Makefile
/usr/src/linux-headers-2.6.32-34/sound/synth/emux/..

```
/usr/src/linux-headers-2.6.32-34/sound/synth/emux/.
/usr/src/linux-headers-2.6.32-34/sound/synth/.
Pasando a las auditorias en /media/1b3cc248-1700-4209-8878-6084d32e6d28/probadas/


Numero de files revisadas: 16085
Perl:54 C/C++: 15954, Bash:73, Python:4
Numero de files vulnerables: 77
Perl:0 C/C++: 75, Bash:2, Python:0
Numero de files con warning: 0
Vulnerabilidades generales 131
Promedio de vulnerabilidades 0.478707

Desea ver los fuentes vulnerables ? s/n
```

ilustración 12: Prueba realizada.

3.6 Resultados obtenidos.

Al realizar dicha prueba con las herramientas rats, flawfinder, novascan y pscan se obtuvo que el tiempo promedio de ejecución para el caso de flawfinder (nivel 4) fue de 3 minutos llegando a detectar 514 vulnerabilidades (Ver Anexo 5), entre las cuales muchas eran de bajo impacto para la seguridad de las aplicaciones, mientras que al ser ejecutada la misma herramienta en su nivel 5 solo fueron detectadas 19 vulnerabilidades, en este caso de alto impacto (Ver Anexo 4). Para el caso de novascan el tiempo promedio de ejecución fue de 1 minuto llegando a detectar 131 vulnerabilidades de medio y alto impacto. La herramienta rats culminó con promedio de 1 minuto y 80 vulnerabilidades encontradas y pscan también tuvo promedio de 1 minuto, detectando 34 vulnerabilidades. Al analizar los resultados de la última prueba, se puede apreciar que la herramienta novascan realiza el chequeo de vulnerabilidades en un tiempo aceptable, debido a que no sobrepasa los tiempos de las otras herramientas y detecta un número considerablemente elevado de vulnerabilidades, entre las cuales se encuentran de alto y medio impacto para la seguridad sin que estas signifiquen falsos positivos.

En la siguiente ilustración se muestra el resultado general de la última prueba, la cual fue realizada sobre 900 paquetes de código fuente.

