Universidad Internacional de La Rioja (UNIR)

ESIT

Máster universitario en Seguridad Informática

Análisis de fiabilidad de las herramientas SAST. Privativo vs Libre

Trabajo Fin de Máster

presentado por: Pérez García, Javier

Director/a: Rodríguez Gómez, Rafael Alejandro

Ciudad: Madrid
Fecha: 19/09/2019

Índice de contenidos

Resumen....10
Abstract....10
Capítulo I: Introducción....11
Introducción a las herramientas SAST....11
Evolución de las herramientas SAST....12
Problemáticas de SAST en la actualidad....13
Estructura de la memoria....14
Capítulo II: Estado del arte....16
Descripción de SAST....16
Situación de SAST en el mercado....17
Estado del arte de las soluciones propuestas....18
Estado del arte de herramientas de estudio SAST....22
Estudios relacionados con la fiabilidad SAST....23
Capítulo III: Objetivos concretos y metodología....26
Objetivos específicos....26
Metodología....26
Capítulo IV: Desarrollo del piloto....28
Descripción del experimento....28
Capítulo V: Presentación de resultados....42
Fortify....42
Lenguaje analizado: C++....42
Lenguaje analizado: Java....48
Lenguaje analizado: PHP....55
SonarQube....61
Lenguaje analizado: C++....61
Lenguaje analizado: Java....63
Lenguaje analizado: PHP....68
VisualCodeGreeper....72

Lenguaje analizado: C++ 72
Lenguaje analizado: Java....78
Lenguaje analizado: PHP....84
Capítulo VI: Discusión de resultados....91
Capítulo VII: Conclusiones y líneas de trabajo futuro 94
Referencias bibliográficas 96
Anexo A: Listados de ficheros por lenguaje....99
A.1 C++ 99
A.1.1 CWE-15: External Control of System or Configuration Setting 99
A.1.2 CWE-23: Relative Path Traversal....99
A.1.3 CWE-259: Use of Hard-coded Password 99
A.1.4 CWE-369: Divide By Zero 99
A.1.5 CWE-404: Improper Resource Shutdown or Release 99
A.2 Java 100
A.2.1 CWE-15: External Control of System or Configuration Setting 100
A.2.2 CWE-23: Relative Path Traversal....100
A.2.3 CWE-259: Use of Hard-coded Password 100
A.2.4 CWE-369: Divide By Zero 100
A.2.5 CWE-404: Improper Resource Shutdown or Release 100
A.3 PHP....100
A.3.1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 100
A.3.2 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')....101
A.3.3 CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')....101
A.3.4 CWE-91: XML Injection (aka Blind XPath Injection) 101
A.3.5 CWE-601: URL Redirection to Untrusted Site ('Open Redirect')....101

Índice de tablas

Tabla 1 - Disposición de máquinas....29
Tabla 2 - Medidas obtenidas Fortify C++ CWE15....43
Tabla 3 - Medidas obtenidas Fortify C++ CWE23....44
Tabla 4 - Medidas obtenidas Fortify C++ CWE259....46
Tabla 5 - Medidas obtenidas Fortify C++ CWE369....47
Tabla 6 - Medidas obtenidas Fortify C++ CWE404....48
Tabla 7 - Medidas obtenidas Fortify Java CWE15....49
Tabla 8 - Medidas obtenidas Fortify Java CWE23....51
Tabla 9 - Medidas obtenidas Fortify Java CWE259....52
Tabla 10 - Medidas obtenidas Fortify Java CWE369....54
Tabla 11 - Medidas obtenidas Fortify Java CWE404....55
Tabla 12 - Medidas obtenidas Fortify PHP CWE79....56
Tabla 13 - Medidas obtenidas Fortify PHP CWE89....57
Tabla 14 - Medidas obtenidas Fortify PHP CWE90....59
Tabla 15 - Medidas obtenidas Fortify PHP CWE91....60
Tabla 16 - Medidas obtenidas Fortify PHP CWE601....61
Tabla 17 - Medidas obtenidas SonarQube Java CWE15....65
Tabla 18 - Medidas obtenidas SonarQube Java CWE23....66
Tabla 19 - Medidas obtenidas SonarQube Java CWE259....67
Tabla 20 - Medidas obtenidas SonarQube Java CWE369....68
Tabla 21 - Medidas obtenidas SonarQube PHP CWE90....71
Tabla 22 - Medidas obtenidas VisualCodeGreeper C++ CWE15....73
Tabla 23 - Medidas obtenidas VisualCodeGreeper C++ CWE23....74
Tabla 24 - Medidas obtenidas VisualCodeGreeper C++ CWE259....75
Tabla 25 - Medidas obtenidas VisualCodeGreeper C++ CWE369....76
Tabla 26 - Medidas obtenidas VisualCodeGreeper C++ CWE404....78
Tabla 27 - Medidas obtenidas VisualCodeGreeper Java CWE15....79
Tabla 28 - Medidas obtenidas VisualCodeGreeper Java CWE23....80
Tabla 29 - Medidas obtenidas VisualCodeGreeper Java CWE259....81
Tabla 30 - Medidas obtenidas VisualCodeGreeper Java CWE369....83
Tabla 31 - Medidas obtenidas VisualCodeGreeper Java CWE404....84
Tabla 32 - Medidas obtenidas VisualCodeGreeper PHP CWE79....85
Tabla 33 - Medidas obtenidas VisualCodeGreeper PHP CWE89....86
Tabla 34 - Medidas obtenidas VisualCodeGreeper PHP CWE90....88
Tabla 35 - Medidas obtenidas VisualCodeGreeper PHP CWE91....89
Tabla 36 - Medidas obtenidas VisualCodeGreeper PHP CWE601....90

Índice de Figuras

Figura 1 - Magic Quadrant de Gartner (abril 2019) 18
Figura 2 - Logo Fortify 19
Figura 3 - Arquitecturas Fortify 20
Figura 4 - Logo SonarQube 20
Figura 5 - Logo VisualCodeGreeper 21
Figura 6 - Interfaz VCG 22
Figura 7 - Logo de SAMATE 22
Figura 8 - Métricas de Fawcett, 2006 24
Figura 9 – Metodología 26
Figura 10 - Arquitectura del experimento 30
Figura 11 - Resultados Fortify 36
Figura 12 - Interfaz de resultados de SonarQube 38
Figura 13 - VCG análisis (I) 38
Figura 14 - VCG análisis (II) 39
Figura 15 - VCG análisis (III) 39
Figura 16 - VCG análisis (IV) 40
Figura 17 - VCG análisis (V) 40
Figura 18 - VCG análisis (VI) 41
Figura 19 - Ficheros analizados Fortify C++ CWE15 42
Figura 20 - CWEs por criticidad Fortify C++ CWE15 43
Figura 21 - Ficheros analizados Fortify C++ CWE23 44
Figura 22 - CWEs por criticidad Fortify C++ CWE23 44
Figura 23 - Ficheros analizados Fortify C++ CWE259 45
Figura 24 - CWEs por criticidad Fortify C++ CWE259 45
Figura 25 - Ficheros analizados Fortify C++ CWE369 46
Figura 26 - CWEs por criticidad Fortify C++ CWE369 47
Figura 27 - Ficheros analizados Fortify C++ CWE404 47
Figura 28 - CWEs por criticidad Fortify C++ CWE404 48
Figura 29 - Ficheros analizados Fortify Java CWE15 49
Figura 30 - CWEs por criticidad Fortify Java CWE15 49
Figura 31 - Ficheros analizados Fortify Java CWE23 50
Figura 32 - Ficheros analizados Fortify Java CWE23 51
Figura 33 - Ficheros analizados Fortify Java CWE259 51
Figura 34 - CWEs por criticidad Fortify Java CWE259 52
Figura 35 - Ficheros analizados Fortify Java CWE369 53
Figura 36 - CWEs por criticidad Fortify Java CWE369 53

Figura 37 - Ficheros analizados Fortify Java CWE404 54
Figura 38 - CWEs por criticidad Fortify Java CWE404 55
Figura 39 - Ficheros analizados Fortify PHP CWE79 55
Figura 40 - CWEs por criticidad Fortify PHP CWE79 56
Figura 41 - Ficheros analizados Fortify PHP CWE89 57
Figura 42 - CWEs por criticidad Fortify PHP CWE89 57
Figura 43 - Ficheros analizados Fortify PHP CWE90 58
Figura 44 - CWEs por criticidad Fortify PHP CWE90 58
Figura 45 - Ficheros analizados Fortify PHP CWE91 59
Figura 46 - CWEs por criticidad Fortify PHP CWE91 60
Figura 47 - Ficheros analizados Fortify PHP CWE601 60
Figura 48 - CWEs por criticidad Fortify PHP CWE601 61
Figura 49 - Ficheros analizados SonarQube C++ CWE15 62
Figura 50 - Ficheros analizados SonarQube C++ CWE23 62
Figura 51 - Ficheros analizados SonarQube C++ CWE259 62
Figura 52 - Ficheros analizados SonarQube C++ CWE369 63
Figura 53 - Ficheros analizados SonarQube C++ CWE404 63
Figura 54 - Ficheros analizados SonarQube Java CWE15 64
Figura 55 - CWEs por criticidad SonarQube Java CWE15 64
Figura 56 - Ficheros analizados SonarQube Java CWE23 65
Figura 57 - CWEs por criticidad SonarQube Java CWE23 65
Figura 58 - Ficheros analizados SonarQube Java CWE259 66
Figura 59 - CWEs por criticidad SonarQube Java CWE259 67
Figura 60 - Ficheros analizados SonarQube Java CWE369 67
Figura 61 - CWEs por criticidad SonarQube Java CWE369 68
Figura 62 - Ficheros analizados SonarQube Java CWE404 68
Figura 63 - Ficheros analizados SonarQube PHP CWE79 69
Figura 64 - Ficheros analizados SonarQube PHP CWE89 69
Figura 65 - Ficheros analizados SonarQube PHP CWE90 70
Figura 66 - CWEs por criticidad SonarQube PHP CWE90 70
Figura 67 - Ficheros analizados SonarQube PHP CWE91 71
Figura 68 - Ficheros analizados SonarQube PHP CWE601 71
Figura 69 - Ficheros analizados VisualCodeGreeper C++ CWE15 72
Figura 70 - CWEs por criticidad VisualCodeGreeper C++ CWE15 73
Figura 71 - Ficheros analizados VisualCodeGreeper C++ CWE23 73
Figura 72 - CWEs por criticidad VisualCodeGreeper C++ CWE23 74
Figura 73 - Ficheros analizados VisualCodeGreeper C++ CWE259 74

