UNIVERSIDAD CARLOS III DE MADRID
INGENIERÍA INFORMÁTICA

Proyecto Fin de Carrera

Herramienta de análisis estático de código para encontrar vulnerabilidades de seguridad en las aplicaciones Web

29/09/2009

AUTORA:

Joanna Maria Filipiak

TUTOR:

Jose María Sierra Cámara

Agradecimientos:

A José María Sierra Cámara.

A todos los miembros del laboratorio “Evalues”

del Parque Científico-Tecnológico de Leganés.

A Israel Forteza.

A Krystyna,Grzegorz i Agnieszka Filipiak.

Contenido

1 INTRODUCCIÓN 11

1.1 "Estado de Arte" sobre la Seguridad en las Aplicaciones Web 11

1.2 Objetivos Generales del Proyecto 14

1.3 Fases del Proyecto 16

2 ANTECEDENTES 19

2.1 Seguridad en Desarrollos Java 19

2.1.1 Modelo de Seguridad en Java 19

2.1.2 Posibles Ataques y Vulnerabilidades en Java 23

2.1.2.1 Ataque de Confusión de Tipos 23

2.1.2.2 Mal Uso de Serialización 23

2.1.2.3 Cross Site Scripting (XSS) 24

2.1.2.4 Inyecciones (Injection Flaw) 25

2.1.2.5 Ejecución Maliciosa de Ficheros 26

2.1.2.6 Referencia Insegura Directa a Objetos 27

2.1.2.7 Cross Site Request Forgery (CSRF) 28

2.1.2.8 Escape de Información y Manejo Inadecuado de Errores 28

2.1.2.9 Autenticación Quebrada y Manejo de Sesiones 29

2.1.2.10 Almacenamiento Criptográfico Inseguro 29

2.1.2.11 Comunicaciones Inseguras 29

2.1.2.12 Acceso a URL sin Restringir 30

2.2 Análisis Estático 31

2.3 LAPSE 33

2.3.1 Tipo de Vulnerabilidades de Seguridad Encontradas por LAPSE 33

2.3.1.1 Ataques de Inyección de Datos Maliciosos en las Aplicaciones Web 34

2.3.1.2 Ataques de Manipulación de Aplicaciones Web con Datos Maliciosos 35

2.3.2 Funcionamiento de LAPSE 36

2.4 PQL – *Program Query Language* 38

2.5 Bddbddb y Datalog 43

2.6 Otros Analizadores Estáticos 47

2.6.1 ITS4 47

2.6.2 Pscan 47

2.6.3 RATS - Rough Auditing Tool for Security 48

2.6.4 FlawFinder 48

2.6.5 BOON 49

2.6.6 MOPS 49

2.6.7 CQual 49

2.6.8 Splint 50

3 IMPLANTACIÓN DE LAPSE EN ECLIPSE 51

3.1 Arquitectura Eclipse Plug-In 52

3.2 Eclipse Java Model 56

3.3 Eclipse Abstract Syntax Tree (AST) 57

3.4 Motor de Búsqueda de Eclipse 59

3.5 Implementación de las Vistas Principales 61

3.5.1 SourceView 62

3.5.2 SinkView 66

3.5.3 Provenenance Tracker 70

4 CONSTRUCCIÓN DE UN PLUG-IN MEJORADO. LAPSE+ 74

4.1 Detección de la Vulnerabilidad “Path Traversal” 75

4.2 Reconocimiento de Diversos Tipos de Expresión “sink” 75

4.3 Inclusión de los Operadores de Derivación 79

5 DETECCIÓN DE LAS VULNERABILIDADES EN LAPSE+ 81

5.1 Fuentes de Datos Maliciosos 82

5.2 Cross Site Scripting (XSS) 85
5.3 Inyección SQL 88
5.4 Ejecución Maliciosa de Ficheros (“Path Traversal”) 91
5.5 Inyección de Comandos 94
6 INCLUSIÓN DE NUEVAS VULNERABILIDADES EN LAPSE+ 96
6.1 Procedimiento de Inclusión de las Nuevas Vulnerabilidades 96
6.2 Especificación de las Nuevas Vulnerabilidades 98
6.2.1 Inyección XPath 99
6.2.1.1 Descripción de la Vulnerabilidad 99
6.2.1.2 Especificación de Inyección XPath en LAPSE 101
6.2.1.3 Detección de Inyección XPath en las Aplicaciones 104
6.2.2 Inyección XML 108
6.2.2.1 Descripción de la Inyección XML 108
6.2.2.2 Especificación de Inyección XML en LAPSE 109
6.2.2.3 Detección de Inyección XML en las Aplicaciones 110
7 COMPARATIVA LAPSE y LAPSE+ 113
8 CONCLUSIONES 116
8.1 Resumen del Trabajo Realizado 116
8.2 Futuras Líneas de Desarrollo 117
8.3 Observaciones Finales 118
9 BIBLIOGRAFÍA 119
ANEXO A: PRESUPUESTO 121
A.1. Recursos 121
A.1.1 Recursos Materiales 121
A.1.1.1. Recursos Software 122
A.1.1.2. Recursos Humanos 122
A.2. Resumen de Costes del Proyecto 123

ANEXO B: MANUAL DE USUARIO LAPSE+ 124
B.1. Instalación de LAPSE+ 124
B.2. Uso de LAPSE+ 124
B.1.1 Uso de la vista “Vulnerabilities Sources” 126
B.2.1 Uso de la Vista “Vulnerability Sinks” 127
B.3.1 Uso de la vista “Provenance Tracker” 129
GLOSARIO DE TERMINOS 132

Índice de Listados de Código

Listado 1: Código Vulnerable al Ataque de “Confusión de Tipos” 23
Listado 2: Código Java Vulnerable al Ataque XSS 25
Listado 3: Ataque XSS - Ejemplo del Script Malicioso....................................25
Listado 4: Ejemplo de Script de Addición de Cookie 25
Listado 5: Código Java Vulnerable al Ataque de “Inyección SQL “ 26
Listado 6: Código Java Vulnerable al Ataque de “Inyección de Comandos” 26
Listado 7: Código Java Vulnerable al Ataque de “Path Traversal” 27
Listado 8: Código Vulnerable al ataque de “Referencia Insegura Directa De Objetos”....................................27
Listado 9: Especificación BNF de Gramática de PQL....................................41
Listado 10: El código Java Vulnerable a la “Inyección SQL” 42
Listado 11: La Vulnerabilidad “Inyección SQL” Expresada en PQL 42
Listado 12: Contenido del Fichero “plug-in.xml”....................................54
Listado 13: Fichero “sources.dtd”....................................62
Listado 14: Fichero "safes.dtd"....................................65
Listado 15: Fichero "sinks.dtd" 66
Listado 16: Fichero "derived.dtd" 79
Listado 17: Consulta "main" de las vulnerabilidades detectadas por LAPSE....................................81
Listado 18: Consulta de derivación desde el objeto "source" a "sink" 82
Listado 19: Consulta PQL para buscar “Manipulación de Parámetros”....................................83
Listado 20: Consulta PQL para buscar “Manipulaciones de Cabeceras” 83
Listado 21: Consulta PQL para buscar “Manipulaciones de Cookies”....................................84
Listado 22: Consulta PQL para buscar “Escape de Información”....................................84
Listado 23: Consulta PQL "sink" para XSS....................................85
Listado 24: Consulta PQL "sink" para “Inyección SQL” 88
Listado 25: Consulta PQL "sink" para “Path Traversal” 91
Listado 26: Consulta PQL "sink" para “Inyección de Comandos” 94
Listado 27: Fichero "usuarios.xml" 99
Listado 28: Consulta XPath....................................100
Listado 29: Fragmento del código vulnerable a la “Inyección XPath”....................................100
Listado 30: Vector para el ataque de “Inyección XPath”....................................100
Listado 31: Consulta PQL "sink" para la “Inyección XPath”....................................102
Listado 32: Actualización de fichero "sink.xml", con la “Inyección XPath”....................................103
Listado 33: Vector de ataque de la “Inyección XML” 108
Listado 34: Consulta PQL sink() para la “Inyección XML” 109
Listado 35: Actualización de fichero "sink.xml", con la “Inyección XML” 109

Índice de Ilustraciones

Ilustración 1: El proceso de desarrollo de software en las organizaciones. Ilustración tomada de [27] 12
Ilustración 2: Fases de Un Ciclo de Vida de Software. Ilustración tomada de [18] 12
Ilustración 3: Arquitectura básica de seguridad en Java 2. Ilustración tomada de [15] 21
Ilustración 4: Diagrama UML del Patrón Visitor (Ilustración Tomada de Wikipedia) 58
Ilustración 5: Diagrama de Clases UML del LAPSE 61
Ilustración 6: Menú Principal de la Vista “Source View” 64
Ilustración 7: Menú Contextual de la Vista “Source View” 64
Ilustración 8: Menú Principal de la Vista "Sink View" 67
Ilustración 9: Menú Contextual de la Vista "Sink View" 68
Ilustración 10: Cuadro de Estadísticas de los Objetos "sink" Encontrados 69
Ilustración 11: La Ventana con Propiedades Modificable de la Vista “Provenance Tracker” 72
Ilustración 12: Menú Contextual de la Vista "Provenance Tracker" 73
Ilustración 13: Menú Principal de la Vista “Provenance Tracker” 73
Ilustración 14: Diagrama de Clases UML de LAPSE+ 74
Ilustración 15: Error de Validación de Expresión "sink" 76
Ilustración 16: Resultado de Mejora del Análisis de Expresiones de Concatenación 77
Ilustración 17: Resultado de Mejorar el Análisis de Expresiones de Concatenación (2) 77
Ilustración 18: Propagación desde una llamada "sink", con invocación de un método como párametro 78
Ilustración 19: Resultado de detección del operador de derivación 80
Ilustración 20: : Resultado de propagación entre el objeto "sink" y objeto "source" para la vulnerabilidad de “Cross Site Scripting” 86
Ilustración 21: Resultado de búsqueda de objetos "sinks" para XSS 87
Ilustración 22: Resultado de búsqueda de objetos "source" para XSS 87
Ilustración 23: Resultado de propagación entre el objeto "sink" y objeto "source" para la vulnerabilidad de “Inyección SQL” 89
Ilustración 24: Resultado de búsqueda de objetos "source" para “Inyección SQL” 90
Ilustración 25: Resultado de búsqueda de objetos "sink" para “Inyección SQL” 90
Ilustración 26: Resultado de propagación entre el objeto "sink" y objeto "source" para “Path Traversal” 92
Ilustración 27: Resultado de búsqueda de objetos "source" para “Path Traversal” 93
Ilustración 28: Resultado de búsqueda de objetos "sink" para “Path Traversal” 93
Ilustración 29: Resultado de propagación entre el objeto "sink" y objeto "source" para “Inyección de Comandos” 95
Ilustración 30: Resultado de búsqueda de objetos "source" para “Inyección de Comandos” 95
Ilustración 31: Resultado de búsqueda de objetos "sink" para “Inyección de Comandos” 95
Ilustración 32: Resultado de búsqueda de objetos "sink" para “Inyección XPath” 104

Ilustración 33: Resultado de búsqueda de objetos "source" para “Inyección XPath”..................105
Ilustración 34: Resultado de propagación para la vulnerabilidad " Inyección XPath "................105
Ilustración 35: Resultado de propagación para la vulnerabilidad " Inyección XPath " (2) 106
Ilustración 36: Resultado de propagación para la vulnerabilidad " Inyección XPath " (3) 107
Ilustración 37: Resultado de búsqueda de objetos "sink" para “Inyección XML”........................110
Ilustración 38: Resultado de búsqueda de objetos "source" para “Inyección XML” 111
Ilustración 39: Resultado de propagación para la vulnerabilidad "Inyección XML" (1)...............112
Ilustración 40: Resultado de propagación para la vulnerabilidad "Inyección XML" (2)...............112
Ilustración 41: Pasos para abrir LAPSE+...124
Ilustración 42: Cuadro de dialogo para abrir vistas LAPSE+..125
Ilustración 43: Vistas LAPSE+ abiertas en eclipse 125
Ilustración 44: Vista "Vulnerability Source" 126
Ilustración 45: Menú principal de la vista " Vulnerability Sources"...126
Ilustración 46: Vista “Vulnerability Sinks” 127
Ilustración 47: Menú principal de la vista " Vulnerability Sinks" 127
Ilustración 48: Estadísticas sobre objetos "sinks" 128
Ilustración 49: Menú contextual de la vista "Vulnerability Sinks" 128
Ilustración 50: Vista "Provenance Tracker"...129
Ilustración 51: La ventana de propiedades modificable de la vista “Provenance Tracker” 130
Ilustración 52: Menú Contextual de la vista "Provenance Tracker" 131
Ilustración 53: Menú Principal de la vista “Provenance Tracker”..131

Índice de Tablas

Tabla 1: Descripción de la Fase 1 del PFC....16
Tabla 2: Descripción de la Fase 2 del PFC....16
Tabla 3: Descripción de la Fase 3 del PFC....17
Tabla 4: Descripción de la Fase 4 del PFC....17
Tabla 5: Descripción de la Fase 5 del PFC....17
Tabla 6: Descripción de la Fase 6 del PFC....17
Tabla 7: Descripción de la Fase 7 del PFC....18
Tabla 8: Descripción de la Fase 8 del PFC....18
Tabla 9: Elementos Básicos de Eclipse Java Model....56
Tabla 10: Número de descriptores "sink", "source" y "derived" utilizados en las pruebas realizadas con LAPSE y LAPSE+ 113
Tabla 11: Aplicaciones Web utilizadas en las pruebas con LAPSE y LAPSE+....114
Tabla 12: Número de vulnerabilidades encontradas por LAPSE y LAPSE+ 114
Tabla 13: Software utilizado para realización del proyecto....122
Tabla 14: Recursos Humanos utilizados en este proyecto....123
Tabla 15: Coste total de este proyecto 123
Tabla 16: Glosario de términos 134

1 INTRODUCCIÓN

En esta sección se analizan las últimas tendencias en la seguridad de entornos Web. Por otro lado se establecen también los objetivos principales de este PFC.

1.1 "Estado de Arte" sobre la Seguridad en las Aplicaciones Web

Las aplicaciones web se han convertido en los últimos años en un apoyo crucial en casi todas las líneas de negocio. Quedan muy pocas cosas que no se puedan hacer en Internet. Conceptos como comercio electrónico, redes sociales, servicios Web, educación on-line, etc. dominan el mundo de las nuevas tecnologías. Los datos manejados por las aplicaciones web también cada vez son más sensibles. Datos personales, financieros, médicos necesitan una protección muy alta. De ahí que la seguridad en las aplicaciones web sea cada vez más importante.

Según el último informe sobre seguridad de CS/FBI, "2008 Computer Crime and Security Survey" [27], los ataques contra las aplicaciones Web ocurren cada vez más a menudo. A pesar de que en los últimos años su frecuencia había bajado, desde hace dos años cada vez ocurren más abusos en las aplicaciones Web. Se repiten ataques a gran escala, como el de la intrusión en una base de datos de la Universidad de California, ocurrido este año [9]. Los atacantes consiguieron acceder a una base de datos sobre salud y robar la información personal de hasta 160.000 estudiantes. Lo peor es que no se descubrió la intrusión hasta pasado medio año, el periodo durante el cual los atacantes tenían vía libre de acceso a la base de datos.

Por otro lado, el informe de CSI/FBI indica que cada vez existe más preocupación sobre la seguridad de entornos Web. Las organizaciones empiezan incluir a los procedimientos de seguridad en los ciclos de desarrollo de software. Sin embargo, como se puede observar en la Ilustración 1, sólo la mitad de las organizaciones emplea los procedimientos formales.

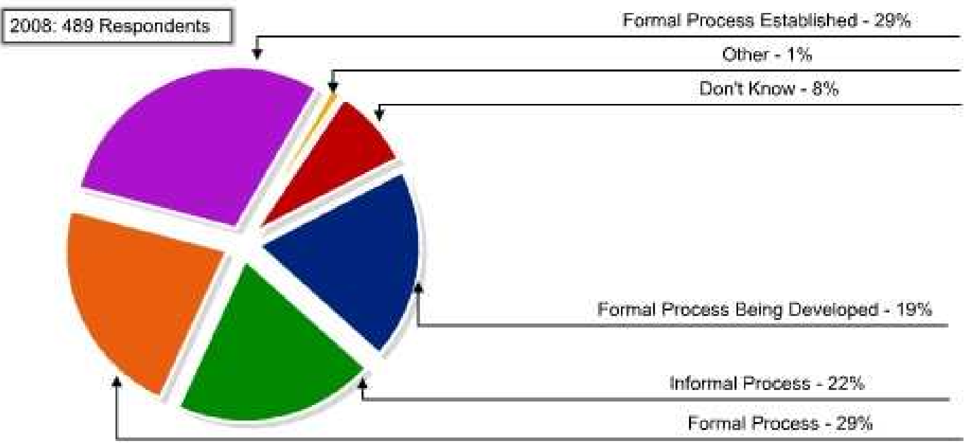


Ilustración 1: El proceso de desarrollo de software en las organizaciones. Ilustración tomada de [27]

La Ilustración 2 muestra fases de un ciclo de vida de software de calidad. Se observa que la seguridad y el análisis de los riesgos forman parte de este desarrollo. Una parte importante es el análisis estático del código (marcado en rojo en la Ilustración 2) cuyo fin es detectar errores de diferentes tipos, todavía antes de realizar las primeras pruebas. En esta fase se centra este trabajo, teniendo en cuenta solamente las vulnerabilidades de seguridad que pueden aparecer.

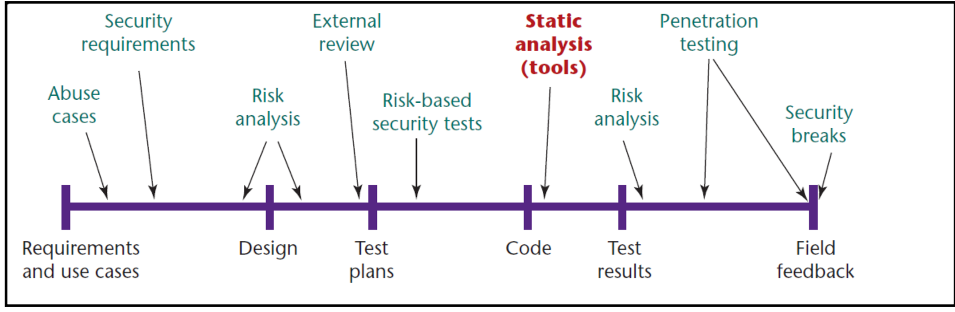


Ilustración 2: Fases de Un Ciclo de Vida de Software. Ilustración tomada de [18]

El rendimiento del análisis estático de código, es inversamente proporcional al tamaño del código. Si para un código pequeño sería todavía viable realizar el análisis manual, para un código grande sería altamente ineficiente. Por ello es necesario utilizar las herramientas diseñadas “ex profeso” para realizar este análisis. Como se describe más adelante en este trabajo existen muchas herramientas “open source”, para realizar el análisis estático de código basado en aspectos de seguridad, sin embargo la mayoría son para el lenguaje C, que se considera altamente inseguro.

Dado que las Aplicaciones Web se desarrollan cada vez más en lenguaje Java, utilizando la tecnología J2EE, es importante asegurar la seguridad en este entorno de desarrollo. Aunque el lenguaje Java se considere más seguro que C, sigue siendo vulnerable a muchos ataques de varios tipos. Por tanto existe una necesidad de herramientas de análisis estático con los fines de seguridad para el código de Java.

La herramienta LAPSE en la que se basa este trabajo es la primera herramienta “open source” para análisis estático de código para Java con el fin de encontrar errores de seguridad. LAPSE fue desarrollado en el año 2005, y se basaba en un informe sobre la seguridad en las aplicaciones web del año 2004 [25]. Por tanto surge la necesidad de actualizar este proyecto para adaptarlo a las vulnerabilidades más recientes.

1.2 Objetivos Generales del Proyecto

Después del análisis inicial del estado de seguridad en las aplicaciones Web y, más en concreto, en los entornos de desarrollo Java surgen los objetivos principales de este proyecto. Con el trabajo realizado en este PFC se pretende:

- **Estudiar los aspectos de seguridad de Java que afectan a las Aplicaciones Web**

  Se analizarán las amenazas más comunes y frecuentes para las aplicaciones Web. Especialmente se tendrá en cuenta las aplicaciones Web creadas con la tecnología J2EE. Se estudiarán los aspectos de seguridad Java y sus vulnerabilidades más extendidas

- **Estudiar las técnicas de análisis de seguridad estático sobre desarrollos java.**

  Se estudiará el trabajo del grupo proyecto de seguridad del grupo “Stanford SUIF Compiler Group” [30] de la Universidad Stanford, en particular los métodos de análisis estático de código que proponen. Se analizará el proyecto LAPSE, de forma detallada, teniendo en cuenta sus antecedentes, funcionamiento, implantación y resultados que obtiene.

- **Mejorar las técnicas de análisis de seguridad estático sobre desarrollos java.**

  Partiendo de los trabajos realizados por el grupo SUIF de la Universidad de Stanford se va a construir una versión actualizada y mejorada de la herramienta para detección de vulnerabilidades de seguridad en el código estático en la nueva herramienta LAPSE+

- **Definir un procedimiento que describa la inclusión de nuevas vulnerabilidades en LAPSE+**

  Se va a definir un procedimiento para la definición e inclusión de nuevas vulnerabilidades en LAPSE+. La integración de nuevas vulnerabilidades en LAPSE+ se podrá realizar sin necesidad de entender el funcionamiento interno

de la herramienta, siendo por tanto un proceso fácil y que pueden realizar los propios desarrolladores

- **Investigar nuevas y más recientes vulnerabilidades de las aplicaciones Web 2.0**

  Se investigará sobre las vulnerabilidades más recientes en las aplicaciones Web, sobre todo en el nuevo tipo de aplicaciones Web 2.0. Se incluirá la detección de estas vulnerabilidades en la herramienta LAPSE+, según el procedimiento de la inclusión anteriormente definida.

Los objetivos enumerados arriba, son los objetivos generales del proyecto. De cada objetivo general se derivan unas fases de proyecto que a su vez implican unos objetivos más específicos. En la siguiente sección se describen las fases del proyecto y sus objetivos.

1.3 Fases del Proyecto

En esta sección se explican las distintas fases y tareas que componen el proyecto realizado, así como los objetivos perseguidos en cada una de ellas.

FASE 1	Estudio inicial del marco teórico.
TAREA1	Recopilación de documentación sobre las técnicas de análisis estático del código.
TAREA 2	Recopilación de documentación sobre las herramientas de análisis estático del código
TAREA 3	Recopilación de documentación sobre las vulnerabilidades en aplicaciones Web
OBJETIVOS	• Entender diferentes técnicas de análisis estático de código y sus aplicaciones • Conocer las herramientas "open source" para el análisis estático de código para encontrar vulnerabilidades de seguridad • Conocer y entender el funcionamiento de las vulnerabilidades comunes en las aplicaciones web

Tabla 1: Descripción de la Fase 1 del PFC

FASE 2	Estudio del proyecto del grupo SUIF de la Universidad de Stanford
TAREA1	Lectura de la documentación del método de análisis estático propuesto por el grupo SUIF
TAREA 2	Estudio detallado del proyecto LAPSE
OBJETIVOS	• Entender en detalle el método de análisis estático de código sensible al contexto, propuesto por el grupo SUIF • Conocer el análisis empleado en el proyecto LAPSE y los modelos de descripción de vulnerabilidades que utiliza • Estudiar los resultados obtenidos aplicando la herramienta LAPSE sobre aplicaciones web reales

Tabla 2: Descripción de la Fase 2 del PFC

FASE 3	Estudio del código de LAPSE 2.5.6
TAREA1	Realización de pruebas con el código LAPSE 2.5.6
TAREA 2	Estudio de las tecnologías en las que se basa LAPSE
TAREA 3	Estudio del código fuente de la aplicación LAPSE
OBJETIVOS	• Entender el funcionamiento de plug-in LAPSE • Entender cómo se implanta LAPSE en eclipse • Comprender el diseño y la implementación de LAPSE 2.5.6 • Detectar los aspectos mejorables del LAPSE 2.5.6

Tabla 3: Descripción de la Fase 3 del PFC

FASE 4	Elaboración de una nueva versión de LAPSE: LAPSE+
TAREA1	Definición de los nuevos módulos a implantar
TAREA 2	Codificación de los nuevos módulos
TAREAS 3	Realización de pruebas con el código LAPSE+
OBJETIVOS	• Creación de una nueva versión del plug-in

Tabla 4: Descripción de la Fase 4 del PFC

FASE 5	Estudio sobre las vulnerabilidades recientes en Web 2.0
TAREA1	Estudio de las vulnerabilidades más recientes en la Web 2.0
TAREA 2	Selección de las vulnerabilidades adaptables al patrón de vulnerabilidades de LAPSE+
OBJETIVOS	• Encontrar las vulnerabilidades típicas en las aplicaciones web 2.0, no incluidas en las previas versiones de LAPSE

Tabla 5: Descripción de la Fase 5 del PFC

FASE 6	Descripción e Inclusión de nuevas vulnerabilidades
TAREA1	Descripción de las vulnerabilidades y su especificación en lenguaje PQL
TAREA 2	Inclusión de las nuevas vulnerabilidades en el código LAPSE+
OBJETIVOS	• Actualizar la herramienta LAPSE para que detecte las vulnerabilidades más recientes

Tabla 6: Descripción de la Fase 6 del PFC

FASE 7	Pruebas de comparación de LAPSE con LAPSE+
TAREA1	Encontrar aplicaciones web reales que puedan ser vulnerables a alguna de las vulnerabilidades detectables por LAPSE
TAREA 2	Realizar las pruebas sobre estas aplicaciones con LAPSE y LAPSE+
TAREA 3	Comparar los resultados obtenidos con las dos herramientas
OBJETIVOS	• Seleccionar un conjunto de aplicaciones web reales y frecuentemente utilizadas, con el fin de probar las herramientas • Realizar las pruebas y comparar los resultados obtenidos con LAPSE+ respecto a los obtenidos con LAPSE

Tabla 7: Descripción de la Fase 7 del PFC

FASE8	Redacción de la memoria del proyecto
TAREA1	Redactar el documento que explique en detalle los resultados obtenidos en cada fase de desarrollo de este PFC
TAREA 2	Elaborar un manual de usuario de la herramienta LAPSE+
OBJETIVOS	• Crear el entregable final de este PFC

Tabla 8: Descripción de la Fase 8 del PFC

2 ANTECEDENTES

En esta sección se describen las cuestiones y conceptos estudiados en el análisis inicial de este trabajo. Primero se analiza la seguridad de tecnología java y las vulnerabilidades que puede poseer una aplicación creada con esta tecnología. Se estudia también el concepto de análisis estático para encontrar vulnerabilidades de seguridad, importancia, ventajas e inconvenientes de su uso, como también las herramientas disponibles para automatizar este tipo de análisis. Finalmente se hace una breve introducción al proyecto de seguridad del grupo “Stanford SUIF Compiler Group” [30] de la Universidad Stanford en cuyo trabajo se basa este PFC. En particular se estudia LAPSE [31], una herramienta que realiza el análisis estático para encontrar vulnerabilidades de seguridad en el código Java EE.

2.1 Seguridad en Desarrollos Java

Java es considerado un lenguaje seguro. Esto se debe a que el verificador embebido, (“bytecode verifier”) asegura que no haya explícita manipulación de punteros y de *arrays*, que las cadenas de caracteres se comprueben automáticamente, que se capturen los intentos de referenciar a un puntero *null* y que las operaciones aritméticas y de conversión de tipos estén bien definidas e independientes de plataforma. Además en Java existen unos buenos mecanismos de seguridad que controlan el acceso a ficheros individuales, *sockets* y otros recursos sensibles. Aún así las aplicaciones Java están expuestas a diferentes ataques, por tanto la auditoría de seguridad debería ser aplicada regularmente como parte del proceso de desarrollo.

2.1.1 Modelo de Seguridad en Java

El lenguaje Java ha sido originalmente utilizado por los desarrolladores en el proceso de creación de los “applets”. Los “applets” son trozos de código Java que se pueden descargar de forma dinámica desde Internet y ejecutar en el navegador Web. Dado que un código descargado de forma dinámica puede proceder de una fuente desconocida o maliciosa la seguridad ha sido un aspecto de diseño muy importante en tecnología Java.

Desde sus principios Java ha incluido unos mínimos servicios de seguridad para prevenir el acceso no autorizado del código a los recursos protegidos de un ordenador, como red o ficheros de entrada/salida, modelo de seguridad conocido como “sandbox”. En este modelo, utilizado en la versión Java 1.0, sólo el código local tenía acceso a

todos los recursos del ordenador mientras que el código descargado de forma remota – como “applets”-, solamente podía acceder a los recursos limitados.

En la siguiente versión, Java 1.1, se ha modificado ligeramente el modelo de “sandbox” permitiendo que el código remoto debidamente autorizado, firmado por una autoridad de certificación de confianza, tuviera los mismos privilegios que el código local.

Finalmente Java 1.2 ha introducido un modelo de seguridad basado en políticas configurables en numerosos dominios de protección. En este modelo tanto el código local como remoto pueden ser obligados a utilizar solamente un específico dominio de recursos de acuerdo a las políticas establecidas en cada momento. Este modelo es mucho más flexible en comparación con modelos anteriores, ya que permite tratar el código local y remoto con las mismas reglas de seguridad.

A lo largo de los años, Java ha progresado, implementando medidas para más aspectos de seguridad, teniendo como objetivos principales dar soporte a SSL, S/MIME, PKI, certificados digitales y servicios de autorización y autenticación. Todos estos servicios se basan en un conjunto ampliamente reconocido y soportado por estándares.

Actualmente la tecnología Java incluye un extenso conjunto de APIs, herramientas e implementaciones de algoritmos, mecanismos y protocolos de seguridad conocidos.

El núcleo de la seguridad en Java está construido por los elementos presentes en la Ilustración 3.

- **Byte Code Verifier:** verifica que el *bytecode* que se carga desde la aplicación Java pero que procede del código externo a la plataforma Java, se mantenga fiel a la sintaxis del lenguaje Java.
- **Class Loader:** es responsable de traducir el *bytecode* en clases Java que pueden ser manipuladas por el entorno de ejecución Java - JRE. Diferentes “Class Loaders” pueden emplear diferentes políticas a la hora de decidir si la clase debería ser cargada en el entorno de ejecución o no.
- **Security Manager:** es el responsable de decidir si las llamadas invocadas al API de la plataforma Java pueden ser ejecutadas o no. El “Class Loader” y “Java Platform Classes” interceptan estas llamadas y las delegan al “Security Manager” para que tome la decisión.
- **Access Controller:** permite una forma más flexible y configurable del control del acceso a los recursos, añadido en la versión Java 2.

- **Permissions:** encapsula formas para designar las limitaciones de acceso y permisos que puedan estar asociados con recursos de valor.
- **Policies:** provee el mecanismo para asociar los permisos "permissions" con los recursos de forma configurable.
- **Protection Domains:** encapsulan partes que necesitan el mecanismo de control de acceso.
- **Runtime Execution Engine :** es el motor que ejecuta finalmente el código

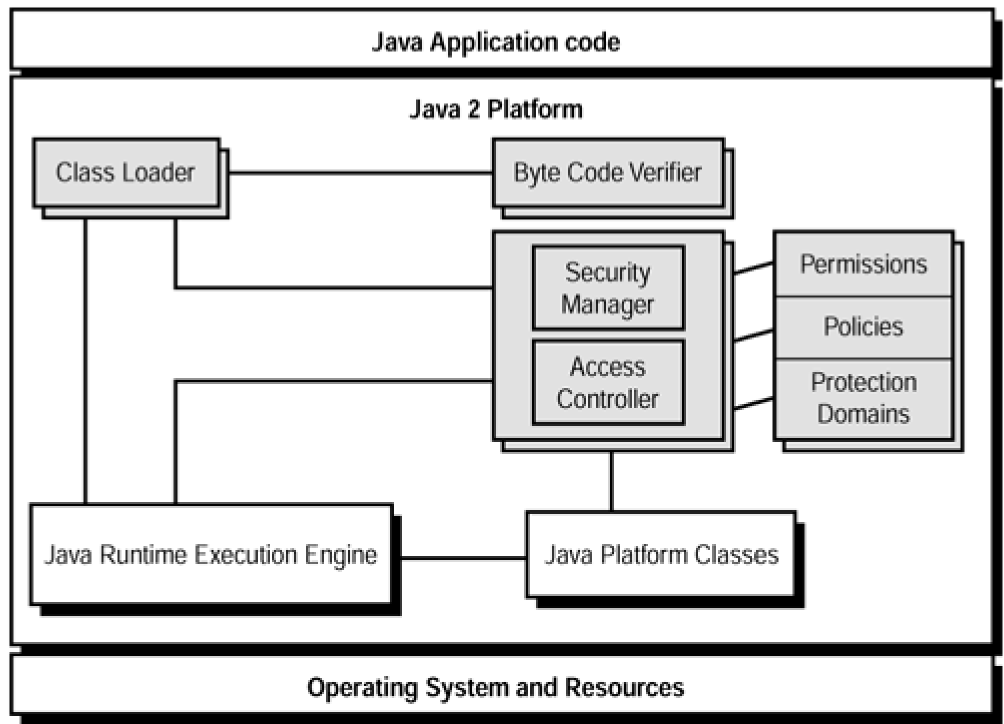


Ilustración 3: Arquitectura básica de seguridad en Java 2. Ilustración tomada de [15]

Además J2SE se construye sobre un conjunto adicional de nuevas tecnologías como:

- **Arquitectura Criptográfica de Java - (JCA)**: Provee la infraestructura para llevar a cabo una básica funcionalidad criptográfica con Java. Además incluye API para el manejo de claves y certificados.
- **Servicio de Autenticación y Autorización Java - (JAAS)** JAAS es un "framework" para proveer los servicios de autenticación y autorización a las aplicaciones Java. Como su nombre indica se divide en dos partes fundamentales, autenticación y autorización
- **Extensión Criptográfica de Java - (JCE)** - JCE es un "framework" para implementación del cifrado, generación de claves y algoritmo MAC para autenticación

- **Extensión Java para sockets seguros - (JSSE):** Provee la interfaz para construir aplicaciones Java con soporte al protocolo SSL. SSL (*Secure Socket Layer*) es un protocolo criptográfico que proporciona conexiones seguras a través de una red de comunicaciones entre cliente y servidor.
- **Estándares para la criptografía de Clave Pública - (PKCS):** Provee el API para PKCS#11, estándar de criptografía de clave pública para *tokens* propuesto por RSA
- **Soporte para Infraestructura de Clave Pública - (PKI):** Provee API para infraestructura de clave pública, manejo de certificados X509, etc.

La mayor ventaja de soporte de seguridad de Java, es la habilidad de proveer el mismo conjunto de servicios de seguridad para entornos heterogéneos de cómputo. Gracias a ello, el conjunto de las características de seguridad del lado del servidor, está disponible en diferentes plataformas y hace que el código y servicios de seguridad Java sean muy portables y capaces de comunicarse con aplicaciones y servicios no-Java.

2.1.2 Posibles Ataques y Vulnerabilidades en Java

A pesar de que Java provee un robusto modelo de seguridad existen formas en las que se puede comprometer la seguridad de las aplicaciones creadas con esta tecnología. A continuación se detallan los ataques más conocidos a los que está expuesto Java.

2.1.2.1 Ataque de Confusión de Tipos

Uno de los problemas más comunes en seguridad de Java es el ataque de confusión de tipos. Este ataque consiste en que un "applet" malicioso crea dos punteros al mismo objeto con tipos incompatibles. De esta forma el "applet" puede escribir en una dirección de memoria a través de un puntero y leer a través del otro, evitando por completo las reglas de seguridad existentes.

Un ejemplo de ataque de este tipo es el siguiente. Un "applet" tiene dos punteros a la misma posición en memoria: uno de tipo T y otro de tipo U. El "applet" puede ejecutar el código mostrado en el Listado 1:

```
class T {  SecurityManager x; }
class U {  MyObject x;}
…
T t = the pointer tagged T;
U u = the pointer tagged U;
t.x = System.getSecurity(); // the Security Manager
MyObject m = u.x;
…
```

Listado 1: Código Vulnerable al Ataque de "Confusión de Tipos"

Como resultado el objeto de tipo *MiObjeto* está apuntando al objeto que representa el Manager de Seguridad Java. Cambiando los campos de *m*, el "applet" puede cambiar el Manager de Seguridad, a pesar de que los campos de este último se han declarado como *prívate.*

2.1.2.2 Mal Uso de Serialización

La serialización permite capturar el estado de un programa Java y escribirlo como un "stream" de bytes. Esto permite que el estado se guarde de forma que puede ser reinstanciado mediante el proceso de deserialización. El hueco de seguridad introducido por la serialización es que ésta capta todos los campos de una clase, incluso los declarados como no públicos, que son normalmente no accesibles. Si el "stream" de bytes en el que se escriben los valores serializados se puede leer, entonces los campos

normalmente no accesibles pueden ser leídos. Esto se puede evitar cifrando el tráfico de bytes.

Aunque existen algunos ataques sobre el código Java tradicional, los programas más sensibles a diversos ataques son las aplicaciones Java EE. OWASP ("Open Web Application Security Project") publicó en el año 2007 la lista de las diez vulnerabilidades más comunes en aplicaciones web, en un proyecto llamado "OWASP Top Ten". OWASP es un proyecto abierto cuyo objetivo es identificar y prevenir causas de software no seguro. Uno de los resultados entregables del proyecto, fue el documento sobre las vulnerabilidades específicas de aplicaciones Java EE. [26]. A continuación se resumen las diez vulnerabilidades más comunes en entornos web, según OWASP.

2.1.2.3 Cross Site Scripting (XSS)

Los errores de XSS ocurren cada vez que las aplicaciones Java EE utilizan los datos introducidos por el usuario y los envían al navegador web sin validarlas o codificarlas primero. XSS permite que el atacante ejecute un script en el navegador de la víctima, el cual pueda secuestrar las sesiones del usuario, modificar las páginas web, introducir gusanos, etc.

Existen tres tipos conocidos de ataques XSS: reflejado, guardado y la inyección DOM. El XSS reflejado consiste en que la página refleja directamente los datos introducidos por el usuario sin validarlos. El XSS guardado primero guarda los datos introducidos por el usuario en un fichero, una base de datos u otro sistema de respaldo y posteriormente los muestra en la página sin ningún tipo de validación. Finalmente el tercer tipo de ataque XSS, la inyección DOM, está basado en el modelo estándar DOM, para representación de los elementos HTML o XML. Este ataque manipula el código y variables de JavaScript con los que esta construida la página.