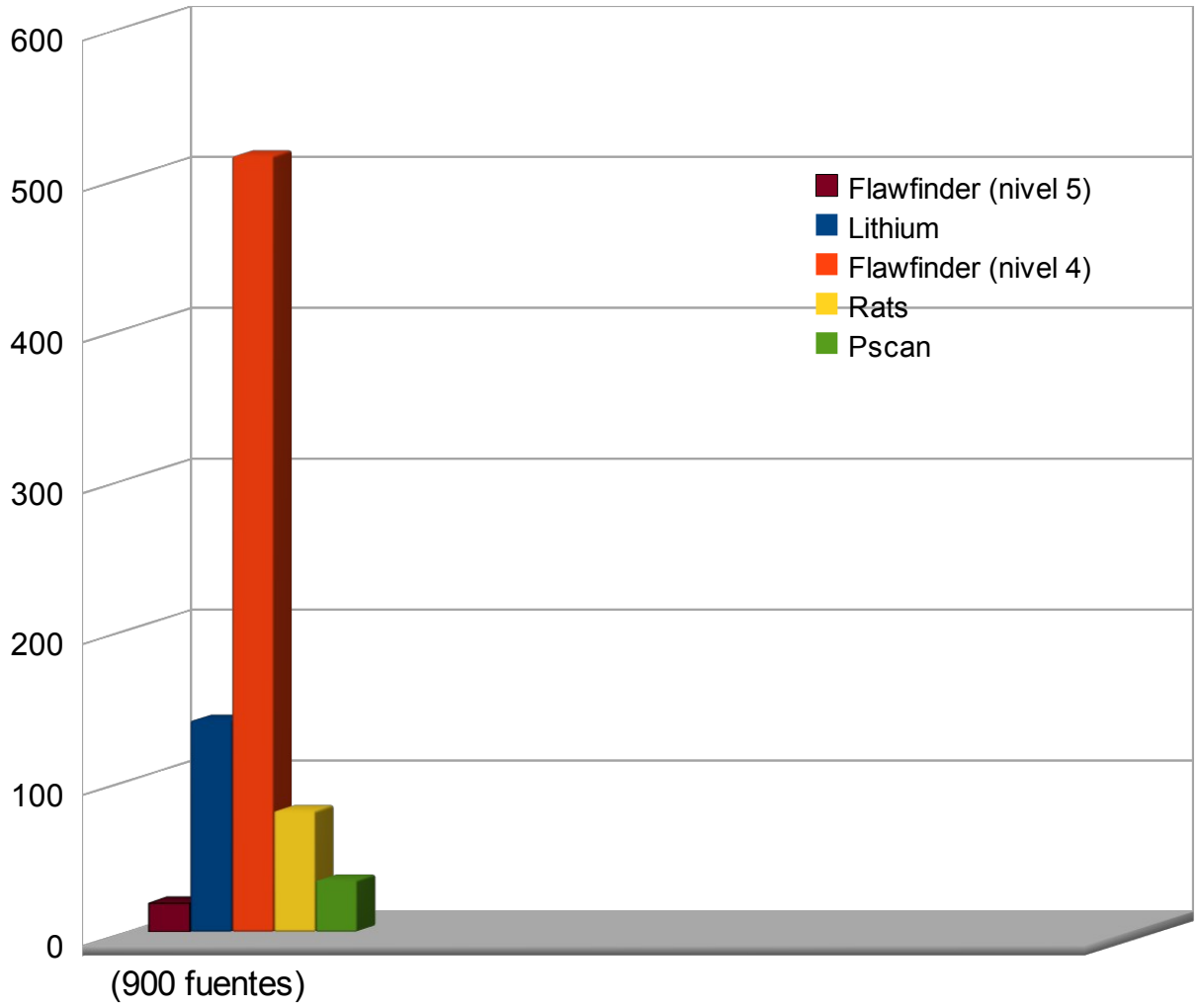


ilustración 13: Resultados de las pruebas.

3.7 Conclusiones parciales

En el presente capítulo se realizó la implementación de la herramienta correspondiente a la solución propuesta. Como mecanismo para ilustrar dicho proceso se mostró parte del código fuente de los métodos fundamentales de la misma. Para realizar la validación de la herramienta se definió usar la técnica de caja negra la cual fue aplicada en el diseño de pruebas, mostrando así las entradas y salidas válidas del sistema según los requerimientos. De igual forma fueron realizadas pruebas reales sobre conjuntos de paquetes de código fuente y repositorios, donde se pudo observar el tiempo de ejecución que se tarda cada herramienta en realizar la búsqueda de vulnerabilidades sobre los mismos, así como el número de vulnerabilidades que dichas herramientas detectan.

CONCLUSIONES

Para el desarrollo de este trabajo de diploma se realizó una investigación asociada a las principales herramientas que realizan auditorías de código, lo cual permitió comprender su funcionamiento y de esta forma diseñar una nueva herramienta que permita detectar vulnerabilidades que las herramientas de auditoría de código existentes no detectan. Se desarrolló y probó dicha herramienta, la cual fue capaz de detectar un mayor número de vulnerabilidades de alto impacto en los paquetes de código fuente almacenados en los repositorios del sistema operativo GNU/Linux Nova, logrando así detectar debilidades en las aplicaciones que hasta el momento no eran detectadas y de esta manera reducir el número de las mismas.

Por lo antes expuesto se concluye que los objetivos propuestos para el desarrollo del presente trabajo se han alcanzado satisfactoriamente. Se incluyen además una serie de recomendaciones que serán útiles para el posterior desarrollo de nuevas funcionalidades y posibles mejoras a las ya existentes.

RECOMENDACIONES

1. Continuar con el estudio de mecanismos para detectar otros tipos de vulnerabilidades en el código fuente.

2. Implementar de un módulo capaz de detectar patrones que coincidan con ataques dirigidos hacia Cuba. Un claro ejemplo de este tipo de ataques es la existencia de aplicaciones que no permiten que los ordenadores ubicados en Cuba, o que realizan peticiones con direcciones ip[18] asignadas a Cuba accedan a servicios determinados, tales como sitios de descargas ubicados en servidores extranjeros, o el propio hecho que algunos de estos sitios descarguen versiones diferentes de aplicaciones, teniendo estas un cifrado diferente (posiblemente vulnerable o con puertas traseras incluidas).

3. Analizar el funcionamiento del software ***Analizador Java Inteligente (AJI),*** el cual es un evaluador y clasificador de código fuente Java, basado en técnicas de inteligencia artificial, en vista a evaluar la posibilidad de incorporar dichas técnicas a la herramienta desarrollada en la presente investigación.