Figura 74 - CWEs por criticidad VisualCodeGreeper C++ CWE259 75
Figura 75 - Ficheros analizados VisualCodeGreeper C++ CWE369 76
Figura 76 - CWEs por criticidad VisualCodeGreeper C++ CWE369 76
Figura 77 - Ficheros analizados VisualCodeGreeper C++ CWE404 77
Figura 78 - CWEs por criticidad VisualCodeGreeper C++ CWE404 77
Figura 79 - Ficheros analizados VisualCodeGreeper Java CWE15 78
Figura 80 - CWEs por criticidad VisualCodeGreeper Java CWE15 79
Figura 81 - Ficheros analizados VisualCodeGreeper Java CWE23 79
Figura 82 - CWEs por criticidad VisualCodeGreeper Java CWE233 80
Figura 83 - Ficheros analizados VisualCodeGreeper Java CWE259 81
Figura 84 - CWEs por criticidad VisualCodeGreeper Java CWE259 81
Figura 85 - Ficheros analizados VisualCodeGreeper Java CWE369 82
Figura 86 - CWEs por criticidad VisualCodeGreeper Java CWE369 82
Figura 87 - Ficheros analizados VisualCodeGreeper Java CWE404 83
Figura 88 - CWEs por criticidad VisualCodeGreeper Java CWE404 83
Figura 89 - Ficheros analizados VisualCodeGreeper PHP CWE79 84
Figura 90 - CWEs por criticidad VisualCodeGreeper PHP CWE79 85
Figura 91 - Ficheros analizados VisualCodeGreeper PHP CWE89 85
Figura 92 - CWEs por criticidad VisualCodeGreeper PHP CWE89 86
Figura 93 - Ficheros analizados VisualCodeGreeper PHP CWE90 87
Figura 94 - CWEs por criticidad VisualCodeGreeper PHP CWE90 87
Figura 95 - Ficheros analizados VisualCodeGreeper PHP CWE91 88
Figura 96 - CWEs por criticidad VisualCodeGreeper PHP CWE91 89
Figura 97 - Ficheros analizados VisualCodeGreeper PHP CWE601 89
Figura 98 - CWEs por criticidad VisualCodeGreeper PHP CWE6011 90
Figura 99 – Comparativa de Precisión y Recall medios de herramientas 91
Figura 100 – Comparativa de Precisión media en Java y C++ de herramientas 92
Figura 101 – Comparativa de Recall medio en Java de Fortify vs SonarQube 92
Figura 102 – Comparativa de detección del CWE 93

Resumen

El presente trabajo tiene como objetivo clarificar la fiabilidad de las herramientas *Static Application Security Testing* (SAST) tanto en el mundo del software libre como en del software privativo, comparando ambos. A lo largo del trabajo, se analiza el estado del arte en técnicas SAST, así como se realiza un experimento piloto.

Para la realización del experimento piloto se ha desarrollado un entorno de análisis que permite llevar a cabo los diferentes análisis de forma metódica, repetible y eficaz. Se han estudiado un total de tres herramientas (Fortify, SonarQube y VisualCodeGreeper) con las que se han analizado tres lenguajes de programación (C++, Java y PHP) en busca de cinco CWE concretos, con muestras provenientes de repositorios de código de SAMATE.

Mediante estos análisis se ha podido concluir que las herramientas SAST aún no son del todo fiables y que el mundo del software privativo se encuentra a la cabeza.

Palabras Clave: Análisis estático, Software Libre, Software Privativo, Fiabilidad

Abstract

The present work aims to clarify the reliability of the Static Application Security Testing (SAST) tools both in the world of free software and in proprietary software, comparing both. Throughout the work, the state of the art in SAST techniques is analyzed, as well as a pilot experiment is carried out.

In order to carry out the pilot experiment, an analysis environment has been developed that allows the different analyses to be carried out in a methodical, repeatable and effective way. A total of three tools have been studied (Fortify, SonarQube and VisualCodeGreeper) with which three programming languages (C++, Java and PHP) have been analysed in search of five specific CWE, with samples coming from SAMATE code repositories.

Through these analyses it has been possible to conclude that SAST tools are still not entirely reliable and that the world of proprietary software is at the forefront.

Keywords: Static analysis, Opensource Software, Private Software, Reliability

Capítulo I: Introducción

El presente trabajo de investigación busca como objetivo determinar la fiabilidad de las herramientas pertenecientes al conjunto *Static Application Security Testing* (SAST), que utilizan en el día a día los profesionales de la ciberseguridad para analizar el software desarrollado durante un ciclo de vida de desarrollo seguro (S-SDLC).

Para ello, se estudiará el análisis SAST y las soluciones disponibles en la actualidad, así como se desarrollará un piloto experimental para estudiar dicha fiabilidad.

Introducción a las herramientas SAST

SAST hace referencia a aquellas técnicas de revisión de seguridad de tipo caja blanca aplicadas sobre el código fuente del software. El objetivo de las técnicas SAST es la búsqueda de defectos de seguridad en el código fuente de las aplicaciones, tras la fase de desarrollo del S-SDLC.

Tal y como indica OWASP (traducido al castellano), se puede definir SAST como:

> "El análisis estático de código (también conocido como análisis SAST) se realiza normalmente como parte de una revisión de código (también conocida como prueba de caja blanca) y se lleva a cabo en la fase de implementación de un ciclo de vida de desarrollo de seguridad (SDL). El análisis de código estático se refiere comúnmente a la ejecución de herramientas de análisis de código estático que intentan resaltar posibles vulnerabilidades dentro del código fuente "estático" (no operativo) mediante el uso de técnicas como el análisis de manchas y el análisis de flujo de datos (OWASP, 2019)".

Por tanto, cuando se habla de SAST, lo lógico es pensar en herramientas SAST, ya que las técnicas SAST están embebidas en estas y no se concibe su aplicación de otra forma en un S-SDLC, ya que supondría un desarrollo excesivo.

En la actualidad, se llevan a cabo análisis estáticos mediante el uso de herramientas SAST y expertos en análisis estático de código realizan triajes sobre los resultados obtenidos. Este proceso es bastante común en los análisis de código estático debido a la gran cantidad de falsos positivos que proporcionan este tipo de herramientas.

Además, suele ser necesario que este equipo de expertos sea capaz de transmitir de forma entendible los problemas de seguridad hallados a los desarrolladores, para que estos sean capaces de solventarlos durante los siguientes ciclos del desarrollo.

Evolución de las herramientas SAST

Como indica German (2003),

"*All software contains errors, and computer programs rarely work the first time. Usually, several rounds of rewriting, testing, and modifying are required before a working solution is produced.*"

Dicho de otro modo, todo el software que se desarrolla está sometido a gran variedad de cambios hasta conseguir el resultado que se buscaba. Es debido a esto, entre otras cosas, que el software tiende a desarrollarse sin tener en cuenta la seguridad, lo que ha incrementado el valor de este tipo de soluciones (Deylami, Ardekani, Muniyandi, & Sarrafzadeh, 2015).

Para luchar contra esto surgió el concepto de *Security by Design* cuyo fin es el de integrar la seguridad durante el diseño del *software* aplicando una serie de principios de seguridad a tener en cuenta a la hora de desarrollar aplicaciones. Entre ellos se encuentran algunos como *Principle of Least privilege*, *Principle of Defense in Depth* o *Minimize attack surface area* (OWASP, 2016).

Dentro de las herramientas SAST han surgido gran cantidad de técnicas (OWASP, 2019):

- Análisis de diagramas de flujo: Este tipo de análisis se utiliza para recopilar información relacionada con los datos en tiempo de ejecución mientras se encuentra en un estado estático (Wögerer, 2005).
  Hay tres principales términos utilizados en los análisis de diagramas de flujo, estos son:
  - Bloque básico: lo que sería el código en sí.
  - Análisis del control del flujo: que representa el flujo de datos.
  - Camino del control del flujo: que representa el camino que los datos toman.
- Grafo de control de flujo: Consiste en una representación gráfica y abstracta de los flujos de la aplicación mediante el uso de nodos que representan bloques básicos, mientras que las flechas dirigidas indican los diferentes caminos que puede tomar el dato.
- Análisis *Taint*: Este tipo de análisis se centra en la asunción de que un dato es malicioso por su procedencia (una entrada de datos de usuario, por ejemplo) y analiza el flujo que realiza para determinar si es capaz de vulnerar alguna función del código en el camino. El destino del dato considerado *taint* se denomina *sink* o sumidero (por su traducción al castellano).
  Si el dato *taint* llega a su destino sin pasar por ningún proceso de validación de este, se considera que existe una vulnerabilidad en ese flujo.

- Análisis léxico: El análisis léxico consiste en la transformación de la sintaxis del código fuente a un lenguaje estándar basado en *tokens* con el objetivo de abstraerse del código fuente y poder manipularlo más fácilmente (Sotirov, 2005).

Las herramientas SAST se han basado en este tipo de técnicas para analizar el código fuente de las aplicaciones, combinando algunas de ellas.

Dentro de las investigaciones realizadas no se ha encontrado como tal una línea temporal que establezca la historia de estas técnicas, pero en base a las referencias de diferentes estudios, como podría ser la tesis doctoral de Bermejo Higuera (2014), se puede determinar que, en los años 70, surgieron las primeras herramientas que realizan análisis estático de código.

Estas primeras herramientas se denominan herramientas de tipo *Lint*, las cuales se basan en la búsqueda de vulnerabilidades en el código fuente mediante el uso de expresiones regulares. Hoy en día se utilizan como apoyo al desarrollador, siendo un complemento al análisis estático al detectar aquellas vulnerabilidades más evidentes en fases más tempranas.