Los ataques suelen ser implementados en JavaScript, un lenguaje de script muy potente, que permite a los atacantes manipular cualquier aspecto de la página atacada. Los ataques pueden ser:

- Añadir nuevos elementos (por ejemplo, un elemento de *login* que reenvía datos sensibles a un página maliciosa)
- Manipular cualquier aspecto de árbol DOM
- Borrar o cambiar el aspecto de las páginas.

Un ejemplo de código Java vulnerable es el mostrado en Listado 2:

```
String nombre= req.getParameter("nombre");
…
out.println("Bienvenido"+nombre);
```

Listado 2: Código Java Vulnerable al Ataque XSS

Un posible ataque podría ser la introducción de un script en el formulario de nombre, que se ejecutara cada vez que se cargue la página. Este Java Script podría contener un código malicioso que una vez ejecutado dañara al sistema, como por ejemplo el mostrado en el Listado 3:

```
<script type="text/javascript"
src="http://scriptmalo/js/scriptMalo.js" />
```

Listado 3: Ataque XSS - Ejemplo del Script Malicioso

Otro ataque que explota la vulnerabilidad de XSS es el ataque de "fijado de sesión" mediante una cookie. El atacante puede engañarle al usuario, haciéndole pulsar un enlace como el mostrado en el Listado 4. El usuario que pulsa el enlace proporciona sus datos de autenticación que en este caso se guardan junto con el nuevo identificador de sesión "1234". Después el atacante puede acceder libremente a la cuenta de la víctima sin autorización, utilizando solamente el ID de sesión previamente fijado.

```
http://online.worldbank.dom<script>document.cookie="sessionid=1234";</sc
ript>
```

Listado 4: Ejemplo de Script de Addición de Cookie

Existen muchas otras manera de explotar la vulnerabilidad de "Cross Site Scripting" sin embargo esta fuera de alcance de este trabajo describirlas todas.

2.1.2.4 Inyecciones (Injection Flaw)

Las inyecciones maliciosas, sobre todo la inyección SQL, son muy comunes en aplicaciones J2EE. Ocurren cuando los datos introducidos por el usuario se mandan al intérprete como parte de un comando o una consulta. Los datos trampa del atacante obligan al intérprete a ejecutar los comandos no deseados e incluso a cambiar los datos.

Un ejemplo clásico de la Inyección SQL consiste en pasar la entrada de un usuario directamente al intérprete, sin validarla a través de consultas generadas dinámicamente. El Listado 5 muestra un ejemplo del código vulnerable a la Inyección SQL.

```
String consulta = "SELECT id_usuario FROM datos_usuario WHERE
nombre_usuario = '" + req.getParameter("userID") + "' y
contrasenia_usuario = '" + req.getParameter("pwd") +"'";
```

Listado 5: Código Java Vulnerable al Ataque de "Inyección SQL "

Si no se validan bien los parámetros "userID" y "pwd", el usuario podría registrarse con la expresión *') OR '1'='1'* como nombre de usuario y contraseña. Dado que la segunda condición de esta expresión es siempre cierta, se le permitirá el acceso.

Otro ataque común es el ataque de Inyección de comandos.

```
Runtime.exec( "C:\\windows\system32\cmd.exe \C netstat -p " +
req.getParameter("proto");
```

Listado 6: Código Java Vulnerable al Ataque de "Inyección de Comandos"

En el ejemplo del Listado 6 el sistema operativo está forzado para ejecutar un comando "shell" con la entrada de usuario sin validar. El usuario podría introducir por ejemplo: `"udp; format c:"` y formatear el disco duro.

2.1.2.5 Ejecución Maliciosa de Ficheros

Un código vulnerable a la inclusión de ficheros remotos permite al atacante incluir código o datos tramposos, lo cual puede provocar ataques devastadores que lleven a la total inhabilitación del servidor. Esta vulnerabilidad afecta a cualquier "framework" Java EE que acepta nombres de ficheros o ficheros del usuario. Los ejemplos típicos incluyen "servlets" que permiten URL con nombres de ficheros como argumentos, o códigos que aceptan la selección de usuario sobre el nombre de fichero para incluir ficheros locales.

Si los datos introducidos por el usuario no se comprueban suficientemente, el contenido arbitrario y tramposo puede ser incluido, procesado o invocado por el servidor web. Esta vulnerabilidad permite a los atacantes tres tipos de ataques:

- La ejecución remota del código mediante el método *runtime.exec().*
- La instalación remota del *rootkit* y un compromiso total del sistema si el atacante consigue instalar puerta trasera.

- Acceso de ficheros sensibles, como *web.xml*, que contienen propiedades de configuración como nombres de usuario y contraseñas para bases de datos del servidor.

Un ejemplo de código vulnerable a ataque de ejecución maliciosa de los ficheros es el mostrado en el Listado 7:

```
String dir = s.getContext().getRealPath("/ebanking")
String file = request.getParameter("file");
File f = new File((dir + "\\" + file).replaceAll("\\\\", "/"));
```

Listado 7: Código Java Vulnerable al Ataque de "Path Traversal"

El atacante podría introducir en el formulario "file" `../../web.xml` , quedando la URL accedida: `www.victima.com/ebanking?file=../../web.xml`. Este ataque permitiría al atacante acceder al contenido de fichero "web.xml", que contiene datos sensibles sobre la aplicación web.

El ataque de ejecución maliciosa de ficheros es un ataque extendido de otro ataque conocido como "Path Traversal". Este permite al usuario navegar por las rutas de ficheros prohibidos.

2.1.2.6 Referencia Insegura Directa a Objetos

La referencia insegura directa a objetos ocurre cuando el desarrollador expone una referencia a un objeto con la implementación interna, como un fichero, un directorio, un registro o una clave de base de datos, en una URL o en un parámetro del formulario. Los atacantes pueden manipular estas referencias para acceder a los objetos sin autorización.

Un ejemplo de este tipo de vulnerabilidad es la referencia a claves de bases de datos. El atacante puede atacar estos parámetros simplemente adivinando otra clave válida. A menudo este tipo de claves son secuenciales, haciendo la búsqueda más fácil. Por ejemplo para el código mostrado en el Listado 8 el usuario puede cambiar el identificador de la tarjeta a cualquier otro y acceder a diferentes tarjetas sin autorización.

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );
String query = "SELECT * FROM table WHERE cartID=" + cartID;
```

Listado 8: Código Vulnerable al ataque de "Referencia Insegura Directa De Objetos"

2.1.2.7 Cross Site Request Forgery (CSRF)

El ataque CSRF fuerza al navegador de la víctima registrada a mandar una petición de pre-autenticación a la aplicación Java EE vulnerable. Esta aplicación a continuación fuerza al navegador de la víctima a llevar a cabo una acción trampa en beneficio del atacante. CSFR puede ser tan potente como la aplicación web a la que ataca. Están en riesgo de este ataque todas las aplicaciones que:

- No tengan verificaciones de autorización para acciones vulnerables
- Procesarán una acción si el *login* por defecto se puede pasar en la petición (e.g.http://www.example.com/admin/doSomething.jsp?username=admin&passwd=admin)
- Autorizan peticiones basadas solamente en los credenciales que se envían de forma automática, tales como los cookie de sesión, si está registrado en la aplicación en el momento, la funcionalidad “recuérdame” si no está registrado o un *token* de “Kerberos”.

2.1.2.8 Escape de Información y Manejo Inadecuado de Errores

Las aplicaciones pueden, sin intención explícita, dar a conocer la información sobre su configuración o su trabajo interno o pueden violar la privacidad a través de una variedad de problemas en las aplicaciones. Los atacantes utilizan estas debilidades para robar los datos sensibles o incluso llevar a cabo ataques más serios. Las aplicaciones a menudo generan mensajes de error y los muestran al usuario. Muchas veces estos mensajes son útiles para atacantes, ya que revelan detalles de la implementación o una información útil para explotar alguna vulnerabilidad. Algunos ejemplos son:

- Un manejo detallado de errores que provoca que se muestre demasiada información como por ejemplo trazas de la pila, sentencias SQL fallidas, u otra información de depuración
- Funciones que producen resultados diferentes para entradas diferentes. Por ejemplo una función de *login* debería producir el mismo texto para el usuario erróneo que para la contraseña errónea, sin embargo muchos sistemas producen un mensaje de error diferente en cada caso

2.1.2.9 Autenticación Quebrada y Manejo de Sesiones

Las credenciales de autenticación y los *tokens* de sesión suelen no estar suficientemente protegidos. Los atacantes descubren contraseñas, claves o *tokens* de autenticación para hacerse pasar por otro usuario.

A pesar de que a veces el mecanismo de autenticación principal contiene defectos de seguridad, la mayoría de las veces las debilidades aparecen en las funciones auxiliares de autenticación, como por ejemplo “logout”, “manejo de contraseñas”, “time-out”, “recuérdame”, “pregunta secreta” o actualización de la cuenta.

2.1.2.10 Almacenamiento Criptográfico Inseguro

Las aplicaciones Java EE rara vez utilizan las funciones criptográficas para proteger los datos y credenciales o las utilizan de forma incorrecta. Los atacantes utilizan los datos mal protegidos para cometer fraudes y otros crímenes. Los problemas más comunes son:

- No cifrar los datos sensibles
- Utilizar algoritmos no estandarizados
- Uso inseguro de algoritmos de cifrado fuerte
- Uso de algoritmos de cifrado flojos (MD5, SHA-1, RC3, RC4)
- Almacenamiento de claves en sitios desprotegidos

2.1.2.11 Comunicaciones Inseguras

Las aplicaciones Java EE pocas veces cifran el tráfico de red cuando es necesario proteger datos sensibles. La falta de cifrado de las comunicaciones sensibles significa que un atacante que puede husmear el tráfico de la red podrá acceder a la conversación, incluidas las credenciales y cualquier información sensible transmitida.

El uso de SSL para comunicaciones con el usuario final es crítico ya que es muy probable que este usuario utilice una red insegura para acceder a la aplicación. Además todo el tráfico autenticado, no solamente la petición de *login*, debería realizarse usando SSL ya que HTTP incluye credenciales de autenticación o *tokens* de sesión en cada petición individual.

Cifrado de las comunicaciones con el servidor “back-end” también es muy importante. Aunque esas redes suelen ser más seguras, la información que transportan es mucho más sensible y por tanto el uso de SSL en este tipo de comunicaciones es crucial.

2.1.2.12 Acceso a URL sin Restringir

Las aplicaciones Java EE a menudo protegen funcionalidades sensibles impidiendo solamente que se muestren enlaces o URLs a los usuarios no autorizados. Los atacantes pueden utilizar estas debilidades para acceder o llevar a cabo operaciones no autorizadas, accediendo esas URLs de forma directa. El ataque principal que explora esta vulnerabilidad se llama “forced browsing” y trata de adivinar los enlaces y encontrar las páginas sin protección mediante técnicas de fuerza bruta.

Algunos ejemplos de este tipo de vulnerabilidades son:

- URLs “ocultos” o “especiales”, reservados solamente para administradores o usuarios privilegiados en la capa de presentación, pero accesibles a todos los usuarios si saben que existen. E.g /admin./adduser.php o /aprobarTransferencia.do.
- Las páginas utilizadas durante el desarrollo o verificación que sirven de maqueta para roles de autorización y se despliegan en entorno de producción.
- Las aplicaciones a menudo permiten acceso a los ficheros “ocultos”, tales como ficheros XML estáticos o informes generados por el sistema, confiando en que no se sabe que éstos existen.
- El código que exige una política de control de acceso pero esta caducado o es insuficiente.
- El código que solamente evalúa los privilegios en la parte cliente y no en la del servidor.

.

2.2 Análisis Estático

Un análisis estático es un tipo de análisis que se emplea sobre el código de la aplicación sin necesidad de ejecutarla. Se puede realizar tanto sobre el código fuente como sobre el código compilado. El análisis estático en busca de vulnerabilidades de seguridad es uno de los puntos más importantes en la lista de buenas prácticas de software. Los mayores inconvenientes de este tipo de análisis son que puede producir "falsos positivos" o "falsos negativos". Los primeros ocurren cuando el programa contiene errores que el análisis no detecta mientras que los segundos cuando el análisis detecta errores que el programa no tiene. Además un análisis estático manual puede llegar a ser muy ineficiente por tanto es deseable realizar el análisis con ayuda de alguna herramienta que automatice el proceso. Existen varios métodos de análisis estático de aplicaciones con el fin de encontrar vulnerabilidades de seguridad. A continuación se detallan los aspectos más importantes en un analizador estático.

El enfoque más simple es utilizando la herramienta UNIX "grep", que escanea el código en busca de cadenas que se le pasa como parámetro. El problema de "grep" es que no percibe la estructura de ficheros, tratando comentarios, cadenas, declaraciones y llamadas a método sin distinción, como flujo de caracteres sobre el cual se realiza la búsqueda.

Un enfoque mejor es el análisis que tiene en cuenta las reglas léxicas del lenguaje en el que está escrito el programa analizado. Las herramientas que se basan en el análisis léxico, procesan y descomponen a símbolos ("tokens") el código antes de analizarlo en busca de vulnerabilidades. Este enfoque tiene la desventaja de producir muchos falsos positivos ya que no tiene en cuenta la semántica del código.

Un buen analizador debería entender el comportamiento del programa una vez en ejecución. La mayoría de los defectos de seguridad no es obvia y requiere una interpretación semántica. Una forma de mejorar la precisión de un analizador es empleando las técnicas del análisis semántico utilizadas por compiladores, tales como el árbol AST ("Abstract syntax Tree").

Otro aspecto a tener en cuenta en un analizador estático es el alcance del análisis. Un análisis local examina una sola función cada vez y no tiene en cuenta las relaciones entre las funciones. Un analizador modular considera una clase o unidad de compilación cada vez, de forma que tiene en cuenta relaciones ente funciones en el mismo modulo, y propiedades aplicables a las clases, pero no las relaciones entre distintos módulos. Un

análisis global involucra un programa entero de modo que tiene en cuenta todas las relaciones entre funciones. El alcance de análisis determina también la cantidad de contexto que el analizador considera. Cuanto más contexto, mejor se reducen los "falsos positivos", pero también hay más computación que ejecutar.

A lo largo de años se han desarrollado varias herramientas para realizar el análisis estático de seguridad con el fin de encontrar vulnerabilidades de seguridad. Aunque existen herramientas para aplicaciones escritas en diferentes lenguajes, la gran mayoría está destinada a revisar el código C o C++. En la sección 2.6 se detallan las herramientas más recientes.

Un tipo particular del análisis estático para seguridad, es el propuesto por el grupo SUIF, un análisis estático "points-to", y sensible al contexto (*context-sensitive points-to*) [21]. Es un análisis escalable a aplicaciones de un tamaño elevado y su enfoque puede ser utilizado para análisis de otros lenguajes orientados a objetos como por ejemplo C#. Este análisis se hace sobre el bytecode de la aplicación, haciendo imprescindible por tanto la disponibilidad de al menos el código fuente de la aplicación a auditar. Las vulnerabilidades analizadas son las causadas por las entradas de usuario que no se comprueban de forma adecuada. En el primer paso del análisis, estas vulnerabilidades se definen en un lenguaje específico de consultas PQL (sección 2.4).

El análisis de seguridad propuesto en [21] tiene como objetivo encontrar todas las violaciones de seguridad potenciales que se ajusten a especificación de vulnerabilidad dada por los descriptores de "source", "sink" y "derivation". Es muy importante saber a qué objetos se pueden referir estos descriptores; este es un problema general conocido como análisis "points-to". La información "points-to" se guarda en una base de datos deductiva llamada *bddbddb*. Los datos se representan con los diagramas de decisión binaria (*BDDs*) y se puede acceder de forma eficiente mediante las consultas escritas en el lenguaje Datalog. Posteriormente se utiliza *bddbddb* para resolver estas consultas.

Es importante conseguir una solución precisa al problema de "points-to" para disminuir el número de falsos positivos resultantes del análisis realizado. Un factor que contribuye de forma positiva a la precisión de los análisis es la sensibilidad al contexto – una característica que permite separar la información obtenida de diferentes contextos de invocación del mismo método.

2.3 LAPSE

LAPSE (**L**ightweight **A**nalysis for **P**rogram **S**ecurity in **E**clipse) es una herramienta para realizar la auditoría de seguridad en las aplicaciones Java. Fue creado por Benjamin Livshits, como parte del proyecto "*Griffin Software Security Project"* de la Universidad de Stanford.[30] LAPSE analiza el código J2EE para encontrar las vulnerabilidades más comunes en aplicaciones Web.

La herramienta tiene como objetivo encontrar las vulnerabilidades de seguridad en las aplicaciones Web, causadas por inadecuada o nula validación de las entradas del usuario. Este tipo de vulnerabilidades es ampliamente reconocido como el más frecuente en las aplicaciones Web. En particular, dos de los ataques de este tipo, el ataque de XSS ("Cross Site Scripting") y la "Inyección SQL", son los más comunes y aparecen como primeros en la lista de "Top Ten 2007" de OWASP [24].

La idea principal de LAPSE, es de ayudar al programador a desinfectar las entradas venenosas "*tainted input"* (entradas envenenadas), problema basado en el modo "*tainted*" de Perl. Los desarrolladores de LAPSE extienden el modo "*tainted*" de Perl definiendo el problema de entradas envenenadas como el problema "*tainted object propagation*", que se explica más en detalle en la sección 2.3.2.

2.3.1 Tipo de Vulnerabilidades de Seguridad Encontradas por LAPSE

Los ataques en las aplicaciones Web causados por entradas de usuario inadecuadamente validadas se pueden dividir en dos grupos. El primer grupo consta de los ataques que inyectan los datos maliciosos en las aplicaciones Web mientras que el segundo grupo son los ataques que manipulan las aplicaciones Web. La combinación de un ataque del primer grupo con su correspondiente del segundo es una vulnerabilidad de seguridad.

2.3.1.1 Ataques de Inyección de Datos Maliciosos en las Aplicaciones Web

Las aplicaciones Web pueden obtener la información del usuario de diferentes maneras, tales como parámetros de formularios HTML, cabeceras HTTP, valores de cookies, etc. Por tanto es difícil proteger las aplicaciones ante entradas de datos maliciosos. La protección en la parte del cliente no es suficiente ya que se puede manipular fácilmente, por tanto es necesario proveer una protección contra este tipo de ataques en la parte del servidor. A continuación se detallan los ataques particulares existentes:

- **Alteración de parámetros:** Este ataque se basa en introducir valores manipulados de forma maliciosa en los campos de texto o formularios HTML. Estos valores son enviados dentro de la petición HTTP.
- **Manipulación de URL:** Este ataque tiene lugar cuando los parámetros de los formularios HTML aparecen como parte de la URL a que se accede después de enviar el formulario con el método HTTP GET. El atacante puede editar la URL de forma directa, añadir los datos trampa en ella y luego acceder a ella de forma maliciosa.
- **Manipulación de campos ocultos:** Los campos ocultos se utilizan en HTTP para mantener el estado entre peticiones. Aunque estos campos no son visibles para el usuario final, se pueden modificar desde el código fuente de la página.
- **Alteración de cabeceras HTTP:** Las cabeceras HTTP normalmente no son visibles para el usuario final, sin embargo algunas aplicaciones sí que procesan estas cabeceras y algunos usuarios podrían pasar un código malicioso a través de ellas a la aplicación.
- **Envenenamiento de Cookies:** Cookie es un pequeño fichero guardado en el ordenador del usuario final y accesible por la aplicación Web. Sirve para guardar la información con el *login* y contraseña u otros identificadores del usuario. Esta información se suele guardar en el ordenador después de la primera integración del usuario con la aplicación Web. Envenenamiento de cookies es una variación de la alteración de cabeceras HTTP, ya que se puede pasar el código maligno guardándolo a través de valores en cookies.
- **Fuentes No Web:** Los datos malignos pueden ser introducidos a la aplicación no solamente a través de las peticiones Web. Aunque es menos común, puede pasar que los datos sean introducidos por línea de comandos, o como argumentos del método *main().*

2.3.1.2 Ataques de Manipulación de Aplicaciones Web con Datos Maliciosos

Una vez los datos maliciosos se han insertado en la aplicación el atacante puede manipularlos mediante diversas técnicas. A continuación se enumeran los ataques conocidos:

- **Inyección SQL:** (véase sección 2.1.2.4).
- **Cross-site Scripting:** (véase sección 2.1.2.3).
- **HTTP response-splitting:** Este ataque se basa en juntar dos respuestas HTTP para una sola petición, mediante el uso de los saltos de línea inesperados. La segunda respuesta HTTP puede ser juntada de forma errónea con la siguiente petición, y controlándola el atacante puede generar varios ataques, tales como envenenar las páginas web del servidor proxy.
- **Path Traversal:** Este ataque explota la vulnerabilidad descrita en (2.1.2.5).
- **Inyección de comandos:** Este ataque ocurre cuando se pasan los comandos de consola a las aplicaciones para su ejecución. Este ataque es menos común en las aplicaciones Web, especialmente las escritas en Java, aunque puede ocurrir también en éstas (véase sección 2.1.2.4).

Todos los ataques descritos en esta sección junto con los descritos en la sección anterior se analizan de forma más detallada en la sección 5. En esa sección se incluyen ejemplos de las aplicaciones vulnerables a diversos ataques y se describe cómo LAPSE detecta estas vulnerabilidades mediante un análisis del código fuente.

2.3.2 Funcionamiento de LAPSE

LAPSE busca todas las potenciales causas de cualquiera de las vulnerabilidades enumeradas. Este tipo de vulnerabilidades se puede generalizar como el problema de "propagación de objetos envenenados" ("*tainted object propagation problem*").

Este patrón de vulnerabilidades consiste en un conjunto de descriptores de fuente (*source*), de hundimiento (*sink*) y de derivación (*derivation*). Un objeto se considera *"tainted"* si se obtiene del objeto fuente aplicando los descriptores de *derivación* cero o más veces. La violación de seguridad ocurre cuando un objeto de hundimiento es *"tainted"*.

Los descriptores fuente especifican en qué forma el usuario puede introducir los datos al sistema. Consisten en un método fuente *m*, número del parámetro *n* y la ruta de acceso *p* que se aplica al argumento *n* para obtener los datos introducidos por el usuario. Se expresan mediante la tupla (*m,n,p*).

Los descriptores de hundimiento especifican las maneras inseguras mediante las que los datos pueden ser utilizados por el programa. Consisten en un método de hundimiento *m* número del parámetro *n* y la ruta de acceso *p* que se aplica a este argumento. Se expresan mediante la tupla (*m,n,p*).

Los descriptores de derivación especifican como los datos se propagan entre objetos de programa. Consisten en un método de derivación *m*, un objeto fuente expresado como argumento n_s y la ruta de acceso a este argumento p_s y el objeto destino n_d y la ruta de acceso a el p_d. Especifican que al llamar al método *m*, el objeto destino obtenido aplicando p_d al n_d se deriva del objeto obtenido aplicando p_s al argumento n_s. Se expresan mediante la tupla (m, n_s, p_s, n_d, p_d).

LAPSE se distribuye como un plug-in de Eclipse. Eclipse es un entorno de desarrollo integrado (IDE), que permite mostrar, mediante vistas específicas, en qué parte del programa se da la vulnerabilidad de seguridad. LAPSE define 3 vistas diferentes:

- **Vulnerability Source** – permite al usuario navegar libremente entre fuentes de datos de usuario.
- **Vulnerability Sink** - permite al usuario navegar libremente entre llamadas que manejan cadenas que serán ejecutadas de forma maliciosa, por ejemplo sobre base de datos en caso de “SQL Injection”.
- **Provenance Tracker** – permite encontrar orígenes de un valor particular en el código fuente siguiendo el flujo de datos hacía atrás.

Más detalles acerca de LAPSE, particularmente acerca de cómo se integra LAPSE en el desarrollo pueden encontrarse en la sección 3 de ese trabajo.

Las pruebas de LAPSE, realizadas por los desarrolladores, detectaron 18 vulnerabilidades en 15 aplicaciones Web open-source de SourceForge. Los resultados obtenidos con LAPSE documentados por sus desarrolladores en [14] no son completos, ya que algunos errores no se llegan a detectar y aparecen falsos positivos como resultado del análisis.

Los autores de LAPSE proponen un análisis más completo, basado en punteros y sensible al contexto como una futura mejora. Este análisis utiliza el lenguaje PQL para definir las vulnerabilidades en una sintaxis parecida a Java. Posteriormente las vulnerabilidades así definidas se traducen a un lenguaje lógico Datalog y se resuelven en una base de datos especifica *bbdbbdd*. (En las siguientes secciones se describen los detalles de cada fase.)

Este análisis a pesar de ser mucho más completo, es difícil de utilizar ya que no se integra bien en el entorno de desarrollo Eclipse. Según los autores, se está desarrollando un plug-in de Eclipse para hacer esta integración más fácil.

2.4 PQL – *Program Query Language*

El lenguaje PQL permite a los usuarios expresar vulnerabilidades en una sintaxis parecida a Java. Es capaz de expresar una variedad de cuestiones sobre la ejecución del programa. PQL se centra en una clase importante de patrones de errores que manejan secuencias de eventos asociadas a un conjunto de objetos relacionados. Esta característica permite, entre otras cosas, especificar el problema de propagación de los objetos venenosos (“tainted object propagation problem”), que como se ha mencionado anteriormente generaliza la descripción de vulnerabilidades causadas por entrada de usuario sin validar.

La construcción de sentencias en PQL sigue una metodología determinada. Primero se modela la ejecución de un programa como secuencia de eventos primitivos, cada uno con su único ID, tipo y listado de atributos. Los objetos se nombran por su identificador único. Dado que PQL se centra en objetos, solamente se escoge del programa las instrucciones que hacen referencia directa a los objetos o que identifican el fin del programa. En particular se tratan como eventos los siguientes tipos de instrucciones:

- Operaciones “load” y “store” sobre los campos. Los atributos son objeto fuente, objeto destino y nombre del campo.
- Operaciones “load” y “store” sobre los *arrays*. Los atributos son objeto fuente, objeto destino, pero se ignora el índice del *array*.
- Las llamadas y retornos de métodos. Los atributos son el método invocado, los objetos pasados como argumentos y el objeto retornado. El parámetro de retorno incluye el ID de su correspondiente evento de llamada.
- Creación de objetos. Los atributos son el nuevo objeto y su clase
- Fin del programa. No tiene atributos y ocurre justo antes de que termine la máquina virtual JVM.

Una vez descrita la traza de ejecución de los eventos, se definen las consultas PQL. Una consulta PQL es un patrón a encontrar durante la traza de ejecución anteriormente descrita, y las acciones a ejecutar una vez encontrado el patrón.

PQL tiene como objetivo seguir la secuencia de invocación de los métodos y accesos de campos y elementos de *arrays* en los objetos relacionados. Por motivos de sencillez PQL no deja referenciar variables o datos primitivos, como enteros, *floats* o caracteres,

ni operaciones primitivas como sumas o multiplicaciones. Es posible que en un futuro se añada la característica de manejar objetos primitivos mediante PQL.

El patrón de la consulta describe la secuencia de eventos dinámicos que involucran a variables, referidas como las instancias dinámicas de objetos, a los que se busca. El "match" de la consulta es un conjunto de objetos y la subsecuencia de trazas que juntos satisfacen el patrón. El patrón de ejecución de la consulta se especifica con un conjunto de eventos primitivos conectados. Además PQL permite crear patrones recursivos o declarar variables dentro de la consulta, que posteriormente se puede utilizar en otra. Las variables de consulta corresponden a los eventos primitivos y se describen con la sintaxis parecida a Java. El Listado 9 describe la gramática de lenguaje PQL.

Las variables de una consulta pueden ser argumentos (pasados de una consulta a otra), variables de retorno o variables internas. Existen dos tipos de variables:

- "**Object":** representan objetos individuales en la pila. Se declaran con el nombre de la clase, opcionalmente precedida de !. El nombre de la clase restringe el tipo de la instancia que debe tomar el objeto declarado; los nombres precedidos por "!" se referirán a aquellas instancias que no debe tomar el objeto declarado. Si la misma variable de tipo "object" aparece más de una vez en la consulta debe representar la misma instancia aunque puede cambiar el contenido del objeto entre distintas clausulas "matches".
- **"Member":** este tipo de variable representa un nombre del campo o método dentro de un objeto. Se declaran a través de un patrón que tiene que coincidir con el nombre. El patrón "*" sustituye a cualquier nombre. Si una variable de tipo "member" aparece más de una vez en la consulta debe representar el mismo campo o método en todos los casos.

Existe también el carácter comodín "_" que se puede utilizar sobre diferentes ocurrencias de variables tanto de tipo "object", como "member", cuando el nombre no es significativo para la consulta. No está permitido utilizar este signo para una variable de retorno.

Cada consulta debe empezar con la palabra **query**. Los puntos y coma se usan en PQL para indicar una secuencia de eventos que ocurre en orden. Las consultas PQL se componen de tres cláusulas principales:

- **“uses”:** declara todas las variables utilizadas por la consulta.
- **“matches”:** especifica la secuencia de eventos sobre las variables que tienen que ocurrir para satisfacer el patrón de la consulta.
- **“return”:** especifica objetos devueltos por consulta siempre y cuando se satisfacen las condiciones declaradas en cláusula **“matches”**.

Dentro de la clausula “matches”, la secuencia de eventos se describe a través de declaraciones (“statements”). Las declaraciones primitivas se corresponden con los tipos de eventos descritos en la “traza de ejecución abstracta”. Aparte de declaraciones primitivas existen invocaciones de métodos, que abarcan todos los eventos desde la llamada al método hasta el retorno de la misma. Todas las referencias que se hagan desde las declaraciones a objetos, tienen que ser objetos declarados o una variable especial representada por el carácter comodín “_”.

Varias declaraciones primitivas se pueden juntar en declaraciones compuestas utilizando operadores de ‘;’ o ‘,’ dependiendo de si el orden de ejecución es importante o no respectivamente. También se puede utilizar el operador de exclusión ‘~’ para indicar que una secuencia no debe ejecutarse.

PQL permite también hacer las consultas recursivas utilizando subconsultas. Estas subconsultas pueden devolver múltiples valores que se asocian a las variables en la consulta llamante. Llamando a las subconsultas de forma recursiva, cada una con su propio conjunto de variables, las consultas pueden involucrar a un número de objetos ilimitado.

Finalmente PQL permite tomar acciones de recuperación utilizando las clausulas “executes” o “replaces”. Para el método “executes” se debería devolver el valor *void*, mientras que para el “replace” el valor del mismo tipo que el evento reemplazado.

```
queries -! query*
query -! query qid ( [decl [, decl]*] )
[var declList ; ]
[within methodInvoc )]
[matches { seqStmt }]
[replaces primStmt with methodInvoc ;]*
[executes methodInvoc [, methodInvoc]* ;]*
methodInvoc -! methodName(idList)
decl -! object [!] typeName id |
member namePattern id
declList -! object [!] typeName id ( , id )*|
member namePattern id ( , id )*
stmt -! primStmt | _ primStmt |
unifyStmt | { seqStmt }
primStmt -! fieldAccess = id |
id = fieldAccess |
id [ ] = id |
id = id [ ] |
id = methodName ( idList ) |
id = new typeName ( idList )
seqStmt -! ( poStmt ; )*
poStmt -! altStmt ( , altStmt )*
altStmt -! stmt ( "|" stmt )*
unifyStmt -! id := id
qid ( idList )
typeName -! id ( . id )*
idList -! [ id ( , id )* ]
fieldAccess -! id . id
methodName -! typeName . id
id,qid -! [A-Za-z ][0-9A-Za-z_ ]*
namePattern -! [A-Za-z*_ ][0-9A-Za-z*_ ]*
```

Listado 9: Especificación BNF de Gramática de PQL

A continuación se muestra un simple ejemplo de uso de PQL.

El fragmento de código en Java que pueda causar el ataque de la inyección SQLm mostrado en el Listado 10 se puede expresar con la consulta PQL mostrada en el Listado 11.

```
String p=request.getParameter("query");
Con.execute(p);
```

Listado 10: El código Java Vulnerable a la "Inyección SQL"