[18]Protocolo de internet.

REFERENCIAS BIBLIOGRÁFICAS

[1] Seguridad Informática. [citado: Noviembre 2011]. Disponible en la dirección web: <http://es.wikipedia.org/wiki/Seguridad_informática>

[2] Metodología de desarrollo de software. [Citado: Noviembre 2010]. Disponible en la dirección web: <http://geeks.ms/blogs/rcorral/archive/2007/01/15/iquest-que-metodolog-iacute-a-de-desarrollo--elegir.aspx>

[3] UML: Casos de Uso. [Citado: Noviembre 2010]. Disponible en la direccion web: <http://www.ingenierosoftware.com/analisisydiseno/casosdeuso.phphttp://www.ingenierosoftware.com/analisisydiseno/casosdeuso.php>

[4] Atacando a Linux – Herramientas para realizar auditorías. [Citado Enero 2012]. Disponible en la dirección web: <http://www.wikilearning.com/tutorial/atacando_linux-herramientas_para_realizar_auditorias/4250-2>

[5] Auditoría de código de aplicaciones. [Citado Enero 2012]. Disponible en la dirección web: <http://es.http://es.wikipedia.org/wiki/Auditoria_de_codigo>

REFERENCIAS BIBLIOGRÁFICAS

[6] Programación segura: Problemas de cadena de formato. [Citado Enero 2012]. Disponible en la dirección web:
<http://www.dragonjar.org/programacion-segura-problemas-de-cadena-de-formato.xhtml>

[7] Programación segura: Desbordamiento de búfer. [Citado Enero 2012]. Disponible en la dirección web:
<http://www.dragonjar.org/programacion-segura-desbordamientos-del-bufer.xhtml>

[8] Ejemplo de auditoría automatizada: flawfinder [citado Febrero 2012]. Disponible en la dirección web:
<http://es.www.debian.org/security/audit/examples/flawfinder.es.html>

[9] Ejemplo de auditoría automatizada: RATS [citado Febrero 2012]. Disponible en la dirección web:
<http://www.debian.org/security/audit/examples/RATS>

[10] Ejemplo de auditoría automatizada: pscan [citado Febrero 2012]. Disponible en la dirección web:
<http://www.debian.org/security/audit/examples/http://www.debian.org/security/audit/examples/pscan>

[11] Unlearning Security [citado Febrero 2012]. Disponible en la dirección web:
<http://unlearningsecurity.blogspot.com/2012/03/explotacion-de-vulnerabilidades-stack.htmlhttp://www.debian.org/security/audit/examples/pscan>

REFERENCIAS BIBLIOGRÁFICAS

[12] Múltiples vulnerabilidades en Python 2.x [citado Febrero 2012]. Disponible en la dirección web:
<http://unaaldia.hispasec.com/2008/08/multiples-vulnerabilidades-en-Python-2x.html>

[13] Inmersión en Python [citado Febrero 2012]. Disponible en la dirección web:
<http://www.rzw.com.ar/seguridad-informatica-2264.html#axzz1uRdE2Lvg>

[14] Alerta 2008-0489 Condición de carrera en Perl [citado Marzo 2012]. Disponible en la dirección web:
<http://www.clcert.cl/show.php?xml=xml/alertas/doc_2008-0489.xml&xsl=xsl/alertas.xsl>

[15] Desbordamiento de búfer en versiones no actualizadas de Python [citado Marzo 2012]. Disponible en la dirección web:
<http://www.rzw.com.ar/seguridad-informatica-1455.html#axzz1uRdE2Lvg>

[16] Boletín de vulnerabilidades [citado Marzo 2012]. Disponible en la dirección web:
<https://www.ccn-cert.cni.es/index.php?option=com_vulnerabilidades&task=view&id=5873&Itemid=96&lang=es>

[17] /dev/null[citado Marzo 2012]. Disponible en la dirección web:
<http://hpantaleev.wordpress.com/tag/vulnerabilidades/>

[18] Como buscar vulnerabilidades en software [citado Marzo 2012]. Disponible en la

dirección web: <http://blog.buguroo.com/?p=344>

[19] Detectar vulnerabilidades en C y C++ con Flawfinder [citado Marzo 2012]. Disponible en la dirección web: <http://blog.buguroo.com/?p=http://backtrackworld.wordpress.com/2008/09/12/detectar-vulnerabilidades-en-c-y-c-con-flawfinder/>