En 2003 la empresa Fortify Software (Microfocus S.L., 2019) desarrolló una herramienta para realizar análisis estático de código de forma diferente a las técnicas *Lint*. Siguiendo a esta, en 2006 se fundaron Checkmarx (Checkmarx Ltd., 2019) y Veracode (Veracode, 2019), sus mayores competidores.

Problemáticas de SAST en la actualidad

Hoy en día, no se puede garantizar que exista una herramienta SAST sin fallos, cuyos resultados sean 100% confiables sin necesidad de que se lleve a cabo un proceso de triaje de resultados. El motivo por el que esto ocurre es debido a que el software es un ente vivo, cuyo lenguaje de programación está sometido a cambios y actualizaciones, así como se descubren errores de seguridad en las funciones de este. Lo habitual es que una herramienta SAST no disponga de analizadores de las últimas versiones de todos los lenguajes que cubre, lo cual dificulta el análisis del software desarrollado.

Además de esto, es cada día mayor el número de lenguajes de programación a los que los desarrolladores tienen acceso, permitiendo desarrollos que utilizan diversas tecnologías mezcladas, dificultando así los flujos de análisis de estas herramientas.

Un ejemplo de esto serían las aplicaciones web, las cuales disponen de diversos lenguajes de programación, como pueden ser HTML, CSS y JavaScript, para su componente *front-end* y de diversos *frameworks* de diferentes lenguajes, como RubyOnRails en Ruby o Django en Python, para su componente *back-end*.

Por estos motivos, entre otros, es relevante determinar la fiabilidad de las herramientas de las que se dispone hoy día, ya que, durante los ciclos de vida de desarrollo software, se establecen fechas límite de entrega en los que se debe incluir la seguridad, la cual se ve directamente afectada por estas métricas.

Para resolver este problema, se propone realizar un análisis de fiabilidad basada en el análisis de resultados de códigos con vulnerabilidades conocidas, permitiendo así identificar la eficacia de la herramienta y determinar los esfuerzos requeridos por los analistas de seguridad para realizar triaje de resultados (o incluso análisis manual).

Estructura de la memoria

A continuación, se presenta la estructura de la memoria comentando cada uno de sus componentes:

- **Estado de arte**: Durante este primer capítulo del presente documento, se busca determinar cómo se encuentra el ámbito científico de la ciberseguridad en relación con el campo a estudiar, las herramientas y técnicas SAST.
  Para realizar esta tarea, se va a realizar una introducción al análisis de código y las técnicas existentes, así como se van a determinar las fases generales de un análisis SAST. Una vez determinado el concepto, se incluirán las herramientas pioneras en el mercado, las cuales formarán parte del posterior estudio.
  Finalmente, se concluirá con referencias a estudios relacionados con este trabajo.
- **Objetivos concretos y metodología**: Una vez se dispone de una visión actualizada del estado del arte en herramientas y técnicas SAST, se plantea determinar los objetivos que se buscan con el presente estudio.
  Se iniciará con un objetivo general y se indicarán posteriormente objetivos concretos menos abstractos. Tras esto, se mostrará la metodología propuesta para lograr dichos objetivos, la cual está compuesta por cinco pasos.
- **Desarrollo del piloto**: Tras explicar la metodología a seguir y los objetivos que se pretenden alcanzar, se llega al capítulo de mayor envergadura de todo el documento. A lo largo de este capítulo, se explicará cada uno de los pasos y cada una de las decisiones que se han tomado a lo largo del experimento para cumplir este con éxito.
  El capítulo consta del siguiente subapartado:
  - Descripción del experimento: En este apartado se explicarán todos y cada uno de los requisitos necesarios para el desarrollo del entorno de pruebas que será necesario para realizar el experimento, así como aquellos requisitos necesarios para la ejecución de este.

- **Presentación de resultados**: Durante este apartado se mostrarán los resultados obtenidos de los análisis realizados durante el experimento. A su vez, consta de un subapartado:
    - Discusión de resultados: En este apartado se plantearán las conclusiones obtenidas de los resultados.
- **Conclusiones y líneas de trabajo futuro**: Una vez realizado el experimento, se enumeran una serie de conclusiones obtenidas a partir de la discusión realizada sobre los resultados. Además de esto, se plantean líneas futuras que seguir a partir del experimento realizado, con el fin de complementarlo o ampliarlo.

Capítulo II: Estado del arte

El objetivo de este capítulo es determinar el estado en el que se encuentra actualmente el campo de las herramientas SAST para tener un punto de partida objetivo, sólido y fiable sobre el que iniciar el presente experimento.

A lo largo de este capítulo, se realizará una investigación del estado del arte de las herramientas SAST, estudiando la técnica actual, su situación en el mercado, analizando las herramientas que se van a utilizar y observando finalmente el estado del arte de bases de datos etiquetadas de código con vulnerabilidades para estudios SAST, puesto que, si no se parte de un conjunto conocido, no se podría hacer una comparativa fiable. Por último, se presentarán también estudios sobre fiabilidad de las herramientas SAST.

Descripción de SAST

En la actualidad, es un hecho que la seguridad debe ser tenida en cuenta e implementada cuando se lleva a cabo un desarrollo *software*, como indica el concepto de *Security by Design* (OWASP, 2016). Muchas son las técnicas utilizadas durante el ciclo de vida del *software* para implementar dicha seguridad. Entre estas, tal y como indica McGraw (2005), el análisis estático de código fuente se considera la actividad más importante de entre las mejores prácticas de seguridad que se han de realizar en el curso del desarrollo de una aplicación. La ventaja que ofrece a las técnicas de *pentesting* es la posibilidad de realizarse durante fases más tempranas del SDLC.

Los análisis de código pueden ser principalmente de 2 tipos:

1. Análisis estático: El análisis de código estático es un análisis del *software* de tipo caja blanca en el cual se analiza el código, ya sea el código fuente o el código objeto generado, del *software* sin ejecutarlo (OWASP, 2019 Mayo 13).
2. Análisis dinámico: El análisis de código dinámico observa el comportamiento del software examinando las salidas que proporciona ante un conjunto de entradas consideradas “anómalas” (se considera “anómala” a toda entrada no esperada por el *software*, como podría ser un nombre en una entrada que espera una fecha de nacimiento) (OWASP, 2009).

Este trabajo se centra en el primer tipo, el análisis estático, concretamente en su técnica más utilizada, el SAST.

Situación de SAST en el mercado

Según Gartner (2019), se puede definir SAST como “un conjunto de tecnologías diseñadas para analizar el código fuente, el *bytecode* y los binarios de la aplicación en busca de condiciones de codificación y diseño que sean indicativas de vulnerabilidades de seguridad. Las soluciones SAST analizan una aplicación ‘desde el interior hacia el exterior’ partiendo de un estado de no ejecución”.

Para realizar este análisis, las herramientas de análisis estático de código, de forma generalizada, cuentan con las siguientes fases:

- Fase de traducción: Consiste en la construcción de un modelo entendible por los analizadores de la herramienta. Para construir dicho modelo, es necesario el código fuente de la aplicación.
- Fase de análisis: Una vez el código es legible por los analizadores que dispone la herramienta, estos se encargan de realizar un análisis del modelo completo, en busca de patrones vulnerables.
- Presentación de resultados: Tras analizar el software, la herramienta muestra los resultados para que un experto en seguridad los revise y determine su veracidad. Este proceso se denomina “triaje de resultados”.

Como todo software, las herramientas SAST también se pueden dividir en herramientas libres y herramientas privativas.

Para determinar cuál sería el conjunto de herramientas privativas por excelencia se va a utilizar el *Magic Quadrant* de Gartner, el cual se muestra en la figura 1:

Figure 1. Magic Quadrant for Application Security Testing

Source: Gartner (April 2019)

Figura 1 - Magic Quadrant de Gartner (abril 2019)

Como se puede observar, la herramienta que destaca más actualmente es la solución de Synopsys, seguida de Veracode, Fortify (solución de Micro Focus) y Checkmarx.

Estado del arte de las soluciones propuestas

Para el presente documento, se utilizará la herramienta Fortify como opción privativa del estudio. En cuanto a las alternativas Opensource, se utilizarán dos soluciones SAST indicadas por OWASP (2019): SonarQube y VisualCodeGreeper. A continuación, se analiza cada una de ellas:

- **Fortify**: Fortify (Microfocus S.L., 2019) es una suite de herramientas de análisis de código de la empresa Microfocus. Fortify da soporte a una gran variedad de lenguajes entre los que se encuentran C, C++, Java y Python. Fortify presenta dos alternativas principales a la hora de su integración en un S-SDLC: versión *On premise* (cuyo logo

se puede observar en la figura 2) y versión *On Cloud* (también denominada Fortify On Demand).

Figura 2 - Logo Fortify

Mientras que la alternativa Fortify On Demand se plantea como una alternativa completamente transparente para el usuario y totalmente gestionada por Microfocus, la versión *On premise* es completamente personalizable, planteando diversas arquitecturas posibles combinando los diversos componentes de los que dispone. Principalmente está compuesto por tres componentes principales:

- Fortify SCA: Componente principal de análisis, su tarea consiste en realizar las tareas de traducción y análisis del código fuente. Para ello se basa en la herramienta de línea de comandos sourceanalyzer, la cual pertenece también a la suite de Fortify y se instala con este componente.

  Este componente dispone de la capacidad de ponerse en modo servicio en escucha para la recepción de modelos traducidos (que son ficheros con extensión .mbs) y dedicarse únicamente a analizarlos, así como es capaz de generar dichos modelos.

  La característica más importante de este componente es que él es capaz, por sí solo, de realizar todas las tareas de análisis de código estático, convirtiéndose en la arquitectura mínima que se puede desplegar de la suite de Fortify.
- Fortify SSC: Este componente es una aplicación web que requerirá de un servidor web en la que alojarse (ej: Tomcat). Mediante este componente, se pueden centralizar los resultados obtenidos por los diferentes análisis realizados, así como realizar una gestión de equipos de triaje de resultados para que cada uno acceda solo a los contenidos que debe triar.