```
query simpleSQLInjection()
uses
object HttpServletRequest r;
object Connection c;
object String p;
matches {
p = r.getParameter(_);
c.execute(p);
}
```

Listado 11: La Vulnerabilidad "Inyección SQL" Expresada en PQL

2.5 Bddbddb y Datalog

En esta sección se describen los detalles de uso de" Bddbddb" y Datalog en el análisis "points-to" y sensible al contexto propuesto por el grupo SUIF de la Universidad de Stanford, descrito en la sección 2.2 de éste trabajo. Aunque LAPSE no utiliza éste análisis, sería conveniente incluirlo como un futuro desarrollo de LAPSE, para disminuir el número de falsos positivos producidos por la herramienta.

Datalog es un lenguaje de reglas y consultas para bases de datos deductivas. Su sintaxis es el subconjunto de la sintaxis de Prolog – lenguaje de programación lógica. Datalog incluye la habilidad de especificar las propiedades de forma recursiva.

En Datalog cada relación es una tupla de *n* atributos. Cada atributo está asociado con un dominio de valores que un determinado atributo puede tomar. La relación $R(x_1,\ldots..,x_n)$ es verdadera si la tupla $(x_1,\ldots..,x_n)$ está en relación R. La relación $R(x_1,\ldots..,x_n)$ es falsa si la tupla $(x_1,\ldots..,x_n)$ no está en la relación R. Un programa en Datalog es un conjunto de reglas de la forma E_0:-E_1,…E_k., donde para cada asignación particular la expresión E_0 (la regla *cabeza*) es verdadera si y solo si las expresiones E_1… E_k (las reglas *subobjetivos*) son ciertas.

Una consulta en Datalog consiste en un conjunto de reglas, escritas en notación Prolog, donde un predicado se define como conjunción de otros predicados. Por ejemplo la regla "D(w,z) :- A(w,x),B(x,y),C(y,z)." especifica que D(w,z) es verdadero si y solo si las reglas A,B,C son verdaderas también.

Bddbddb, Binary Decision Diagram – Based Deductive Database es una base de datos deductiva que implementa a Datalog. Bddbddb acepta las consultas de Datalog e implementa las relaciones como diagramas de decisión binaria - BDDs. Fue escrita por John Whaley, investigador del "Stranford SUIF compiler group". Esta base de datos deriva su eficiencia de las numerosas optimizaciones especificas de cada nivel de abstracción: Datalog, algebra relacional y BDD. La bddbddb se utiliza sobre todo como una herramienta para el análisis del programa, que permite representar todo el programa a través de relaciones de la base de datos.

Las consultas PQL pueden ser traducidas a la aproximación Datalog utilizando un simple enfoque de traducción "dirigida por sintaxis". Es un mecanismo en el que analizador sintáctico es el encargado de hacer toda la traducción. Cada consulta PQL se convierte en una relación de Datalog definida sobre un *bytecode*, un nombre de

campo un método o las variables de la pila. Un punto de programa con la secuencia de eventos más larga de la consulta PQL se convierte en un *bytecode.* Una variable de la consulta PQL se convierte en un campo o nombre del método y un objeto de la consulta se convierte en una variable de pila. Los literales y caracteres comodines no cambian en Datalog, respecto al PQL.

Para representar el programa se guarda toda la información disponible en el código fuente como conjunto de relaciones de la base de datos. Esta representación permite abstraer la sintaxis del programa y hace que la información sea fácilmente accesible. Los dominios de interés para el análisis de punteros son:

- Java Bytecode **B**
- Variables de Programa **V**
- Métodos **M**
- Campos **F**
- Contexto de invocación **C**
- Los objetos de pila llamados por su lugar de asignación **H**
- Enteros **Z**

El analizador traduce el programa fuente en un conjunto de relaciones de entrada. Las relaciones relevantes para el análisis estático "point-to" son:

- actual – paso de parámetros
- assign – asignación entre dos variables
- vPo – la referencia directa de una variable a un objeto
- hP – la referencia del campo de un objeto al otro objeto
- ret – retorno de un método
- fldld – "load" sobre un campo
- fldst – "store" sobre un campo
- arrayld – "load" sobre un array
- arrayst – "store" sobre un array

Además existe una relación de "uno en uno" entre los atributos de las declaraciones primitivas en PQL y los atributos de relaciones. Los ejemplos de relaciones son:

- **assign**: *VxV.assign(v1,v2)* significa asignación v1=v2.

- **vPo**: VxH.vPo(v,h) significa que el programa pone la referencia a objeto de pila h directamente en la variable v.
- **fldld**: *BxVxFxV. fldld(b,v1,f,v2)* significa que el *bytecode* b ejecuta v1=v2.f.
- **fldst**: BxVxFxV.*fldst*(b,v1,f,v2) significa que el *bytecode* b ejecuta v1.f=v2.
- **actual**: *BxZxV. actual(b,z,v)* significa que la variable v es el argumento número z del método invocado en el *bytecode* b.
- **ret**: *BxV.ret(b,v)* – significa que la variable es un valor de retorno del método en el *bytecode* b.

Utilizando las relaciones definidas se puede expresar un análisis "points-to" en Datalog. Las reglas *vPo, fldst, fldld, assign*, son reglas de entrada. Además el análisis "points-to" produce dos nuevas reglas de salida:

- **vP(v,h):** VxH: es verdadero si la variable v apunta a un objeto de pila h en cualquier punto durante la ejecución del programa.
- **hP(h1,f,h2):** HxFxH es verdadera si el campo f del objeto pila h1 apunta al otro objeto h2.

El análisis "points-to" consiste en aplicar cuatro reglas de forma repetida hasta que no se produzcan nuevos elementos en consecuencia de aplicación de cualquiera de las reglas. Las reglas que se aplican son:

1. vP(v, h) : – vP0(v, h).
2. vP(v1, h) : – assign(v1, v2), vP(v2, h).
3. hP(h1, f, h2) : – fldst(_, v1, f, v2), vP(v1, h1), vP(v2, h2).
4. vP(v2, h2) : – fldld (_, v1, f, v2), vP(v1, h1), hP(h1, f, h2).

La regla 1 incorpora la primera relación "points-to" a la regla de salida vP. La regla 2 calcula el cierre transitivo respecto a la asignación. Si la variable v2 puede apuntar al objeto h y a la variable v1 se le asigna v2, entonces v1 puede también apuntar al objeto h. La regla 3 describe el efecto de guardar variables en campos de objeto. Si la variable v1 apunta al objeto h1 y v2 apunta al objeto h2 entonces la regla v1.f=v2 implica que h1.f puede apuntar a h2. La regla 4 modula la carga desde los campos de objetos. Dado la regla v2=v1.f, si v1 puede apuntar al h1 y h1.f puede apuntar a h2 entonces v2 puede apuntar al h2.

Las reglas descritas arriba para llevar a cabo el análisis "points-to", realizan un análisis insensible al contexto, que como se ha explicado anteriormente puede producir muchos

falsos positivos a la hora de analizar el programa en busca de potenciales vulnerabilidades. Un enfoque novedoso y eficiente, propuesto por SUIF es el enfoque “cloning-based”, en el que se crea una copia separada del método por cada contexto llamante [11]. Este enfoque es muy sencillo desde el punto de vista algorítmico ya que se aplica el análisis “points-to” insensible al contexto del programa clonado. Además aunque el programa crece exponencialmente muchos de los contextos se parecen entre sí. Estas similitudes se pueden aprovechar utilizando los diagramas de decisión binaria (BDD).

El análisis “points-to” sensible al contexto consiste en varios pasos. Primero se utiliza el análisis insensible al contexto con el fin de definir el grafo de llamadas a funciones. Posteriormente se procede a clonar todos los métodos en el grafo, produciendo una copia por cada contexto de interés. Cada clon recibe un único contexto *c*. Este paso genera una nueva relación, respecto al análisis insensible al contexto:

- **Call:** *CxBxCxM.calls(c1,b,c2,m)* – la llamada al método en el bytecode b, ejecutada en el contexto c1, puede invocar el contexto c2 del método m.

Además las reglas utilizadas en el último paso del análisis “points-to” sensible al contexto, son casi idénticas a las reglas del mismo análisis insensible al contexto.

1. vPc(_, v, h) : – vP0(v, h).
2. vPc(c1, v1, h) : – assignc(c1, v1, c2, v2), vPc(c2, v2, h).
3. hP(h1, f, h2) : – fldst(_, v1, f, v2), vPc(c, v1, h1), vPc(c, v2, h2).
4. vPc(c, v2, h2) : – fldld (_, v1, f, v2), vPc(c, v1, h1), hP(h1, f, h2).

A cada variable se le añade su correspondiente contexto en el programa.

Las reglas Datalog se corresponden con las secuencias de operaciones expresadas en algebra relacional y se codifican como funciones booleanas. Las funciones booleanas grandes se pueden expresar de forma eficiente utilizando diagramas de decisión binaria (*BDDs*). Estos diagramas han sido inventados para verificación de hardware con el fin de guardar de forma eficiente un elevado número de estados, que comparten varias características comunes. Un BDD es un grafo dirigido, no cíclico con un único nodo fuente y dos nodos terminales que representan constantes cero o uno. Cada nodo no terminal se marca con una variable de decisión de entrada y tiene exactamente dos aristas de salida: la arista alta que se sigue en caso de que la variable de entrada del nodo es cierta y la arista baja que se sigue en caso contrario.

2.6 Otros Analizadores Estáticos

Existen varias herramientas “open source”, aparte de LAPSE, para realizar auditoria de seguridad sobre el código fuente de la aplicación, mediante el análisis estático. Sin embargo la mayoría de estas herramientas está destinada a auditar aplicaciones escritas en C o C++. LAPSE, según sus desarrolladores, es la primera herramienta de análisis estático en busca de vulnerabilidades de seguridad para Java. A continuación se describen varios ejemplos de herramientas existentes. Las cuatro primeras, (ITS4, Pscan, RATS y FlawFinder) utilizan solamente el análisis léxico para escaneo del código. Este tipo de análisis provoca muchos falsos positivos, como se ha explicado en la sección 2.2. Los cuatro analizadores descritos como últimos (BOON, CQual, MOPS y Splint) tienen en cuenta también la semántica del lenguaje que analizan, gracias a lo que consiguen resultados más precisos.

2.6.1 ITS4

La primera herramienta para detectar los problemas de seguridad mediante el análisis de código fuente fue ITS4 desarrollada en principios de 2000 por “cdigital”. ITS4 es una herramienta muy simple que escanea el código C y C++ para encontrar potenciales vulnerabilidades de seguridad. Es una herramienta que funciona por línea de comandos en plataformas Windows y Linux.

ITS4 escanea el código en busca de llamadas a funciones que pueden ser peligrosas. La herramienta utiliza un conjunto de simples reglas en las que se basa para encontrar las posibles vulnerabilidades. Por ejemplo, el uso de la función strcpy() debería ser evitado. Para algunas de las llamadas ITS4 intenta analizar qué nivel de riesgo tienen. Para cada caso la herramienta produce un informe sobre el problema encontrado, con una corta descripción y sugerencias de cómo puede arreglarse el código.

2.6.2 Pscan

Pscan es un escáner para detectar los problemas de seguridad en código C. En particular pscan detecta los problemas de formato de cadenas que pueden provocar ataques de desbordamiento de buffer.

Pscan detecta como vulnerabilidad los métodos de la librería printf que toman las cadenas como parámetros y ninguno de los argumentos es el “especificador de conversión” (eg. %s, %c, etc.). Este tipo de expresiones podría provocar el ataque de desbordamiento de buffer, ya que el usuario podría introducir su propio “especificador

de conversión" dentro de la variable pasada como argumento. Un método de la librería printf, si no se le pasa explícitamente a un especificador de tipos, tratará como tal cualquier cadena de tipo %c que encuentre dentro de la variable que toma como argumento.

Esta herramienta es muy limitada en comparación con otras diseñadas para el análisis estático existente, ya que solamente detecta un tipo de errores.

2.6.3 RATS - Rough Auditing Tool for Security

RATS, desarrollado por los ingenieros del "Secure Software", que actualmente pertenece a "Fortify Software", es una herramienta para escanear el código fuente de las aplicaciones escritas en C, C++, Perl, PHP y Python con el fin de encontrar errores de seguridad como desbordamiento de buffer o condiciones de carrera TOCTOU (Time Of Check to Time Of Use). RATS utiliza una base de datos para encontrar sentencias del código que pueden ser una vulnerabilidad potencial. El usuario puede decidir que base de datos utilizar en el escaneo y que nivel de riesgo asumir.

Por cada problema potencial encontrado RATS provee la descripción del mismo y sugiere una solución al respecto. Además indica el grado de severidad del problema encontrado para facilitarle al auditor la asignación de prioridades. RATS ejecuta un análisis básico para descartar condiciones que no causan problemas de seguridad. El análisis con RATS no es completo ya que no encuentra todos los errores existentes y puede destacar como errores cosas que no lo son.

2.6.4 FlawFinder

FlawFinder es una herramienta, escrita en Phyton, que analiza el código fuente de los programas escritos en C o C++ para encontrar posibles vulnerabilidades de seguridad y ordena los problemas encontrados según el nivel del riesgo. Como en caso de RATS el análisis no es completo, ya que pueden producirse falsos positivos, como también pueden existir vulnerabilidades no detectadas por el FlawFinder. FlawFinder fue desarrollado casi simultáneamente con RATS y las dos herramientas son muy parecidas.

FlawFinder, como RATS, funciona utilizando una base de datos embebida con funciones C/C++ que pueden provocar problemas de seguridad conocidos, tales como desbordamiento de buffer, condiciones de carrera, o problema con el formato de

cadenas. El escaneo consiste en analizar el código fuente de la aplicación en busca de estas funciones.

2.6.5 BOON

"Buffer Overun DetectiON" es una herramienta que sirve para encontrar vulnerabilidades del desbordamiento del buffer en el código C. Boon aplica un análisis de "rango del integer" para determinar si un programa C puede indexar algún *array* fuera de sus límites. Para ello modela cada buffer como un par de rangos del *integer*, el primer rango sirve para saber el número de bytes asignado, y el otro rango indica el número de bytes en uso. El análisis verifica por cada buffer si su tamaño de asignación es al menos tan grande como su máxima longitud.

BOON es capaz de encontrar muchos errores, aun así no es muy preciso ya que ignora el orden de las sentencias del código, y es incapaz de modelar dependencias entre funciones o referencias de punteros.

2.6.6 MOPS

MOPS, *MOdelchecking Programs for Security properties,* utiliza un enfoque formal de chequeo del modelo para buscar violaciones de las propiedades de seguridad. MOPS construye un modelo formal del programa y de la propiedad de seguridad y posteriormente analiza este modelo.

Los desarrolladores pueden modelar sus propias propiedades de seguridad y algunos han utilizado la herramienta para detectar errores de manejo de privilegios, la construcción incorrecta de "chroot jails", condiciones de carrera en acceso a ficheros e o esquemas incorrectos de ficheros temporales.

2.6.7 CQual

Cqual es una herramienta de análisis basada en tipos, que sirve para detectar las vulnerabilidades de formato de cadenas en C como también errores de confianza entre espacio de usuario y espacio del kernel. CQual extiende el sistema de tipos de C, mediante unos calificadores. Se basa en modo "taint mode" de Perl, el cual trata y utiliza calificadores de tipo para realizar análisis "taint".

El programador debe anotar varias variables como "tainted" o "untainted" y luego utilizar las reglas de inferencia para propagar calificadores. De esta forma no tiene que añadir

muchas anotaciones, ya que con unos pocos se infieren automáticamente al todo el programa. Una vez propagados los calificadores el sistema puede detectar vulnerabilidades de cadenas mediante chequeo de tipos. Para analizar el programa, CQual atraviesa el árbol AST y genera una serie de restricciones que capturan relaciones entre calificadores. La solución a estas restricciones da una asignación válida de los calificadores de tipo a variables en el programa. Si la restricción no tiene solución, ocurre una inconsistencia de calificadores de tipo la cual indica una potencial vulnerabilidad.

2.6.8 Splint

Splint es una herramienta de análisis estático ligero “lightweight” para el código C. Extiende el concepto de “lint”, herramientas de análisis estático del código, al mundo de seguridad. Splint encuentra típicas vulnerabilidades de seguridad del lenguaje C, como desbordamiento del buffer, formato de cadenas, etc. Aparte de las verificaciones embebidas, Splint provee un mecanismo para definir nuevos chequeos y anotaciones para detectar nuevas vulnerabilidades.

Splint encuentra potenciales vulnerabilidades comprobando si el código fuente es consistente con propiedades emergentes de anotaciones. Añadiendo anotaciones los desarrolladores pueden permitir a la herramienta encontrar diversas vulnerabilidades, tales como violaciones de abstracción, modificaciones de variables globales sin anunciar o posibles usos sin inicialización.

3 IMPLANTACIÓN DE LAPSE EN ECLIPSE

LAPSE, como se ha indicando en la sección de Antecedentes (sección 2) se distribuye como un plug-in Eclipse, bajo la licencia GNU General Public License (GPL), siendo por tanto una aplicación "Open Source". Para realizar este PFC, se ha utilizado la versión 2.5.6 de LAPSE para estudiar y mejorar el funcionamiento de la herramienta. A continuación se describe cómo está implementado el plug-in. La versión de Eclipse utilizada es eclipse 3.2, siendo esta la única compatible con el LAPSE 2.5.6.

Eclipse [8], como definen sus fundadores, es una comunidad de código abierto, cuyos proyectos se centran en construir una plataforma de desarrollo abierta, compuesta de unos "frameworks" extensibles, herramientas y *runtimes* para construcción, despliegue y manejo de software durante su ciclo de vida.

Además LAPSE utiliza Eclipse JDT APIs para manipular el código fuente que debe analizar. El JDT ("Java Development Tool") provee una API para acceder y manipular el código fuente de Java. Permite acceder, leer y modificar proyectos existentes en el *workspace* de Eclipse, como también crear proyectos nuevos. JDT permite además lanzar programas Java. Al código fuente Java se puede acceder en JDT mediante "Java Model" o "Abstract Syntaxt Tree" (AST). LAPSE utiliza las dos modalidades para tratar el código.

LAPSE implementa tres vistas. La mayor funcionalidad está implementada en la vista "Provenance Tracker" ya que en ella se muestra al usuario el camino de la propagación desde objeto "sink" al objeto "source". En la sección 3.5 se describen los detalles de la implementación de estas tres vistas.

A continuación se describen las tecnologías y los "frameworks" con los que está creado Eclipse. En la sección 3.1 se ofrecen algunos detalles sobre la arquitectura de un plug-in de Eclipse. En las secciones siguientes se explica qué es y cómo se pueden utilizar el "Java Model" (sección 3.2) "AST Tree" (sección 3.3) y "Java Search Engine" (sección 3.4).

3.1 Arquitectura Eclipse Plug-In

Eclipse, tal como se ha definido anteriormente, es una plataforma extensible para la construcción de IDE ("Integrated Development Environment"). Cada una de las herramientas Eclipse, que trabajan juntas para dar soporte a las tareas de programación, se puede encapsular en un solo componente "de enchufe" de ahí el nombre inglés Eclipse plug-in. Además los nuevos plug-ins pueden añadir los elementos de procesado a los plug-ins existentes de ahí la definición de plataforma extensible.

Un plug-in de Eclipse es un componente que provee un cierto tipo de servicios dentro del contexto de Eclipse "workbench". El motor de ejecución de Eclipse proporciona una infraestructura para dar soporte a activación y operación de un conjunto de plug-ins que trabajan juntos para proveer entorno para actividades de desarrollo. Un plug-in está embebido en una instancia de clase plug-in. Cada clase plug-in tiene que extender el *org.eclipse.core.runtime.Plug-in,* una clase abstracta que provee unas facilidades genéricas para manejo de los plug-ins. El directorio de instalación de Eclipse incluye una carpeta con el nombre "plug-ins" donde están desplegados los plug-ins individuales. Cada plug-in se describe en un fichero de manifiesto XML, llamado "plugin.xml" y contiene información necesaria para el motor de ejecución Eclipse respecto a la activación del plug-in.

Eclipse incluye un *kernel* especifico para plug-ins, llamado Eclipse *platform* o Eclipse *runtime* y algunos plug-ins centrales que se incluyen en todos los entornos de desarrollo eclipse. Sus identidades están codificadas en la plataforma Eclipse, la cual sabe cuándo activarlas. Otros plug-ins (no centrales) se activan cuando son necesitados por otros plug-ins.

En el modelo eclipse un plug-in puede ser relacionado con otro plug-in atavés de uno de los dos tipos de relaciones existentes:

- **Dependencia**: Los roles en esta relación son el plug-in dependiente y plug-in de prerrequisito. El de prerrequisito respalda las funciones del plug-in dependiente.
- **Extensión**: Los roles en esta relación son plug-in de host y plug-in de extensión. El plug-in de extensión, como su nombre indica, extiende las funciones del plug-in de host.

Estas relaciones se especifican de forma declarativa en los ficheros de manifiesto "plug-in.xml", a través de los elementos *requieres* y *extension.* Un plug-in no central que se ha desplegado en Eclipse, sólo puede ser activado en una instancia ejecutable de Eclipse si está relacionado de forma transitiva con algún plug-in central, a través de unión de las relaciones de dependencia y/o extensión.

El usuario puede acceder a las funcionalidades ofrecidas por un plug-in, a través de los elementos de interfaz de usuario, que deben ser añadidos al "workbench" de Eclipse para que el plug-in pueda ser utilizado. El proceso de adición de estos elementos se llama "extensión". Una "extensión" se define con el plug-in de "extensión" y causa el plug-in que la utiliza, llamado el plug-in de "host", a modificar su comportamiento. Normalmente se añaden unos nuevos elementos de procesamiento al plug-in de "host" (por ejemplo: adición de nuevos elementos de un menú) y se personaliza su comportamiento mediante los servicios definidos en el plug-in de "extensión".

En el Listado 12 se observa el contenido de fichero "plugin.xml" para el LAPSE. En este fichero se puede apreciar que LAPSE extiende a tres plug-ins diferentes. Cada extensión está definida en el fichero "plug-in.xml" como un punto de extensión (*extension point*). En LAPSE existen tres puntos de extensión:

- ***org.eclipse.ui.views:*** permite a los plug-ins que lo extienden añadir vistas al "workbench" de Eclipse. LAPSE añade tres vistas: "ProvenanceTracker" definido en la clase lapse.views.LapserView, "SourceView" definido en la clase *lapse.views.SourceView* y "SinkView" definido en la clase *lapse.views.SinkView*. Las tres clases correspondientes a las vistas extienden la clase *ViewPart* que especifica la interfaz para todas las vistas Eclipse.
- ***org.eclipse.ui.perspectiveExtensions*** permite a los plug-ins añadir sus propios conjuntos de acciones, vistas o atajos a las perspectivas existentes. El elemento "targetId" indica la perspectiva a la cual el plug-in añade un nuevo comportamiento. En caso de LAPSE el targetId es *eclipse.ui.resourcePerspective.* A esta perspectiva se le añade una vista nueva *lapse.views.lapse.* Es necesario que una vista nueva se relacione con una ya existente en la perspectiva objeto, y en caso de LAPSE se relaciona a la vista *org.eclipse.ui.views.TaskList*. Finalmente el elemento "relationship" especifica el tipo de relación entre las vistas indicadas, en caso de LAPSE a la nueva vista se colocará en la parte derecha de la vista relacionada, ocupando 50% de la misma (ratio 0,5).

- ***org.eclipse.ui.perspectives*** permite a los plug-ins definir sus propias perspectivas. La clase que implementa una nueva perspectiva LAPSE es *lapse.PerspectiveFactory*

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plug-in>
     <extension
         point="org.eclipse.ui.views">
      <category
            name="LAPSE: Static Analysis for Security"
            id="lapse">
      </category>
      <view
            allowMultiple="false"
            class="lapse.views.LapserView"
            icon="icons/e_search_menu.gif"
            category="lapse"
            name="Provenance Tracker"
            id="lapse.views.lapse">
      <view
            allowMultiple="false"
            category="lapse"
            class="lapse.views.SourceView"
            icon="icons/d/setfocus.gif"
            id="lapse.views.SQLSourceView"
            name="Vulnerability Sources"/>
      <view
            allowMultiple="false"
            category="lapse"
            class="lapse.views.SinkView"
            icon="icons/d/setfocus.gif"
            id="lapse.views.SinkView"
            name="Vulnerability Sinks"/>
   </extension>
   <extension
         point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
            targetID="org.eclipse.ui.resourcePerspective">
         <view
               ratio="0.5"
               relative="org.eclipse.ui.views.TaskList"
               relationship="right"
               id="lapse.views.lapse">
         </view>
      </perspectiveExtension>
   </extension>
   <extension
         point="org.eclipse.ui.perspectives">
      <perspective
            class="lapse.PerspectiveFactory"
            name="Static analysis"
            id="lapse.perspective"/>
   </extension>
</plug-in>
```

Listado 12: Contenido del Fichero “plug-in.xml”

Para la implementación gráfica de las vistas LAPSE utiliza SWT ("Standard Widget Toolkit") y JFace. SWT es una herramienta de Eclipse para creación de interfaces de usuario adaptables al sistema operativo donde se implementa la aplicación, mientras que JFace, otra herramienta Eclipse, posibilita separar el modelo de datos de la interfaz del usuario. JFace actúa como un puente entre los componentes SWT de bajo nivel y los objetos de dominio del programa personalizado. Los componentes SWT interactúan con el sistema operativo y no tienen constancia de los objetos de dominio del programa concreto.

Una de las maneras en las que JFace conecta a los componentes SWT con los objetos del dominio del programa es a través de vistas. Vistas JFace están formadas por los componentes SWT (arboles, tablas, etc.) y los objetos de dominio. El programador proporciona a los componentes la información que necesitan para rellenarlos. La vista es capaz de filtrar, organizar y actualizar los objetos de dominio del programa concreto. En las siguientes secciones, dedicadas a explicar la implementación de cada una de las tres vistas de LAPSE, se explican los detalles de uso de SWT y JFace para implementar la interfaz de usuario.

Las acciones son las últimas, pero no menos importantes, características de Eclipse utilizadas por LAPSE son las acciones. Las acciones se sujetan a los botones y opciones del menú en las aplicaciones de Eclipse. Cada acción tiene implementado un método *run,* que se invoca cada vez que el usuario pulsa el botón o el elemento del menú. Cada una de las vistas LAPSE define sus propias acciones, que se explicarán en las secciones correspondientes.

3.2 Eclipse Java Model

"Eclipse Java Model" es una representación de proyecto Java, sencilla y tolerante a fallos. No contiene tanta información como AST pero es rápida a la hora de volver a crearse en caso de modificaciones. El "Java Model" se define en el paquete *org.eclipse.jdt.core* y se representa mediante una estructura de árbol, cuyos elementos están descritos en la siguiente tabla:

Elemento del Proyecto	**Elemento de Java Model**	**Descripción**
Workspace	*IJavaModel*	Workspace de Eclipse
Proyecto Java	*IJavaProject*	El proyecto Java que contiene todos los demás objetos
Carpeta "src" o "bin" o una librería externa	*IPackageFragmentRoot*	Contiene ficheros fuente o binarios. Puede ser una carpeta o una librería (fichero zip/jar)
Cada carpeta	*IPackageFragment*	Cada paquete o subpaquete esta debajo de IPackageFragmentRoot,
El fichero con el código fuente Java	*ICompilationUnit*	El fichero fuente esta siempre debajo del nodo paquete
Tipos/Campos/Metodos	*IType / IField / IMethod*	Los tipos, campos y métodos

Tabla 9: Elementos Básicos de Eclipse Java Model

LAPSE utiliza "Java Model" para acceder a los proyectos abiertos en el "workspace" actual.

3.3 Eclipse Abstract Syntax Tree (AST)

AST es una detallada representación del código Java, en forma de árbol. AST define API para modificar, crear, leer y borrar el código fuente. El API AST está en el paquete *org.eclipse.jdt.core.dom* en el plug-in *org.eclipse.jdt.core*. Cada elemento en un fichero fuente Java es representado por una subclase de *ASTNode*. Cada nodo AST específico provee una información sobre el objeto que representa. Por ejemplo, existe el nodo *MethodDeclaration* para métodos, *VariableDeclarationFragment* para declaraciones de variables y *SimpleName* para cada cadena que no sea una palabra clave de Java. El AST es típicamente creado en base a *ICompilationUnit* del modelo Java. Cada nodo AST pertenece a una única instancia, llamada instancia dueño. Cada hijo de un nodo AST siempre tiene el mismo dueño que su nodo padre. La clase *ASTParser* es la encargada de parsear el código fuente Java y devolver el AST que lo represente.

Es muy común utilizar el patrón Visitor para procesar el AST. Visitor es un patrón que permite definir una nueva operación para una jerarquía, sin tener que modificar sus clases. Su funcionamiento es el siguiente:

1. Se añade el método *accept()*, a alguna o a todas las clases en la jerarquía de clases. Cada implementación de este método acepta como argumento, el objeto que implemente la interfaz creada para el patrón especifico.
2. Se crea una interfaz con un conjunto de operaciones que comparten el mismo nombre; el nombre suele ser –*visit-,* pero que aceptan diferentes tipos de argumentos. Declara una operación de este tipo para cada clase de jerarquía que permite la extensión de Visitor.

La condición necesaria para Visitor es que para cada clase que implemente el método *accept()* exista un método *visit* que tome como argumento una instancia de esta clase. En la Ilustración 4 se puede observar el esquema UML del patrón Visitor. Hay dos visitores concretos y dos elementos visitados. Cada uno de los visitores concretos implementa el método *visit* con cada uno de los elementos concretos como argumento. Estos elementos por su parte implementan el método *accept* para la interfaz del Visitor.

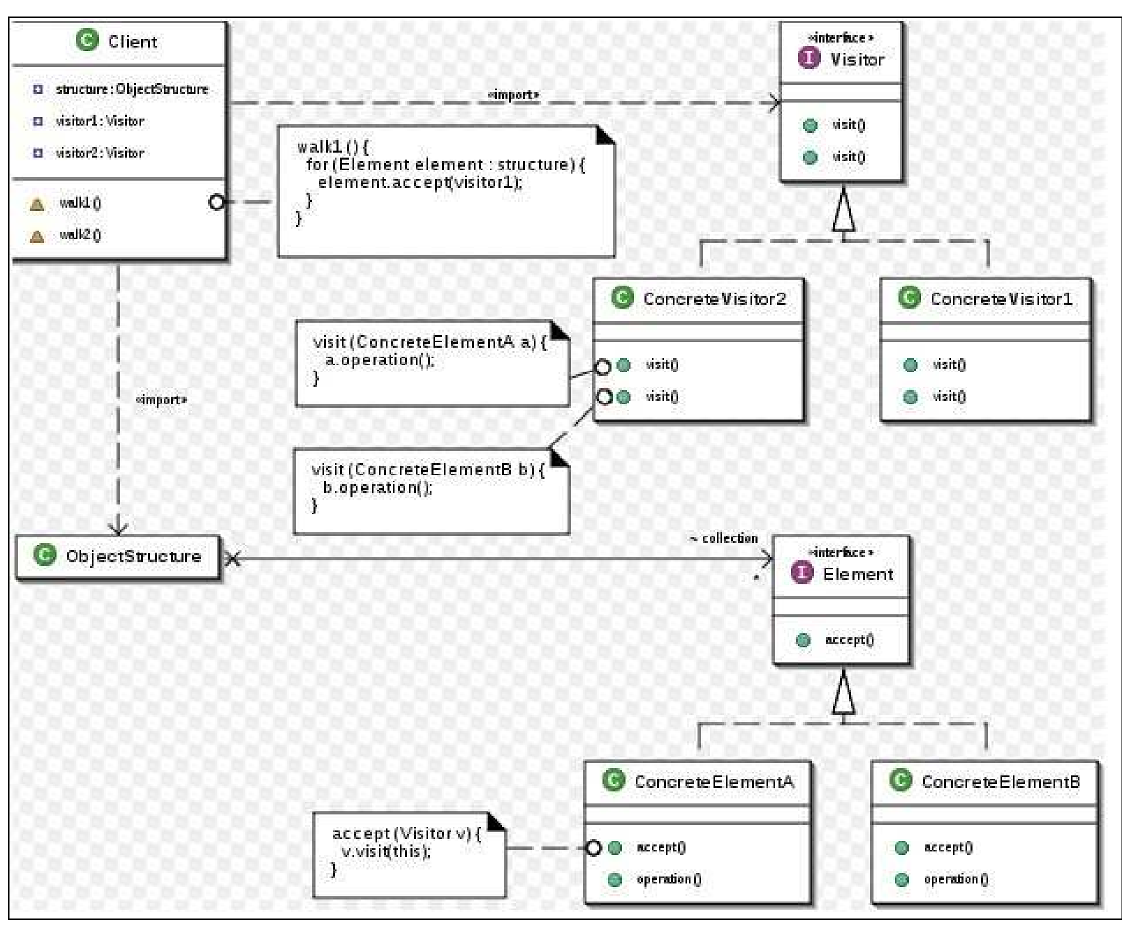


Ilustración 4: Diagrama UML del Patrón Visitor (Ilustración Tomada de Wikipedia)

LAPSE define una clase NodeFinder que representa a un visitor concreto. Esta clase extiende a un Visitor Genérico, *ASTVisitor,* que forma parte del AST de Eclipse y está definido en el paquete *org.eclipse.jdt.core.dom.* En ella implementa el método *VisitNode* que se invoca sobre cada nodo AST visitado. Este método comprueba si el nodo pasado como argumento está cubierto (su posición en el código es menor o igual) que el nodo de la instancia de *NodeFinder* actual. El nodo de la instancia *NodeFinder* define el patrón de búsqueda para el análisis en cada momento.

3.4 Motor de Búsqueda de Eclipse

Para realizar un escaneo eficiente y fijar el patrón de búsqueda LAPSE utiliza el motor de búsqueda de JDT. En concreto el paquete que define el API necesario para el funcionamiento de motor es *org.eclipse.jdt.core.search.* Este API permite realizar búsquedas de los elementos Java, tales como referencias a métodos, declaraciones de campos, implementaciones de interfaces, etc. en los proyectos Java del workspace de Eclipse.

El punto de entrada del motor de búsqueda de JDT es la clase *SearchEngine*, la que realiza búsqueda de un patrón específico, creado con el método *CreatePattern*, dentro de un alcance definido con el método *createJavaSearchScope*. Una vez definido el patrón y el alcance, se utiliza el método *search* para encontrar los resultados. Posteriormente estos resultados se acceden a través de una extensión específica de la clase *SearchRequestor*. Esta extensión sobrescribe el método *acceptSearchMatch,* que se invoca cada vez que el motor encuentra un resultado. Este resultado viene en forma de la instancia de la clase *SearchMatch.* El método comprueba el grado de adecuación con el que se recupera el resultado ("match") concreto y define las acciones que hay que tomar cada vez que se acepta un "match".

LAPSE realiza dos tipos de búsqueda utilizando el motor *SearchEngine*. Uno para encontrar todas las declaraciones de un método específico y otro para encontrar todas las referencias (invocaciones) a un método especifico. Estos dos procesos de búsqueda siguen los siguientes pasos:

1) Se crea el patrón de búsqueda con el método *SearchPattern.createPattern().* El patrón está constituido por el nombre del método a encontrar, el tipo del método (constructor o método normal), el tipo de resultado de búsqueda, que en caso de LAPSE son o bien declaraciones (*IJavaSearchConstants.DECLARATIONS*) o bien las referencias (*IJavaSearchConstants.REFERENCES)* y la regla de ajuste que es del tipo exacto y sensible al tamaño de la letra. *(R_EXACT_MATCH | R_CASE_SENSITIVE).*
2) Se invoca el método *SearchEngine.search(),* con el patrón definido en el punto anterior y la extensión de la clase *SearchRequestor*, definida en LAPSE como *MethodDeclarationSearchRequestor* para las declaraciones y *MethodReferencesSearchRequestor* para las referencias. El alcance de

búsqueda está constituido por el proyecto actual, incluyendo la carpeta fuente, las librerías de la aplicación y las librerías de sistema.

3) Se recogen los resultados de la búsqueda. Por cada resultado se obtiene una instancia de la clase *SearchMatch*, de la cual se recupera la instancia del *ICompilationUnit* (ver Tabla 9) a la que pertenece el elemento encontrado, el nodo AST correspondiente a la declaración encontrada y el recurso que contiene el resultado (fichero, paquete, etc.). Si se trata de la referencia al método, se comprueba si el nodo AST recuperado o su padre es una invocación del método o una creación de instancia y se guarda el resultado de la comprobación en forma de una instancia de la clase *Expression* (parte de AST). En caso de la declaración del método se guarda el nodo AST recuperado o su padre como instancia de clase *MethodDeclaration* (parte de AST). Estos cuatro datos sobre el resultado se encapsulan en un objeto *MethodDeclarationUnitPair* si se trata de la declaración o *ExprUnitResourceMember* si se trata de la referencia y se añaden al listado con los resultados encontrados hasta este momento.
4) Se devuelve el listado con todos los resultados encontrados.

En las secciones siguientes se explica cómo este proceso se utiliza por cada una de las vistas definidas en LAPSE.

3.5 Implementación de las Vistas Principales

LAPSE, como se ha indicado en la parte de Antecedentes, se distribuye en forma de plug-in de eclipse, que está constituido por tres vistas principales. Cada una de estas vistas tiene su propia implementación, y además LAPSE utiliza otros plug-ins y librerías de Eclipse que facilitan la tarea de análisis del código. Los detalles de las tres librerías más importantes, y la forma en la que LAPSE hace uso de ellas, se han descrito en las secciones anteriores. LAPSE implementa el proceso de registro de errores, de forma que todos los avisos sobre los comportamientos no típicos de la aplicación se guardan en un fichero *.log* en la carpeta *.metadata* ubicada en el "workspace" del proyecto.

En la Ilustración 5 se muestra el diagrama de clases UML del LAPSE. Este diagrama tiene como objetivo mostrar el diseño general de la herramienta con sus clases y métodos más importantes.

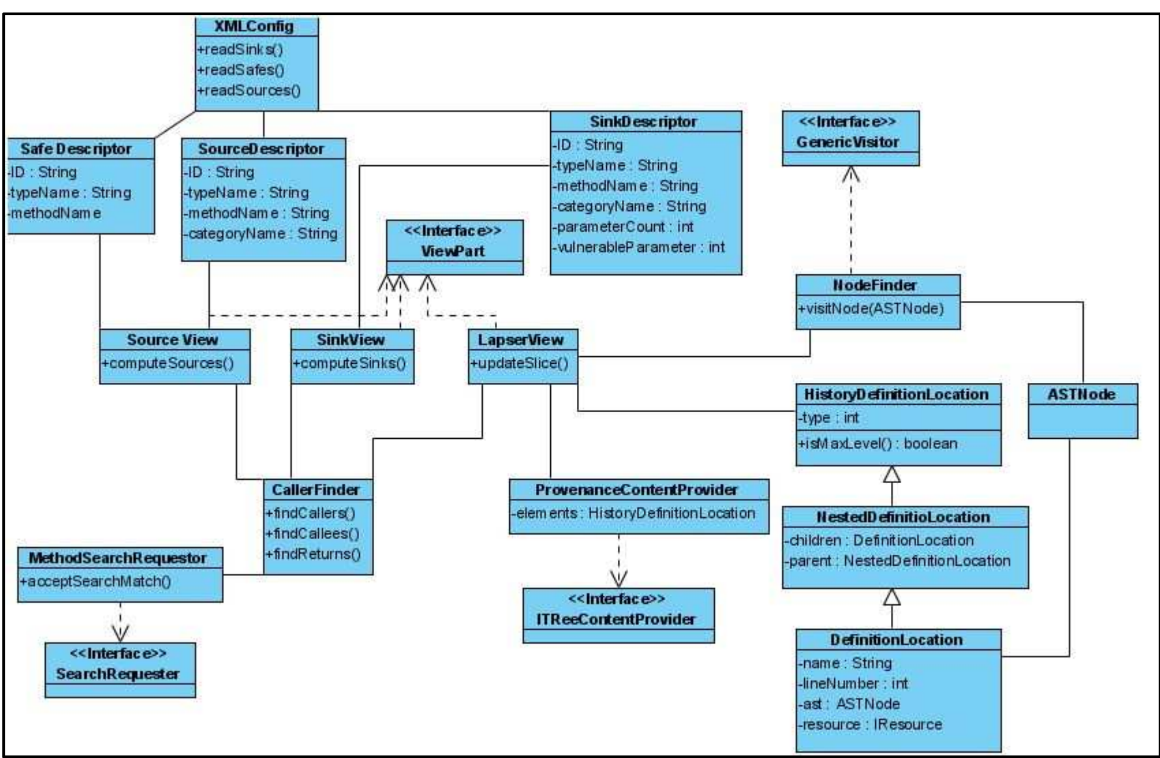


Ilustración 5: Diagrama de Clases UML del LAPSE

A continuación se describe de forma breve cómo está implementando el plug-in y cómo se integra en el desarrollo Java. La descripción se divide en las tres vistas que implementa LAPSE.

3.5.1 SourceView

En esta vista se muestran todas las posibles fuentes "*sources*" de datos de usuario. Para el análisis en busca de objetos de tipo "source" LAPSE utiliza un fichero "*sources.xml*". En este fichero se describen todos los elementos "source" que pueden ser fuente de una posible vulnerabilidad. El fichero tiene la estructura especificada en el dtd descrito en el Listado 13

```
<!ELEMENT SOURCES (source*)>
<!ELEMENT SOURCE (category)>
<!ELEMENT CATEGORY #CDATA>
<!ATTLIST SOURCE  ID CDATA #REQUIRED>
```

Listado 13: Fichero "sources.dtd"

El atributo "ID" del elemento "SOURCE" expresa el nombre completo del método cuya invocación puede ser una posible fuente de la vulnerabilidad, mientras que el elemento "CATEGORY" expresa el tipo de la vulnerabilidad.

Una vez leído el listado de objetos "source", se recorre el código en busca de líneas donde se invocan estos métodos. Para buscar las invocaciones en el código, de los métodos definidos en el fichero "sources.xml", se hace uso del mecanismo de búsqueda *SearchEngine* de Java. (ver sección 3.4).

En LAPSE el proceso de búsqueda de los objetos "source" de las vulnerabilidades es el siguiente:

1) Se lee la entrada <source> del fichero "source.xml".
2) Se encapsula la entrada leída en la clase *SourceDescriptor*, donde aparte del nombre completo del método (incluyendo el paquete en el que está definido), se guarda la información sobre la categoría de la vulnerabilidad que puede ser provocada por la invocación del método.
3) Se busca a través del proceso definido en sección 3.4 todas las referencias (invocaciones) del método "source" actual.
4) Los pasos de 1) a 3) se repiten por cada entrada del fichero "sources.xml" y por cada proyecto abierto en el workspace actual del Eclipse.
5) Finalmente se recupera el listado con todos los resultados obtenidos en el proceso de búsqueda. Por cada elemento del listado se comprueba si su nodo

AST es del tipo *MethodInvocation*, es decir si se trata de una invocación de método, y si es así se añade el resultado en la vista "SourceView" de usuario.

Para mostrar los resultados de búsqueda al usuario "Source View", implementa las siguientes vistas de JFace:

- **Table Viewer:** muestra una tabla con los nombres de columnas determinadas por el programador. En caso de "SourceView", estas columnas se corresponden con los campos de la clase *SourceDescriptor*, descritos en el paso 2) del proceso de búsqueda. La implementación de esta vista contiene también el menu contextual que el usuario puede invocar desde cualquier fila de la tabla. Este menu permite iniciar la búsqueda de los objetos "source" o copiar la selección actual al portapapeles.
- **StructuredContentProvider:** proveedor del contenido para la vista "Table Viewer", contiene un listado con los elementos "sink" encontrados. Cada elemento esta encapsulado en una clase "ViewMatch" y contiene datos acerca de un objeto "source" concreto, como su nodo AST, el recurso donde aparece, etc.
- **TableLabelProvider:** contiene métodos para proveer el texto e imagen de cada columna para un elemento concreto, en caso de LAPSE el elemento concreto es una instancia de "ViewMatch", del que se extraen datos para llenar cada columna.
- **ViewerSorter:** provee métodos para comparar datos en cada columna, para poder reorganizar el contenido de la vista. Se puede ordenar los datos según el nombre de cada columna, de forma ascendente o descendente.

La vista "Source View" proporciona también un menú principal (ver Ilustración 6), el cual permite filtrar los resultados. Se pueden filtrar los resultados para ver los objetos "source" sin código fuente, por ejemplo los que aparecen en las librerías. También se pueden ver las declaraciones de todos los métodos *main* y métodos de tipo "set" de las tareas de "Ant". Tanto los métodos *main*, como los métodos *set* de "Ant", aunque no es muy común en las aplicaciones web, pueden ser una posible fuente de la vulnerabilidad por el hecho de aceptar los argumentos de usuario en la invocación por consola o desde el fichero "build.xml" en caso de "Ant".