[20] Diagrama de clases [citado Marzo 2012]. Disponible en la dirección web: <http://www.scribd.com/doc/53551175/11/>

[21] Diseño UML [citado Marzo 2012]. Disponible en la dirección web: <http://egdamar877.blogspot.com/2009/05/expocicion.html>

BIBLIOGRAFíA CONSULTADA

- IRAM-ISO-IEC 17799 Tecnología de la información. Código de práctica para la gestión de la seguridad de la información. [Online] [Cited feb 15, 2012]

  http://www.construsur.com.ar/module-Normas-view-nid-7953.html

- Lenguajes-de-programacion.[Online] [Cited:ene 5, 2012.].

  http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml.

- **López, Patricia**. OpenCourseWare. [Online] [Cited: feb 10, 2012]

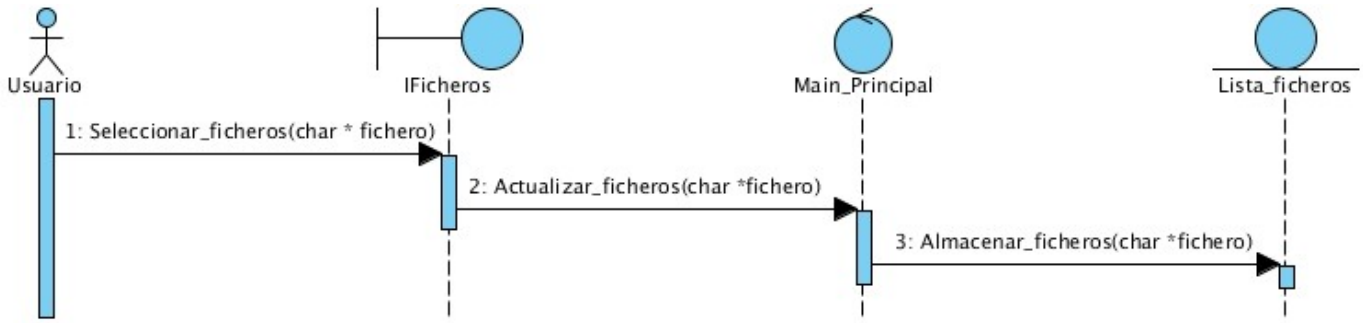
  http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/practicas-1/is1-p01-trans.pdf.

- Netbeans. [Online] [Cited: ene 8, 2012.] http://netbeans.org/index_es.html.

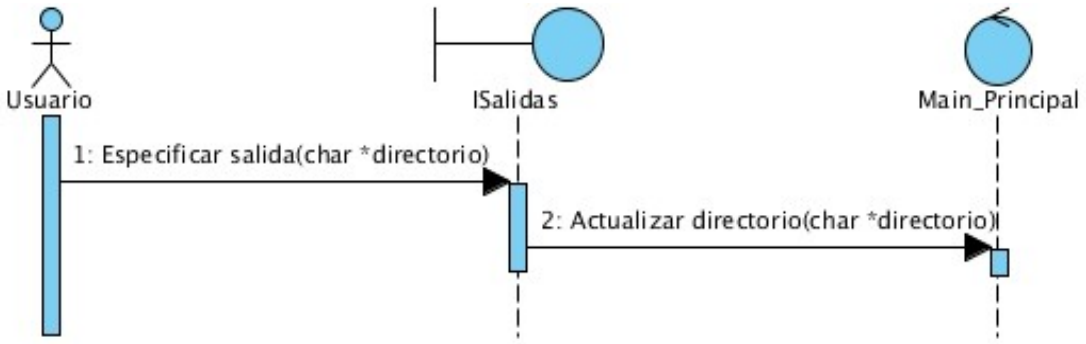
- Visual-paradigm. [Online] [Cited: ene 11, 2012.] http://www.visual-paradigm.com/.