  Fortify SSC ofrece un servicio en escucha para la recepción de ficheros de análisis (con extensión .fpr), los cuales parsea y despliega en la plataforma, permitiendo así realizar las tareas de análisis y triaje que se realizarían a través de la interfaz de Fortify SCA.
- Cloudscan: Cloudscan es un componente de gestión de colas y de reparto de tareas entre los componentes de la suite. Su función consiste en recibir modelos (ficheros .mbs) y mandarlos a analizadores que se encuentren libres para que estos les devuelvan el fichero de resultados (extensión .fpr). Una vez dispone de los resultados los almacena de forma temporal (siete días por

defecto), siendo opcional la posibilidad de enviarlos a un componente Fortify SSC.

Con estos 3 componentes (4 si se divide el componente Fortify SCA en sus 2 modalidades: analizador y traductor) se dispone de 3 posibles arquitecturas principales, como se muestra en la figura 3:

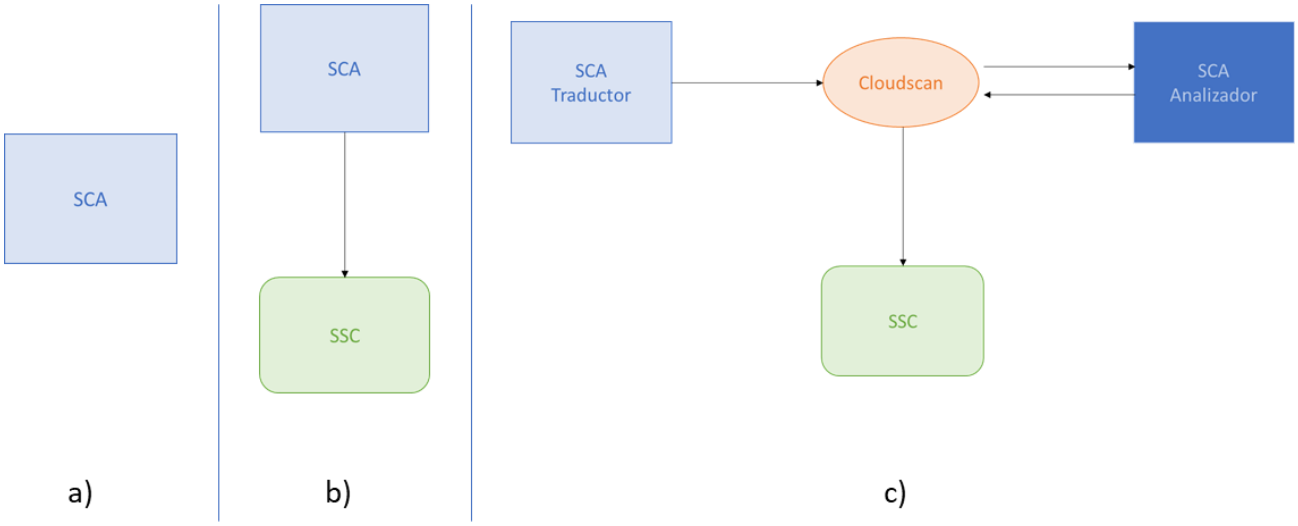

Figura 3 - Arquitecturas Fortify

- **SonarQube**: SonarQube (SonarSource, 2019), cuyo logo se encuentra en la figura 4, es una herramienta de análisis estático de la calidad y seguridad del código. Da soporte a una gran variedad de lenguajes entre los que se encuentran C++, Java, C#, PHP, Python, etc. SonarQube dispone de cuatro modalidades: *Community Edition*, *Developer Edition*, *Enterprise Edition*, *Data Center Edition*.
  En este documento, con el fin de realizar el experimento, se ha utilizado la versión *Community Edition*, por lo que todo lo relativo a esta herramienta en el presente documento se referirá a dicha versión.

Figura 4 - Logo SonarQube

SonarQube dispone de dos componentes principales:

- SonarScanner: Este componente es similar a la herramienta sourceanalyzer mencionada previamente en Fortify, ya que se trata de una herramienta de línea de comandos sin interfaz cuyo fin es analizar código. La principal diferencia entre ambas es que, mientras que sourceanalyzer te proporciona un

fichero de salida con los resultados obtenidos, SonarScanner los envía a un servidor para que los visualices ahí, por lo que es dependiente del mismo.
Si no dispone de un servidor, no es posible llevar a cabo el análisis, ya que, a diferencia de sourceanalyzer, cuando se inicia un nuevo análisis, el conjunto de reglas se descarga desde el servidor al que se enviarán los resultados.

- SonarQube: Aunque se llama igual que la solución global, la herramienta SonarQube es un servidor en el que alojar los resultados de los análisis realizados con SonarScanner, así como la solución SSC de Fortify. Además de esto, es el servidor donde se almacenan y gestionan todas las reglas y traductores que utiliza SonarScanner para su ejecución.

Como se puede observar, ambos componentes son dependientes el uno del otro de forma simbiótica, formando una única posible arquitectura que consta de una o más máquinas con SonarScanner que conectan a un servidor de SonarQube.

SonarQube está basado en *plugins* de análisis, disponiendo de una gran variedad de *plugins* de análisis para cubrir todos los lenguajes. La principal diferencia entre las diferentes modalidades mencionadas previamente reside en la cantidad de *plugins* disponibles.

- **VisualCodeGreeper**: VisualCodeGreeper (VCG) (NCC Group, 2016), cuyo logo se encuentra en la figura 5, es una herramienta automática de análisis de seguridad en el código que cuenta con soporte para los siguientes lenguajes: C++, C#, VB, PHP, Java y PL/SQL.

Figura 5 - Logo VisualCodeGreeper

A diferencia del resto de soluciones mostradas, la herramienta VisualCodeGreeper es una solución más sencilla, no pensada para entornos centralizados pues no dispone de ningún componente ajeno a la propia herramienta, la cual muestra una interfaz sencilla y eficaz, tal y como se muestra en la siguiente imagen:

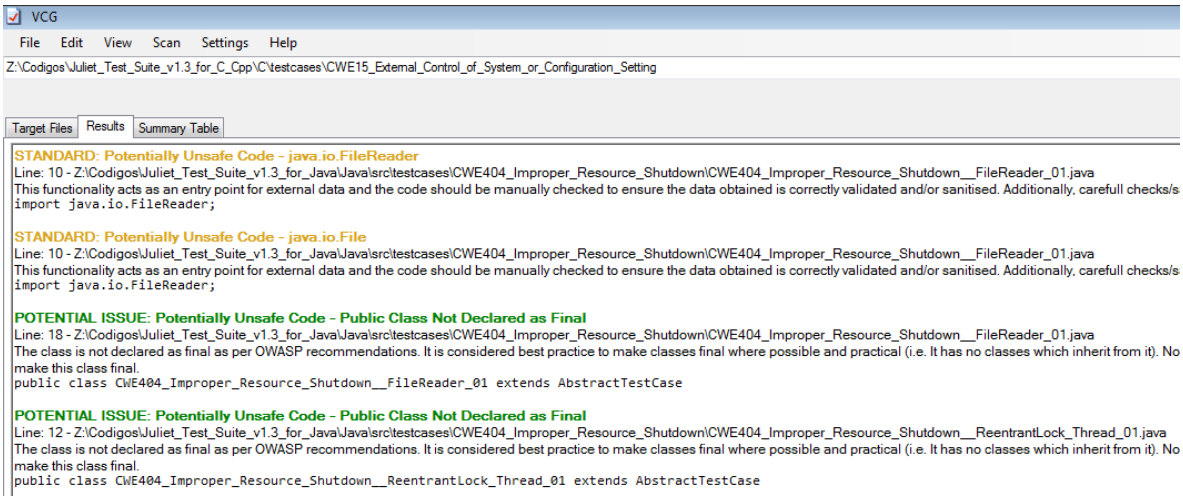

Figura 6 - Interfaz VCG

La arquitectura y la forma de trabajar con esta herramienta sería similar a la primera de las opciones planteadas con la suite de Fortify.

Estado del arte de herramientas de estudio SAST

Una vez establecido el conjunto de herramientas que se va a utilizar, es necesario estudiar también el conjunto de fragmentos de código que se va a analizar. Para poder realizar un estudio objetivo, es necesario el uso de fragmentos de código con vulnerabilidades ya preestablecidas. Gracias a esto, es posible determinar con exactitud la ratio de Falsos Positivos, Verdaderos Positivos, Falsos Negativos y Verdaderos Negativos.

El proyecto *NIST Software Assurance Metrics And Tool Evaluation* (SAMATE), tal y como se indica en la web de su proyecto, está dedicado a mejorar la seguridad del software mediante el desarrollo de métodos que permitan evaluar las herramientas de análisis de software, medir la eficacia de las herramientas y técnicas, e identificar las lagunas en las herramientas y métodos (National Institute of Standards and Technology, s.f.). Se puede encontrar su logo en la figura 7.

El alcance del proyecto SAMATE es amplio: desde sistemas operativos a cortafuegos, desde SCADA a aplicaciones web, desde analizadores de seguridad de código fuente a métodos de corrección por construcción.

Figura 7 - Logo de SAMATE

Dentro de SAMATE, se encuentra el *Software Assurance Reference Dataset* (SARD), el cual definen en SAMATE como un repositorio comunitario de código de ejemplo y otros artefactos para ayudar a los usuarios finales a evaluar las herramientas y a los desarrolladores a probar sus métodos (National Institute of Standards and Technology, s.f.). A partir de noviembre de 2016, SARD consta de más de 171 000 casos de prueba que abarcan una amplia variedad de defectos y lenguajes de programación.

Es necesario utilizar este tipo de repositorios etiquetados porque establecen de forma fiable la veracidad de los resultados obtenidos por las herramientas, tal y como hizo (Bermejo Higuera, 2014) en el desarrollo de su tesis.

Estudios relacionados con la fiabilidad SAST

Tras analizar la situación del mercado de las herramientas SAST y los recursos disponibles para su estudio, se observa su situación en el entorno académico en relación con el estudio propuesto: análisis de la fiabilidad de las herramientas SAST.