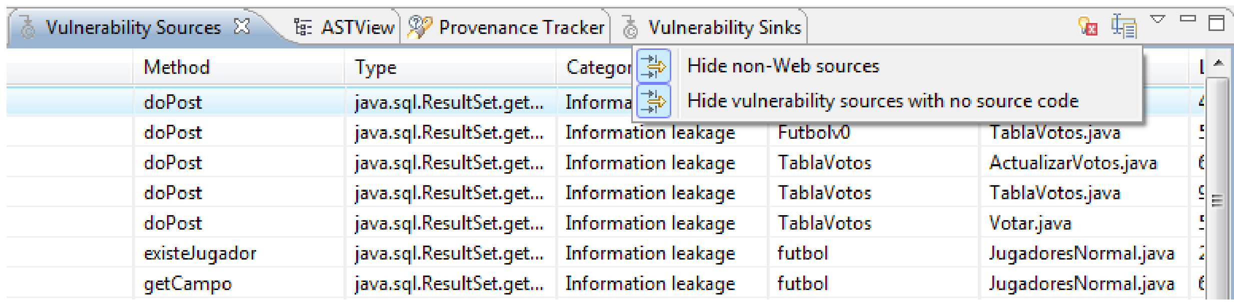


Ilustración 6: Menú Principal de la Vista "Source View"

Finalmente, "Source View" implementa cinco acciones. Dos se corresponden con el menú contextual e implementan la búsqueda de objetos "source" y la realización de copia de la selección a portapapeles. Las otras dos se corresponden con las opciones del menú principal, descritas con anterioridad. Las funciones del menú contextual aparecen también en la barra de tareas, como se puede observar en la Ilustración 7. La última acción, correspondiente a un "doble click" sobre una fila, abre el fichero fuente del objeto "source" y selecciona la línea correspondiente a la invocación del mismo, en el editor de Eclipse.

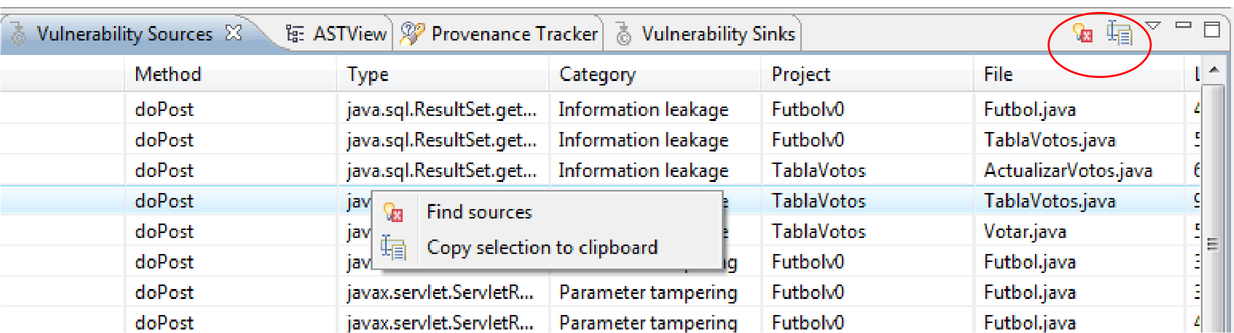


Ilustración 7: Menú Contextual de la Vista "Source View"

LAPSE utiliza también el fichero "safes.xml", donde se especifican todos los métodos que se consideren seguros. Por ejemplo si se sabe que para una vulnerabilidad particular un método siempre es seguro, aunque aparezca en el fichero "sources.xml", se puede incluir en el fichero "safes.xml". Cuando se realiza la propagación en la vista "Provenance Tracker" se comprueba por cada invocación de un método si pertenece a "safes.xml" y si es así se muestra en el color verde para indicar que no hay peligro.

El Listado 14 muestra el esquema para el fichero "safes.xml". El atributo "ID" del elemento "SAFE" expresa el nombre completo del método considerado seguro. Los

elementos “TYPE” y “METHOD”, expresan el tipo del método y su nombre simple respectivamente y el elemento “PARAMCOUNT” indica el número de parámetros que posee el método. El elemento “CATEGORY” expresa el tipo de la vulnerabilidad para la que se considera seguro el método.

```
<!ELEMENT SAFES (safe*)>
<!ELEMENT SAFE (type, method, paramCount, category)>
<!ELEMENT TYPE  #CDATA>
<!ELEMENT METHOD  #CDATA>
<!ELEMENT PARAMCOUNT #CDATA>
<!ELEMENT CATEGORY #CDATA>
<!ATTLIST SAFE  ID CDATA #REQUIRED>
```

Listado 14: Fichero "safes.dtd"

3.5.2 SinkView

La vista “SinkView” implementa la búsqueda y la presentación al usuario de llamadas que manejan cadenas que serán ejecutadas de forma maliciosa. Todos los métodos de tipo “sink”, se definen en el fichero “sinks.xml”. Este fichero tiene una estructura definida con el “dtd” en el Listado 15.

```
<!ELEMENT SINKS (sink*)>
<!ELEMENT SINK (paramCount, vulnParam, category)>
<!ELEMENT PARAMCOUNT #CDATA>
<!ELEMENT VULNPARAM  #CDATA>
<!ELEMENT CATEGORY #CDATA>
<!ATTLIST SINK  ID CDATA #REQUIRED>
```

Listado 15: Fichero "sinks.dtd"

El atributo “ID” del elemento “SINK” expresa el nombre completo del método cuya invocación sobre un objeto sin validar puede ser un posible ataque. El elemento “PARAMCOUNT” indica el número de parámetros que posee el método y el elemento “VULNPARAM” especifica el número de parámetro vulnerable, empezando por 0. El elemento “category” expresa el tipo de la vulnerabilidad provocada por este método “sink”, e.g “Cross Site Scripting”.

Una vez leído el listado de objetos “sink”, se recorre el código en busca de líneas donde se invocan estos métodos. Para buscar las invocaciones en el código, de los métodos definidos en el fichero “sink.xml”, se hace uso, como en caso de “sources.xml” del mecanismo de búsqueda *SearchEngine* de Java (ver sección 3.4).

En LAPSE el proceso de búsqueda de los objetos “sink” de las vulnerabilidades es el siguiente:

1. Se lee la entrada <sink> del fichero “sink.xml”.
2. Se encapsula la entrada leída en la clase *SinkDescriptor*, donde aparte del nombre completo del método (incluyendo el paquete en el que está definido), se guarda la información sobre el número de parámetros que posee el método, el número de parámetro vulnerable y la categoría de la vulnerabilidad de seguridad que puede provocar la invocación del método.
3. Se busca a través del proceso definido en (ver sección 3.4) todas las referencias (invocaciones) del método “sink” actual.

4. Se comprueba si el parámetro de llamada es o apunta a una cadena constante, para poder diferenciar las invocaciones peligrosas de las que no lo son.
5. Los pasos de 1) a 3) se repiten por cada entrada del fichero “sinks.xml” y por cada proyecto abierto en el “workspace” actual del Eclipse.
6. Se recupera el listado con todos los resultados obtenidos en el proceso de búsqueda. Por cada elemento del listado se comprueba si su nodo AST es de tipo *MethodInvocation* o *ClassInstanceCreation*, es decir si se trata de una invocación de método o creación de la instancia de clase respectivamente y si es así se comprueba el argumento definido como vulnerable para averiguar si la invocación es segura o no. Todos los argumentos que no sean una cadena constante se tratan como inseguros “tainted”. Esto incluye referencias a otro objeto, concatenación de cadenas con alguna no constante o invocación de método
7. Finalmente se añaden los resultados en la vista “SinkView” de usuario.

Para mostrar los resultados de búsqueda al usuario “Sink View” implementa las mismas cuatro vistas de JFace que el “Source View” (ver sección anterior).

En la vista “SinkView”, de manera parecida que en la “SourceView” se implementan varios filtros, aplicables a los resultados del análisis. Dado que en el proceso de búsqueda se comprueba si la invocación del método “sink” es peligrosa o no (ver 7), se pueden filtrar los resultados de forma que se muestren solamente los definidos como inseguros. Además, como en el caso de “SourceView”, se pueden ver también los resultados encontrados sin código fuente (los que aparecen en las librerías del proyecto). Por otro lado “SinkView” permite filtrar los resultados para ver únicamente los de una sola categoría de vulnerabilidad, por ejemplo solamente los de “Cross Site Scripting”. Todas estas opciones de filtrado aparecen en menú principal de la vista (ver Ilustración 8).

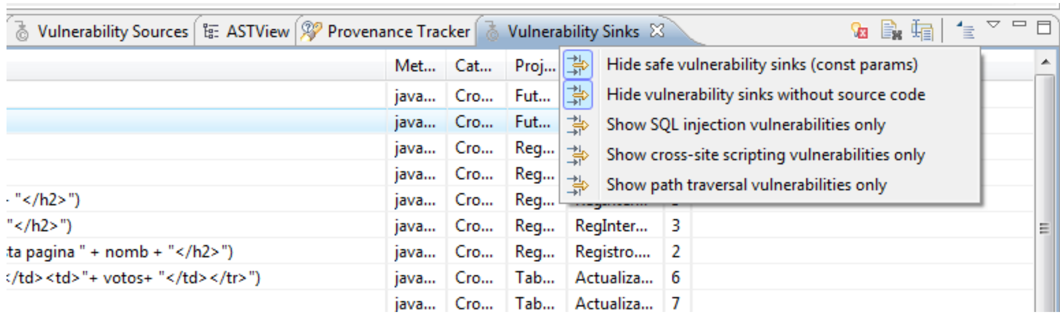


Ilustración 8: Menú Principal de la Vista "Sink View"

La vista “Sink View” implementa también un menú contextual (ver Ilustración 9). Una de las opciones de este menú permite hacer la propagación hacía atrás para encontrar camino a fuente “source” de los datos utilizados en el método “sink”. Esto es el proceso clave de LAPSE. Si de una sentencia de tipo “sink”, se puede llegar a un objeto de tipo “source”, mediante propagación reversa, a través de declaraciones de variables, invocaciones de métodos, asignaciones, etc. Esto significa que el programa es vulnerable al ataque de la categoría especificada. Cuando el usuario escoge la opción de “propagación hacía atrás” desde un resultado “sink” elegido, se muestra automáticamente la vista “Provenance Tracker”, en la que se enseña el camino buscado. Dado que la vista “Provenance Tracker” es la responsable de implementar esta propagación, los detalles de cómo se lleva acabo se describen en la siguiente sección. Las otras dos opciones del menú contextual son parecidas a las vistas en el mismo menú de “Source View” una sirve para buscar todos los objetos de tipo “sink” y a otra para copiar la selección al portapapeles.

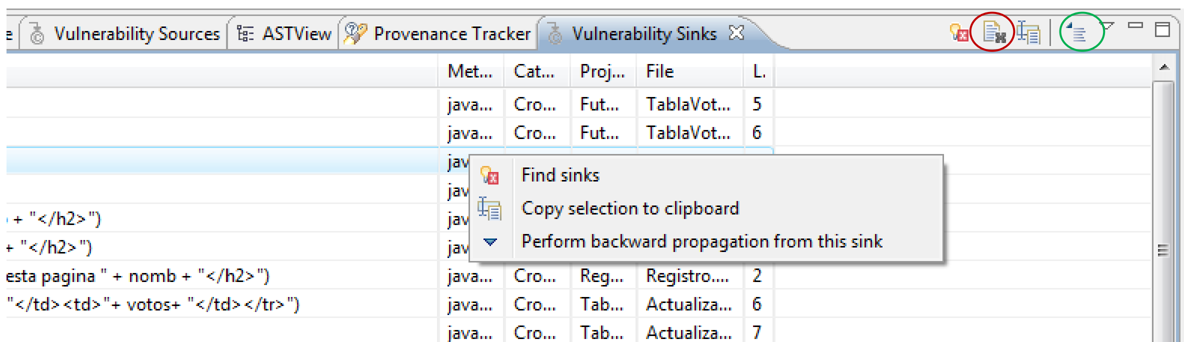


Ilustración 9: Menú Contextual de la Vista "Sink View"

La vista “Sink View” define once acciones. Tres correspondientes al menú contextual, cinco correspondientes al menú principal y una correspondiente a la acción “doble click” que abre el código fuente de la invocación del objeto “sink” seleccionada en el editor de Eclipse. Además implementa dos acciones que se pueden invocar desde la barra de tareas y son “toggle safe state”, marcado con el círculo rojo en la Ilustración 9 y “Get sink statistics” marcado con el circulo verde en la misma ilustración.

La acción de “toggle safe status”, permite cambiar el estado de un objeto “sink” de seguro a inseguro y al revés, lo cual es útil cuando el usuario quiere reflejar que un objeto “sink” es inseguro a pesar de que LAPSE no lo haya detectado como tal.

La acción “get sink statistics” muestra unas estadísticas del proceso de búsqueda de objetos “sink”. Como se puede apreciar en la Ilustración 10, las estadísticas muestran el número total de los objetos “sink” encontrados, el número total de los inseguros y el número total y de los inseguros de cada categoría definida. También indica el número de ocurrencias de los objetos “sink” en el código fuente. (Los demás ocurren en fuentes binarias, como librerías).

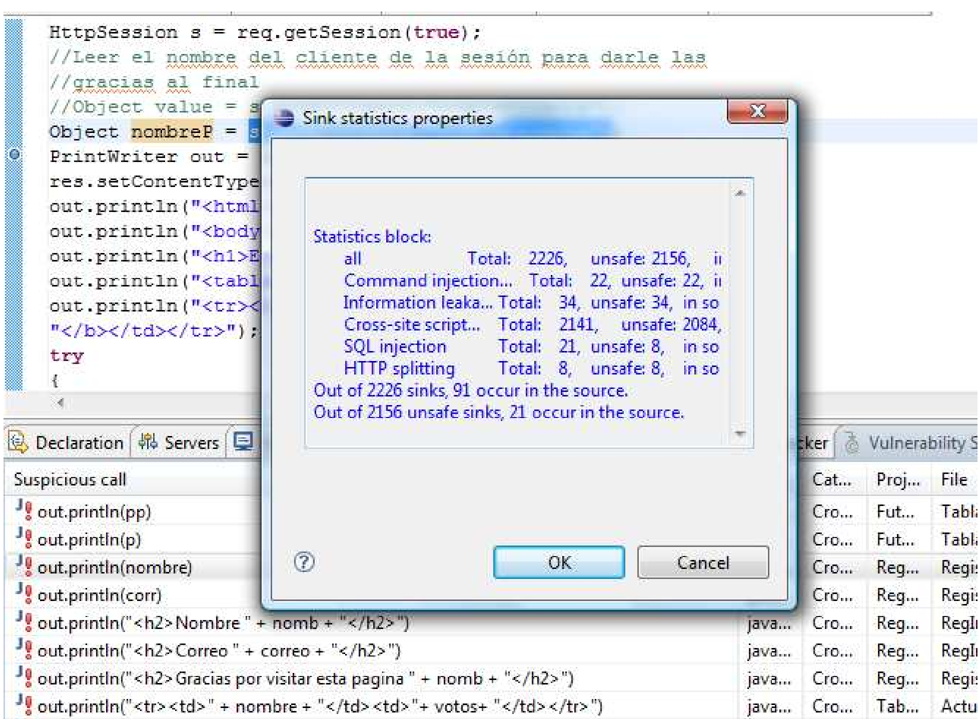


Ilustración 10: Cuadro de Estadísticas de los Objetos "sink" Encontrados

3.5.3 Provenenance Tracker

La vista “ProvenanceTracker” implementa el proceso de propagación de los datos manejados en métodos identificados como “sink” al origen de las mismas. Si el origen coincide con alguna invocación del tipo “source”, el programa es vulnerable al ataque definido.

El elemento en el cual se empieza la propagación se puede elegir de dos formas. Escogiendo un método “sink” en la vista “sink view” y eligiendo en el menú contextual la opción “Perform Backward Propagation from the Sink” o bien seleccionado alguna expresión en uno de los ficheros fuente abiertos en el editor en “workspace”. EL proceso para realizar la propagación desde un método identificado como “sink” es el siguiente:

1. Se comprueba si el argumento vulnerable del método “sink” es un identificador simple.
2. Si es la primera vez que se accede a un determinado fichero fuente (*ICompilationUnit*), se recorre dicho fichero con el patrón ASTVisitor (ver sección 3.3) y se guarda la información sobre las variables en una tabla Hash. Por cada declaración se especifica si es global o local y, en caso de ser local, en que método está declarado; también se guarda la información sobre las variables de tipo “final”.
3. Por cada identificador que se comprueba se busca su correspondiente declaración, previamente guardada en la tabla Hash (después de visitar los nodos AST con el *visitor* específico - *DeclarationFinderVisitor*) y se procesa dicha declaración. La declaración se diivide entre la declaración formal de un parámetro de método y la declaración de variable.
4. Si el identificador esta declarado como parámetro de algún método:
   4.1. Se buscan todas las invocaciones de este método.
   4.2. Por cada invocación se recupera el argumento correspondiente (que aparece en la misma posición) al identificador y se le encapsula como expresión junto con su *CompiltationUnit* y su recurso.
   4.3. Se procesa por separado cada expresión encontrada en b, según el proceso descrito en 5.
5. En caso de que el identificador sea una variable se comprueba su inicialización, en concreto la parte que aparece a la derecha en la expresión de la inicialización.
   5.1. Si la expresión es nombre de alguna otra variable, se trata de una asignación y se invoca el proceso 3 para el identificador en cuestión.

5.2. Si la expresión es la invocación de un método, se buscan las declaraciones de métodos con el mismo nombre. Si se encuentra la declaración adecuada y la opción de análisis dentro de métodos está activada (se activa en el menu de la vista del usuario) se siguen los pasos descritos a continuación:

5.2.1. Se busca con el patrón *Visitor* todas las sentencias del tipo *return* para la declaración en cuestión.

5.2.2. Se procesan todas las expresiones *return* devueltas

5.3. Si es una concatenación de dos cadenas se procesa por separado la parte izquierda y derecha de la expresión, aplicando el proceso de nuevo para cada una.

5.4. Si la expresión es una simple cadena de caracteres u otra expresión indefinida se termina de procesar la declaración.

Además por cada expresión procesada se guarda su historial compuesto de nombre de la expresión, fichero o paquete donde se encuentra, el número de línea en el código, el nodo AST y el tipo pudiendo ser ese último uno de los siguientes:

- Parámetro de llamada al método
- Llamada al método
- Un parámetro formal del método
- Expresión de retorno de un método
- Declaración
- Campo de una clase
- Una cadena constante
- Una concatenación de cadenas
- Asignación
- No definido
- Inicial

Toda esta información se muestra al usuario en la vista “Provenance tracker”. Las expresiones procesadas se organizan en forma de un árbol con la expresión de tipo inicial (el nombre de la variable) como raíz. Cada nodo del árbol guarda la lista de todos sus nodos hijos y de esta forma se puede comprobar que no se procesen más de una vez las mismas expresiones dando lugar a un bucle infinito.

La vista utilizada para mostrar el árbol es la vista JFace “TreeViewer”. El modelo de datos para la implementación de esta vista está formado por el historial de uso de la

variable desde la que se propaga. La apariencia del árbol, es decir el color y las imágenes de los elementos se definen en una implementación de la vista JFace "LabelProvider".

La vista "Provenance Tracker" permite al usuario personalizar algunas de sus propiedades. Por un lado se puede establecer el límite de propagación entre expresiones o se puede optar por no poner ningún límite (este límite por defecto es 10). Por otro lado se puede decidir si se quiere proceder a la propagación dentro de métodos, a través de las expresiones de retorno y el límite de llamadas para esta propagación (este límite por defecto es 10). Finalmente se permite filtrar los resultados de la propagación, incluyendo o no el uso inicial de la variable, los argumentos de llamada, los parámetros formales y las declaraciones de llamada. La ventana que permite configurar todos estos parámetros es la que se muestra en la Ilustración 11.

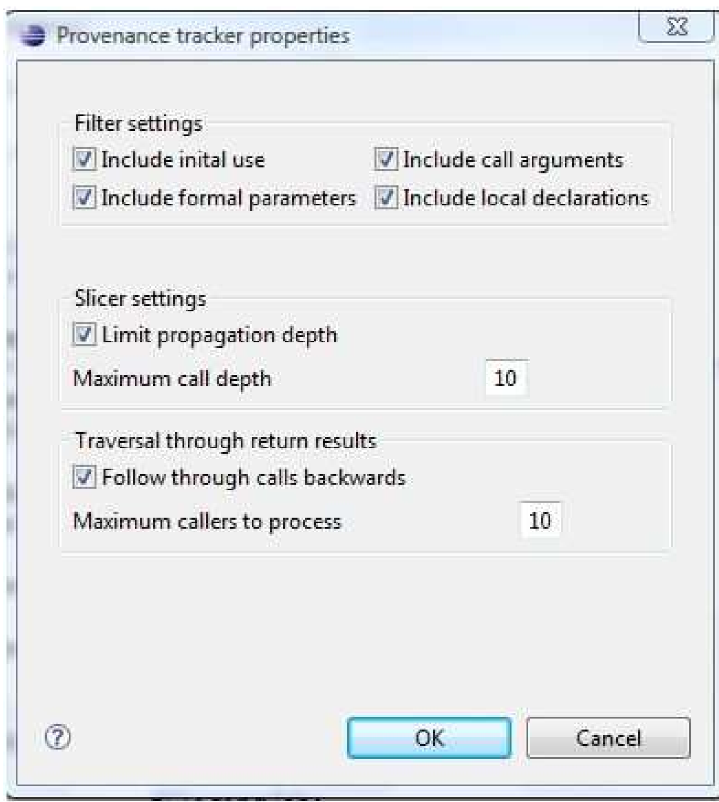


Ilustración 11: La Ventana con Propiedades Modificable de la Vista "Provenance Tracker"

Además de mostrar la ventana de propiedades, "Provenance Tracker" implementa diez acciones más. Las dos que corresponden al menu contextual (ver Ilustración 12) permiten seleccionar todas las ramas del árbol y copiar la selección al portapapeles. Las tres acciones siguientes, son las que se corresponden con los tres primeros iconos en la

barra de tareas (marcadas con el círculo rojo en la Ilustración 12). El primer icono permite realizar la propagación desde una variable seleccionada en el editor de Eclipse. El segundo y tercer icono sirven para expandir y comprimir la vista de árbol respectivamente. Finalmente el último icono que aparece en la barra de tareas (marcada con el círculo verde en la Ilustración 12) se corresponde con la acción de mostrar el historial de diferentes propagaciones realizadas. Este historial guarda los nombres de todos los objetos "sink" desde los que se ha realizado la propagación.

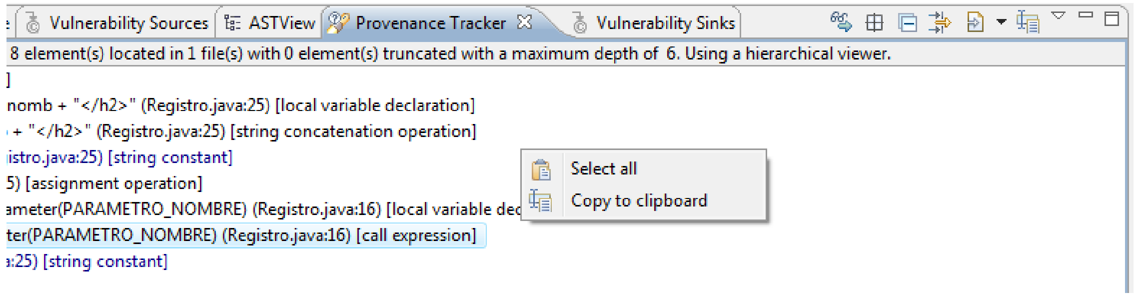


Ilustración 12: Menú Contextual de la Vista "Provenance Tracker"

Por otro lado existen dos acciones más, relacionadas con el menú contextual (ver Ilustración 13).

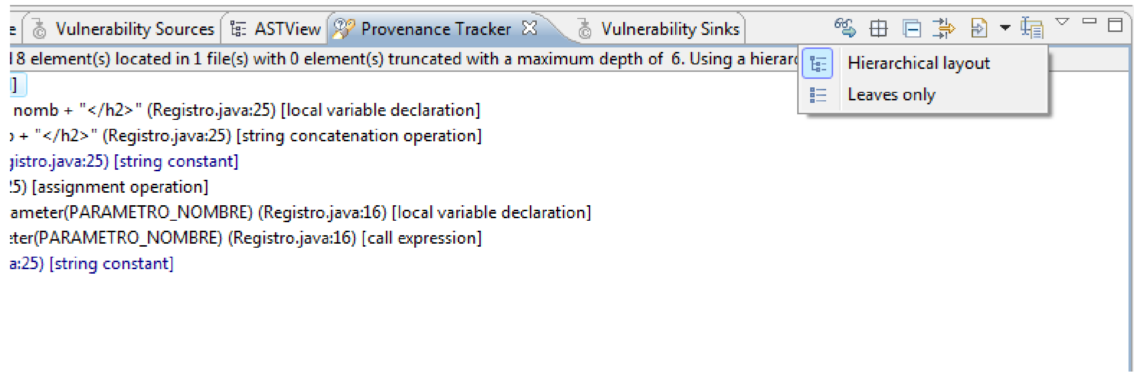


Ilustración 13: Menú Principal de la Vista "Provenance Tracker"

Finalmente la última acción es, como en las otras vistas, la acción de "doble click", que permite abrir la expresión seleccionada en la vista en el editor Eclipse con el correspondiente código fuente.

4 CONSTRUCCIÓN DE UN PLUG-IN MEJORADO. LAPSE+

Después del análisis de LAPSE y las primeras pruebas realizadas con el plug-in, se han detectado varios aspectos mejorables. Con el fin de mejorar el plug-in se ha creado una nueva versión LAPSE+. Este nuevo plug-in esta basado en el diseño original de LAPSE, al que se han introducidos modificaciones con el fin de construir una nueva versión LAPSE+.

En la Ilustración 14 se puede observar el diagrama de clases UML de LAPSE+. El diseño de la aplicación es casi igual que en caso de LAPSE. Lo que cambia sobre todo son algunos detalles de la implementación relacionadas con el funcionamiento del plug-in.

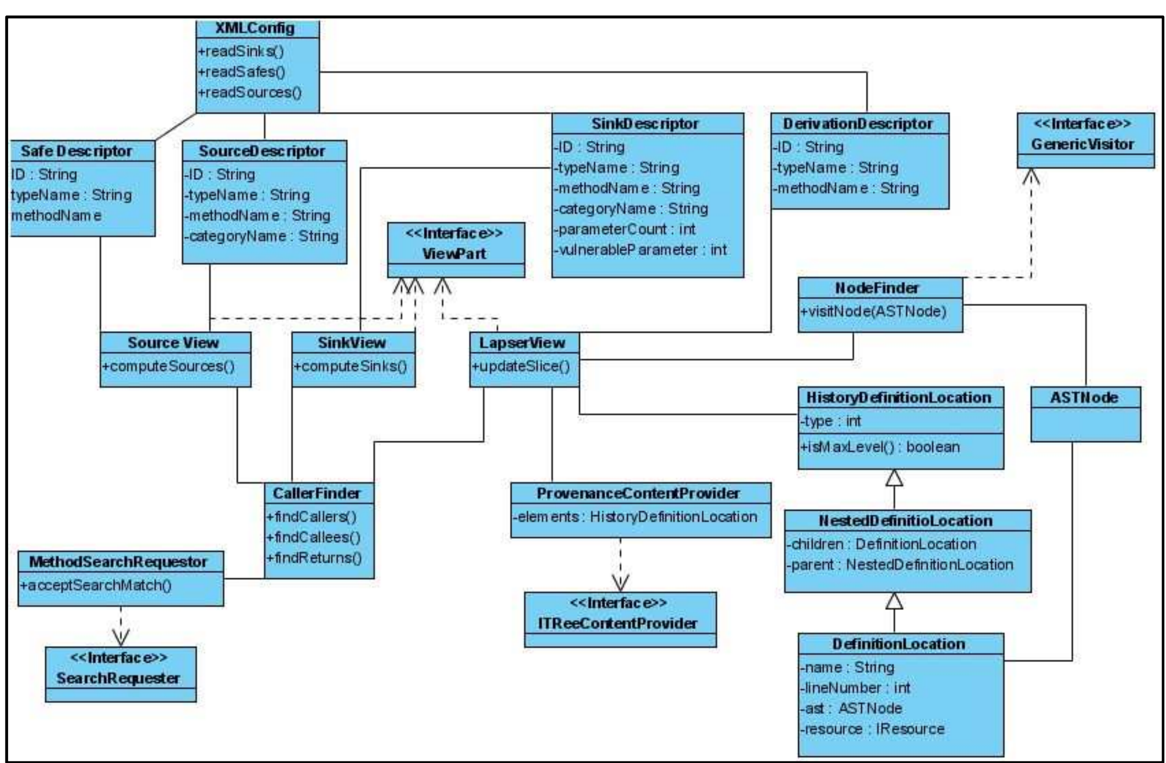


Ilustración 14: Diagrama de Clases UML de LAPSE+

A continuación se detallan las modificaciones realizadas sobre el código original de LAPSE.

4.1 Detección de la Vulnerabilidad "Path Traversal"

El plug-in LAPSE, a pesar de tener definidas en el fichero "sinks.xml" varios métodos vulnerables al ataque de "Path Traversal", nunca llegaba a detectar realmente invocaciones de estas llamadas en el código. Después del análisis detallado del código se ha observado que el plug-in no detectaba las invocaciones de llamadas de tipo "source" o "sink" en el código, si se trataba de un constructor en vez de un método normal. Dado que las llamadas vulnerables al "Path Traversal" son en la mayoría los constructores, sus invocaciones en el código no se detectaban. Se ha corregido este error, incluyendo como una posible invocación de "sink" o "source", a objetos de clase *ClassInstanceCreation.*

4.2 Reconocimiento de Diversos Tipos de Expresión "sink"

Otra debilidad de LAPSE encontrada fue el hecho de aceptar como objeto "sink" en una invocación del método "sink" solamente a nombres simples. Sin embargo es muy frecuente que el parámetro vulnerable del método "sink" sea una invocación del método, una concatenación, etc. Si se intentaba propagar desde el parámetro vulnerable de algún método "sink" que no fuese un nombre simple se obtenía el error de que no se identificaba bien al parámetro. (En la Ilustración 15 se puede observar el mensaje de error). A continuación se describen los tipos de expresiones incluidas en LAPSE+ como posibles objetos "sink".

Uno de los parámetros ignorados fueron los accesos a un *array*. Por ejemplo ante la llamada *stmt.execute(valores[3])*, la propagación hacia atrás no se realizaba bien, dado que LAPSE no identificaba un acceso a un *array* como una expresión válida. Estos es una debilidad significativa dado que los objetos "source" (los devueltos de las invoaciones de llamadas "source") pueden ser *arrays* como por ejemplo la llamada *req.getParameterValues(_)* que devuelve un *array* de cadenas con contenido potencialmente malicioso. Se ha corregido este error dándole un trato especial a las expresiones de tipo *ArrayAccess.*

También se ha incluido el reconocimiento como objetos "sink" las expresiones en paréntesis, las expresiones de casting y el literal null. Anteriormente al encontrarse con alguno de estos dos tipos de expresiones, LAPSE mostraba el error del objeto no identificable. Se ha modificado de forma que en caso de que una expresión aparezca en paréntesis se evalúe la expresión que viene dentro de paréntesis, como puede ser un retorno de un método. En cuanto al literal "null" se ha modificado el código de plug-in

para que le indique al usuario el hallazgo de una asignación de tipo null, ya que esto suele significar que existe una inicialización verdadera en algún punto ulterior del código.

Por otro lado se ha encontrado que LAPSE no propagaba bien desde un método "sink" que tuviera como parámetro una concatenación de varias cadenas. Por ejemplo, para la expresión: *new File(_blojsom2Path + "/resources/" + blojsom2ID + "/"),* si se intentaba propagar hacia atrás, se obtenía el error mostrado en la Ilustración 15.

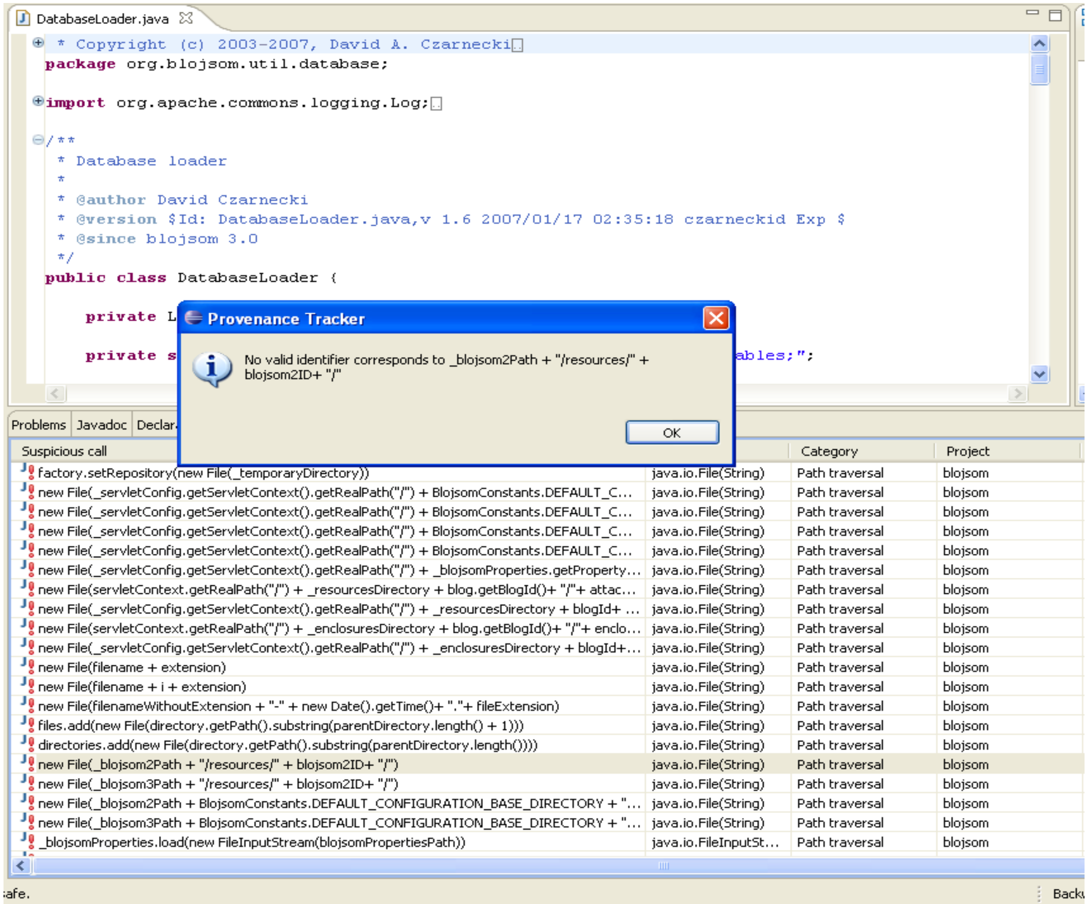


Ilustración 15: Error de Validación de Expresión "sink"

Dado que este tipo de expresión se repite en muchos ejemplos probados con LAPSE, se ha decidido mejorar esta característica. Se ha modificado el plug-in para que tuviera en cuenta los parámetros de tipo compuesto y para que analizara cada parte de la concatenación por separado. El efecto de modificación para el mismo ejemplo, se puede observar en la Ilustración 16.

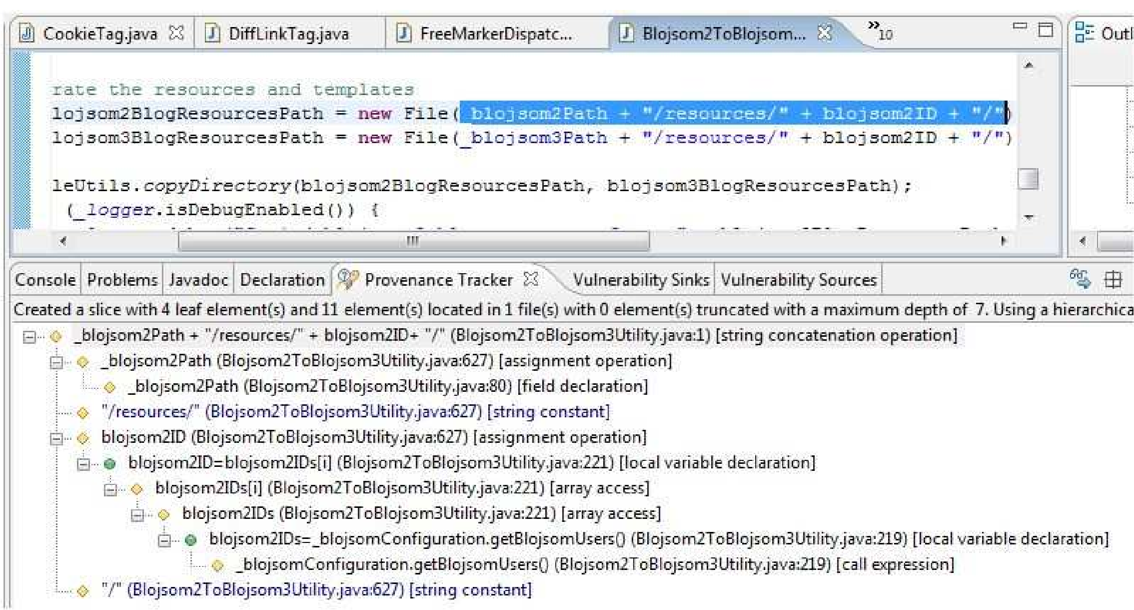


Ilustración 16: Resultado de Mejora del Análisis de Expresiones de Concatenación

En la Ilustración 17 se observa la propagación desde un objeto que es una concatenación de una variable normal con un acceso a un *array*. Se observa que LAPSE+ es capaz de detectar tanto la concatenación, como el acceso a un *array,* por lo que la propagación se realiza correctamente.