ANEXOS

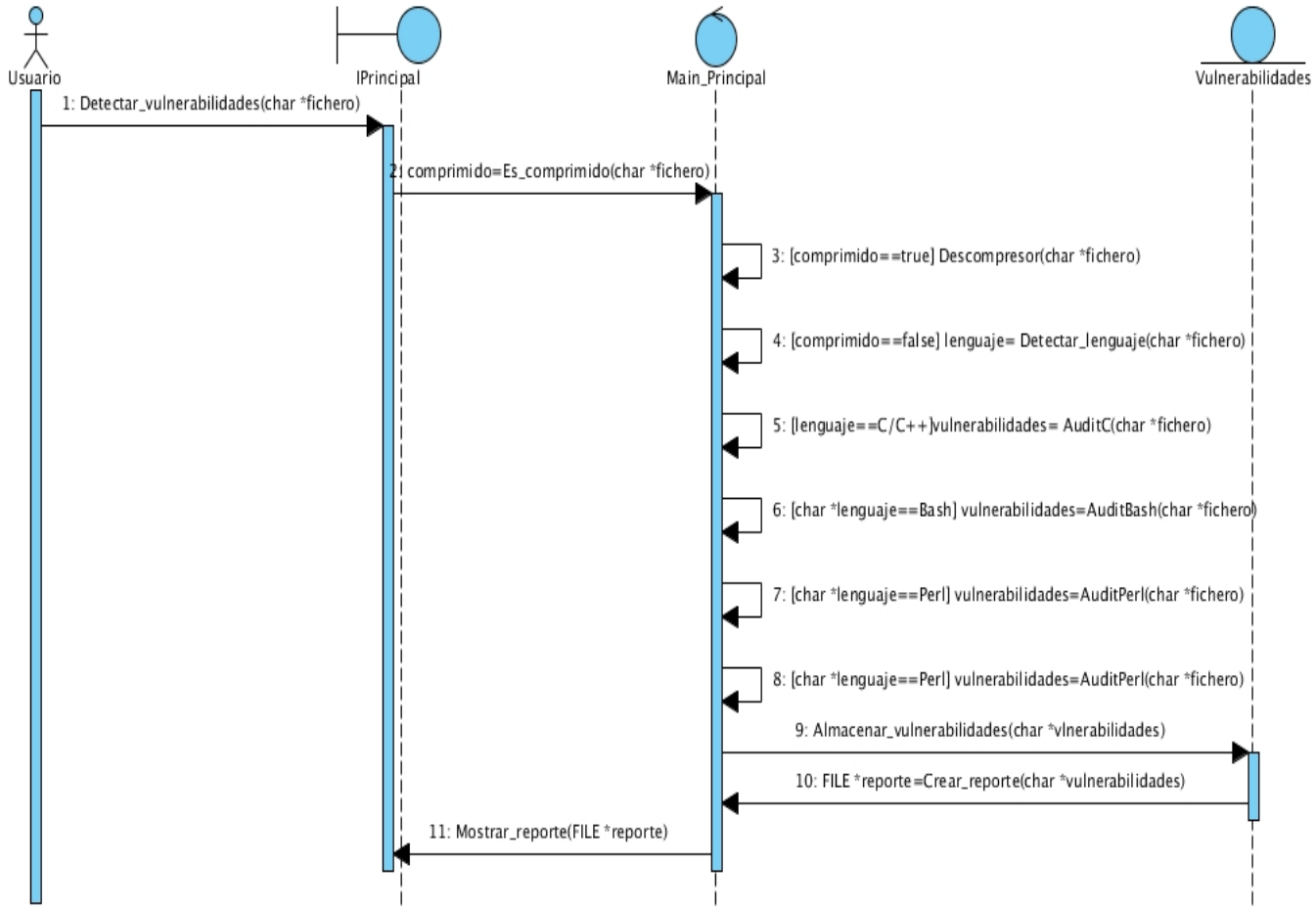
Anexo 1: Diagrama de secuencia “Seleccionar fichero”



Anexo 2: Diagrama de secuencia “Especificar directorio de salidas”



Anexo 3: Diagrama de secuencia “Detectar vulnerabilidades”



Anexo 4: Flawfinder (nivel 5)