Según Fawcett (2006), se puede determinar la fiabilidad de las herramientas SAST partiendo de 4 métricas iniciales básicas que se pueden obtener de cualquier herramienta de este tipo: Falsos Positivos, Verdaderos Positivos, Falsos Negativos, Verdaderos Negativos (a partir de ahora FP, TP, FN y TN respectivamente).

De estas 4 métricas, Fawcett obtiene las siguientes medidas:

- FP rate: Esta métrica determina la cantidad de Falsos Positivos que ha encontrado la herramienta.
- TP rate: Determina la ratio de Verdaderos Positivos hallados.
- Precision: Se define como $\frac{TP}{TP+FP}$
- Recall: Esta variable se define como $\frac{TP}{P}$
- Accuracy: Se define como $\frac{TP+TN}{P+N}$
- F-measure: Variable establecida de la siguiente forma: $\frac{2}{1/precision + 1/recall}$

A continuación, se puede ver en la figura 8 un resumen de todas estas métricas obtenidas a partir de la matriz de confusión de resultados obtenidos por las herramientas:

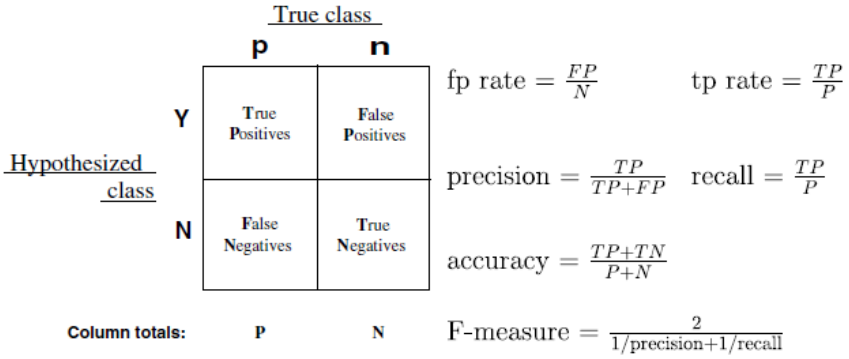

Figura 8 - Métricas de Fawcett, 2006

Más adelante, Antunes & Vieira (2010) utilizaron estas medidas para generar *benchmarks* de este tipo de herramientas. Para ello, redujeron el número de métricas a tres: *Precision, Recall y F-measure*. Según comentan, estas métricas, que están acotadas entre 0 y 1, permiten comparar de forma eficaz diferentes herramientas de análisis de vulnerabilidades, ya que hay herramientas que muestran resultados por vulnerabilidad y otras, en cambio, lo hacen por líneas vulnerables, por lo que el número de vulnerabilidades no sería una métrica fiable que permita comparar correctamente dos herramientas diferentes.

Estas métricas serán la base de la comparativa que se realizará en el presente documento.

A continuación, se pueden encontrar los siguientes trabajos relacionados con este experimento:

- **Metodología de evaluación de herramientas de análisis automático de seguridad de aplicaciones web para su adaptación en el ciclo de vida de desarrollo** (Bermejo Higuera, 2014): A lo largo de esta tesis se lleva a cabo una metodología para evaluar herramientas de análisis automático de aplicaciones web. El presente trabajo puede servir como apoyo al análisis de nuevas herramientas para ampliar el conjunto de resultados de la tesis, ya que las métricas utilizadas son las mismas, así como el repositorio de muestras.
- **Análisis de seguridad y calidad de aplicaciones (Sonarqube)** (Ospina Delgado, 2015): El estudio realizado por Ospina presenta un análisis de la herramienta SonarQube en un conjunto de vulnerabilidades pertenecientes al OWASP top 10, por lo que los resultados obtenidos por parte de SonarQube amplian y complementan los obtenidos por este estudio.
- **Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services** (Antunes & Vieira, 2010): El estudio de Antunes y Vieira ha servido de base para la obtención de métricas

utilizadas durante el piloto para la obtención de resultados, así como permite justificar la necesidad de utilizar únicamente las tres medidas utilizadas. El piloto experimental realizado puede considerarse un caso práctico del contenido teórico que plantean estos autores.

Capítulo III: Objetivos concretos y metodología

El objetivo del presente documento consiste en el estudio de las herramientas de análisis estático de código más extendidas en el mercado con el fin de determinar su fiabilidad. Además de esto, se busca realizar una comparación entre el mundo del software libre y el del software privativo en el campo SAST.

Objetivos específicos

Para cumplir con el objetivo mencionado previamente, se pretenden cumplir los siguientes objetivos específicos:

- Establecer un entorno de trabajo que cuente con diferentes lenguajes.
- Conocer el conjunto de herramientas de análisis a utilizar con el fin de entender cómo funciona cada una de ellas.
- Determinar las vulnerabilidades que se van a utilizar como base.
- Tomar medidas para calcular las métricas de fiabilidad de las herramientas: FP, FN, TP y TN.
- Calcular las variables *Precision*, *Recall* y *F-measure* para cada vulnerabilidad y herramienta.
- Comparar los resultados obtenidos por cada herramienta y vulnerabilidad y extraer conclusiones.
- Identificar qué herramienta y en qué situaciones se obtienen mejores resultados.

Metodología

Con el fin de llevar a cabo los objetivos indicados en el apartado anterior, la metodología que se va a implementar consta de cinco fases principales, las cuales se muestran en la siguiente imagen:

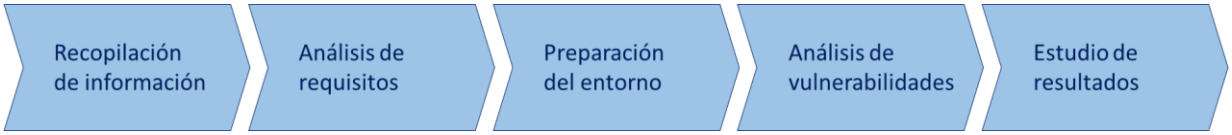

Figura 9 – Metodología

1. **Recopilación de información**: Durante esta primera fase, se realiza una investigación inicial con el fin de determinar el conjunto de análisis y el conjunto de herramientas de las que va a disponer el piloto desarrollado. Esta fase termina cuando se ha

determinado el conjunto de herramientas, así como el conjunto de lenguajes de programación y el repositorio de análisis.

2. **Análisis de requisitos**: Una vez determinado cada uno de los componentes del entorno, se realiza un análisis de los requisitos de cada uno de ellos para poder buscar una solución en la que conviva el conjunto completo de herramientas y lenguajes. El análisis de requisitos se da por finalizada cuando se dispone de una solución en la cual se muestra el conjunto completo de herramientas y lenguajes, permitiendo analizar cada uno de los lenguajes con cada una de las herramientas. De ser necesario, por motivos de rendimiento, se podrán incluir un conjunto de restricciones a la hora de llevar a cabo los análisis.
3. **Preparación del entorno**: Tras finalizar el análisis de requisitos, se procede a la preparación del entorno en el que se va a ejecutar el piloto, el cual debe cumplir con el conjunto de requisitos obtenidos en la fase anterior. Durante la preparación del entorno, se incluye la definición del conjunto de análisis, el cual es un subconjunto del repositorio seleccionado durante la fase de Recopilación de información. Esta fase cuenta con una subfase de *testing* en la cual se comprueba que todo funciona de forma adecuada. Una vez los *test* son pasados de forma exitosa, se procede a la siguiente fase. Durante esta fase se podrá modificar el listado de requisitos.
4. **Análisis de vulnerabilidades**: Una vez se dispone de entorno desplegado y probado, se puede proceder a analizar el conjunto establecido para el análisis. Durante el desarrollo de esta fase, cada código y lenguaje establecido durante la fase de Recopilación de información será analizado con cada una de las herramientas. Los resultados obtenidos serán clasificados en base a las vulnerabilidades analizadas.
5. **Estudio de resultados**: Los resultados obtenidos durante la fase anterior son analizados con el fin de obtener las métricas para determinar la fiabilidad de las herramientas SAST. Una vez analizados, se compararán para sacar conclusiones del estudio realizado.

Capítulo IV: Desarrollo del piloto

A lo largo de este capítulo se llevará a cabo el desarrollo del piloto del estudio. Para su correcto desarrollo y un mejor entendimiento, se va a dedicar un apartado a describir todo el proceso que se ha seguido, así como la descripción de cada uno de los componentes del entorno para así permitir su reproducibilidad.

Descripción del experimento

El experimento consiste en la realización de una serie de escaneos de códigos que sirven como ejemplo de vulnerabilidades conocidas. El objetivo de estos escaneos consiste en la obtención de unas métricas objetivas basadas en un código del que ya se conocen todas sus vulnerabilidades, clasificadas por su CWE.

Con el fin de llevar a cabo esto se han definido una serie de entornos que cumplan con los requisitos de todos los componentes. El entorno definido cuenta con un conjunto de máquinas (virtuales), herramientas de escaneo y lenguajes de programación.

El conjunto de máquinas es el siguiente:

- 9 máquinas Windows 7, de ahora en adelante W_1, W_2, …, W_9, con 10 GB de RAM y 1 core asignado cada una.
- 1 máquina Ubuntu 18.04 (LTS), de ahora en adelante U_1, con 4 GB de RAM y 1 core asignado.

El uso de máquinas virtuales puede afectar al rendimiento del escaneo ya que, por norma, los analizadores de código estático suelen aprovechar la totalidad de recursos de la máquina. Aun así, es necesario el uso de máquinas virtuales para garantizar la pureza del entorno, ya que se parte de un conjunto de recursos limitado (la máquina local del alumno). El caso ideal constaría de una máquina física por cada máquina virtual de las mencionadas previamente, evitando así los problemas de rendimiento.

Dentro de estas máquinas se alojarán las diferentes herramientas y lenguajes ya que, para que las herramientas de análisis sean capaces de generar su propio modelo, es necesario que la máquina analizadora disponga de un compilador o intérprete del lenguaje de programación analizado.

El listado de herramientas de escaneo es el siguiente:

- Fortify v4.40 versión *On premise*: Se utiliza esta versión ya que es la proporcionada por la UNIR (HPE, 2015).