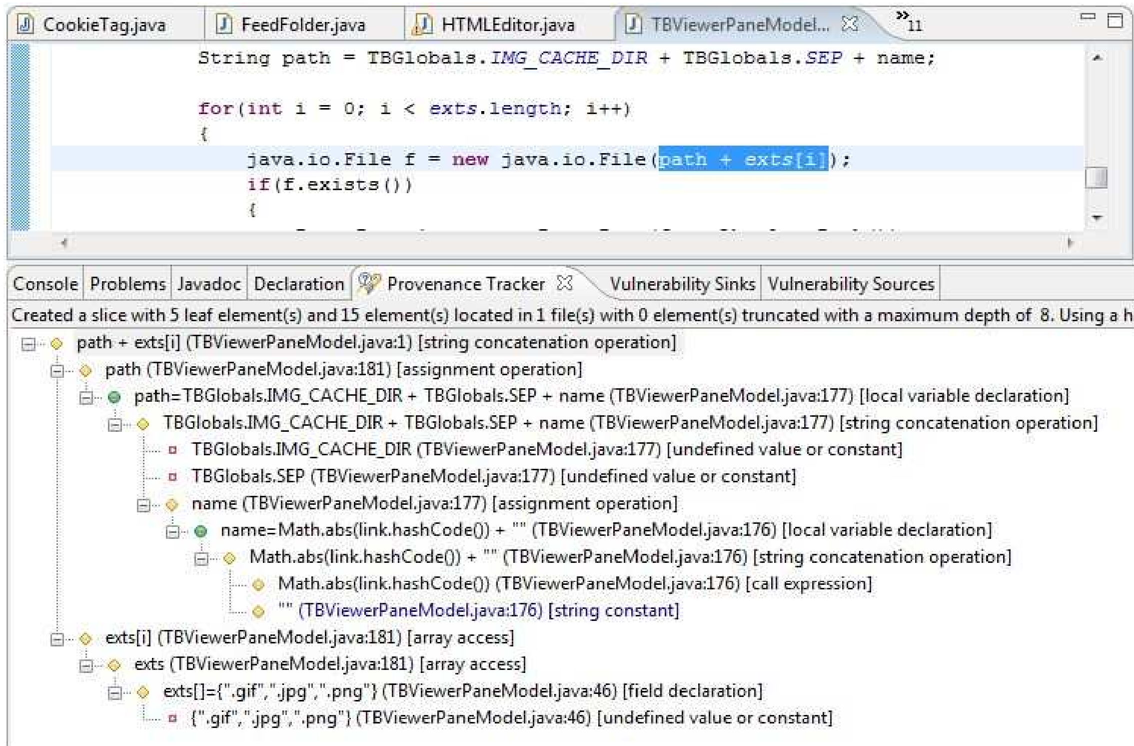


Ilustración 17: Resultado de Mejorar el Análisis de Expresiones de Concatenación (2)

Finalmente se ha dado soporte a las invocaciones de método (*MethodInvocation*) o constructores (*ClassInstanceCreation*), como posibles objetos “sink”. Por ejemplo en la llamada “sink” *new File(f.getParent())* el parámetro vulnerable es una invocación del método; sin embargo LAPSE+ realiza bien la propagación, como se puede observar en la Ilustración 18.

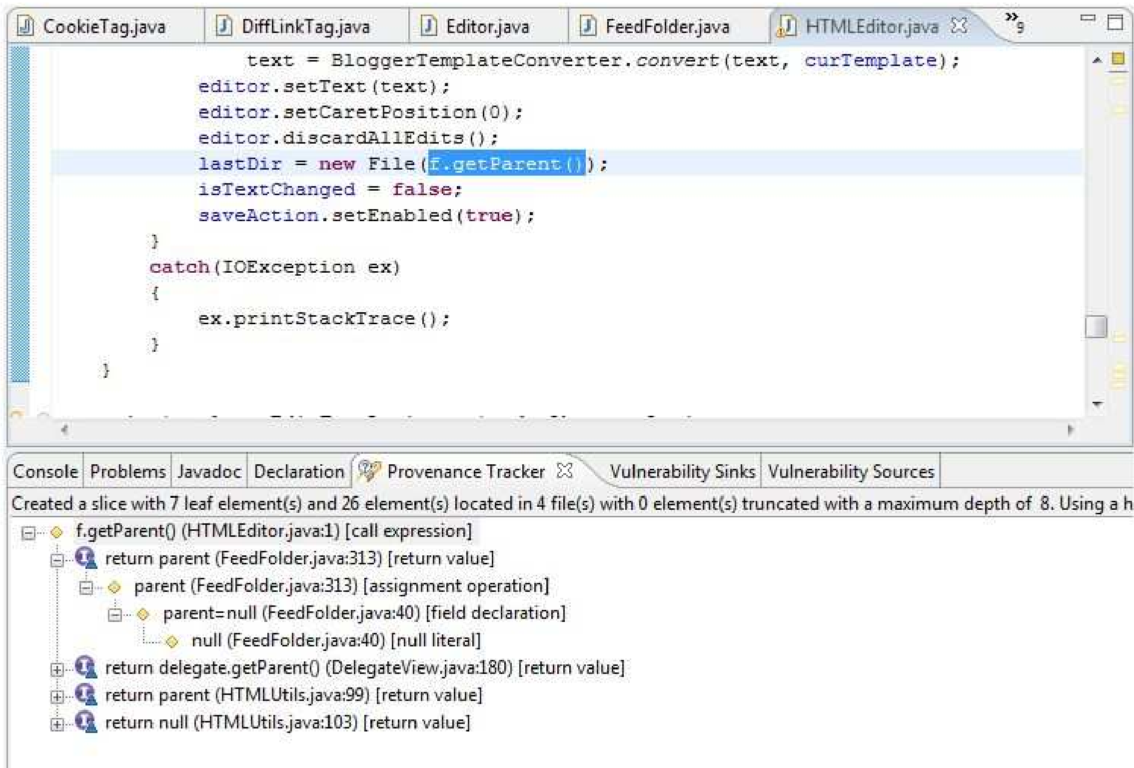


Ilustración 18: Propagación desde una llamada "sink", con invocación de un método como parámetro

4.3 Inclusión de los Operadores de Derivación.

Con el fin de acoplar mejor el plug-in con el modelo de los objetos "tainted" se ha incluido el procesamiento de la parte faltante del modelo "tainted object propagation", la parte de derivación. La propagación desde un objeto "sink" hacia uno de "source" en la versión original de LAPSE se realizaba de forma interna a través de asignaciones, invocaciones de métodos, etc. Sin embargo no se tenían en cuenta unos operadores de derivación específicos como métodos toString(), append(), etc. Dado que son unos métodos pertenecientes al API de Java, al no encontrarse su código fuente en el proyecto analizado se terminaba de propagar ante invocaciones de estos métodos. Se ha modificado esta característica de forma que si se encuentra invocación de un método no definido en el proyecto se comprueba si este método está definido como un método de derivación. En caso afirmativo se procesa la expresión de derivación, que bien puede ser un parámetro del método o el objeto que lo invoca.

En el Listado 16 se observa el esquema para fichero "derived.xml". Se observa que tiene un Identificador ID que está formado por el nombre completo del método de derivación. Otros datos son tipo "type" que especifica el tipo al que pertenece el método así como método "method" especifica el nombre simple del mismo.

```
<!ELEMENT DERIVATIONS (derivation*)>
<!ELEMENT DERIVATION (type, method)>
<!ELEMENT TYPE #CDATA>
<!ELEMENT METHOD #CDATA>

<!ATTLIST DERIVATION  ID CDATA #REQUIRED>
```

Listado 16: Fichero "derived.dtd"

En la Ilustración 19 se observa cómo se detecta y trata una derivación encontrada en el código. El objeto "sink" es de tipo invocación de método *templatePath.toString()*.Al detectar que se trata de una invocación del método, LAPSE+ recorre el código fuente en busca de la declaración del mismo. Sin embargo si no encuentra la declaración comprueba si se trata de un método de derivación especial, definido en el fichero "derived.xml". Dado que el método *toString()* es un método de derivación, se procede a analizar la variable de la cual se deriva el objeto "sink" a través de este método, que en este caso es la variable *templatePath*.

.

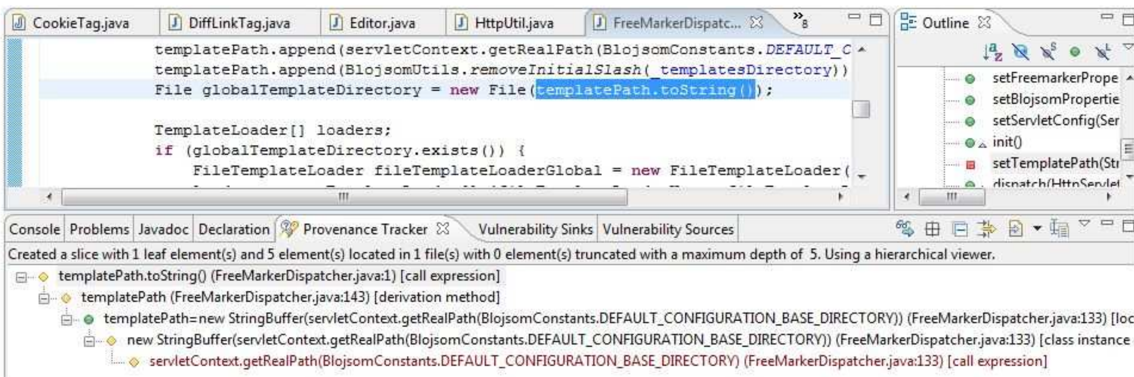


Ilustración 19: Resultado de detección del operador de derivación

También se ha reimplementado el método que comprueba si una invocación es la invocación de algún método de tipo “source”. Esta comprobación no se realizaba correctamente en la anterior versión de LAPSE y por tanto cuando la propagación mostraba el resultado final no diferenciaba bien si se llegaba a “source” o no. LAPSE+ muestra una llamada de tipo “source” en el color rojo, como se puede observar en la Ilustración 19, y una llamada de tipo “safe” en el color verde.

5 DETECCIÓN DE LAS VULNERABILIDADES EN LAPSE+

En esta sección se describen las vulnerabilidades que LAPSE es capaz detectar. Todas ellas, tienen un punto en común, y es que son causadas por la entrada de usuario utilizada por la aplicación sin validarla previamente. Esta característica permite que todas ellas se puedan describir mediante el modelo “tainted object propagation”. (véase sección 2.3.2). A continuación se describe cada vulnerabilidad en particular, su especificación en PQL, como son detectadas en el código por LAPSE, etc.

Dado que todas las vulnerabilidades siguen el modelo “tainted object propagation” el primer paso es la definición de objetos “source“, objetos “sinks” y la manera en la que se pueden derivar los últimos de los primeros. Para esta definición, se va a utilizar el lenguaje PQL cuya sintaxis es muy parecida a Java y por tanto facilita entender cómo se producen las vulnerabilidades en el propio código.

Todas las vulnerabilidades tienen una parte de especificación PQL en común. La consulta principal *main()* (ver Listado 17) y la consulta de derivación *derived*()* (ver Listado 18) son comunes para todos los problemas pertenecientes al modelo “tainted object propagation”.

```
query main()
returns
object Object sourceObj, sinkObj;
matches    {    sourceObj    :=    sourceParam()|    sourceHeader()|
sourceCookie()|sourceEscape();
sinkObj := derived*(sourceObj);
sinkObj := sink();}
```

Listado 17: Consulta "main" de las vulnerabilidades detectadas por LAPSE

La consulta *main()* busca todos los objetos de tipo “sink” tales que puedan ser derivados recursivamente de los objetos “source” mediante la consulta *derived()*. La consulta *derived()* a su vez define todos los posibles métodos de derivación. Los métodos de derivación son los métodos de API de String de Java, que permiten realizar diferentes cambios a las cadenas de caracteres, como concatenación, obtención de subcadenas, etc. Además de estos operadores existe la derivación general mediante la cual un objeto de tipo “source” se puede convertir en uno de tipo “sink”, a través de todo tipo de asignaciones simples, invocaciones de métodos, paso de parámetros, etc. Este tipo de derivación no se especifica en PQL, ya que es igual para todo tipo de vulnerabilidades y se realiza mediante el parseo interno de código.

```
query derived*(object Object x)
returns
object Object y;
uses
object Object temp;
matches { y := x |
temp := derived(x); y := derived*(temp);}

query derived(object Object x)
returns
object Object y;
matches { y.append(x)
| y = _.append(x)
| y = new String(x)
| y = new StringBuffer(x)
| y = x.toString()
| y = x.substring(_ ,_);}
```

Listado 18: Consulta de derivación desde el objeto "source" a "sink"

Por otra parte los objetos “source” pueden ser compartidos por varias vulnerabilidades. Esto se debe a la distinción de ataques de inyección de datos maliciosos, de ataques que utilizan estos datos. Como se ha explicado en la sección 2.3.1.1 los datos maliciosos pueden ser introducidos a través de varios ataques. Además estos datos, una vez introducidos, pueden ser utilizados a su vez para realizar diferentes ataques (ver sección 2.3.1.2). Por lo tanto en lugar de especificar una consulta *source()* para cada vulnerabilidad por separado, se definen varias consultas *source()* que se referenciarán desde la especificación completa de cada vulnerabilidad.

5.1 Fuentes de Datos Maliciosos

La manera más común en la que los datos maliciosos pueden ser introducidos en una aplicación Web es mediante la “alteración de parámetros”. El usuario proporciona valores a los parámetros rellenando un formulario de una página Web. Estos valores posteriormente se incluyen en la petición Web o en la URL a la que se redirige la petición, desde donde la aplicación puede recuperarlos. Por tanto todas las llamadas cuyo objetivo es recuperar los parámetros se consideran vulnerables y se especifican en la consulta *sourceParam()* en el Listado 19.

```
query sourceParam()
returns
object Object sourceObj;
uses
object String[] sourceArray;
object Map sourceMap;
object HttpServletRequest req;
matches { sourceObj = req.getParameter(_)
|sourceObj = req.getHeader(_)
|sourceArray = req.getParameterValues(_);
|sourceObj = req.getParameterNames();
|sourceMap=getParameterMap()
|sourceObj= getRequestURL()
|sourceObj= req.getQueryString()
;}
```

Listado 19: Consulta PQL para buscar "Manipulación de Parámetros"

Otra fuente de datos modificable por el usuario son las cabeceras HTTP. Es más difícil modificar las cabeceras que los parámetros ya que no se pueden acceder directamente por el navegador, pero aún así se pueden alterar mediante unos programas específicos que permiten el acceso a ellas. El Listado 20 muestra el PQL del ataque de "Manipulación de Cabeceras" que contiene todas las posibles llamadas que obtienen la información de una cabecera HTML de la petición web concreta.

```
query sourceHeader()
returns
object Object sourceObj;
object Object[] sourceArray;
uses
object ServletRequest req;
matches { sourceObj = req.getScheme()
|sourceObj = req.getProtocol()
|sourceObj = req.getContentType()
|sourceObj = req.getServerName()
|sourceObj = req.getRemoteAddr()
|sourceObj = req.getRemoteHost()
|sourceObj = req.getRealPath()
|sourceObj = req.getLocalName()
|sourceObj = req.getLocalAddr()
|sourceObj = req.getHeader()
|sourceArray= req.getHeaders()
|sourceObj = req.getAuthtype()
|sourceObj = req.getRequestURI();}
```

Listado 20: Consulta PQL para buscar "Manipulaciones de Cabeceras"

En el Listado 21 se puede apreciar la consulta PQL que recupera todas las llamadas a métodos de acceso a cookies. Una cookie contiene parámetros que pueden ser manipulados por el usuario, de la misma forma que la cabecera HTTP, ya que se incluyen en la petición HTTP. Por tanto la falta de validación de los valores obtenidos de la cookie de la petición Web puede ser una fuente de datos maliciosos.

```
query sourceCookies()
returns
object Object sourceObj;
object Object[] sourceArray;
uses
object javax.servlet.http.Cookie cookie;
matches { sourceObj = cookie.getName()
|sourceObj = cookie.getPath()
|sourceObj = cookie.getDomain()
|sourceObj = cookie.getComment()
|sourceObj = cookie.getValue();}
```

Listado 21: Consulta PQL para buscar "Manipulaciones de Cookies"

Otro tipo de vulnerabilidad que LAPSE es capaz de detectar es el escape de información ("Information Leakege"). Aunque no se trata de un tipo de inyección de datos malignos a la aplicación (ver sección 2.3.1.1.), se considera fuente de un posible ataque. Si al usuario se le revela la información sensible acerca de la aplicación, este puede aprovecharla para realizar ataques. Por ejemplo, si se revelan los nombres de las tablas de la base de datos de la aplicación, y además la aplicación es vulnerable a la Inyección SQL, el atacante puede realizar cambios sin autorización en la base de datos e incluso borrar tablas enteras de la misma. En el Listado 22 se observa el PQL de la vulnerabilidad de Escape de Información, que contiene los métodos sensibles.

```
query sourceEscape()
returns
object Object sourceObj;
uses
object java.sql.ResultSet res;
matches { sourceObj = res.getString(int)
|sourceObj = res.getString(String)
|sourceObj = res.getObject(int)
|sourceObj = res.getObject(String);}
```

Listado 22: Consulta PQL para buscar "Escape de Información"

5.2 Cross Site Scripting (XSS)

Esta vulnerabilidad descrita en sección 5.2 se puede producir cada vez que la aplicación imprime algo recibido con anterioridad de usuario sin validar. Por lo tanto las llamadas de tipo “source” son todas las llamadas que permiten que el usuario proporcione alguna información. Esta información puede venir por la manipulación de parámetros, cabeceras o cookies. Por tanto un objeto “source” para XSS será cualquier ocurrencia de las consultas *sourceParma*(), *sourceHeader*() y *sourceCookie();* también se considera como fuente la consulta *sourceEscape()*, dado que el XSS se basa en la impresión de cosas en la interfaz de usuario, por lo que podría imprimir la información sensible recogida por la consulta *sourceEscape()*.

Por otra parte las llamadas de tipo “sink” son todas las llamadas que realizan el proceso de impresión de los datos. Esta impresión puede ser realizada en la respuesta de un servlet, en una página JSP en una página HTML, etc. En el resultado se obtiene la especificación PQL de la consulta “sink”, mostrado en el Listado 23.

```
query sink()
returns
object Object sinkObj;
uses
object javax.servlet.ServletOutputStream strm;
object javax.servlet.jsp.JspWriter jsp;
object java.io.PrintWriter print;
matches { strn.println(sinkObj)
| strn.printl(sinkObj)
| strn.println(sinkObj)
| jsp.println(sinkObj)
| jsp.print(sinkObj)
| print.println(sinkObj)
| print.print(sinkObj);}
```

Listado 23: Consulta PQL "sink" para XSS

Una vez definidas las llamadas de tipo “source” y las de tipo “sink” se proceden a buscar las vulnerabilidades de tipo XSS en el código Java. Los proyectos analizados son un conjunto de aplicaciones J2EE para votación online, que en su día realizamos como práctica de la asignatura de Programación J2EE.

La Ilustración 20 muestra como se propaga hacia atrás desde una llamada “sink” de tipo XSS, mediante asignaciones, invocaciones de métodos, etc., hasta llegar finalmente a la llamada de tipo “source”, la cual confirma que la vulnerabilidad existe. La llamada de tipo “source” se muestra en color rojo, para diferenciar con claridad que se ha llegado al origen de la vulnerabilidad.

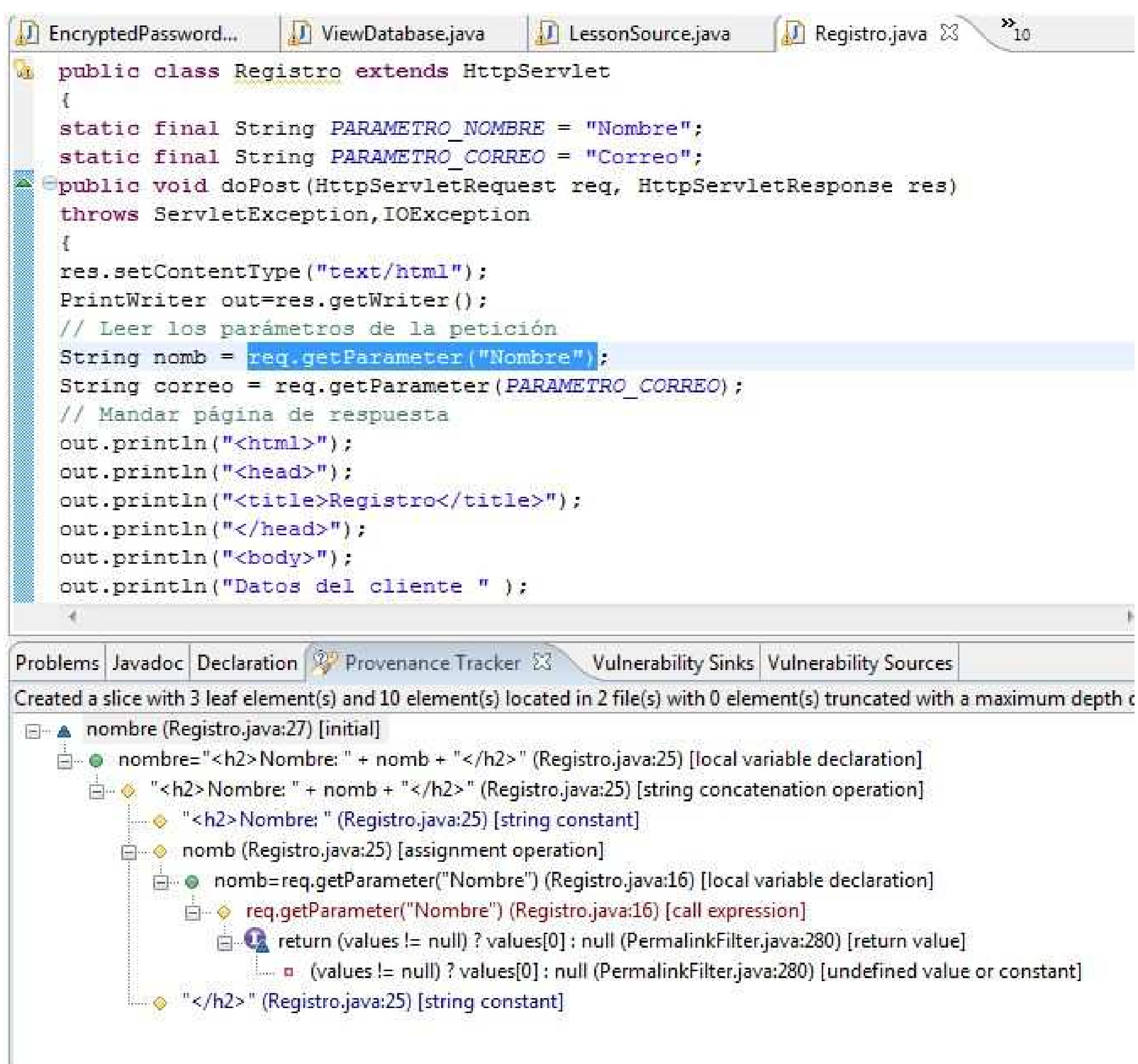
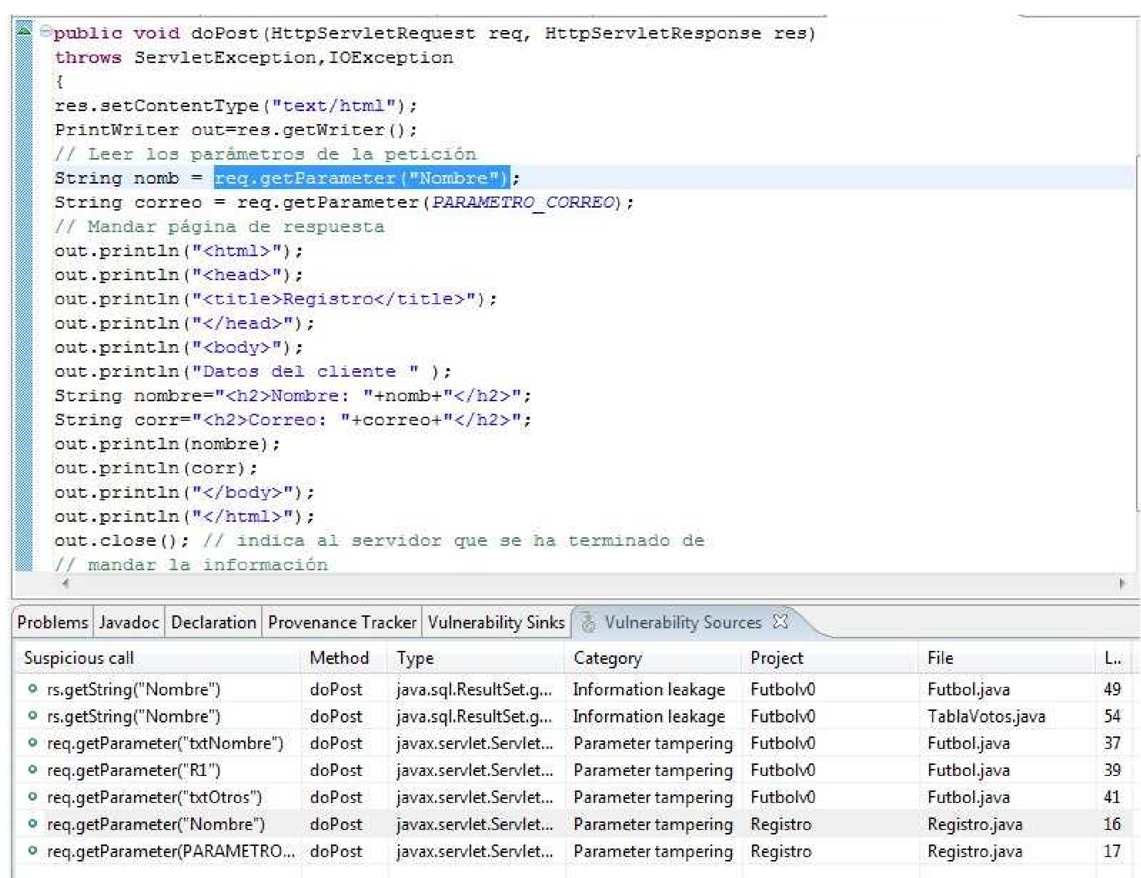


Ilustración 20: : Resultado de propagación entre el objeto "sink" y objeto "source" para la vulnerabilidad de "Cross Site Scripting"

La Ilustración 21 muestra las llamadas de tipo "source" detectadas en la aplicación escaneada y se puede apreciar que realmente se ha detectado como "source" la misma llamada a la que se llega propagando desde "sink". Se observa también que el tipo de fuente de datos es un ataque de manipulación de parámetros. La

Ilustración 22 muestra las llamadas de tipo "sink", la seleccionada es desde la que se ha propagado hacía atrás para llegar a "source". Tanto en la vista "Source View", como en la de "Sink View" si se hace "doble click" sobre un resultado seleccionado se abre en el editor código fuente donde esta seleccionada la llamada en cuestión.

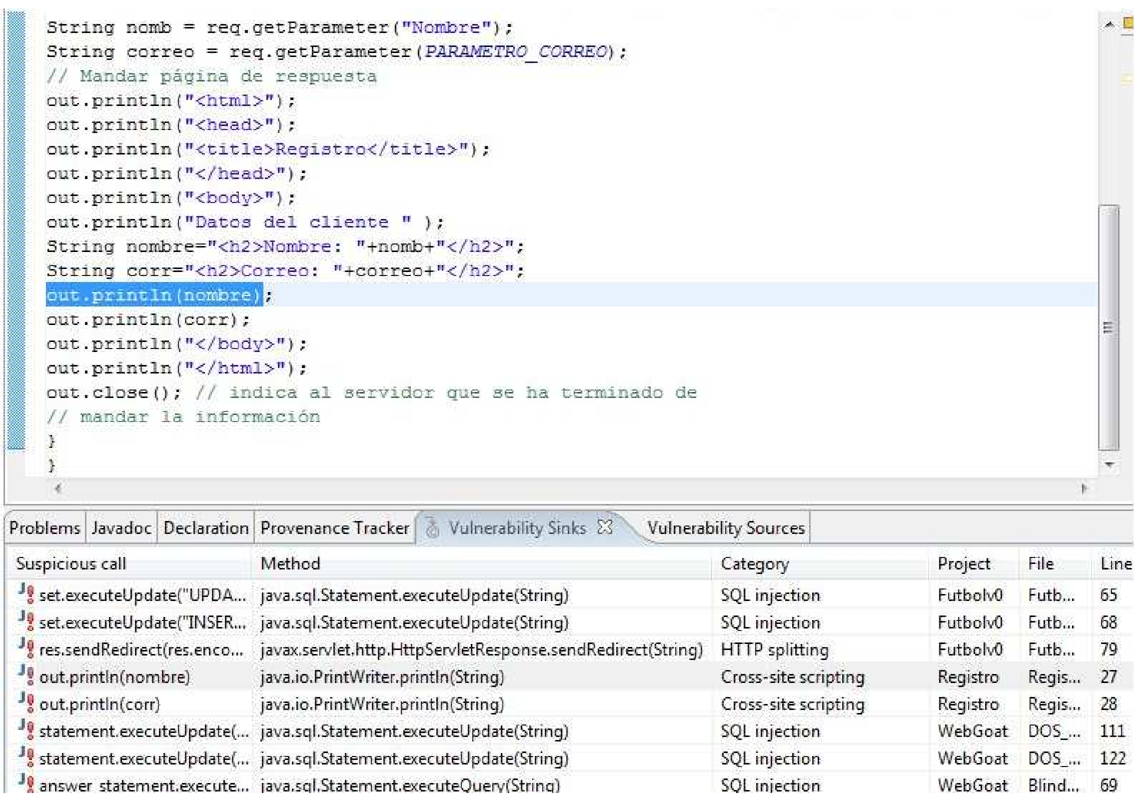


```
public void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter out=res.getWriter();
// Leer los parámetros de la petición
String nomb = req.getParameter("Nombre");
String correo = req.getParameter(PARAMETRO_CORREO);
// Mandar página de respuesta
out.println("<html>");
out.println("<head>");
out.println("<title>Registro</title>");
out.println("</head>");
out.println("<body>");
out.println("Datos del cliente " );
String nombre="<h2>Nombre: "+nomb+"</h2>";
String corr="<h2>Correo: "+correo+"</h2>";
out.println(nombre);
out.println(corr);
out.println("</body>");
out.println("</html>");
out.close(); // indica al servidor que se ha terminado de
// mandar la información
```

Problems | Javadoc | Declaration | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Suspicious call	Method	Type	Category	Project	File	L..
rs.getString("Nombre")	doPost	java.sql.ResultSet.g...	Information leakage	Futbolv0	Futbol.java	49
rs.getString("Nombre")	doPost	java.sql.ResultSet.g...	Information leakage	Futbolv0	TablaVotos.java	54
req.getParameter("txtNombre")	doPost	javax.servlet.Servlet...	Parameter tampering	Futbolv0	Futbol.java	37
req.getParameter("R1")	doPost	javax.servlet.Servlet...	Parameter tampering	Futbolv0	Futbol.java	39
req.getParameter("txtOtros")	doPost	javax.servlet.Servlet...	Parameter tampering	Futbolv0	Futbol.java	41
req.getParameter("Nombre")	doPost	javax.servlet.Servlet...	Parameter tampering	Registro	Registro.java	16
req.getParameter(PARAMETRO...	doPost	javax.servlet.Servlet...	Parameter tampering	Registro	Registro.java	17

Ilustración 21: Resultado de búsqueda de objetos "sinks" para XSS



```
String nomb = req.getParameter("Nombre");
String correo = req.getParameter(PARAMETRO_CORREO);
// Mandar página de respuesta
out.println("<html>");
out.println("<head>");
out.println("<title>Registro</title>");
out.println("</head>");
out.println("<body>");
out.println("Datos del cliente " );
String nombre="<h2>Nombre: "+nomb+"</h2>";
String corr="<h2>Correo: "+correo+"</h2>";
out.println(nombre);
out.println(corr);
out.println("</body>");
out.println("</html>");
out.close(); // indica al servidor que se ha terminado de
// mandar la información
}
}
```

Problems | Javadoc | Declaration | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Suspicious call	Method	Category	Project	File	Line
set.executeUpdate("UPDA...	java.sql.Statement.executeUpdate(String)	SQL injection	Futbolv0	Futb...	65
set.executeUpdate("INSER...	java.sql.Statement.executeUpdate(String)	SQL injection	Futbolv0	Futb...	68
res.sendRedirect(res.enco...	javax.servlet.http.HttpServletResponse.sendRedirect(String)	HTTP splitting	Futbolv0	Futb...	79
out.println(nombre)	java.io.PrintWriter.println(String)	Cross-site scripting	Registro	Regis...	27
out.println(corr)	java.io.PrintWriter.println(String)	Cross-site scripting	Registro	Regis...	28
statement.executeUpdate(...	java.sql.Statement.executeUpdate(String)	SQL injection	WebGoat	DOS_...	111
statement.executeUpdate(...	java.sql.Statement.executeUpdate(String)	SQL injection	WebGoat	DOS_...	122
answer_statement.execute...	java.sql.Statement.executeQuery(String)	SQL injection	WebGoat	Blind...	69

Ilustración 22: Resultado de búsqueda de objetos "source" para XSS

5.3 Inyección SQL

La especificación de la Inyección SQL es la definida con la consulta "sink" en el Listado 24. Como en el caso de XSS, los objetos "sink" utilizados pueden provenir de cualquier consulta de tipo "source" de las especificadas antes. Las llamadas "sink", en cambio, son todas las que manipulan una base de datos.

```
query sink()
returns
object Object sinkObj;
uses
object java.sql.Statement stmt;
object java.sql.Connection con;
matches { stmt.executeQuery(sinkObj)
| stmt.execute(sinkObj)
| stmt.execute(sinkObj,_)
| con.prepareStatement(sinkObj)
| con.prepareStatement(sinkObj,_)
| con.prepareStatement(sinkObj,_,_)
| stmt.executeUpdate(sinkObj)
| stmt.executeUpdate(sinkObj,_)
| stmt.executeQuery(sinkObj)
| stmt.addBatch
| con.prepareCall(sinkObj)
| con.prepareCall(sinkObj,_)
| con.prepareCall(sinkObj,_,_);}
```

Listado 24: Consulta PQL "sink" para "Inyección SQL"

En las siguientes ilustraciones se muestra como se detecta en LAPSE la vulnerabilidad de tipo Inyección SQL. La aplicación analizada es WebGoat, una aplicación diseñada por OWASP [36], con las vulnerabilidades de seguridad implantadas a propósito, con fines educativos.

La Ilustración 23 muestra cómo se propaga hacia atrás desde la llamada potencialmente vulnerable, mediante asignaciones, invocaciones de métodos, etc., hasta llegar finalmente a la llamada de tipo "source" peligrosa (marcada en rojo en la Ilustración 23) lo cual confirma que la vulnerabilidad existe.

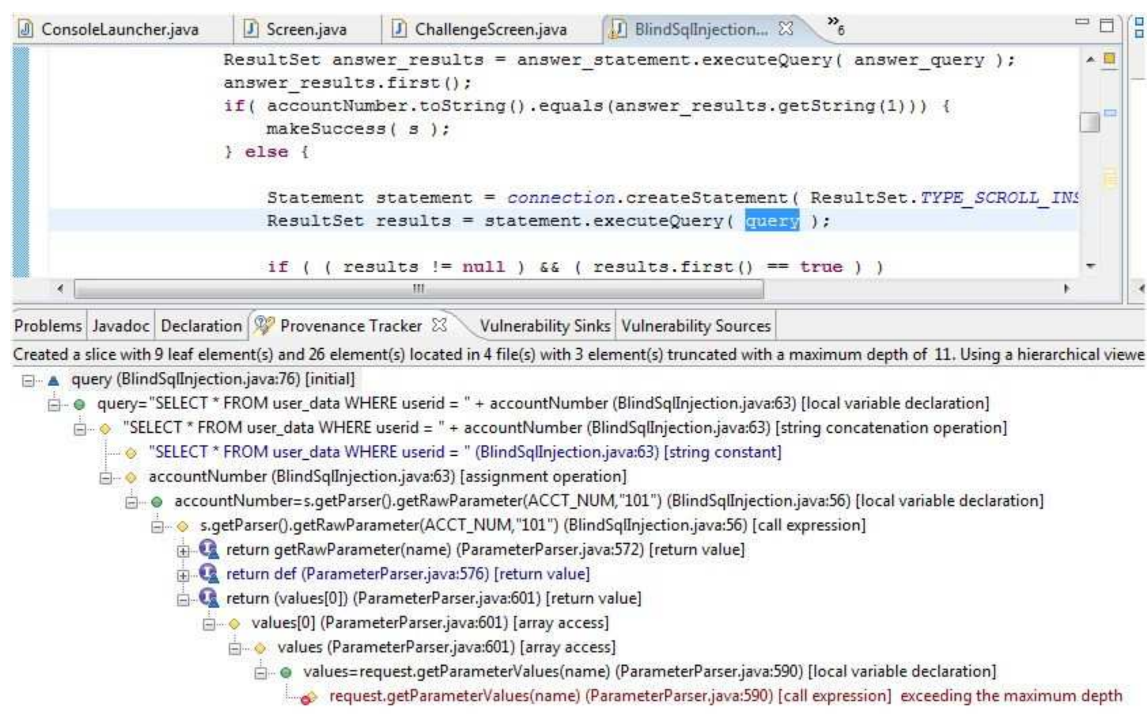


Ilustración 23: Resultado de propagación entre el objeto "sink" y objeto "source" para la vulnerabilidad de "Inyección SQL"

La Ilustración 24 muestra las llamadas de tipo "source" detectadas para la aplicación escaneada y se puede apreciar que realmente se ha detectado como "source" la misma llamada a la que se llega propagando desde "sink". Como en el caso anterior, la fuente de datos es la manipulación de parámetros. La Ilustración 25 muestra las llamadas de tipo "sink"; la seleccionada es desde la que se ha propagado hacia atrás para llegar a "source".