e) readlink:
This accepts filename arguments; if an attacker can move those files
or change the link content, a race condition results. Also, it does not
terminate with ASCII NUL. Reconsider approach.
/usr/src/linux-headers-2.6.35-25/arch/powerpc/include/asm/systbl.h:187: [5] (race) chown:
This accepts filename arguments; if an attacker can move those files,
a race condition results.. Use fchown( ) instead.
/usr/src/linux-headers-2.6.35-25/include/linux/fs.h:1527: [5] (race) readlink:
This accepts filename arguments; if an attacker can move those files
or change the link content, a race condition results. Also, it does not
terminate with ASCII NUL. Reconsider approach.
/usr/src/linux-headers-2.6.35-25/include/linux/nfs_xdr.h:1024: [5] (race) readlink:
This accepts filename arguments; if an attacker can move those files
or change the link content, a race condition results. Also, it does not
terminate with ASCII NUL. Reconsider approach.
/usr/src/linux-headers-2.6.35-25/scripts/basic/docproc.c:100: [5] (buffer) strncat:
Easily used incorrectly (e.g., incorrectly computing the correct
maximum size to add). Consider strlcat or automatically resizing strings.
Risk is high; the length parameter appears to be a constant, instead of
computing the number of characters left.
/usr/src/linux-headers-2.6.35-25/scripts/basic/docproc.c:181: [5] (buffer) strncat:
Easily used incorrectly (e.g., incorrectly computing the correct
maximum size to add). Consider strlcat or automatically resizing strings.
Risk is high; the length parameter appears to be a constant, instead of
computing the number of characters left.

Hits = 19
Lines analyzed = 2305959 in 167.56 seconds (13803 lines/second)
Physical Source Lines of Code (SLOC) = 1499920
Hits@level = [0] 0 [1] 0 [2] 0 [3] 0 [4] 0 [5] 19
Hits@level+ = [0+] 19 [1+] 19 [2+] 19 [3+] 19 [4+] 19 [5+] 19
Hits/KSLOC@level+ = [0+] 0.0126673 [1+] 0.0126673 [2+] 0.0126673 [3+] 0.0126673 [4+] 0.0126673

[5+] 0.0126673
Symlinks skipped = 2311 (--allowlink overrides but see doc for security issue)
Dot directories skipped = 2 (--followdotdir overrides)
Minimum risk level = 5

Anexo 5: Flawfinder (nivel 4)

/usr/src/linux-headers-2.6.35-25/scripts/mod/modpost.c:1627: [4] (format) vsnprintf:
If format strings can be influenced by an attacker, they can be
exploited, and note that sprintf variations do not always \0-terminate. Use a
constant for the format specification.
/usr/src/linux-headers-2.6.35-25/scripts/mod/modpost.c:2053: [4] (buffer) sprintf:
Does not check for buffer overflows. Use snprintf or vsnprintf.
/usr/src/linux-headers-2.6.35-25/scripts/mod/modpost.h:99: [4] (format) printf:
If format strings can be influenced by an attacker, they can be
exploited. Use a constant for the format specification.
/usr/src/linux-headers-2.6.35-25/scripts/mod/sumversion.c:318: [4] (buffer) sprintf:
Does not check for buffer overflows. Use snprintf or vsnprintf.
/usr/src/linux-headers-2.6.35-25/scripts/mod/sumversion.c:321: [4] (buffer) sprintf:
Does not check for buffer overflows. Use snprintf or vsnprintf.
/usr/src/linux-headers-2.6.35-25/scripts/mod/sumversion.c:404: [4] (buffer) sprintf:
Does not check for buffer overflows. Use snprintf or vsnprintf.
/usr/src/linux-headers-2.6.35-25/scripts/pnmtologo.c:79: [4] (format) printf:
If format strings can be influenced by an attacker, they can be
exploited. Use a constant for the format specification.
/usr/src/linux-headers-2.6.35-25/scripts/pnmtologo.c:417: [4] (format) vfprintf:
If format strings can be influenced by an attacker, they can be
exploited. Use a constant for the format specification.
/usr/src/virtualbox-ose-3.2.8/vboxdrv/linux/SUPDrv-linux.c:1085: [4] (format) vsnprintf:
If format strings can be influenced by an attacker, they can be
exploited, and note that sprintf variations do not always \0-terminate.

Hits = 514
Lines analyzed = 2305959 in 150.99 seconds (15323 lines/second)

Physical Source Lines of Code (SLOC) = 1499920
Hits@level = [0] 0 [1] 0 [2] 0 [3] 0 [4] 495 [5] 19
Hits@level+ = [0+] 514 [1+] 514 [2+] 514 [3+] 514 [4+] 514 [5+] 19
Hits/KSLOC@level+ = [0+] 0.342685 [1+] 0.342685 [2+] 0.342685 [3+] 0.342685 [4+] 0.342685 [5+] 0.0126673
Symlinks skipped = 2311 (--allowlink overrides but see doc for security issue)
Dot directories skipped = 2 (--followdotdir overrides)
Minimum risk level = 4
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