- SonarQube v7.8 Community Edition: Se utiliza esta por compatibilidad con el plugin de análisis de C++, la versión LTS (v7.9) no está soportada por el plugin opensource de C++, el disponible pertenece a versiones de pago (las cuales están fuera del alcance del estudio) (SonarSource, 2019).
- Sonar Scanner v4.0.0.1744: Esta es la versión más actualizada y es compatible con la versión seleccionada de SonarQube (SonarSource, 2019).
- VisualCodeGreeper v2.1.0: Esta versión es la última disponible (npdunn, 2016).

Se ha elegido este conjunto de herramientas por encontrarse dentro de las más utilizadas hoy en día dentro de los dos tipos mencionados previamente: software libre y software privativo.

Estas herramientas analizarán los siguientes lenguajes:

- C++ proveniente de la suite de Visual Studio 2010.
- Java 6.
- PHP 5.5.30.

Las versiones de los lenguajes indicados están establecidas por los requisitos de los repositorios de código a analizar. Se han elegido estos lenguajes por ser de los más utilizados y por encontrarse dentro de la intersección generada por el listado de lenguajes de programación del repositorio y los lenguajes soportados de cada una de las herramientas.

La disposición de elementos queda representada de la siguiente forma:

	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	U_1
Fortify	✓	✓	✓							
SonarScanner				✓	✓	✓				
SonarQube										✓
VisualCodeGreeper							✓	✓	✓	
C++ (VS2010)	✓			✓			✓			
Java 6		✓			✓			✓		
Java 11										✓
PHP 5.5.30			✓			✓			✓	

Tabla 1 - Disposición de máquinas

Para la realización del experimento, se han analizado diferentes alternativas tanto para las herramientas de análisis como para los lenguajes que serán analizados. Las versiones de las herramientas y lenguajes están establecidas en base a los requisitos de los repositorios de código, así como de las propias herramientas.

En lo relativo a red, todas las máquinas son máquinas aisladas a excepción de las máquinas W_4, W_5, W_6 y U_1. Las W_x tienen visibilidad de la máquina U_1. Esta arquitectura es necesaria para poder analizar código en Java 6 con el cliente SonarScanner y ejecutar el servidor de SonarQube con Java 11 (requisito de SonarQube v7.8).

A continuación, se muestra la arquitectura de forma gráfica:

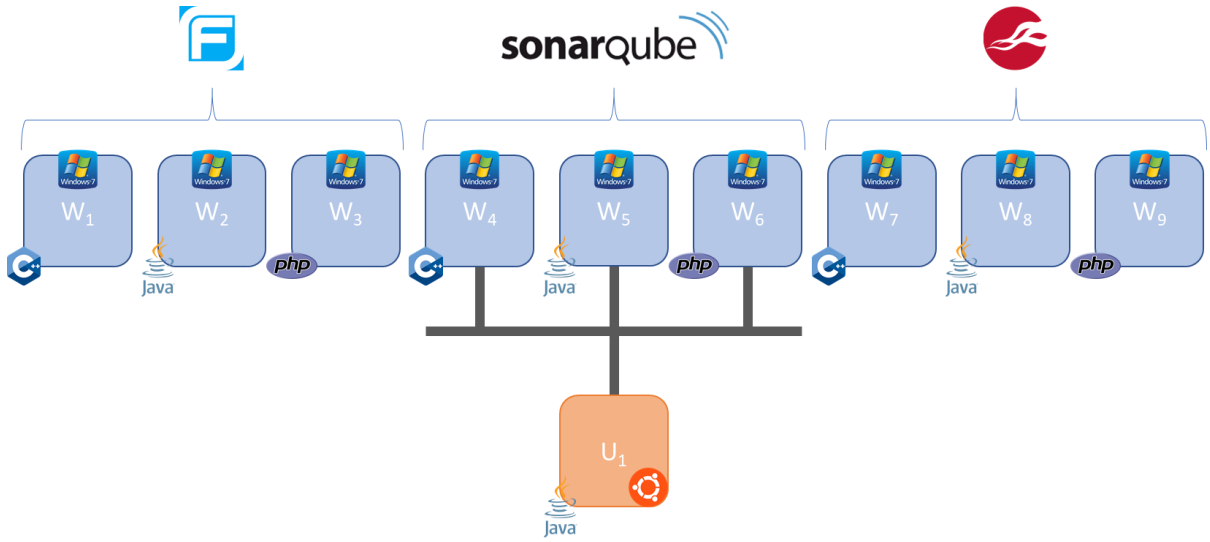

Figura 10 - Arquitectura del experimento

Para realizar el experimento, se han obtenido los códigos vulnerables de los repositorios de la herramienta SARD de SAMATE, concretamente el repositorio Juliet para C++ y Java y el repositorio de SARD para PHP (National Institute of Standards and Technology, s.f.).

SAMATE es un repositorio del NIST cuya finalidad es proveer de códigos de ejemplo con vulnerabilidades conocidas a los investigadores para poder realizar estudios con herramientas de análisis de código estático.

Del conjunto total de códigos recopilados Java y C++ provienen del repositorio Juliet, por lo que coinciden la gran mayoría de vulnerabilidades, mientras que en PHP las vulnerabilidades de las que dispone SARD son diferentes de las que se encuentran en Juliet, por lo que se ha decidido coger 5 vulnerabilidades para analizar Java y C++ y de esta forma poder hacer una comparación adicional; y coger otras 5 vulnerabilidades para PHP. Además, para cada una de estas vulnerabilidades, se va a truncar la cantidad de fragmentos de código vulnerables en 500, ya que es un conjunto suficientemente grande para el estudio.

De esta forma, las vulnerabilidades tomadas para Java y C++ son:

1. **CWE-15: External Control of System or Configuration Setting**
   "Permitir el control externo de la configuración del sistema puede interrumpir el servicio o hacer que una aplicación se comporte de forma inesperada y potencialmente maliciosa."
2. **CWE-23: Relative Path Traversal**
   "El software utiliza una entrada externa para construir una ruta de acceso que debería estar dentro de un directorio restringido, pero no neutraliza adecuadamente secuencias como ".." que pueden resolverse a una ubicación que esté fuera de ese directorio."
3. **CWE-259: Use of Hard-coded Password**
   "Una contraseña escrita en claro en el código normalmente conduce a un fallo de autenticación significativo que puede resultar difícil de detectar para el administrador del sistema.
   Una vez detectado, puede ser difícil de arreglar, por lo que el administrador puede verse obligado a desactivar el producto por completo. Hay, principalmente, dos opciones posibles:
   - *Inbound*: el software contiene un mecanismo de autenticación que comprueba la existencia de una contraseña escrita en claro en el código.
   - *Outbound*: el software se conecta a otro sistema o componente, y contiene una contraseña escrita en claro en el código para conectarse a ese componente.

   En la alternativa *Inbound*, se crea una cuenta de administración predeterminada y se introduce una contraseña sencilla en el producto y se asocia a esa cuenta. Esta contraseña codificada es la misma para cada instalación del producto, y por lo general no puede ser cambiada o desactivada por los administradores del sistema sin modificar manualmente el programa, o de otro modo parcheando el software. Si la contraseña es descubierta o publicada (algo común en Internet), entonces cualquier persona con conocimiento de esta contraseña puede acceder al producto. Por último, dado que todas las instalaciones del software tendrán la misma contraseña, incluso en diferentes organizaciones, esto permite que se produzcan ataques masivos como los gusanos.
   La opción *Outbound* se aplica a los sistemas *front-end* que se autentican con un servicio *back-end*. El servicio *back-end* puede requerir una contraseña fija que puede ser fácilmente descubierta. El programador puede simplemente introducir en claro en el código esas credenciales de *back-end* en el software de *front-end*. Cualquier usuario de ese programa puede extraer la contraseña. Los sistemas del lado del cliente con

contraseñas codificadas representan una amenaza aún mayor, ya que la extracción de una contraseña de un binario suele ser muy sencilla."

4. **CWE-369: Divide By Zero**
   "Esta debilidad ocurre típicamente cuando se proporciona un valor inesperado al producto, o si se produce un error que no se detecta correctamente. Normalmente, ocurre en cálculos que involucran dimensiones físicas tales como tamaño, longitud, ancho y alto."
5. **CWE-404: Improper Resource Shutdown or Release**
   "Cuando se crea o se asigna un recurso, el desarrollador es responsable de liberarlo correctamente y de tener en cuenta todas las posibles vías de expiración o invalidación, como un período de tiempo determinado o la revocación."

Todas las descripciones se han obtenido de MITRE (Common Weakness Enumeration, 2019) y se han traducido al castellano. Se han decidido estas vulnerabilidades por ser poco comunes, entendiendo que el software estará testeado, al menos, con el OWASP Top 10 (OWASP, 2019).

Por otro lado, las vulnerabilidades de PHP son:

1. **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**
   "Las vulnerabilidades de *Cross-site Scripting* (XSS) se producen cuando:
   a) Los datos no confiables entran en una aplicación web, normalmente desde una solicitud web.
   b) La aplicación web genera dinámicamente una página web que contiene estos datos no fiables.
   c) Durante la generación de la página, la aplicación no impide que los datos contengan contenido ejecutable por un navegador web, como JavaScript, etiquetas HTML, atributos HTML, eventos de ratón, Flash, ActiveX, etc.
   d) Una víctima visita la página web generada a través de un navegador web, que contiene secuencias de comandos maliciosas que se inyectaron utilizando los datos no confiables.
   e) Dado que el script proviene de una página web enviada por el servidor web, el navegador web de la víctima ejecuta el script malicioso en el contexto del dominio del servidor web.
   f) Esto viola efectivamente la intención de la política del mismo origen del navegador web, que establece que los scripts en un dominio no deberían poder acceder a los recursos o ejecutar código en un dominio diferente.