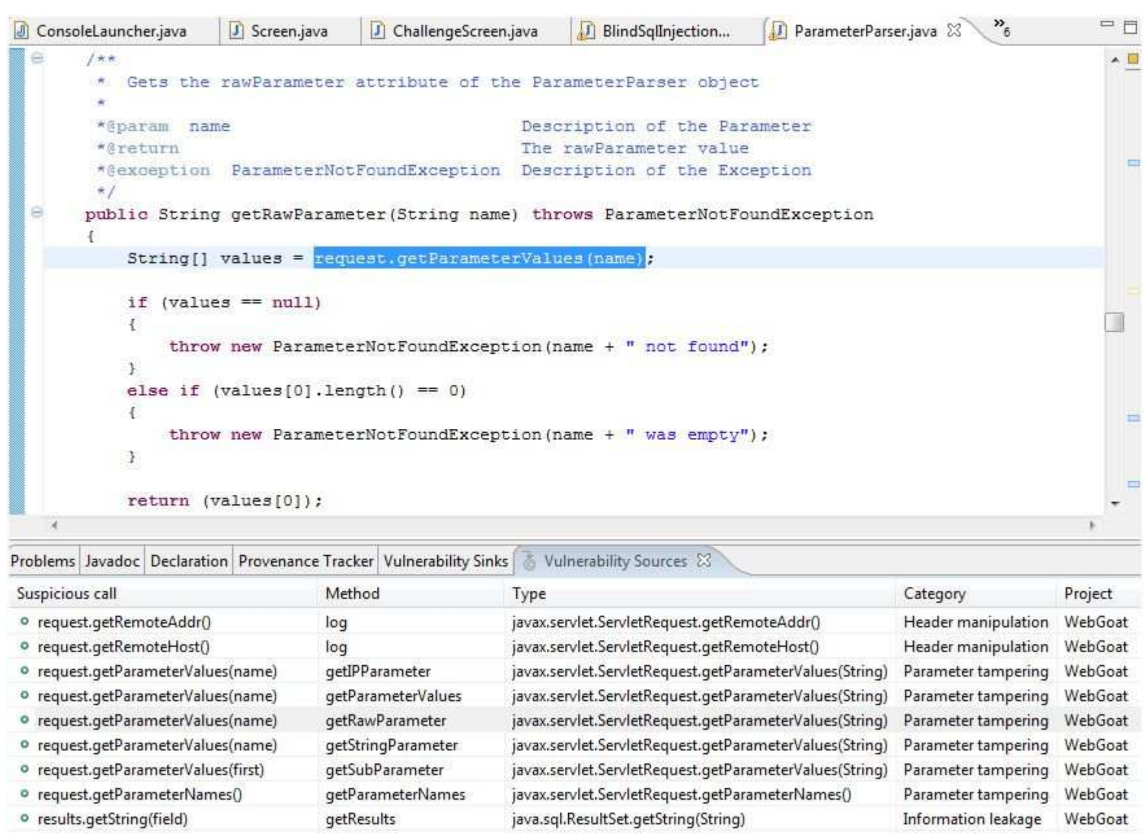


Ilustración 24: Resultado de búsqueda de objetos "source" para “Inyección SQL”

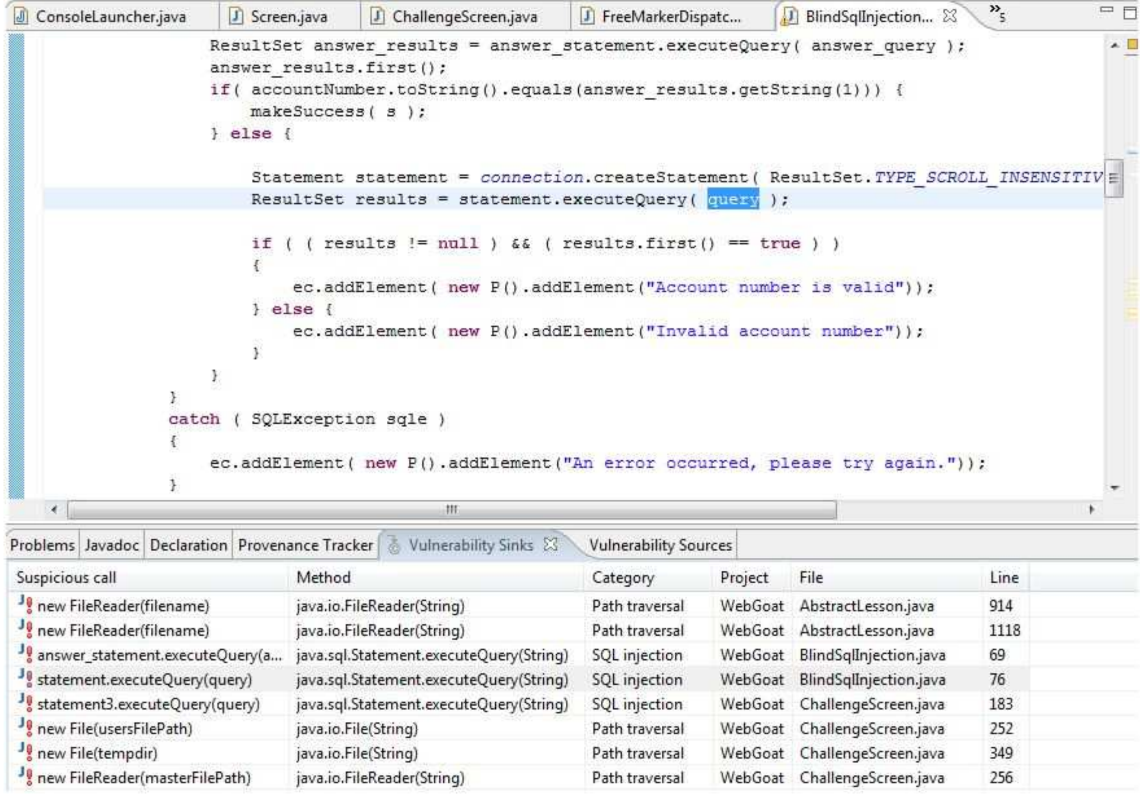


Ilustración 25: Resultado de búsqueda de objetos "sink" para “Inyección SQL”

5.4 Ejecución Maliciosa de Ficheros ("Path Traversal")

Los ataques de "ejecución maliciosa de ficheros" o de "Path Traversal", descritas en la sección 2.1.2.5, pueden ocurrir cada vez que la entrada procedente del usuario sin validar se utiliza en la creación, acceso de lectura o escritura a los ficheros y directorios. Por tanto todos los métodos que manipulen acceso a los ficheros se consideran vulnerables y los objetos que manipulan se convierten en objetos de tipo "sink". En el Listado 25 se describe la consulta PQL completa que busca los objetos "sink" vulnerables al "Path Traversal".

```
query sink()
returns
object Object sinkObj;
uses
object java.io io;
matches { io.File(sinkObj)
|io.RandomAccessFile(sinkObj,_)
|io.FileInputStream(sinkObj)
|io.FileReader(sinkObj)
|io.FileWriter(sinkObj)
|io.FileOutputStream(sinkObj)
```

Listado 25: Consulta PQL "sink" para "Path Traversal"

Para demonstrar cómo detecta LAPSE este tipo de vulnerabilidades, se ha analizado un proyecto de "open source" encontrado en un repositorio importante "sourceforge" [29] con una alta relevancia y frecuentemente utilizado. La aplicación se llama "Blojsom" y es una aplicación multi-blog, multi-usuarios.

En la Ilustración 26 se observa cómo se propaga de una llamada de tipo "sink" hasta encontrar una llamada de tipo "source" (marcada en rojo). Se puede apreciar que el objeto "sink" inicial es una concatenación de dos variables. Gracias a la mejora introducida en el plug-in (ver sección 3.6), la propagación se realiza correctamente y una de las variables resulta ser infectada con la manipulación de parámetros.

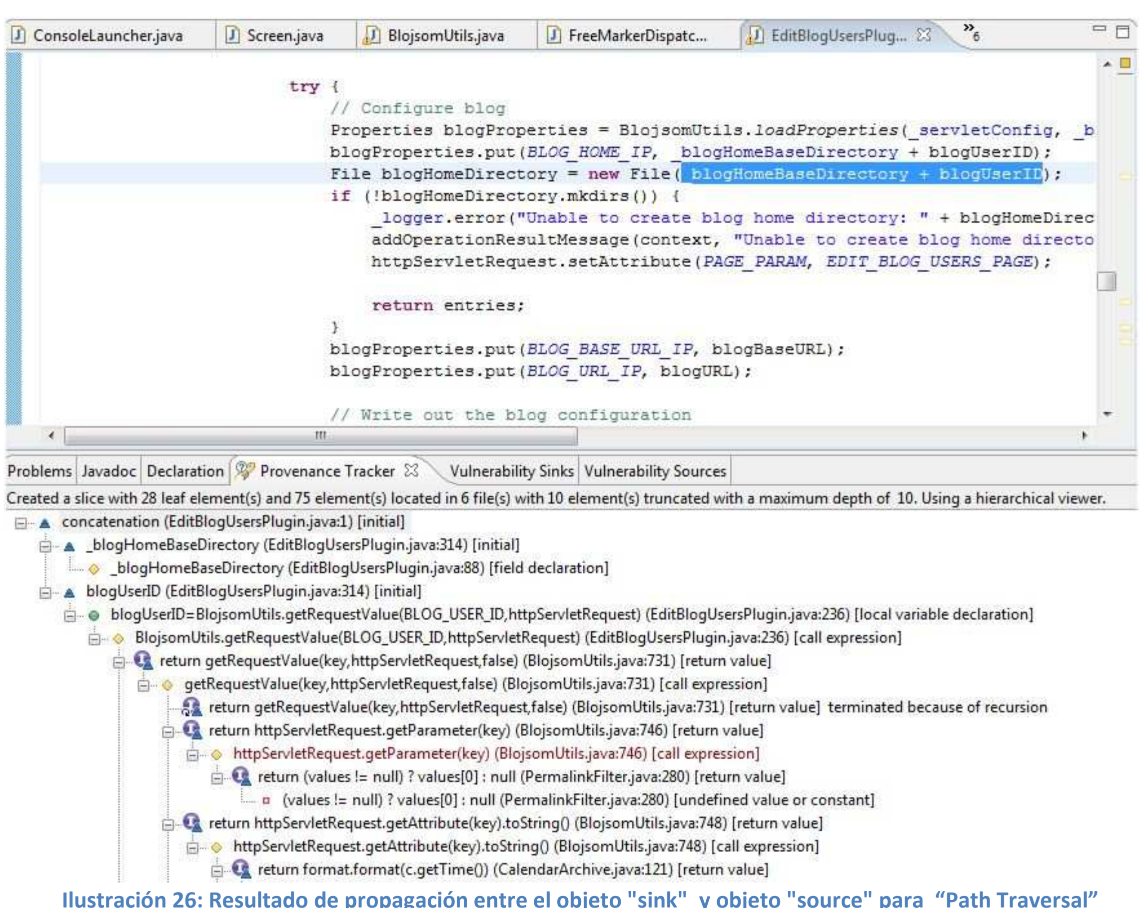
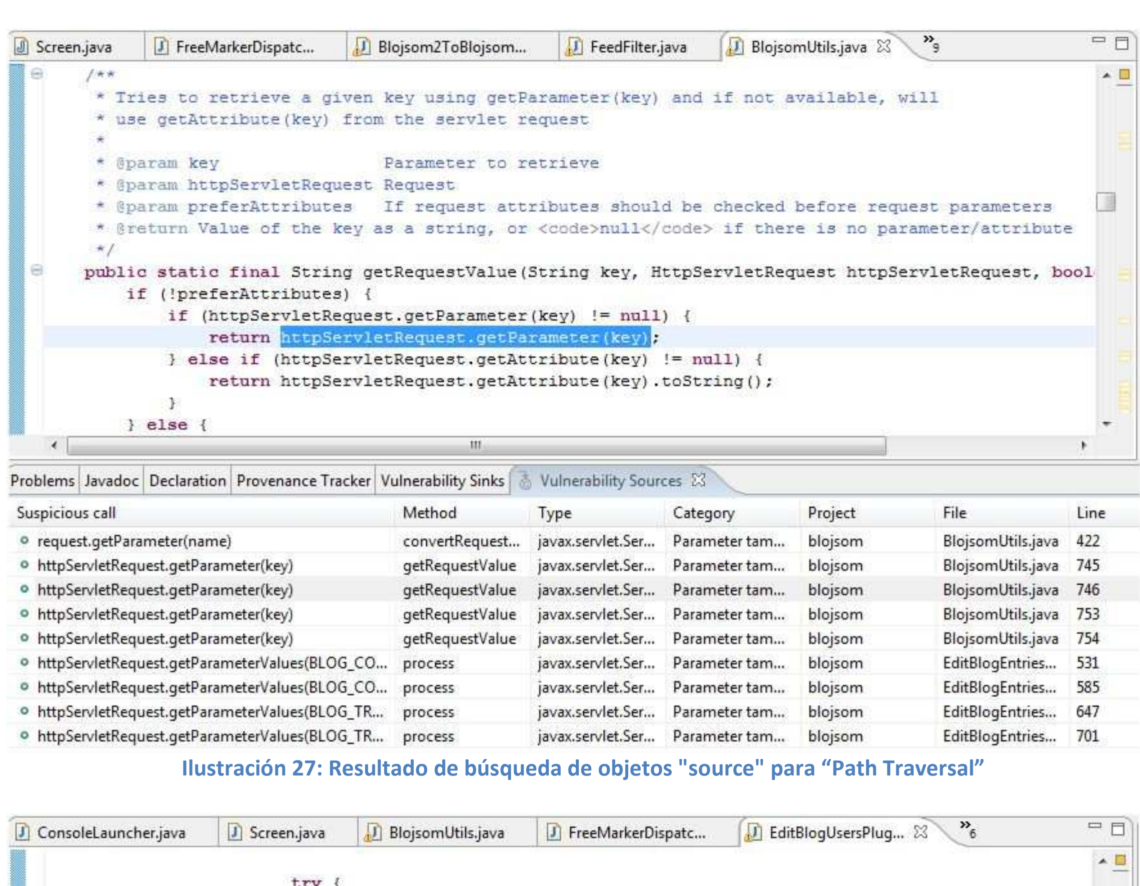


Ilustración 26: Resultado de propagación entre el objeto "sink" y objeto "source" para "Path Traversal"

Las Ilustraciones 27 y 28 muestran los resultados de búsqueda de objetos de tipo "source" y "sink" respectivamente. Se observa que se corresponden con los utilizados en la propagación de la Ilustración 26.



Suspicious call	Method	Type	Category	Project	File	Line
request.getParameter(name)	convertRequest...	javax.servlet.Ser...	Parameter tam...	blojsom	BlojsomUtils.java	422
httpServletRequest.getParameter(key)	getRequestValue	javax.servlet.Ser...	Parameter tam...	blojsom	BlojsomUtils.java	745
httpServletRequest.getParameter(key)	getRequestValue	javax.servlet.Ser...	Parameter tam...	blojsom	BlojsomUtils.java	746
httpServletRequest.getParameter(key)	getRequestValue	javax.servlet.Ser...	Parameter tam...	blojsom	BlojsomUtils.java	753
httpServletRequest.getParameter(key)	getRequestValue	javax.servlet.Ser...	Parameter tam...	blojsom	BlojsomUtils.java	754
httpServletRequest.getParameterValues(BLOG_CO...	process	javax.servlet.Ser...	Parameter tam...	blojsom	EditBlogEntries...	531
httpServletRequest.getParameterValues(BLOG_CO...	process	javax.servlet.Ser...	Parameter tam...	blojsom	EditBlogEntries...	585
httpServletRequest.getParameterValues(BLOG_TR...	process	javax.servlet.Ser...	Parameter tam...	blojsom	EditBlogEntries...	647
httpServletRequest.getParameterValues(BLOG_TR...	process	javax.servlet.Ser...	Parameter tam...	blojsom	EditBlogEntries...	701

Ilustración 27: Resultado de búsqueda de objetos "source" para "Path Traversal"

ConsoleLauncher.java | Screen.java | BlojsomUtils.java | FreeMarkerDispatc... | EditBlogUsersPlug...

```
try {
    // Configure blog
    Properties blogProperties = BlojsomUtils.loadProperties(_servletConfig, _b
    blogProperties.put(BLOG_HOME_IP, _blogHomeBaseDirectory + blogUserID);
    File blogHomeDirectory = new File(_blogHomeBaseDirectory + blogUserID);
    if (!blogHomeDirectory.mkdirs()) {
        _logger.error("Unable to create blog home directory: " + blogHomeDirec
        addOperationResultMessage(context, "Unable to create blog home directo
        httpServletRequest.setAttribute(PAGE_PARAM, EDIT_BLOG_USERS_PAGE);

        return entries;
    }
    blogProperties.put(BLOG_BASE_URL_IP, blogBaseURL);
    blogProperties.put(BLOG_URL_IP, blogURL);

    // Write out the blog configuration
```

Problems | Javadoc | Declaration | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Suspicious call	Method	Category	Project	File	Line
new File(userDirectory)	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	297
new File(_blogHomeBaseDirectory + blogUserID)	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	314
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	326
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	343
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	351
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getR...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	404
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	418
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	EditBlogUsersPlu...	446
new File(_blojsomConfiguration.getInstallationDirectory() + _resourcesDirectory + user.g...	java.io.Fil...	Path trav...	blojsom	FileUploadPlugin...	202
new File(_blojsomConfiguration.getInstallationDirectory() + _resourcesDirectory + user.g...	java.io.Fil...	Path trav...	blojsom	FileUploadPlugin...	258
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	ForgottenPasswo...	194
new File(_blojsomConfiguration.getInstallationDirectory() + _blojsomConfiguration.getB...	java.io.Fil...	Path trav...	blojsom	ThemeSwitcherP...	141

Ilustración 28: Resultado de búsqueda de objetos "sink" para "Path Traversal"

5.5 Inyección de Comandos

Esta vulnerabilidad descrita en la sección 2.1.2.4, ocurre cuando el usuario proporciona alguna cadena para ejecutar en un comando y no se valida esta cadena antes de utilizarla en la ejecución. Por tanto todos los métodos que ejecutan comandos del sistema son vulnerables. En el Listado 26 se describe la consulta PQL para esta vulnerabilidad.

```
query sink()
returns
object Object sinkObj;
object Object[] sinkArray;
uses
object java.lang.Runtime run;
matches { run.exec(sinkObj)
| run.exec(sinkArray)
| run.exec(sinkObj,_)
| run.exec(sinkArray,_)
| run.exec(sinkArray,_,_)
;}
```

Listado 26: Consulta PQL "sink" para “Inyección de Comandos”

El ejemplo de la Ilustración 29 muestra la propagación desde el objeto “sink” hasta el objeto “source” para la vulnerabilidad de “Inyección de Comandos”. El proyecto analizado es WebGoat, el mismo que para Inyección SQL. Para esta propagación ha sido necesario aumentar el número de pasos de propagación que viene por defecto, ya que no era suficiente para llegar al “source”. Las Ilustraciones 30 y 31 muestran los objetos de tipo “source” y de tipo “sink” utilizadas en la propagación, detectadas en sus respectivas vistas.

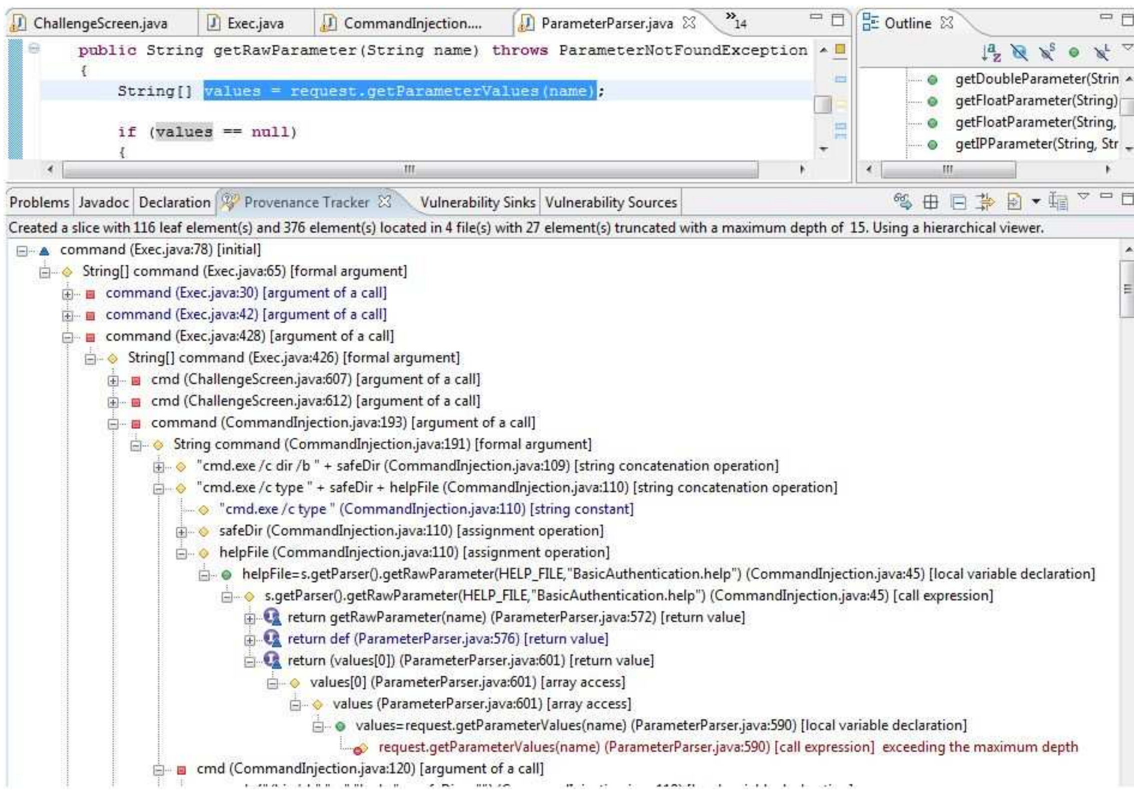


Ilustración 29: Resultado de propagación entre el objeto "sink" y objeto "source" para "Inyección de Comandos"

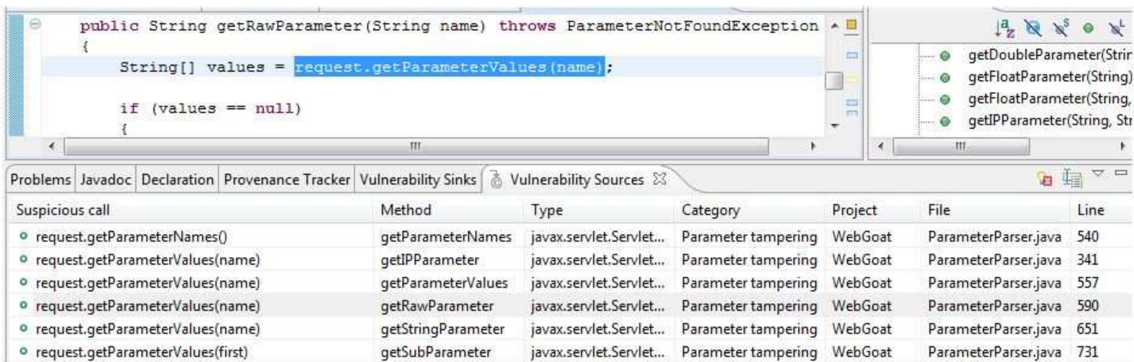


Ilustración 30: Resultado de búsqueda de objetos "source" para "Inyección de Comandos"

Ilustración 31: Resultado de búsqueda de objetos "sink" para "Inyección de Comandos"

6 INCLUSIÓN DE NUEVAS VULNERABILIDADES EN LAPSE+

6.1 Procedimiento de Inclusión de las Nuevas Vulnerabilidades

LAPSE ha sido diseñado como un plug-in de Eclipse para poder integrarse bien en el entorno de desarrollo, de forma que el análisis de seguridad pueda constituir una de las fases de desarrollo del software de calidad. Por otro lado, LAPSE permite a los desarrolladores incluir nuevas vulnerabilidades que no estén especificados en la versión oficial. Este proceso es muy amigable, ya que no exige entendimiento del funcionamiento interno del plug-in.

El procedimiento que debe seguirse con el fin de incluir nuevas vulnerabilidades tiene los siguientes pasos:

1. Definir en PQL la vulnerabilidad que se desea incluir. La especificación debería seguir el formato de modelo “tainted object propagation” (ver sección 2.3.2), es decir, se debe indicar claramente la consulta *sink(), derived()* y la consulta *source().*
2. Incluir las llamadas especificadas como “matches” de la consulta *source()* en el fichero “sources.xml”, especificando la categoría de la vulnerabilidad a la que pertenece. Por ejemplo:
   ```
   matches{
   sourceObj = cookie.getPath()
   …
   ```
   aparece en *“sources.xml”* como:
   ```
   <source id="javax.servlet.http.Cookie.getPath()">
      <category>Cookie poisoning</category>
   </source>
   ```
3. Incluir las llamadas especificadas como “matches” de la consulta *sink()* en el fichero “*sinks.xml*”, especificando la categoría de la vulnerabilidad a la que pertenece, el número de parámetros menos uno, y el parámetro vulnerable. Los parámetros no vulnerables en PQL aparecen como ‘_’, sin embargo en *“sinks.xml”* hay que especificar su tipo. Por ejemplo:

```
matches { …
|io.RandomAccessFile(sinkObj,_)
…
```

aparece en *“sinks.xml”* como:

```
<sink id="java.io.RandomAccessFile(String,String)">
   <vulnParam>0</vulnParam>
   <paramCount>1</paramCount>
   <category>Path traversal</category>
</sink>
```

4. Incluir las llamadas especificadas como “matches” de la consulta *derived()* en el fichero “*derived.xml*”, especificando el nombre del método y el tipo al que pertenece. Por ejemplo:

```
matches { …
| y = new String(x)
…
```

aparece en *“derived.xml”* como:

```
<derivation id="java.lang.String.String(String)">
   <type>java.lang.String</type>
   <method>String</method>
</derivation>
```

5. Opcionalmente se puede añadir alguna entrada al fichero “Safes.xml”, especificando el nombre del método, el tipo al que pertenece y la categoría de la vulnerabilidad.
6. Exportar el fichero *jar* de la aplicación a la carpeta principal del proyecto. Marcar la opción “generate new Manifest File”, en el cuadro de dialogo de la exportación.
7. Desplegar el plug-in, copiando la carpeta del proyecto en la carpeta plug-ins del directorio donde reside Eclipse.

6.2 Especificación de las Nuevas Vulnerabilidades

LAPSE fue creado en el año 2006. Como se indica en la sección de Antecedentes de este trabajo las vulnerabilidades para cuya detección fue diseñado LAPSE son las vulnerabilidades clasificadas como más comunes en las aplicaciones Web por OWASP en el proyecto “TOP TEN 2004”. En el año 2007, OWASP actualizó el proyecto “TOP TEN” y sacó una nueva lista de vulnerabilidades más comunes en las aplicaciones web, haciendo una adaptación especial para las aplicaciones J2EE. En la sección 2.1.2 de este trabajo se describen las vulnerabilidades mencionadas en “TOP TEN J2EE 2007”. Aunque se han añadido algunas vulnerabilidades nuevas respecto a la versión previa del proyecto, la mayoría no ha cambiado. En las primeras posiciones seguían las vulnerabilidades relacionadas con los ataques de “Cross Site Scripting”, “Inyección SQL” y “Path Traversal”. Las demás vulnerabilidades definidas en esa lista son difíciles de detectar mediante el análisis estático de código.

Uno de los objetivos de este trabajo, ha sido actualizar el proyecto LAPSE con el fin de que sea capaz de detectar los problemas de seguridad recientes. Una proceso reciente en las aplicaciones Web ha sido centrarse más en el usuario final, y adaptarse a la Web 2.0. En particular este año, hace varios meses apareció un documento con las vulnerabilidades más comunes en las aplicaciones Web 2.0. Este documento fue creado por Secure Enterprise 2.0 Forum [7].

Algunas de las vulnerabilidades allí definidas son las mismas que aparecen en el proyecto OWASP 2007, tales como “Cross Site Scripting”, “Cross Site Forgery”, etc. Sin embargo hay una clase de ataques que son muy comunes en las aplicaciones Web recientes y sin embargo no están incluidas en LAPSE. Estos ataques son los nuevos tipos de Inyecciones, tales como Inyección XML, Inyección XPath, etc. Las aplicaciones Web 2.0 son particularmente vulnerables a estos ataques ya que dependen en un grado alto del código de la parte cliente y procesan entradas del cliente que el atacante puede fácilmente desviar. Los servicios Web y AJAX son las tecnologías clave de la Web 2.0 y ambos utilizan XML. Además estos ataques suelen ser menos conocidos por los desarrolladores que Inyección SQL o “Cross Site Scripting” y por tanto puede que haya más aplicaciones vulnerables.

6.2.1 Inyección XPath

XPath es un lenguaje utilizado para referirse a parte de un documento XML. Puede ser utilizado directamente para consultar los documentos XML por la aplicación o puede formar parte de una operación más grande, como la transformación de un XSLT a un documento XML o la aplicación de XQuery, el lenguaje diseñado para consultar colecciones de datos XML.

6.2.1.1 Descripción de la Vulnerabilidad

Las aplicaciones XML a menudo utilizan XPath como un lenguaje de consulta para manipular datos de la misma forma que SQL se utiliza para manejar las bases de datos relacionales. La Inyección XPath es un ataque en el que las entradas especialmente diseñadas utilizadas por la aplicación dentro de una consulta XPath, alteran la consulta para conseguir los objetivos del atacante.

El funcionamiento de la Inyección XPath es muy parecido al funcionamiento de Inyección SQL. Solamente que en lugar de base de datos aparecen ficheros XML, y en lugar del lenguaje SQL las consultas XPath. A continuación se muestra un ejemplo de implementación del ataque de Inyección de XPath.

En el Listado 27 se describe un ejemplo de fichero XML que constituye un repositorio de usuarios.

```
<?xml version="1.0" encoding="UTF-8"?>
<usarios>
      <usuario>
          <nombre>Alberto</nombre>
          <apellido>Gonzalez</apellido>
          <loginID>abc</loginID>
          <contraseña>test123</contraseña>
      </usuario>
      <usuario>
          <nombre>Almudena</nombre>
          <apellido>Grande</apellido>
          <loginID>almudena</loginID>
          <contraseña>000</contraseña>
      </usuario>
      <usuario>
          <nombre>Francisco</nombre>
          <apellido>Lopez</apellido>
          <loginID>fran</loginID>
          <contraseña>345</contraseña>
      </usuario>
</usuarios>
```

Listado 27: Fichero "usuarios.xml"

La consulta para la comprobación de nombre de usuario y la contraseña es la que se muestra en el Listado 28.

```
//usuarios/usuario[loginID/text()='abc' and
contraseña/text()='test123']
```

Listado 28: Consulta XPath

Esta consulta se puede utilizar en Java para autenticar a los usuarios. En el Listado 29 se observa un fragmento del código Java que realiza la autenticación de usuarios. Sin embargo, si no se valida bien la entrada, se podría pasar una consulta que siempre devuelva verdadero como la especificada en Listado 30.

```
…
XPath xpath = factory.newXPath();
         XPathExpression expr = xpath.compile("//usuarios/usuario
[loginID/text()='"+loginID+"'
and contraseña/text()='"+contraseña+"' ]/nombre/text()");
     Object result = expr.evaluate(doc, XPathConstants.NODESET);
         NodeList nodes = (NodeList) result;
//imprimir los nombres a la consola
         for (int i = 0; i < nodes.getLength(); i++) {
             System.out.println(nodes.item(i).getNodeValue());}
…
```

Listado 29: Fragmento del código vulnerable a la "Inyección XPath"

```
//usuarios/usuario[loginID/text()='' or 1=1 or ''='' and
contraseña/text()='' or 1=1 or '''='']
```

Listado 30: Vector para el ataque de "Inyección XPath"

6.2.1.2 Especificación de Inyección XPath en LAPSE

Con el fin de incluir la búsqueda con LAPSE de esta vulnerabilidad en el código fuente se va a seguir el procedimiento descrito en la sección 4.1.

El primer paso exige especificar la vulnerabilidad en el lenguaje PQL. Dado que la entrada de usuario puede venir por los mismos caminos que en el caso de las vulnerabilidades ya incluidas en LAPSE, las consultas *source()* definidas en sección 3.7 son aplicables a la Inyección XPath. También es aplicable el descriptor *derived(),* dado que el objeto infectado es una cadena de tipo String.

La única consulta que es preciso definir es la consulta *sink().* Se considerarán vulnerables a la Inyección XPath todos los métodos Java que procesan las consultas de tipo XPath. Hasta hace poco la API mediante la cual los programas Java hacían consultas XPath dependía del motor de procesamiento XPath. Para independizar la aplicación del motor XPath particular, Java 5 introdujo el paquete `javax.xml.xpath`, que provee un motor que no depende de ninguna librería particular. Este paquete define varios métodos para compilación y evaluación de consultas XPath. Todos estos métodos se van a incluir en la especificación de consulta PQL *sink().*

Sin embargo, después de analizar ejemplos de aplicaciones “open source” se ha observado que existen todavía muchos programas que utilizan un motor particular en vez del estándar java. Por tanto se ha decidido dar soporte a las API específicas en la implementación de detección de esta vulnerabilidad en LAPSE.

Una librería conocida que procesa el lenguaje XPath es “Xalan” de apache [3]. En concreto la clase `org.apache.xpath.XPath` encapsula la expresión XPath y provee servicios para su ejecución. Los diferentes constructores de esta clase inicializan el compilador XPath y compilan la expresión. Por tanto se van a añadir a la consulta *sink().*

Otra librería utilizada por las aplicaciones es JXPath de apache. JXPath es un simple intérprete de expresiones XPath

Existe también un modulo de la XML:DB, una especificación de base de datos XML(veáse la sección siguiente) que provee un servicio para compilación de expresiones XPath. Este servicio se implementa en la clase `org.xmldb.api.modules.XPathQueryService` y define dos métodos para la compilación de consultas XPath.

Finalmente después de analizar diferentes librerías Java XPath, se obtiene la consulta *sink()* completa, mostrada en Listado 31.

```
query sink()
returns
object Object sinkObj;
uses
javax.xml.xpath.XPath path
org.apache.xpath apath
org.xmldb.api.modules.XPathQueryService qs
matches { path.compile(sinkObj)
|path.evaluate(sinkObj,_)
|path.evaluata(sinkObj,_,_)
|apath.XPath(sinkObj,_,_,_)
|apath.XPath(sinkObj,_,_,_,_)
|apath.XPath(sinkObj,_,_,_,_,_)
|qs.query(sinkObj)
|qs.query(_,sinkObj)
;}
```

Listado 31: Consulta PQL "sink" para la "Inyección XPath"

Una vez especificada la consulta PQL se procede a incluir las llamadas en el fichero "sinks.xml". El fichero "source.xml" y "derived.xml" no necesita actualización ya que no se han añadido nuevas llamadas de estos dos tipos. Las entradas añadidas al fichero "sink.xml" se especifican en el Listado 32. Se añaden los nombres completos de llamadas vulnerables, con el tipo y número de parámetros que contienen y se especifica el número del parámetro vulnerable. La categoría definida para todas estas llamadas es la "Inyección XPath", el nombre inglés de la vulnerabilidad.

```
<sink id="javax.xml.xpath.XPath.compile(String)">
         <vulnParam>0</vulnParam>
         <paramCount>1</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink id="javax.xml.xpath.XPath.evaluate(String,InputSource)">
         <vulnParam>0</vulnParam>
         <paramCount>2</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink id="javax.xml.xpath.XPath.evaluate(String,InputSource,QName)">
         <vulnParam>0</vulnParam>
         <paramCount>3</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink id="javax.xml.xpath.XPath. evaluate(String,Object)">
         <vulnParam>0</vulnParam>
         <paramCount>2</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink id="javax.xml.xpath.XPath.evaluate(String,Object,QName)">
         <vulnParam>0</vulnParam>
         <paramCount>3</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink
id="org.apache.xpath.XPath(String,SourcelLocator,PrefixResolver,type)">
         <vulnParam>0</vulnParam>
         <paramCount>4</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink
id="org.apache.xpath.XPath(String,SourcelLocator,PrefixResolver,type,
ErrorListener)">
         <vulnParam>0</vulnParam>
         <paramCount>5</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink
id="org.apache.xpath.XPath(String,SourcelLocator,PrefixResolver,type,
Errorlistener,FunctionTable)">
         <vulnParam>0</vulnParam>
         <paramCount>4</paramCount>
         <category>XPath Inyection</category>
</sink>
</sink>
<sink id="org.xmldb.api.modules.XPathQueryService.query(String)">
         <vulnParam>0</vulnParam>
         <paramCount>1</paramCount>
         <category>XPath Inyection</category>
</sink>
<sink
id="org.xmldb.api.modules.XPathQueryService.query(String,String)">
         <vulnParam>1</vulnParam>
         <paramCount>2</paramCount>
         <category>XPath Inyection</category>
</sink>
```

Listado 32: Actualización de fichero "sink.xml", con la “Inyección XPath”

6.2.1.3 Detección de Inyección XPath en las Aplicaciones

La aplicación analizada es una agenda de direcciones que se distribuye como ejemplo de uso de una base de datos de apache Xindice [4]. En la Ilustración 32 se puede apreciar que el escaneo de la aplicación con LAPSE ha detectado dos posibles vulnerabilidades de tipo Inyección XPath en el código fuente.