Anexo 6: Fragmento de código fuente del método AuditPython.

```
/*buscar llamada al sistema*/
int isystem = 0;
char *csystem = "os.system(";
for (i = 0; i < strlen(csystem); i++) {
   if (csystem[i] == partida[i]) {
      isystem++;
   }
}
if (isystem == strlen(csystem) && buscalineasystem == 0) {
   buscalineasystem = 1;

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".bug");
   if (filaregistrada == 0) {
      char tsalida[2020];
      sprintf(tsalida, "%s %s %s", "echo ", param, ">> /tmp/files.vul ");
      system(tsalida);
      filaregistrada = 1;
   }
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "\nLa llamada al sistema 'os.system', puede causar que se ejecuten procesos indeseados.\n", linea);
   fprintf(salida, "%s ", partida);
   fclose(salida);
   vulner = 1;

} else if (buscalineasystem == 1) {

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".bug");
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "%s ", partida);
   fclose(salida);
}
```

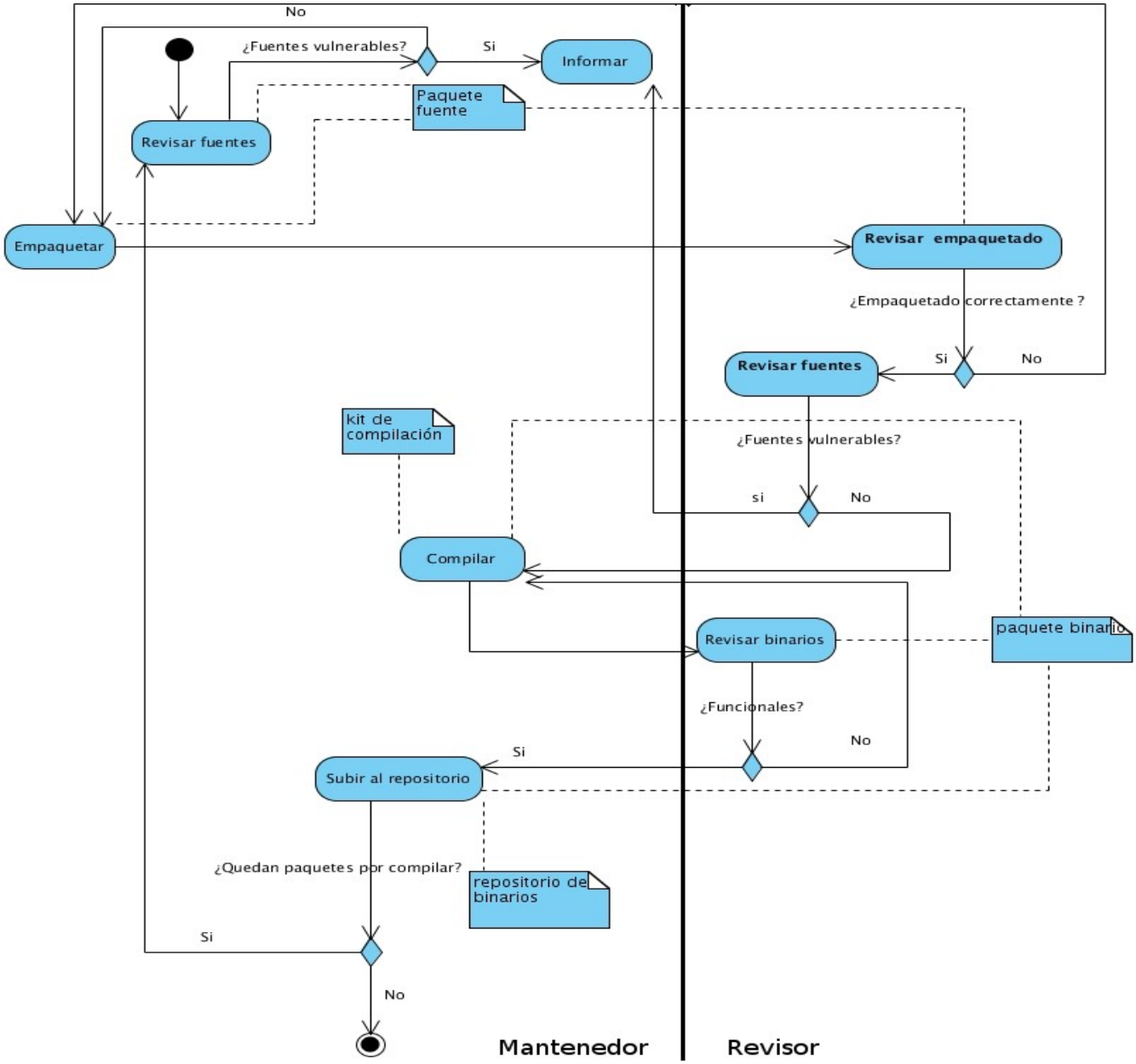
Anexo 7: Fragmento de código fuente del método AuditC

```
/*buscar sockets abiertos*/
int isocket = 0;
char *csocket = "socket(";
for (i = 0; i < strlen(csocket); i++) {
   if (csocket[i] == partida[i]) {
      isocket++;
   }
}
if (isocket == strlen(csocket) && buscalineasocket == 0) {
   buscalineasocket = 1;

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".bug");
   if (filaregistrada == 0) {
      char tsalida[2020];
      sprintf(tsalida, "%s %s %s", "echo ", param, ">> /tmp/files.vul ");
      system(tsalida);
      filaregistrada = 1;
   }
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "\nSe ha abierto una conexion con la funcion 'socket', esto puede causar conexiones no deseadas\n",
   fprintf(salida, "%s ", partida);
   fclose(salida);
   vulner = 1;

} else if (buscalineasocket == 1) {

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".bug");
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "%s ", partida);
   fclose(salida);
}
```