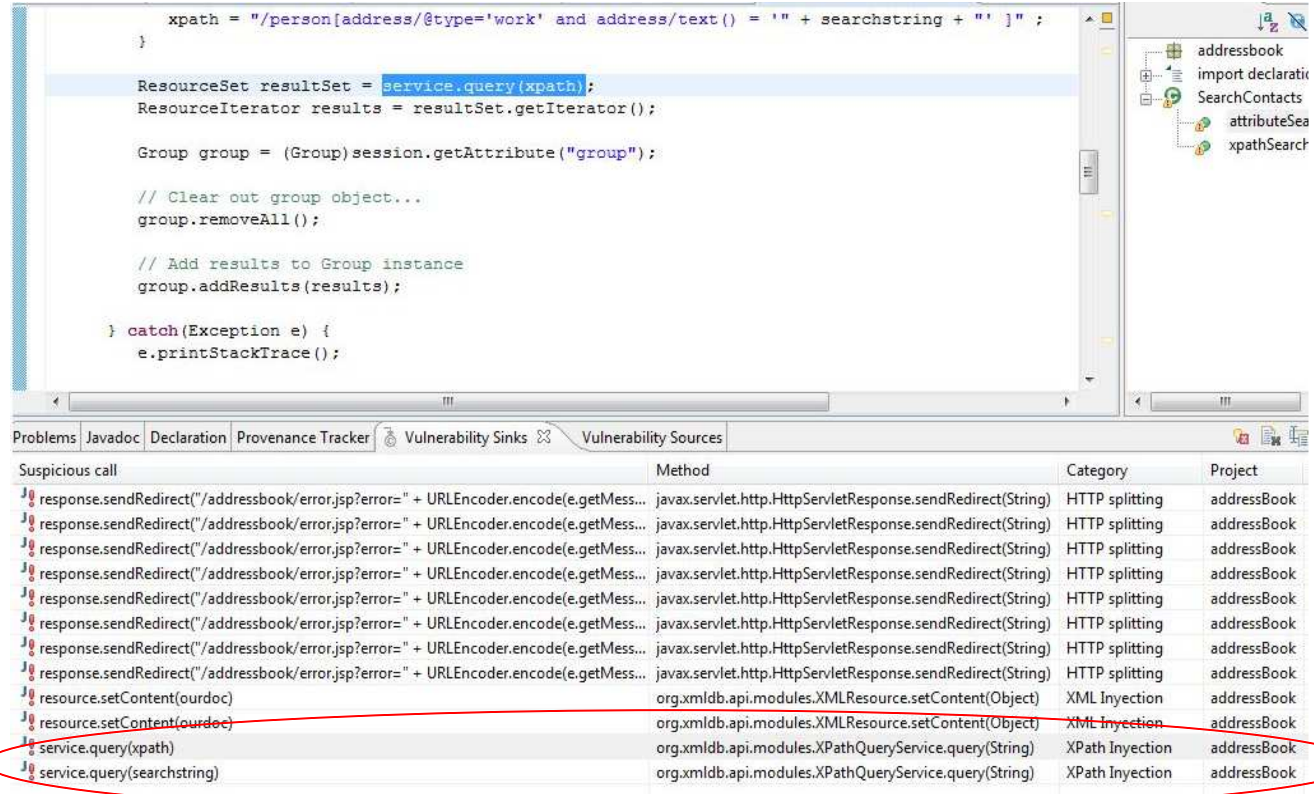


Ilustración 32: Resultado de búsqueda de objetos "sink" para "Inyección XPath"

En la Ilustración 33 se observa el resultado de escanear la misma aplicación en busca de llamadas de tipo "source". Dado que se han detectado varias llamadas de tipo "source" se procede a ejecutar la propagación, para ver si la vulnerabilidad potencial es real.

Para el primer método de tipo "sink" detectado en la Ilustración 32 la propagación en muy pocos pasos llega a un objeto de tipo "source". El resultado de esta propagación se observa en la Ilustración 34.

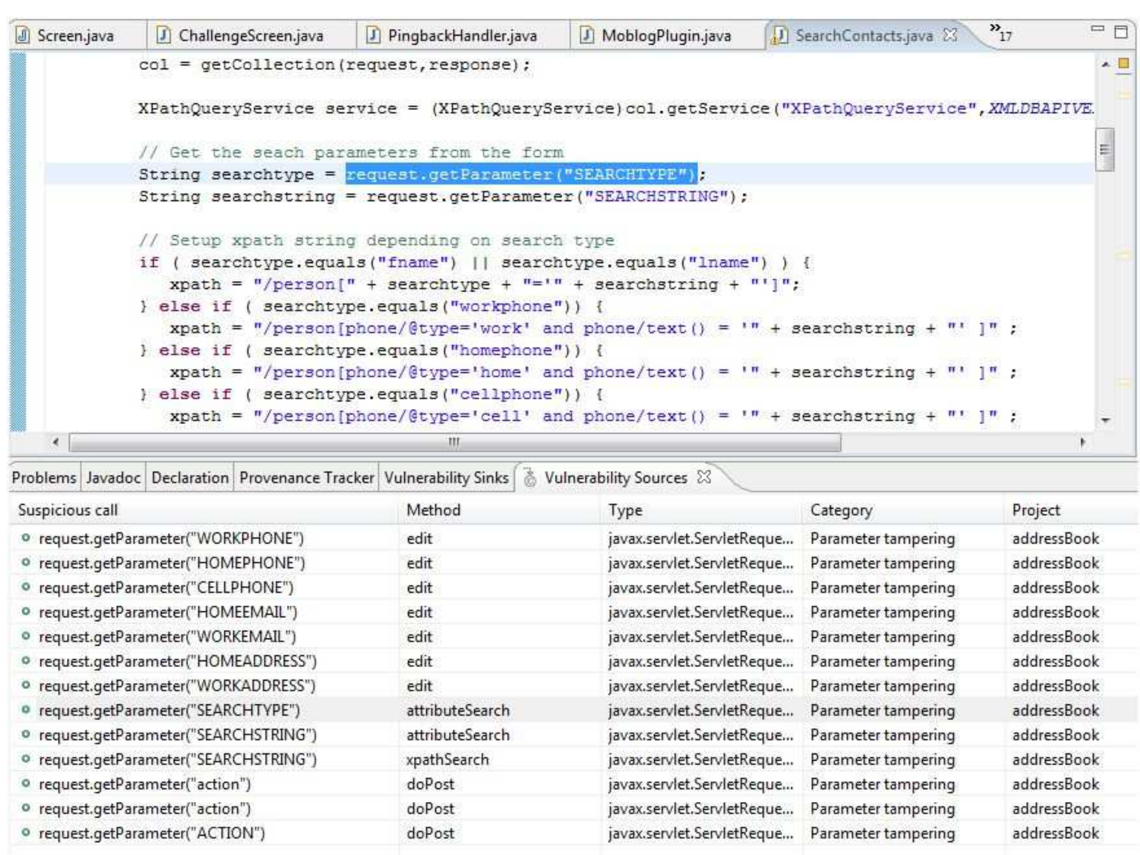


Ilustración 33: Resultado de búsqueda de objetos "source" para "Inyección XPath"

```
// Get the seach parameters from the form
String searchstring = request.getParameter("SEARCHSTRING");

ResourceSet resultSet = service.query(searchstring);
ResourceIterator results = resultSet.getIterator();

Group group = (Group)session.getAttribute("group");

// Clear out group object...
group.removeAll();

// Add results to Group instance
group.addResults(results);
```

Problems | Javadoc | Declaration | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Created a slice with 1 leaf element(s) and 3 element(s) located in 1 file(s) with 0 element(s) truncated with a maximum depth of 3. Using a hierarchical viewer.

- searchstring (SearchContacts.java:121) [initial]
  - searchstring=request.getParameter("SEARCHSTRING") (SearchContacts.java:119) [local variable declaration]
    - request.getParameter("SEARCHSTRING") (SearchContacts.java:119) [call expression]

Ilustración 34: Resultado de propagación para la vulnerabilidad " Inyección XPath "

En caso de la segunda llamada "sink", la propagación exige dos pasos. En el primer paso se realiza una propagación desde el método detectado como "sink", sin embargo se detecta que la variable, objeto "sink", esta inicializada al *null*. (Ilustración 35). Esto significa que probablemente en un punto ulterior del código se inicialice la variable con el valor real.

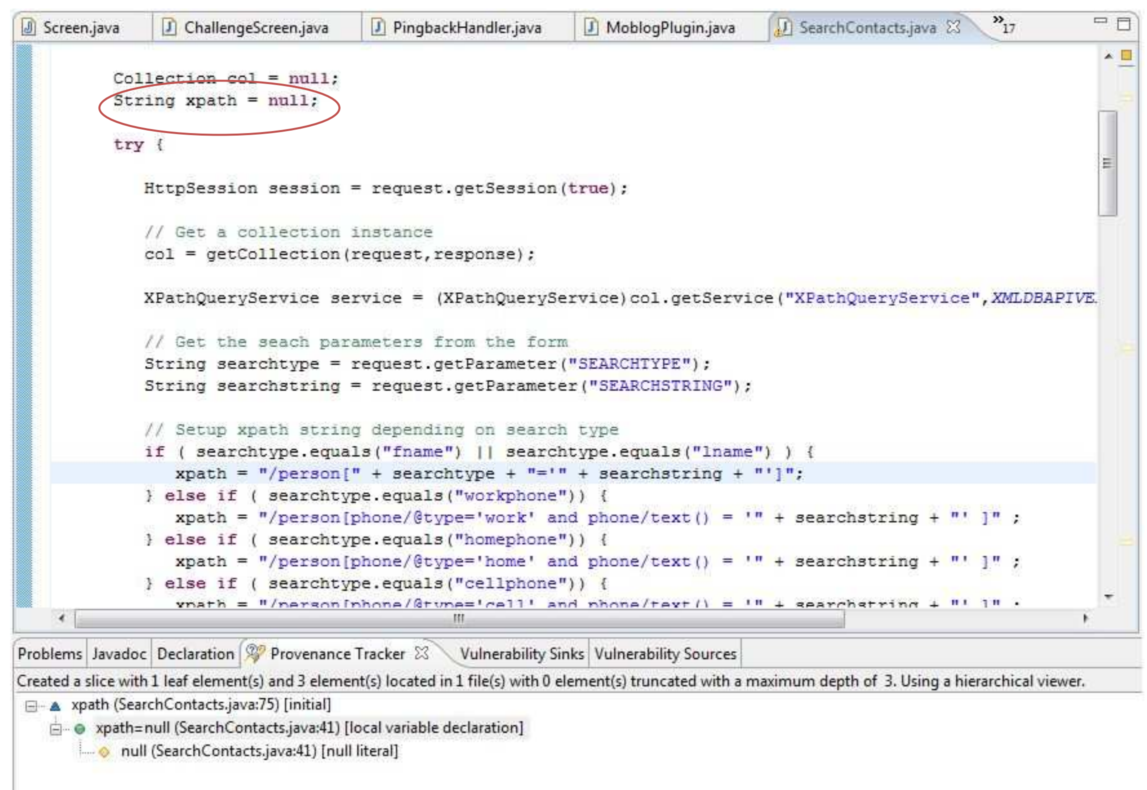


Ilustración 35: Resultado de propagación para la vulnerabilidad " Inyección XPath " (2)

Analizando el código de forma manual se observa que unas líneas más abajo de la primera inicialización a null, la variable XPath vuelve a ser inicializada con un valor real. Por tanto el siguiente paso consiste en seleccionar la parte derecha de esta asignación y seguir propagando. Como se puede observar en la Ilustración 36, la parte derecha es una concatenación de cadenas en la cual dos son objetos de tipo "source".

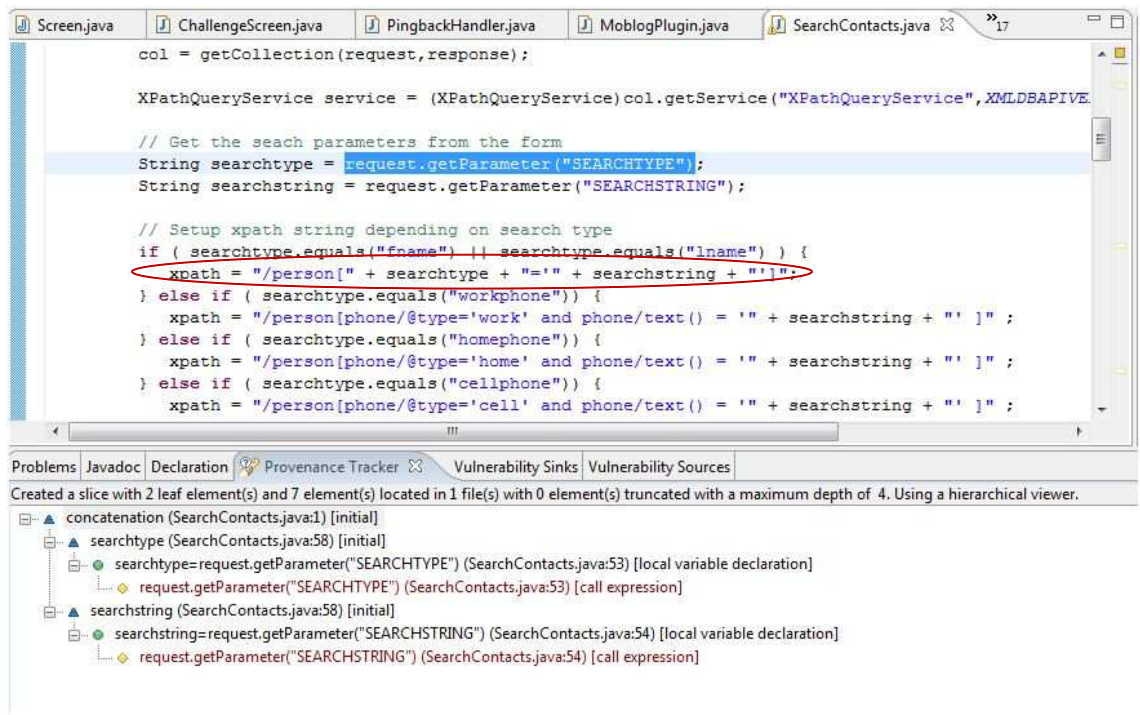


Ilustración 36: Resultado de propagación para la vulnerabilidad " Inyección XPath " (3)

6.2.2 Inyección XML

La Inyección XML es posible en las aplicaciones que utilizan las bases de datos XML. Existen dos tipos de estas bases de datos: las que convierten en XML el contenido de las bases de datos relacionales, y las que utilizan los documentos XML como el único sistema de almacenamiento. Los dos tipos son igual de vulnerables ante el ataque de Inyección XML sino validan bien las entradas del usuario final. Dado que las bases de datos XML representan una nueva tecnología que no posee una especificación formal, surgió una iniciativa llamada XML:DB [6] cuyo objetivo es estandarizar la industria de las bases de datos XML.

6.2.2.1 Descripción de la Inyección XML

La Inyección XML es un ataque en el cual las entradas de usuario sin validar se insertan en los campos de un mensaje XML. Las entradas inyectadas modifican la estructura de los campos XML añadiendo no solamente contenido sino también etiquetas. La inyección de un contenido o estructuras malignas en un mensaje XML pude modificar la lógica programada de la aplicación y comprometer a la misma o al servicio que proporciona. Si el flujo XML se guarda en una base de datos se llega a comprometer también la base de datos.

Un ejemplo del ataque de Inyección XML es cuando un usuario, para darse de alta en el sistema, introduce sus datos. Estos datos se van a añadir al fichero de usuarios especificado en el Listado 27. Si se introducen los datos especificados en el Listado 33 y no se validan bien, se añadiría un nuevo usuario con el nombre hacker y el *login* “abc”. Muchas veces si se inserta un *login* ya existente el usuario simplemente se sobrescribe con lo cual el atacante puede ganar el acceso al sistema con todos los privilegios que desea.

```
Nombre: Alejandro
Apellido: Gomez
LoginId: alej_gomez
Contraseña:
abc123</contraseña></usuario><usuario><nombre>Hacker</nombre><apellido>h
acker</apellido><loginId>abc<loginId><contraseña>123</contraseña></usuar
io>
```

Listado 33: Vector de ataque de la “Inyección XML”

6.2.2.2 Especificación de Inyección XML en LAPSE

El primer paso consiste en definir las consultas PQL que especifiquen la vulnerabilidad. Como en caso de Inyección XPath las consultas *source()* y *derived()* para esta vulnerabilidad son las definidos en la sección 3.7.1. Esto se debe a que las entradas de datos al sistema son iguales que en vulnerabilidades anteriores y el objeto vulnerable es cadena String siendo validas también las llamadas de derivación anteriormente definidas.

Para definir la consulta *sink()* es necesario buscar todos los métodos Java que escriben datos en un mensaje o flujo XML. La librería utilizada para este fin es la "deXML:DB", en particular el paquete `org.xmldb.api.modules.XMLResource`, el cual representa el recurso XML de la base de datos. Los métodos vulnerables a la Inyección XML son todos los métodos que establecen el contenido del recurso XML *XMLResource.*

```
query sink()
returns
object Object sinkObj;
uses
org.xmldb.api.modules.XMLResource res
matches {
res.setContent(sinkObj)
;}
```

Listado 34: Consulta PQL sink() para la "Inyección XML"

Como en caso de XPath Inyection, una vez especificada la consulta PQL se procede a incluir las llamadas en el fichero "sinks. xml". Las entradas añadidas al fichero "sink.xml" se especifican en el Listado 35.

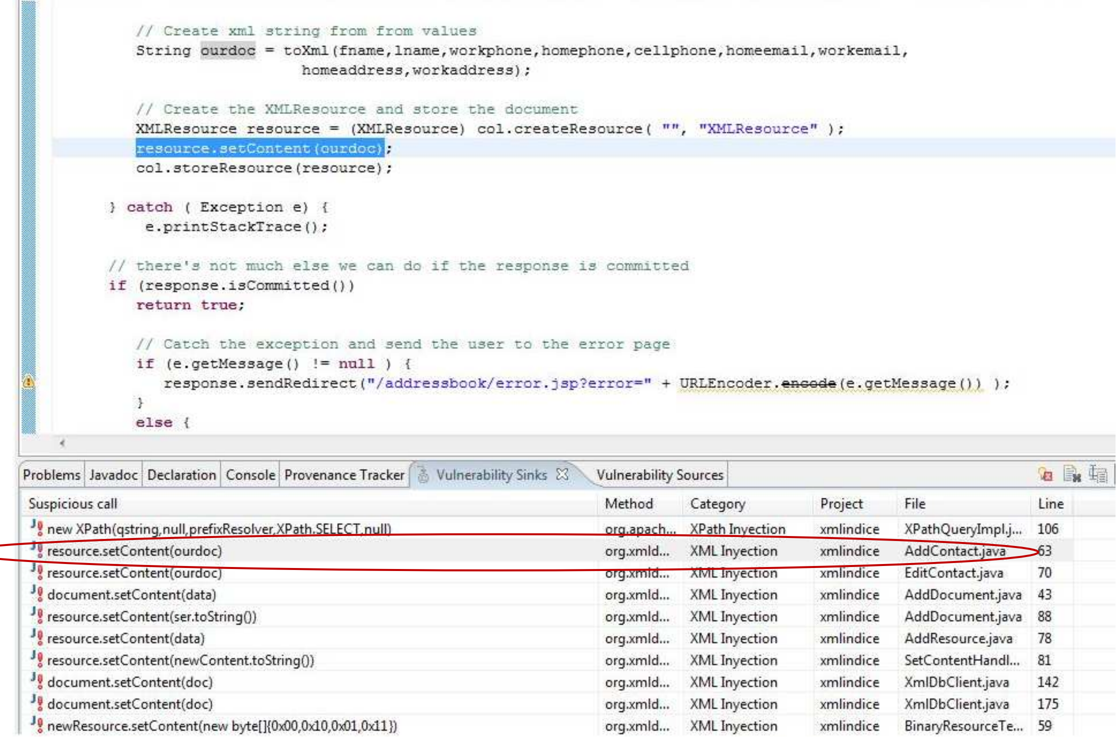
```
<sink id="org.xmldb.api.modules.XMLResource.setContent(Object)">
         <vulnParam>0</vulnParam>
         <paramCount>1</paramCount>
         <category>XML Inyection</category>
</sink>
```

Listado 35: Actualización de fichero "sink.xml", con la "Inyección XML"

6.2.2.3 Detección de Inyección XML en las Aplicaciones

Como hemos visto en la sección anterior, al analizar la aplicación de libreta de direcciones con el fin de encontrar las vulnerabilidades de tipo Inyección XPath, se han detectado también posibles vulnerabilidades ante el ataque de la Inyección XML. Por eso motivo se va a analizar la misma aplicación en esta sección. La única diferencia es que se analiza no solamente la aplicación de libreta sino todo el proyecto "xindice" de Apache.

En la Ilustración 37 se observa el resultado de búsqueda de objetos "sink" para esta aplicación. LAPSE encuentra varias vulnerabilidades potenciales ante el ataque de la Inyección XML. A continuación se va a analizar la llamada "sink" marcada con el círculo rojo en la Ilustración 37.



Suspicious call	Method	Category	Project	File	Line
new XPath(qstring,null,prefixResolver,XPath.SELECT,null)	org.apach...	XPath Inyection	xmlindice	XPathQueryImpl.j...	106
resource.setContent(ourdoc)	org.xmld...	XML Inyection	xmlindice	AddContact.java	63
resource.setContent(ourdoc)	org.xmld...	XML Inyection	xmlindice	EditContact.java	70
document.setContent(data)	org.xmld...	XML Inyection	xmlindice	AddDocument.java	43
resource.setContent(ser.toString())	org.xmld...	XML Inyection	xmlindice	AddDocument.java	88
resource.setContent(data)	org.xmld...	XML Inyection	xmlindice	AddResource.java	78
resource.setContent(newContent.toString())	org.xmld...	XML Inyection	xmlindice	SetContentHandl...	81
document.setContent(doc)	org.xmld...	XML Inyection	xmlindice	XmlDbClient.java	142
document.setContent(doc)	org.xmld...	XML Inyection	xmlindice	XmlDbClient.java	175
newResource.setContent(new byte[]{0x00,0x10,0x01,0x11})	org.xmld...	XML Inyection	xmlindice	BinaryResourceTe...	59

Ilustración 37: Resultado de búsqueda de objetos "sink" para "Inyección XML"

El segundo paso, antes de realizar la propagación, consiste en ver si existen también las llamadas de tipo "source" en el código. La Ilustración 38 demuestra que efectivamente LAPSE encuentra varios métodos que producen los objetos vulnerables de tipo "source".

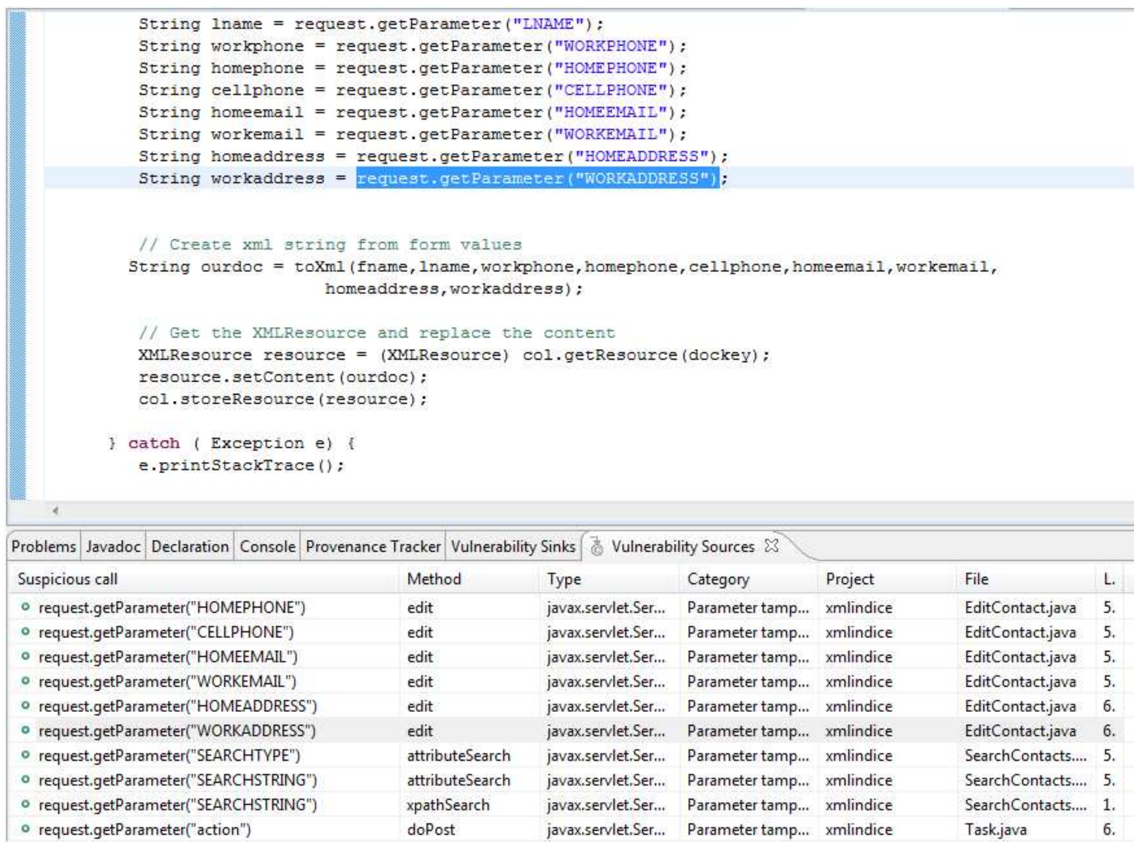


Ilustración 38: Resultado de búsqueda de objetos "source" para "Inyección XML"

Una vez detectados los métodos "source" y "sink" se procede a comprobar mediante la propagación si la vulnerabilidad es real. La Ilustración 39 muestra el primer paso de la propagación. La variable infectada es "ourdoc" sin embargo la propagación en el primer paso llega solamente al paso de la declaración local. Si se investiga manualmente el código se observa que a la variable "ourdoc" se le concatenan muchas cadenas con el formato de mensajes XML. A modo de ejemplo, se ha propagado solamente desde la última línea inspeccionada. El resultado de este segundo paso de la propagación se observa en la Ilustración 40. La variable resulta ser infectada, ya que la propagación llega a un objeto de tipo "source".

```
        // Create xml string from from values
        String ourdoc = toXml(fname,lname,workphone,homephone,cellphone,homeemail,workemail,
                              homeaddress,workaddress);

        // Create the XMLResource and store the document
        XMLResource resource = (XMLResource) col.createResource( "", "XMLResource" );
        resource.setContent(ourdoc);
        col.storeResource(resource);

    } catch ( Exception e) {
        e.printStackTrace();

    // there's not much else we can do if the response is committed
    if (response.isCommitted())
        return true;

        // Catch the exception and send the user to the error page
        if (e.getMessage() != null ) {
            response.sendRedirect("/addressbook/error.jsp?error=" + URLEncoder.encode(e.getMessage()
        }
        else {
```

Problems | Javadoc | Declaration | Console | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Created a slice with 1 leaf element(s) and 6 element(s) located in 2 file(s) with 0 element(s) truncated with a maximum depth of 6. Using a hierarchical viewer.

- ourdoc (AddContact.java:63) [initial]
  - ourdoc=toXml(fname,lname,workphone,homephone,cellphone,homeemail,workemail,homeaddress,workaddress) (AddContact.java:58) [local variable
    - toXml(fname,lname,workphone,homephone,cellphone,homeemail,workemail,homeaddress,workaddress) (AddContact.java:58) [call expression]
      - return ourdoc (Action.java:141) [return value]
        - ourdoc (Action.java:141) [assignment operation]
          - ourdoc (Action.java:125) [local variable declaration]

Ilustración 39: Resultado de propagación para la vulnerabilidad "Inyección XML" (1)

```
/**
 * Create a String XML document from passed in parameters
 */
public String toXml(String fname, String lname, String workphone,
        String homephone, String cellphone, String homeemail, String workemail,
        String homeaddress, String workaddress) {

    String ourdoc;

    // Build up the string to represent our document
    ourdoc = "<person>";
    ourdoc = ourdoc + "<fname>" + fname + "</fname>";
    ourdoc = ourdoc + "<lname>" + lname + "</lname>";
    ourdoc = ourdoc + "<phone type='work'>" + workphone + "</phone>";
    ourdoc = ourdoc + "<phone type='home'>" + homephone + "</phone>";
    ourdoc = ourdoc + "<phone type='cell'>" + cellphone + "</phone>";
    ourdoc = ourdoc + "<email type='home'>" + homeemail + "</email>";
    ourdoc = ourdoc + "<email type='work'>" + workemail + "</email>";
    ourdoc = ourdoc + "<address type='home'>" + homeaddress + "</address>";
    ourdoc = ourdoc + "<address type='work'>" + workaddress + "</address>";
    ourdoc = ourdoc + "</person>";
```

Problems | Javadoc | Declaration | Console | Provenance Tracker | Vulnerability Sinks | Vulnerability Sources

Created a slice with 8 leaf element(s) and 30 element(s) located in 7 file(s) with 0 element(s) truncated with a maximum depth of 10. Using a hierarchical viewer.

- workaddress (Action.java:137) [initial]
  - String workaddress (Action.java:123) [formal argument]
    - workaddress (AddContact.java:59) [argument of a call]
      - workaddress=request.getParameter("WORKADDRESS") (AddContact.java:55) [local variable declaration]
        - request.getParameter("WORKADDRESS") (AddContact.java:55) [call expression]

Ilustración 40: Resultado de propagación para la vulnerabilidad "Inyección XML" (2)

7 COMPARATIVA LAPSE y LAPSE+

Con el fin de probar la nueva herramienta LAPSE+ y realizar la comparativa con la antigua versión de LAPSE se ha realizado el análisis estático para detectar los problemas especificados como “tainted object propagation”, descritos en la sección 2.3.2.

El número total de los descriptores “sink”, “source” y de “derivación”, utilizado en cada versión de LAPSE, se especifica en la Tabla 10. Los descriptores utilizados son los especificados en las consultas PQL en la sección 5.

	Descriptores “sink”	Descriptores “source”	Descriptores de “derivación”
LAPSE	35	42	0
LAPSE+	47	43	5

Tabla 10: Número de descriptores "sink", "source" y "derived" utilizados en las pruebas realizadas con LAPSE y LAPSE+

Se observa que el número de descriptores de tipo “sink” utilizado por LAPSE+ es mucho mayor que el utilizado por LAPSE. Esto se debe a que LAPSE+ incluye detección de nuevas vulnerabilidades, cada una con su correspondiente conjunto de métodos vulnerables de tipo “sink”. El número de descriptores “source” es casi igual en LAPSE que en LAPSE+, dado que LAPSE+ utiliza las mismas llamadas de tipo “source” que LAPSE. Finalmente LAPSE+ utiliza 5 descriptores de derivación, mientras que LAPSE no utiliza ninguno.

Con el fin de llevar acabo las pruebas de comparación entre LAPSE y LAPSE+, ha sido necesario elaborar un conjunto de aplicaciones de prueba. Estas aplicaciones provienen de dos repositorios de aplicaciones “open-source”, frecuentemente utilizados. El primer repositorio es Java-source [10] donde se encuentran aplicaciones escritas solo en Java, como su propio nombre indica. Otro repositorio es el sourceforge [29], que aunque contienen aplicaciones escritas en diferentes lenguajes de programación, es un repositorio más grande y más utilizado que el primero.

La aplicaciones web seleccionadas para realizar las pruebas, son programas reales, frecuentemente utilizados, y del ámbito de web 2.0, tales como blogs, wikipedias, bases de datos XML, etc. El listado completo de las aplicaciones utilizadas se muestra en la Tabla 11. Las versiones utilizadas son las más recientes de entre las que tuvieran el código fuente disponible.

Nombre	Versión	Tipo	Repositorio
Apache xindice – Address Book	1.1	XML	http://xml.apache.org/xindice/
Blogunity	0.1	Blog	http://blogunity.sourceforge.net/
Blojsom	3.1	Blog	http://java-source.net/open-source/bloggers/blojsom
Chikiwiki	0.24	Wikipedia	http://sourceforge.net/projects/chiki/
Elvis	1.8	e-Biblioteca	http://sourceforge.net/projects/elvis-dl/
JavaBlogLib	1.1	Blog	http://sourceforge.net/projects/javabloglib/
JSP-Wiki	2.8.2	Wiki	http://www.jspwiki.org/
Thingamablog	1.0.6	Blog	http://java-source.net/open-source/bloggers/thingamablog
Washi	23	Blog	http://sourceforge.net/projects/washi/
Xalan-ejemplos	2_7_0	XML	http://xml.apache.org/xalan-j/

Tabla 11: Aplicaciones Web utilizadas en las pruebas con LAPSE y LAPSE+

Se han realizado los análisis de todas las aplicaciones enumeradas en la Tabla 11, con el plug-in LAPSE y con el plug-in LAPSE+. En la Tabla 12 se pueden observar los resultados. Se observa que los resultados obtenidos con el LAPSE+ son muchos más completos que los obtenidos con el LAPSE. LAPSE+ ha detectado tres tipos de vulnerabilidades más que LAPSE. Además el número total de las vulnerabilidades encontradas con LAPSE+ es cuatro veces mayor que el número de las encontradas con LAPSE.

	Inyección SQL	XSS	Path Traversal	Inyección de Commandos	Inyección XPath	Inyección XML	Total
LAPSE	1	7	0	0	0	0	**8**
LAPSE+	1	7	19	0	3	5	**35**

Tabla 12: Número de vulnerabilidades encontradas por LAPSE y LAPSE+

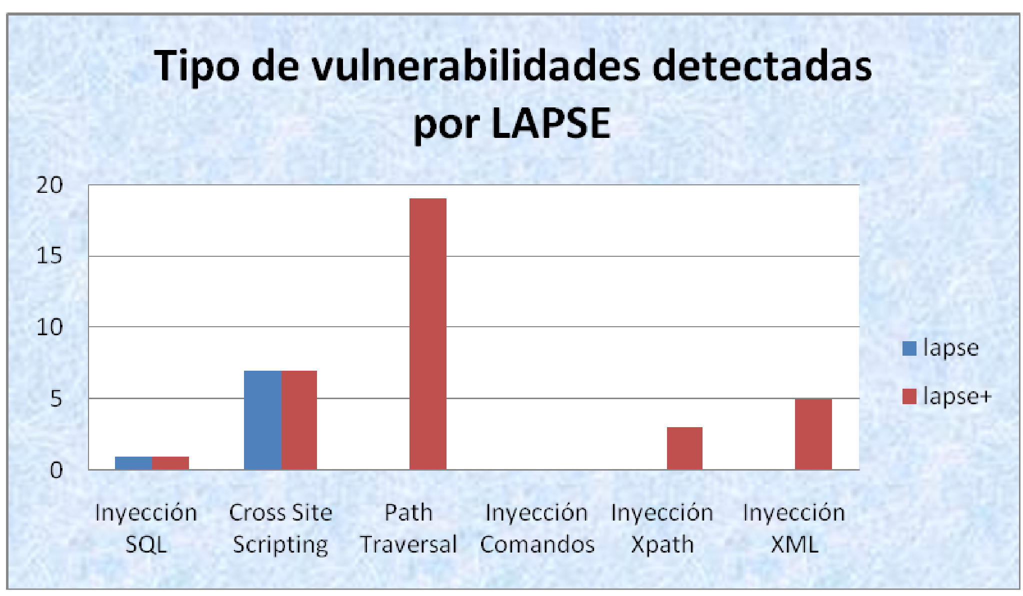


Figura 1: Comparativa de tipos de vulnerabilidades detectadas por LAPSE y LAPSE+

En la Figura 1 se observan los diferentes tipos de vulnerabilidades detectados por LAPSE. Se puede apreciar que la vulnerabilidad más frecuente es "Path Traversal" y sin embargo la antigua versión de LAPSE no es capaz de detectarla. Las dos nuevas vulnerabilidades relacionadas con el uso de bases de datos XML, la "Inyección XML" y la "Inyección XPath" también aparecen con bastante frecuencia y solamente son detectadas por LAPSE+. Las vulnerabilidades de "Cross Site Scripting" y "Inyección SQL" se detectan utilizando cualquiera de las dos versiones de LAPSE. Es interesante observar que la Inyección SQL es cada vez menos frecuente, sin embargo aparecen las Inyecciones XML y XPath. Esto indica que por un lado las aplicaciones Web 2.0, tienden a trabajar más con las bases de datos XML que con las bases de datos tradicionales, y por otro lado que las inyecciones asociadas con el uso de XML, son ataques menos conocidos y los desarrolladores no los tienen en cuenta a la hora de construir su aplicación.

8 CONCLUSIONES

En este apartado se extraen una serie de conclusiones sobre el trabajo realizado a lo largo de duración de este proyecto. Se especifica un resumen final y las líneas de futuro desarrollo.

8.1 Resumen del Trabajo Realizado

En este trabajo se ha intentado mostrar cómo se pueden utilizar las técnicas de análisis estático del código para construir un software más seguro ante diversos ataques. En concreto se ha estudiado el trabajo del grupo SUIF de la Universidad de Stanford y las técnicas de análisis estático propuestas por este grupo. Se han tenido en cuenta sobre todo las vulnerabilidades en aplicaciones web cuyo origen se basa en la entrada de datos de usuario que no se valida de forma adecuada antes de ser utilizada por la aplicación, problema definido por el grupo SUIF como “tainted object propagation problem” (“propagación de objetos envenenados”).

Como el resultado de todo el análisis realizado, se desarrolló una herramienta para la detección de las vulnerabilidades de seguridad en el código Java, teniendo en cuenta las vulnerabilidades más recientes en las aplicaciones web. Esta herramienta es una versión mejorada de LAPSE – un plug-in de Eclipse desarrollado por el grupo SUIF de la Universidad de Stanford. LAPSE es una herramienta que se integra bien en el entorno de desarrollo habitual como es Eclipse y la inclusión de nuevas vulnerabilidades es muy poco costosa ya que no involucra cambios en el código de la herramienta. Las modificaciones y actualizaciones realizadas sobre LAPSE han originado una herramienta nueva: LAPSE+.

Las pruebas realizadas con LAPSE+ sobre 10 aplicaciones Web reales han detectado 35 vulnerabilidades en total, siendo la más común “Path Traversal”. Se han detectado también las vulnerabilidades de tipo “Inyección SQL”, “Cross Site Scripting” y las vulnerabilidades menos conocidas relacionadas con el uso de la tecnología XML, como “Inyección XML“ e “Inyección XPath”. Además LAPSE+ ha detectado tres tipos de vulnerabilidades más que LAPSE y el número total de las vulnerabilidades detectadas por LAPSE+ fue más de cuatro veces superior al número total de las vulnerabilidades detectadas por LAPSE.

8.2 Futuras Líneas de Desarrollo

LAPSE+ es muy fácil de utilizar, y se integra muy bien en el entorno de desarrollo Java – Eclipse. Sin embargo, en el futuro podría mejorarse para ofrecer una solución de seguridad más completa. A continuación se detallan las modificaciones y actualizaciones sobre LAPSE+ que se podrían llevar acabo:

- Actualizar LAPSE+ con las nuevas vulnerabilidades que se detecten en las aplicaciones Web y que se ajustan al patrón “tainted object propagation”. Este proceso es muy amigable y puede ser realizado por cualquiera que utilice LAPSE+.
- La mayor desventaja de LAPSE+, es un alto número de falsos positivos que produce. Un falso positivo es un resultado de análisis. Por tanto, una mejora significativa sería actualizar LAPSE+ para que realice un análisis más acertado. Convendría adaptar el análisis realizado por LAPSE+, al análisis sensible al contexto propuesto por el grupo SUIF. Este análisis, como se ha explicado en la sección 2.5, utiliza Datalog y la base de datos “bbdbbdb” y ha demostrado producir los resultados mucho más adecuados. La solución para LAPSE+ sería incluir el uso de Datalog y bbdbbd en el plug-in.
- Incluir en LAPSE+ soporte para otros modelos de vulnerabilidades, además de las definidas como “tainted object propagation”. Existen vulnerabilidades comunes en las aplicaciones Web, que sin embargo no se pueden detectar en LAPSE+ dado que no es posible definirlas con el patrón “tainted object propagation”. En un futuro se podría definir e incluir más patrones para especificar las vulnerabilidades.
- Exportar la herramienta LAPSE+ y el modelo en el que se basa a otros entornos de desarrollos web frecuentemente utilizados, como por ejemplo el entorno .NET.

.

8.3 Observaciones Finales

La inclusión de las herramientas en el análisis estático de código para objetivos de seguridad es un procedimiento todavía poco común en los desarrollos de software y sin embargo tiene mucha importancia. El software de calidad hoy en día es un software seguro y la seguridad construida desde las primeras fases de desarrollo es mucho más robusta.

Es importante destacar que siempre que se habla de seguridad Informática, la solución propuesta no es absoluta. También en este caso la herramienta desarrollada no pretende resolver todos los problemas relacionados con la seguridad de las aplicaciones web. Es solamente una ayuda a los procesos de desarrollo de software de calidad, que tengan como objetivo construir un código seguro. LAPSE+ debería ser utilizado junto con otras soluciones de seguridad para Java, como pueden ser los “frameworks” dedicados a ofrecer los servicios de seguridad de autenticación, confidencialidad, autorización, etc. Además una buena aplicación debería incluir la seguridad en la fase de mantenimiento, de forma que todas las posibles amenazas que surgen a lo largo de la vida del software, puedan ser remediadas sin tener que rehacer toda la aplicación. LAPSE+ permite realizar esta fase del mantenimiento de seguridad con poco esfuerzo.

En resumen, la realización de este proyecto ha sido muy interesante. He podido investigar sobre las últimas tendencias en desarrollos seguros, como también ver las amenazas de seguridad más recientes en el mundo web. Me gustaría tener la oportunidad de profundizar más sobre este tema en un futuro.

9 BIBLIOGRAFÍA

[1] Amsden, J. (2003). *"Your first plug-in" - http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlug-in.html*.

[2] Apache. (s.f.). *http://ant.apache.org/*.

[3] Apache. (s.f.). *http://xml.apache.org/xalan-j/*.

[4] Apache. (s.f.). *http://xml.apache.org/xindice/*.

[5] Benjamin Livshits, M. S. (2004). Finding Security Vulnerabilities in Java Applications with Static Analysis.

[6] Community, X. (2003). *http://xmldb-org.sourceforge.net/*.

[7] Forum, S. E. (s.f.). *http://secure-enterprise20.org/node/38*.

[8] Foundation, E. (s.f.). *http://www.eclipse.org/*.

[9] InternetNews. (2009). *http://www.internetnews.com/security/article.php/3819726/UC+Berkeley+Says+Hackers+Stole+Data+in+2008.htm*.

[10] Java-Source. (s.f.). *http://java-source.net/*.

[11] John Whaley, M. S. (2004). Cloning Based Context Sensitive Pointer Alias Analysis Using Binary Decision Diagrams.

[12] Klein, A. (2004). Blind XPath Injection - http://www.packetstormsecurity.org/papers/bypass/Blind_XPath_Injection_20040518.pdf.

[13] Kolsek, M. (2007). Session Fixation Vulnerability in Web-based Applications.

[14] Livshits, B. (2004). Findings Security Errors in Java Applications Using Lightweight Static Analysis.

[15] Marco Pistoia, N. N. (2004). *Enterprise Java™ Security: Building Secure J2EE™ Applications.*

[16] Margues, M. (2005). *"Exploring Eclipse's AST parser" - http://www.ibm.com/developerworks/opensource/library/os-ast/*.

[17] McGraw, G. (s.f.). *Securing Java.* Obtenido de www.securingjava.com.

[18] McGraw, G. (2004). Static Analysis for Security.

[19] Metsker, S. J., & Wake, W. C. Design Patterns in Java. En S. J. Metsker, & W. C. Wake.

[20] Michael Martin, B. L. (2005). Finding Application Errors and Security Flaws Using PQL: a Program Query Language.

[21] Monica S Lam, J. W. (2005). Context Sensitive Program Analysis as Database Queries.

[22] Monica S. Lam, B. L. (s.f.). Securing Web Applications with Static and Dynamic Information Flow Tracking.

[23] OWASP. (s.f.). OWASP Code Review Guide.

[24] OWASP. (2007). OWASP TOP 10: THE TEN MOST CRITICAL WEB APPLICATION SECURITY VULNERABILITIES 2007 UPDATE.

[25] OWASP. (2004). OWASP TOP 10: THE TEN MOST CRITICAL WEB APPLICATION VULNERABILITIES.

[26] OWASP. (2007). THE TEN MOST CRITICAL WEB APPLICATION SECURITY VULNERABILITIES FOR JAVA ENTERPRISE APPLICATIONS.

[27] Richardson, R. (2008). CSI Computer Crime & Security.

[28] Sen, R. (2007). Avoid the dangers of XPath injection - http://www.ibm.com/developerworks/xml/library/x-xpathinjection.html#resources.

[29] SourceForge. (s.f.). *http://sourceforge.net/*.

[30] SUIF. (s.f.). *http://suif.stanford.edu/~livshits/work/griffin/*.

[31] SUIF. (s.f.). *http://suif.stanford.edu/~livshits/work/lapse/*.

[32] Thomas Kuhn, E. M. (2006). *Eclipse Corne Article - Abstract Syntax Tree*.

[33] Vogel, L. (2009). *Eclipse JDT - http://www.vogella.de/articles/EclipseJDT/index.html*.

[34] w3. (s.f.). *http://www.w3.org/TR/xpath*.

[35] WASC. (s.f.). XML Injection - http://projects.webappsec.org/XML-Injection.

[36] WebGoat_Project-OWASP. (s.f.). *http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project*.

[37] Whaley, J. (s.f.). *http://bddbddb.sourceforge.net/*.

ANEXO A: PRESUPUESTO

En este anexo se especifica un presupuesto detallado de los costes de desarrollo del proyecto realizado. Para ello se tendrán en cuenta las tareas en las que se ha dividido el proyecto, su planificación temporal, el análisis de los recursos utilizados durante el mismo y, finalmente, se evaluarán los costes totales.

A.1. Recursos

En esta sección se especifican todos los recursos utilizados a lo largo del proyecto y los costes implicados.

A.1.1 Recursos Materiales

El principal recurso material con el que se ha realizado este proyecto consiste en un ordenador tipo PC, con las siguientes características:

- **Procesador:** Intel Core 2 Quad 2100.
- **Memoria:** 3 GB
- **Capacidad de disco duro:** 200 GB
- **Interfaz de red:** Ethernet.
- **Coste:** 800€.

El ordenador utilizado tiene una obsolescencia de unos 3 años. Puesto que el proyecto se ha realizado a lo largo de 6 meses y medio, los costes totales ascienden a 250€.

A.1.1.1. Recursos Software

El software y el coste de las licencias utilizadas para la realización de este proyecto son:

Nombre del Software	Coste
Windows XP Home Edition	150€
Microsoft Office 2007	500€
Eclipse	0€
LAPSE	0€
TOTAL:	**650€**

Tabla 13: Software utilizado para realización del proyecto

Si consideramos una vida útil al software de 4 años los costes totales correspondientes al proyecto son de 169 €.

A.1.1.2. Recursos Humanos

Se han estimado los siguientes costes horarios para un Ingeniero Informático:

- Coste de ejecución por hora normal: **10 €.**
- Coste de ejecución por hora extra: **15 €.**

A.2. Resumen de Costes del Proyecto

En las siguientes tablas se resumen los costes debidos a las tareas realizadas en el proyecto (se ha supuesto una dedicación diaria de 8 horas).

SUMARIO	Hombre * día	Importe €
Estudio inicial del marco teórico	1 *40	3.200
Estudio del código de LAPSE 2.5.6	1 *15	1.200
Elaboración de una nueva versión de LAPSE: LAPSE+	1 * 21	1.680
Estudio sobre las vulnerabilidades recientes en Web 2.0	1 * 21	1.680
Descripción e Inclusión de nuevas vulnerabilidades	1 * 21	1.680
Pruebas de comparación de LAPSE con LAPSE+	1 * 15	1.200
Redactar el documento explicativo sobre el proyecto	1 * 21	1.680
TOTAL	**1 * 154**	**12.320**

Tabla 14: Recursos Humanos utilizados en este proyecto

A continuación se añade a los costes de ejecución anteriores el resto de costes involucrados en el proyecto.

Concepto	Importe €
Software	169
Hardware	250
Recursos Humanos	12.320
12% Beneficio	1.528,68
15% Riesgo	1.848
Total sin IVA	16.115,68
16% IVA	2.578,51
Total con IVA	**18.694,20**

Tabla 15: Coste total de este proyecto

El coste total del proyecto asciende a:

DIECIOCHO MIL SEISCIENTOS NOVENTA Y CUATRO EUROS CON VEINTE CÉNTIMOS

ANEXO B: MANUAL DE USUARIO LAPSE+

B.1. Instalación de LAPSE+

LAPSE+ es un plug-in de Eclipse. El primer paso consta en instalar Eclipse, versión 3.2.x. Eclipse es un software libre y puede descargarse de la siguiente dirección web: http://archive.eclipse.org/eclipse/downloads/index.php

Una vez instalado Eclipse, hay que copiar la carpeta LAPSE+ al directorio *$HOME_ECLIPSE/eclipse/plug-ins*. La variable *$HOME_ECLIPSE* indica la ruta de instalación de eclipse. Es necesario asegurarse de que solamente hay una versión de LAPSE en este directorio. Después es necesario reiniciar la aplicación Eclipse.

B.2. Uso de LAPSE+

LAPSE+ consta de tres vistas eclipse. Cada una funciona independientemente de la otra, por tanto se pueden abrir de forma independiente. Para abrir una vista LAPSE+, se selecciona en el menú principal de Eclipse *Window* -> *Show View*, tal como se muestra en Ilustración 41. Para escoger una vista concreta de LAPSE+, hay que seleccionarla en el cuadro de dialogo mostrado en la Ilustración 42.

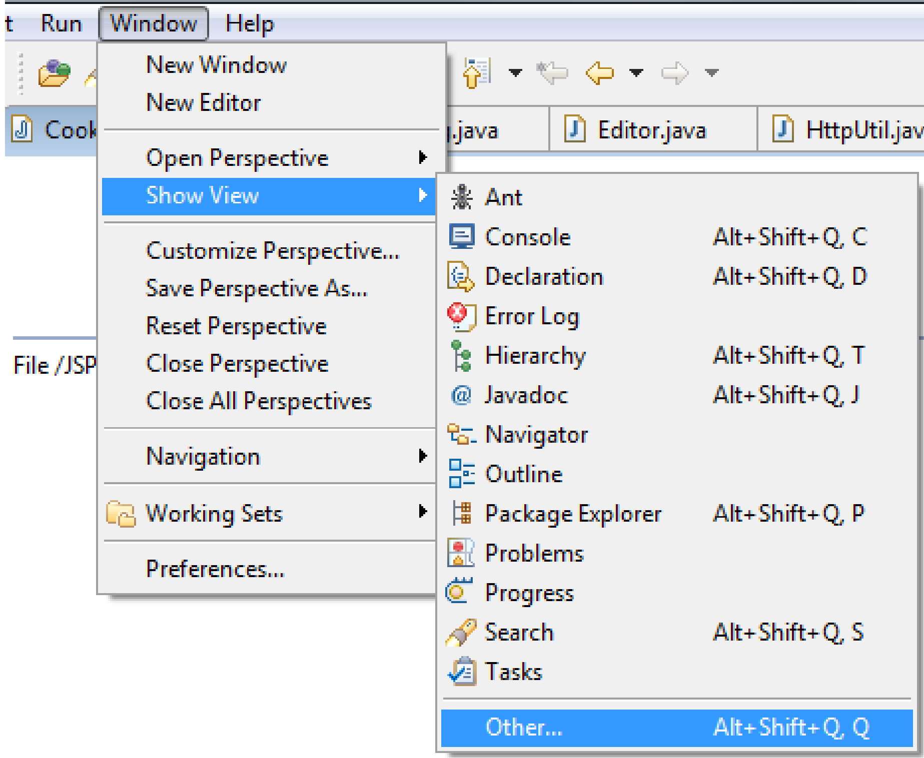


Ilustración 41: Pasos para abrir LAPSE+

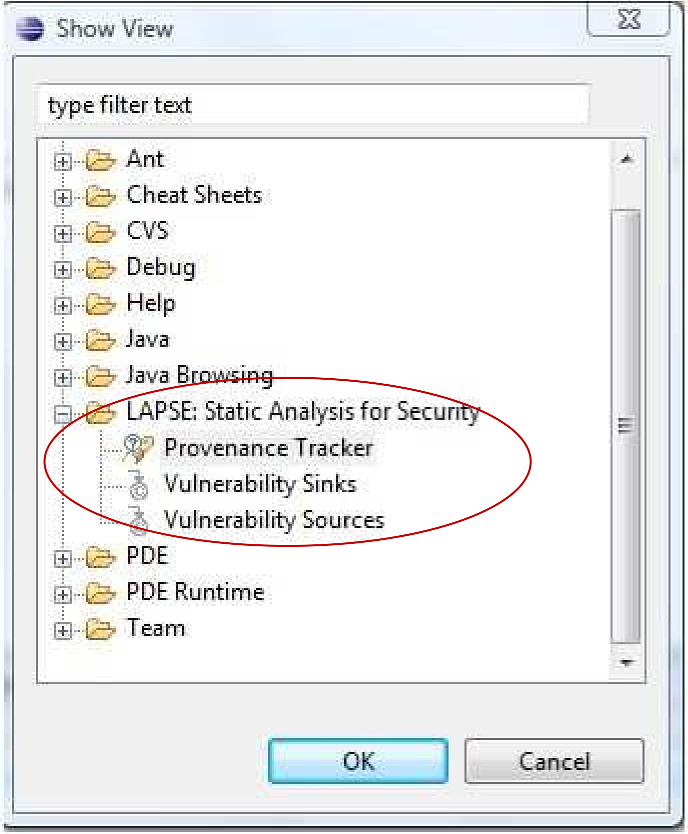


Ilustración 42: Cuadro de dialogo para abrir vistas LAPSE+

Una vez abiertas, las vistas aparecen en la parte de vistas del editor Eclipse, como se muestra en la Ilustración 43.

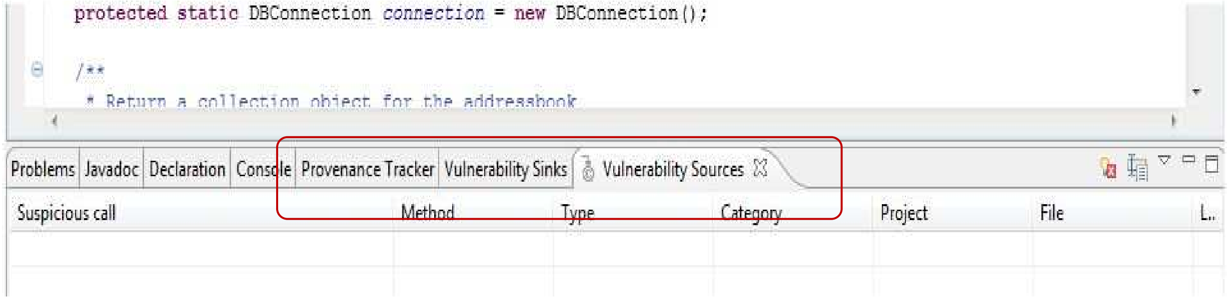


Ilustración 43: Vistas LAPSE+ abiertas en eclipse

B.1.1 Uso de la vista “Vulnerabilities Sources”

Para poder analizar el programa en busca de invocaciones de métodos tipo “source” hace falta abrir la vista “SourceView”. Una vez abierta la vista, pulsar en el icono marcado en el círculo rojo de la Ilustración 44 para empezar el escaneo del código.

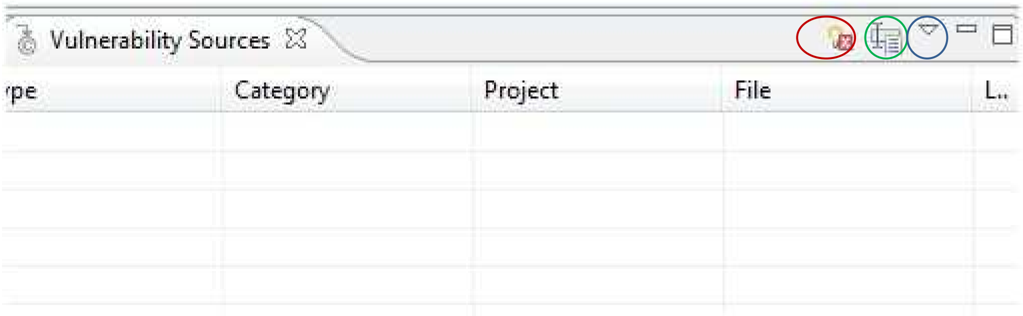


Ilustración 44: Vista "Vulnerability Source"

Se puede filtrar los resultados para ver los métodos no web, como *main* o métodos *set* de Ant, desmarcando la opción “Hide non-Web sources” en el menú desplegable mostrado en la Ilustración 45. También se pueden ver los métodos sin código fuente, como los que aparecen en las librerías o ficheros binarios. Para ello hay que desmarcar la opción “Hide vulnerability sources with no source code”. Para desplegar el menú hay que pulsar el icono marcado con el círculo azul en la Ilustración 44. El icono marcado con el círculo verde en la misma ilustración, sirve para copiar al portapapeles los métodos “source” seleccionados.

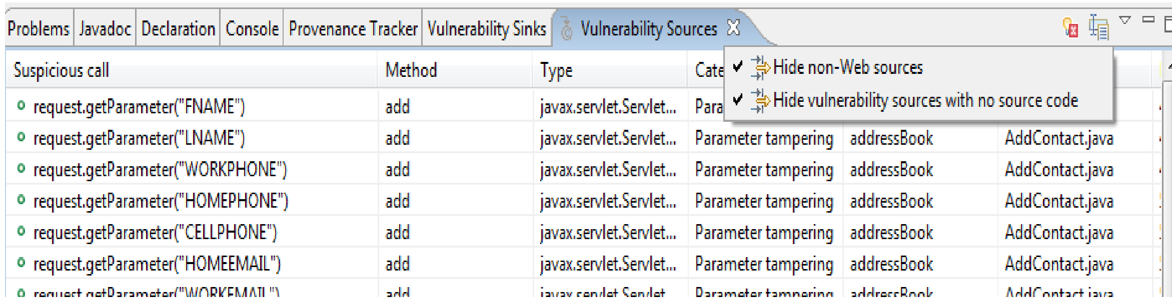


Ilustración 45: Menú principal de la vista " Vulnerability Sources"

Para ver dónde aparece el método en el código fuente, hay que hacer “doble click” sobre el resultado seleccionado y se abrirá el código, con la línea correspondiente marcada.

B.2.1 Uso de la Vista "Vulnerability Sinks"

Para poder analizar el programa en busca de invocaciones de métodos tipo "sinks" hace falta abrir la vista "Vulnerabilities Sinks". Una vez abierta la vista, pulsar el icono marcado en el circulo rojo de la Ilustración 46 para empezar el escaneo del código.

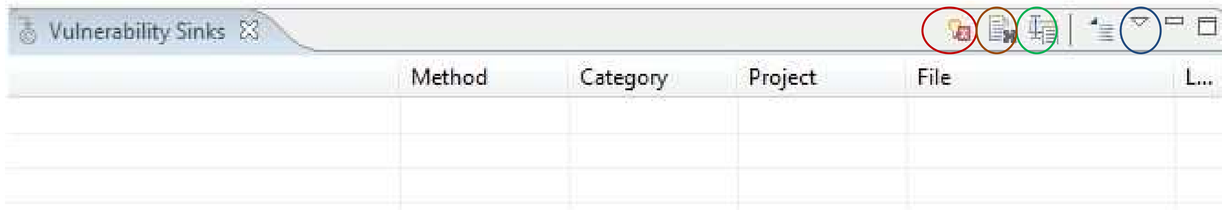


Ilustración 46: Vista "Vulnerability Sinks"

Se pueden filtrar los resultados para ver los métodos que tengan una cadena constante como parámetro y por tanto se consideren seguros, desmarcando la opción de "Hide safe vulnerability sinks" en el menú desplegable mostrado en la Ilustración 47.También se pueden ver los métodos sin código fuente, como los que aparecen en las librerías o ficheros binarios. Para ello hay que desmarcar la opción "Hide vulnerability sinks with no source code". Las demás opciones del menú sirven para filtrar el tipo de la vulnerabilidad que se quiere ver. Para desplegar el menú hay que hacer pulsar el icono marcado con el círculo azul en la Ilustración 46. El icono marcado con el círculo verde en la misma ilustración, sirve para copiar al portapapeles los métodos "source" seleccionados. El icono marcado con el círculo marrón, sirve para cambiar el estado de un objeto "sink" determinado por LAPSE+ (de seguro a no seguro y viceversa).

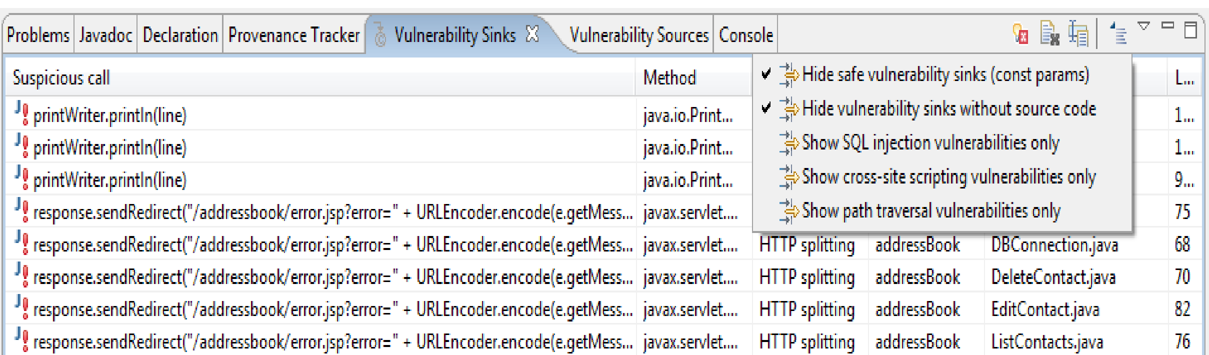


Ilustración 47: Menú principal de la vista " Vulnerability Sinks"

La vista “Vulnerabilities Sinks” permite también mostrar las estadísticas sobre el análisis realizado. Para obtener estas estadísticas es necesario pulsar el icono que aparece en el círculo rojo en la Ilustración 48. En esta ilustración aparece también la ventana de estadísticas que se abre al pulsar el botón.

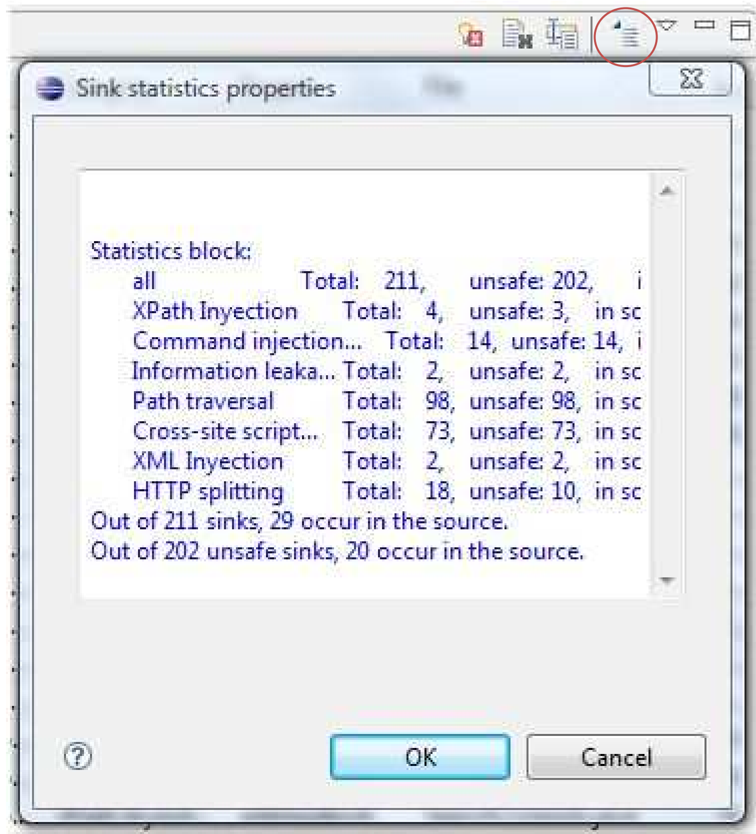


Ilustración 48: Estadísticas sobre objetos "sinks"

Desde la vista “Vulnerabilites Sinks”, se puede iniciar la propagación para ver si se llega a algún objeto definido como “source”. Esta propagación se inicia escogiendo la opción “Perform backward propagation from sink” desde el menú contextual que se abre pulsando el botón izquierdo del ratón sobre el resultado desde el que se quiere propagar (veáse Ilustración 49). Antes de escoger esta opción, es necesario abrir la vista “Provenance Tracker”.

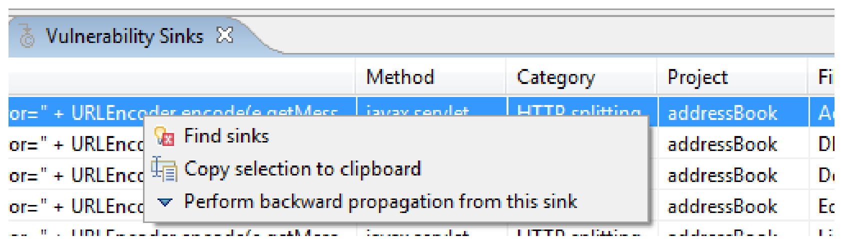


Ilustración 49: Menú contextual de la vista "Vulnerability Sinks"

B.3.1 Uso de la vista "Provenance Tracker"

La vista "Provenance Tracker" muestra la propagación hacia atrás desde un objeto "sink". Se puede inicializar esta propagación de dos maneras. La primera es inicializarla desde la vista "Vulnerabilities Sinks" desde un resultado seleccionado (véase sección anterior). La otra opción es pulsando el icono marcado con el círculo rojo en la Ilustración 50, en cuyo caso la propagación se realiza desde el objeto "sink" seleccionado en ese momento en el editor de código de Eclipse

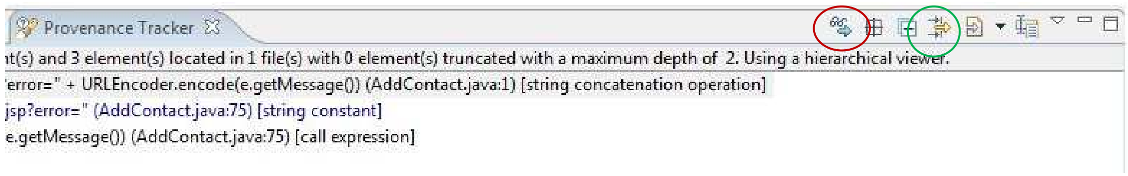


Ilustración 50: Vista "Provenance Tracker"

La vista "Provenance Tracker" permite al usuario personalizar algunas de sus propiedades. La ventana que permite configurar los diferentes parámetros es la que se muestra en la Ilustración 51. Para acceder a ella es necesario pulsar el botón destacado con el círculo verde en la Ilustración 50.

S puede establecer el límite de propagación entre expresiones, o bien optar por no poner ningún límite (este límite por defecto es 10). Para cambiarlo hace falta escribir un nuevo valor entre 1 y 99 en el segundo recuadro ("Slicer Setings").

Por otro lado se puede decidir si se quiere proceder la propagación dentro de métodos o a través de las expresiones de retorno, así como el límite de llamadas para esta propagación, siendo este último 10 por defecto. Para cambiarlo hace falta escribir un nuevo valor entre 1 y 99 en el tercer recuadro ("Traversal trough return results").

Finalmente se permite filtrar los resultados de la propagación, incluyendo o no el uso inicial de la variable, los argumentos de llamada, los parámetros formales y las declaraciones de llamada. Todas estas opciones se pueden cambiar a través del primer recuadro ("Filter Settings").

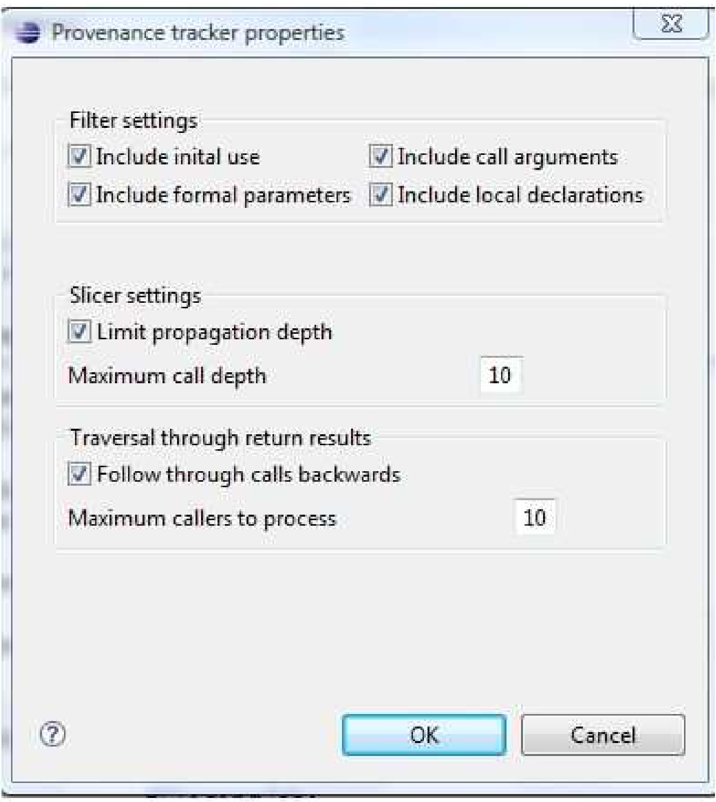


Ilustración 51: La ventana de propiedades modificable de la vista "Provenance Tracker"

Además de mostrar la ventana de propiedades, "Provenance Tracker" permite mostrar el historial de todas las propagaciones realizadas. Este hstorial guarda los nombres de todos los objetos "sink" desde los que se ha realizado la propagación. Para acceder a él hace falta pulsar el icono marcado con el círculo rojo en la Ilustración 52.

El menu contextual que se muestra en la Ilustración 52 permite seleccionar todas las ramas del árbol y copiar la selección al portapapeles.

El segundo y tercer icono que aparece en la barra de tareas sirven para expandir y comprimir la vista de árbol formada por los elementos de propagación respectivamente. Finalmente el último icono que aparece en la barra de tareas permite copiar la selección al portapapeles.

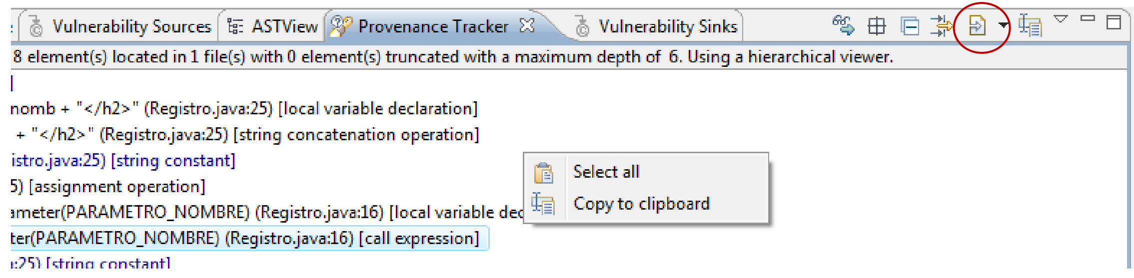


Ilustración 52: Menú Contextual de la vista "Provenance Tracker"

Para cambiar el aspecto del árbol con soluciones de propagación, es necesario escoger una opción del menú principal. Para acceder a este menú hay que pulsar el icono en el círculo rojo, en la Ilustración 53. Se puede escoger la vista jerárquica de todo el árbol o la vista que muestra solamente hojas.

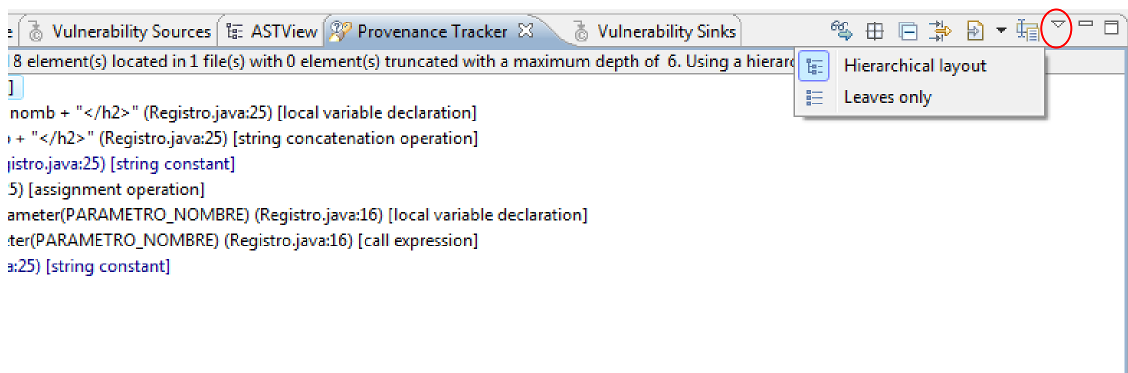


Ilustración 53: Menú Principal de la vista “Provenance Tracker”

Como en las otras vistas, la acción de “doble click”, permite abrir el resultado de propagación seleccionado en la vista en el editor Eclipse tal como aparece en el código fuente.

GLOSARIO DE TERMINOS

Termino	Definición
Ant	Ant es un herramienta Java para la realización de tareas de construcción (“build”), ejecutadas en las fases de compilación. Ant es un software de Apache.
API	“Application Programming Interface” - es una interfaz a través de la cual una aplicación puede solicitar servicios de librerías o sistemas operativos.
Applet	Applet es un pequeño programa escrito en Java que se incluye en las páginas HTML.
DOM	“Document Object Model” – es una convención independiente del lenguaje y plataforma, utilizada para representar e interactuar con los objetos de los lenguajes XML, HTML y XHTML.
Eclipse	Eclipse es una comunidad de código abierto, cuyos proyectos se centran en construir una plataforma de desarrollo abierta, compuesta de “frameworks “ extensibles y herramientas para la construcción, el despliegue y el manejo de software a lo largo de su ciclo de vida.
Framework	El Framework de software es un diseño reutilizable para un sistema de software, que puede incluir librerías de código, programa de soporte o lenguaje de scripts para ayudar en el desarrollo de los diferentes componentes de un sistema software.
HTML	“Hyper Text Markup Language” – es un lenguaje de mercado utilizado para la construcción de páginas web.
HTTP	“Hypertext Transfer Protocol” es un protocolo de red de nivel de aplicación, utilizado para transacciones de la Web.
IDE	“Integrated developmnet enviornment” es una aplicación software que provee facilidades a los desarrolladores de software.
J2EE	“Java 2 Enterprise Edition” es una plataforma para la programación del servidor en lenguaje Java.
Java Script	Es un lenguaje de script, orientado a objetos, utilizado principalmente en páginas web para construir el contenido dinámico y una interfaz de usuario mejorada.
JVM	“Java Virtual Machine” es un conjunto de programas software y estructuras de datos que utilizan el modelo de máquina virtual para la ejecución de otros programa o scripts. El modelo utilizado por JVM acepta un lenguaje

	intermedio llamado Java Bytecode.
Kerberos	Es un protocolo de autenticación en las redes de ordenadores que permite a los nodos que se comunican sobre una red no segura demostrar su identidad unos a otros de forma segura.
LAPSE	“Lightweight Analysis for Program Security” es una herramienta diseñada para ayudar a auditar las aplicaciones J2EE para encontrar vulnerabilidades de seguridad en las aplicaciones web.
MD5	“Message-Digest Algorithm 5” es un algoritmo de resumen criptográfico, o de función hash, ámpliamente utilizado. MD-5 utiliza el valor de hash de 128 bits.
PKCS	“Public-Key Cryptography Standards” son especificaciones producidas por RSA con el fin de acelerar el despliegue de criptografía de clave pública.
PKI	“Public Key Infraestructure” es una infraestructura de clave pública que contiene hardware, software, gente, políticas y procedimientos necesarios para crear, manejar, guardar, distribuir y revocar los certificados digitales.
Plug-in	Es una programa que interactúa con otra aplicación para proveer alguna función específica.
RC-4	Es el mecanismo de cifrado de flujo, muy utilizado, sobre todo en los protocolos SSL y WEP.
Rootkit	Es un sistema software que consiste en una combinación de varios programas diseñados para ocultar el hecho de que el sistema ha sido atacado.
S/MIME	“Secure / Multipurpose Internet Mail Extensions” es un estándar para el cifrado de clave pública y de firmado de correos encapsulados en MIME.
RSA	Es un algoritmo criptográfico de clave pública, descrito por Rivest, Shamir y Adleman y frecuentemente utilizado en comercio electrónico.
Servlet	Los Servlets son objetos del lenguaje de programación Java que procesan peticiones y construyen respuestas de forma dinámica.
SHA-1	“Secure Hash Algorithm 1” es una función de resumen criptográfico, función hash de 160 bits.
Socket	Un socket en redes se define como un extremo de comunicación, entre un cliente y un servidor.
SQL	“Structured Quey Lenguage” es un lenguaje de consultas, diseñado para manejar datos en un sistema gestor de bases de datos relacionales.
SSL	“Secure Socket Layer” es un protocolo criptográfico para proveer

	conexiones seguras sobre redes como Internet.
Token	Es un dispositivo electrónico de seguridad, que se entrega al usuario autorizado para permitirle realizar el proceso de autenticación.
UNIX	Es un sistema operativo multitarea, desarrollado en el año 1991, por los investigadores del laboratorio Bell de AT&T.
XML	"Extensible Markup Language" es un lenguaje extensible de etiquetas desarrollado por *World Wide Web Consortium* W3C, que se propone como estándar para el intercambio de información entre plataformas heterogéneas.
XPath	Es un lenguaje diseñado para referirse a las partes de mensajes XML.

Tabla 16: Glosario de términos