Anexo 8: Fragmento de código fuente del método AuditPerl

```
int iurllib = 0;
char *curllib = "urllib.open(";
for (i = 0; i < strlen(curllib); i++) {
   if (curllib[i] == partida[i]) {
      iurllib++;
   }
}

if (iurllib == strlen(curllib) && buscalinesurl == 0) {
   buscalinesurl = 1;

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".warning");
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "\nWarning : Controle la redireccion de la funcion 'urllib.open'\n");
   fprintf(salida, "%s ", partida);
   fclose(salida);
   //vulner = 1;
   warning = 1;

} else if (buscalinesurl == 1) {

   char nombre[strlen(param) + 2000];
   sprintf(nombre, "%s%s", param, ".warning");
   FILE *salida = fopen(nombre, "a");
   fprintf(salida, "%s ", partida);
   fclose(salida);
}
```

Anexo 9: Proceso de compilación de paquetes.



GLOSARIO DE TÉRMINOS

C

CASE: Computer Aided Software Engineering (Ingeniería de Software Asistida por Ordenador), aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el coste de las mismas en términos de tiempo y de dinero.

Código Fuente: Puede definirse como:

- Un conjunto de líneas que conforman un bloque de texto, escrito según las reglas sintácticas de algún lenguaje de programación destinado a ser legible por humanos.
- Un programa en su forma original, tal y como fue escrito por el programador, no es ejecutable directamente por el computador, debe convertirse en lenguaje de máquina mediante compiladores, ensambladores o intérpretes. Normalmente está destinado a ser traducido a otro código, llamado código objeto, ya sea lenguaje máquina nativo para ser ejecutado por una computadora o código byte para ser ejecutado por un intérprete.

E

Empaquetar: Operación que permite que grupo de ficheros (o uno solamente) se incluyan dentro de otro fichero, ocupando así menos espacio. El empaquetado es similar a la compresión de ficheros. La diferencia entre empaquetado y compresión es la herramienta con que se realiza la operación. Por ejemplo, la herramienta tar se utiliza para empaquetar, mientras que la herramienta zip o gzip -WinZip- se utiliza para comprimir.

GLOSARIO DE TÉRMINOS

I

IDE: Integrate Development Enviroment: Entorno de desarrollo integrado. Herramienta que se usa para facilitar el desarrollo de software.

R

Repositorio: El repositorio podría definirse en el dominio de las herramientas CASE como la base de datos fundamental para el diseño; no solamente guarda datos, sino también algoritmos de diseño y, en general, elementos software necesarios para el trabajo de programación.

S

Shell: Es una aplicación por la cual un usuario puede comunicarse con el sistema operativo en modo texto.

Sistema Operativo: Un sistema operativo es un software de sistema, es decir, un conjunto de programas de computadora destinado a permitir una administración eficaz de sus recursos. Comienza a trabajar cuando se enciende el computador y gestiona el hardware de la máquina desde los niveles básicos, permitiendo también la interacción con el usuario.

Software: Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

M

Metodología: Define quién hace qué, cómo y cuándo.