Hay tres tipos principales de XSS:

- Tipo 1: XSS reflejado (o no persistente): el servidor lee los datos directamente de la solicitud HTTP y los refleja en la respuesta HTTP. Las vulnerabilidades XSS reflejadas se producen cuando un atacante hace que una víctima suministre contenido peligroso a una aplicación web vulnerable, que luego se refleja de vuelta a la víctima y es ejecutado por el navegador web. El mecanismo más común para entregar contenido malicioso es incluirlo como parámetro en una URL que se publica o se envía por correo electrónico directamente a la víctima. Las URL construidas de esta manera constituyen el núcleo de muchos esquemas de phishing, en los que un atacante convence a una víctima para que visite una URL que hace referencia a un sitio vulnerable. Después de que el sitio refleja el contenido del atacante a la víctima, el contenido es ejecutado por el navegador de la víctima.
- Tipo 2: XSS almacenado (o persistente): la aplicación almacena datos peligrosos en una base de datos, un foro de mensajes, un registro de visitantes u otro almacén de datos de confianza. Posteriormente, los datos peligrosos se vuelven a leer en la aplicación y se incluyen en el contenido dinámico. Desde la perspectiva de un atacante, el lugar óptimo para inyectar contenido malicioso es en un área que se muestra a muchos usuarios o a usuarios particularmente interesantes. Los usuarios interesantes suelen tener privilegios elevados en la aplicación o interactuar con datos confidenciales que son valiosos para el atacante. Si uno de estos usuarios ejecuta contenido malicioso, el atacante puede realizar operaciones privilegiadas en nombre del usuario u obtener acceso a datos confidenciales pertenecientes al usuario. Por ejemplo, el atacante puede inyectar XSS en un mensaje de registro, que puede no ser manejado correctamente cuando un administrador ve los registros.
- Tipo 0: XSS basado en DOM - En XSS basado en DOM, el cliente realiza la inyección de XSS en la página; en los otros tipos, el servidor realiza la inyección. El XSS basado en DOM generalmente implica un script de confianza controlado por el servidor que se envía al cliente, como por ejemplo Javascript, que realiza comprobaciones de sanidad en un formulario antes de que el usuario lo envíe. Si el script suministrado por el servidor procesa los datos suministrados por el usuario y luego los inyecta de nuevo en la página web (por ejemplo, con HTML dinámico), entonces XSS basado en DOM es posible.

Una vez inyectado el script malicioso, el atacante puede realizar una variedad de actividades maliciosas. El atacante podría transferir información privada, como cookies

que pueden incluir información de sesión, desde el equipo de la víctima al atacante. El atacante podría enviar solicitudes maliciosas a un sitio web en nombre de la víctima, lo que podría ser especialmente peligroso para el sitio si la víctima tiene privilegios de administrador para administrarlo. Los ataques de phishing se pueden utilizar para emular sitios web de confianza y engañar a la víctima para que introduzca una contraseña, lo que permite al atacante comprometer la cuenta de la víctima en ese sitio web. Finalmente, el script podría explotar una vulnerabilidad en el propio navegador web que posiblemente se apodere de la máquina de la víctima, a veces denominada *drive-by hacking*.

En muchos casos, el ataque puede ser lanzado sin que la víctima lo sepa. Incluso con usuarios cuidadosos, los atacantes utilizan con frecuencia una variedad de métodos para codificar la parte maliciosa del ataque, como la codificación de URL o Unicode, por lo que la solicitud parece menos sospechosa."

2. **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

   "Sin la suficiente eliminación o cita de la sintaxis SQL en las entradas controlables por el usuario, la consulta SQL generada puede hacer que esas entradas se interpreten como SQL en lugar de como datos normales del usuario. Esto se puede utilizar para alterar la lógica de la consulta y pasar por alto las comprobaciones de seguridad, o para insertar instrucciones adicionales que modifiquen la base de datos de *back-end*, incluyendo posiblemente la ejecución de comandos del sistema.

   La inyección SQL se ha convertido en un problema común en los sitios web basados en bases de datos. El defecto se detecta fácilmente y se explota fácilmente, y como tal, cualquier sitio o paquete de software con una base de usuarios mínima es probable que sea objeto de un intento de ataque de este tipo. Este fallo depende del hecho de que SQL no hace ninguna distinción real entre los planos de control y de datos."

3. **CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')**

   "El software construye toda o parte de una consulta LDAP utilizando una entrada de influencia externa de un componente de flujo ascendente, pero no neutraliza o neutraliza incorrectamente los elementos especiales que podrían modificar la consulta LDAP prevista cuando se envía a un componente de flujo descendente."

4. **CWE-91: XML Injection (aka Blind XPath Injection)**

   "Dentro de XML, los elementos especiales podrían incluir palabras o caracteres reservados como "<", ">", "", "", y "&", que podrían utilizarse para añadir nuevos datos o modificar la sintaxis XML."

5. **CWE-601: URL Redirection to Untrusted Site ('Open Redirect')**

> "Un parámetro HTTP puede contener un valor de URL y puede hacer que la aplicación web redirija la petición a la URL especificada. Al modificar el valor de la URL de un sitio malicioso, un atacante puede iniciar con éxito una estafa de phishing y robar las credenciales de usuario. Debido a que el nombre del servidor en el enlace modificado es idéntico al sitio original, los intentos de phishing tienen una apariencia más confiable."

Todas las descripciones se han obtenido de MITRE (Common Weakness Enumeration, 2019) y se han traducido al castellano. Se han tomado estas vulnerabilidades porque son las disponibles en el repositorio de SARD.

Para cada una de las vulnerabilidades, los métodos a implementar para su análisis dentro de cada una de las herramientas eran diferentes. A continuación, se comenta cada uno de ellos:

- **Fortify**: Para la ejecución de este estudio, se ha optado por la arquitectura más sencilla de las disponibles en la suite de Fortify, la cual consta únicamente del componente Fortify SCA. Los análisis realizados con Fortify se han realizado a través de la herramienta sourceanalyzer. Esta herramienta es una herramienta de línea de comandos, por lo que se ha creado un script para Windows (un fichero .bat) para realizar los análisis. Los scripts tenían la siguiente estructura:

```
sourceanalyzer -b mybuild -clean

sourceanalyzer -b mybuild cl /I"..\..\testcasesupport" /W3
/MT /GS /RTC1 /bigobj /EHsc /nologo /c main.cpp CWE*.cpp
CWE*.c ..\..\testcasesupport\io.c
..\..\testcasesupport\std_thread.c

sourceanalyzer -b mybuild -scan -f result.fpr
```

  Primero se eliminan los elementos que pudiera haber en la sesión "mybuild" y después se traduce el código en esa sesión y se analiza. Este ejemplo en concreto es un script sacado de los utilizados para analizar C++.

  A continuación, se muestra una imagen de ejemplo de resultados obtenidos con la herramienta:

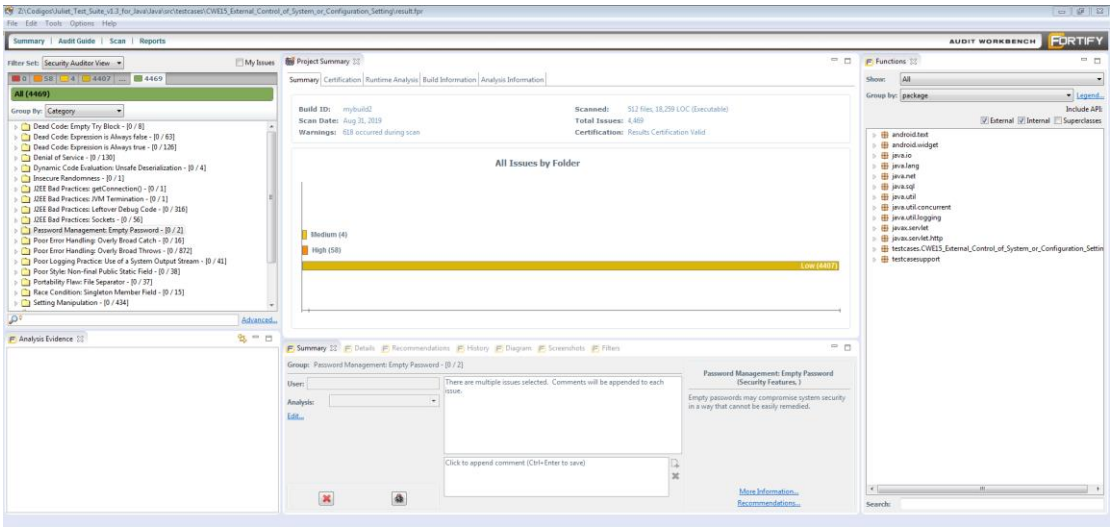

Figura 11 - Resultados Fortify

- **SonarQube**: En el caso de SonarQube, se ha tenido que utilizar el cliente SonarScanner, el cual, partiendo de un fichero de configuración, analiza el código y sube los resultados al servidor de SonarQube.
  Las propiedades necesarias para realizar los escaneos y clasificar de forma adecuada los resultados han sido las siguientes:

```
# must be unique in a given SonarQube instance

sonar.projectKey=Java_project_CWE23

# --- optional properties ---

# defaults to project key

sonar.projectName=Java project CWE23

# defaults to 'not provided'

sonar.projectVersion=1.0

sonar.projectBaseDir=Z:\\Codigos\\Juliet_Test_Suite_v1.3_for_Java\\Java\\src\\testcases\\CWE23_Relative_Path_Traversal

sonar.login=admin

sonar.password=admin

sonar.exclusions=**/*.py

sonar.java.binaries=Z:\\Codigos\\Juliet_Test_Suite_v1.3_for_Java\\Java\\src\\testcases\\CWE23_Relative_Path_Traversal\\antbuild\\testcases\\CWE23_Relative_Path_Traversal

# Path is relative to the sonar-project.properties file. Defaults to .

#sonar.sources=.

# Encoding of the source code. Default is default system encoding

#sonar.sourceEncoding=UTF-8
```

Este fichero ejemplo ha sido recopilado de uno de los ficheros utilizados para analizar Java. Para el caso de Java, en concreto, fue necesario añadir una propiedad que indicara dónde se localizaban los ficheros .class de Java. Para el resto, lo relacionado con ficheros que debieran ser referenciados se hacía dentro de la configuración del propio plugin en el servidor de SonarQube.
A continuación, se muestra una imagen de la interfaz de resultados de SonarQube:

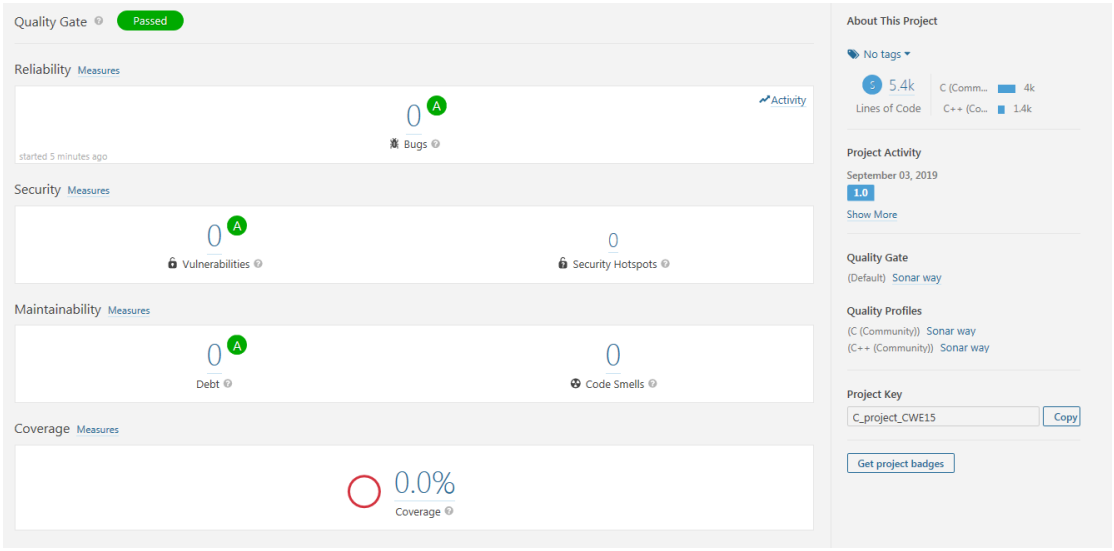

Figura 12 - Interfaz de resultados de SonarQube

Como se puede observar, se dividen los resultados en diferentes categorías, mostrando resultados de seguridad, de mantenimiento y de fiabilidad.

- **VisualCodeGreeper**: VisualCodeGreeper ha resultado la más cómoda de las tres herramientas, ya que es la única que dispone de una interfaz directamente, sin necesidad de dividirse en diferentes componentes (componentes de análisis, componentes de visualización y clasificación de resultados, etc.).
  Para realizar análisis dentro de VisualCodeGreeper, simplemente se tienen que seguir los pasos mostrados en las siguientes imágenes:

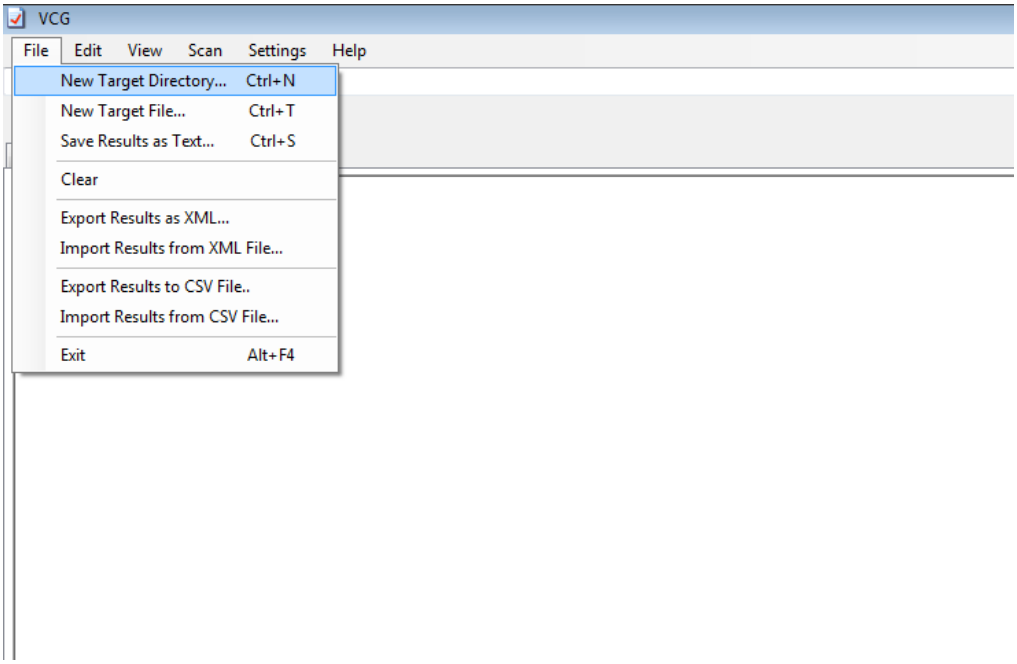

Figura 13 - VCG análisis (I)

1. El primer paso a realizar cuando se quiere realizar un análisis es indicar los ficheros a analizar. En este caso, como se trata de un análisis de más de un fichero, se a de seleccionar un directorio de ficheros.

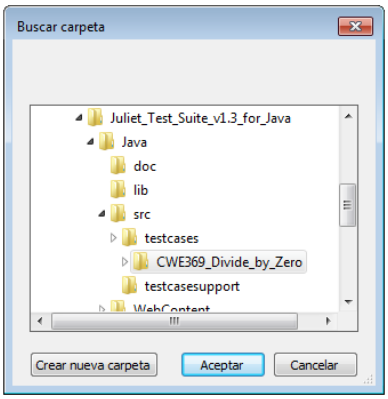

Figura 14 - VCG análisis (II)

2. Entonces aparecerá un árbol de directorios para que se indique la carpeta donde se encuentran los ficheros que se desean analizar.

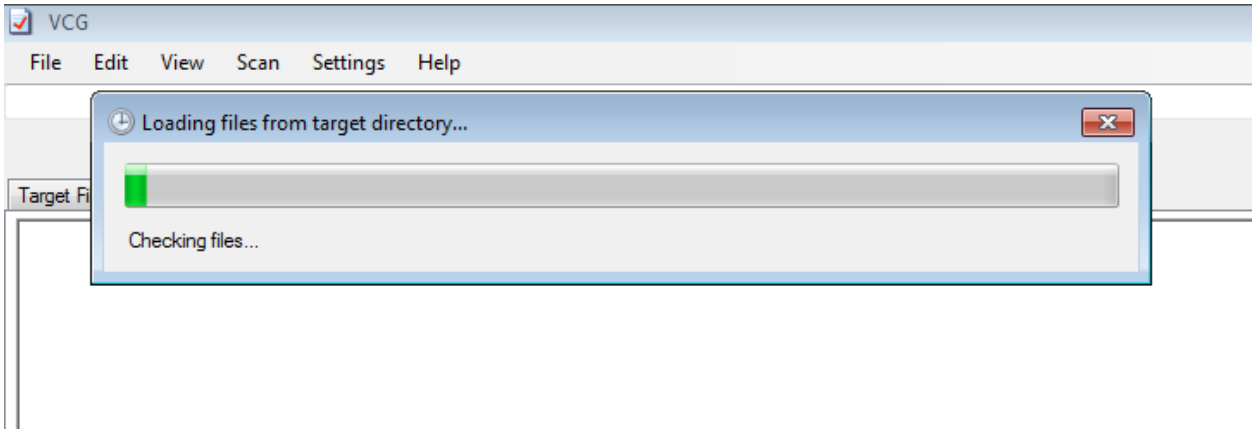

Figura 15 - VCG análisis (III)

3. Entonces, aplica filtros para determinar qué ficheros son del lenguaje especificado en la configuración (esta se debe indicar antes de realizar un análisis).

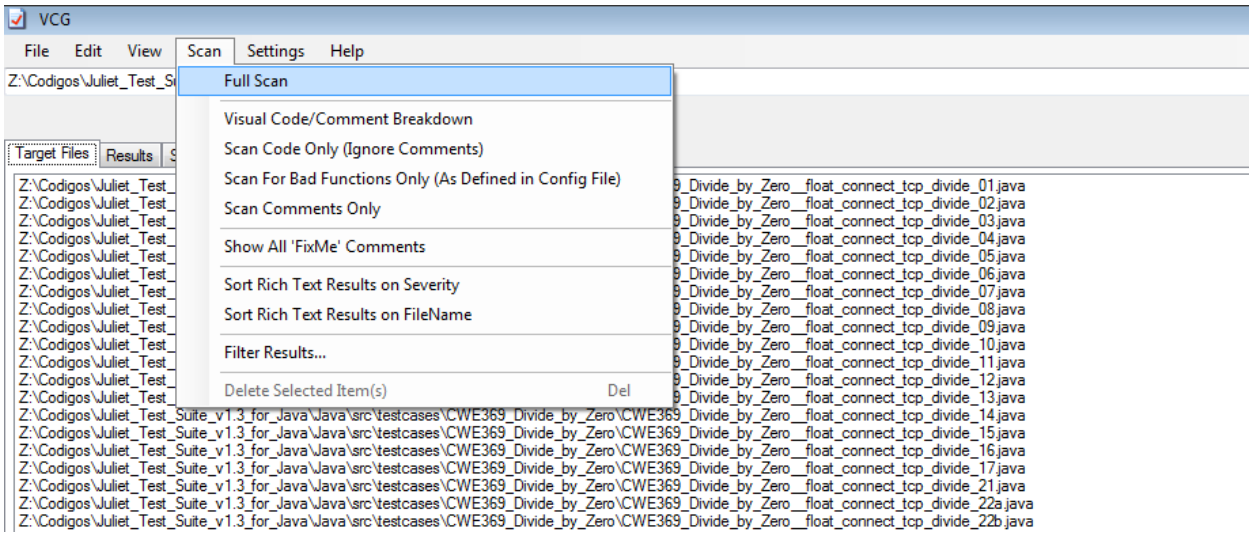

Figura 16 - VCG análisis (IV)

4. Aparecerá entonces el listado de ficheros que van a ser analizados. Para analizarlos hay que darle a *Scan* → *Full Scan*.

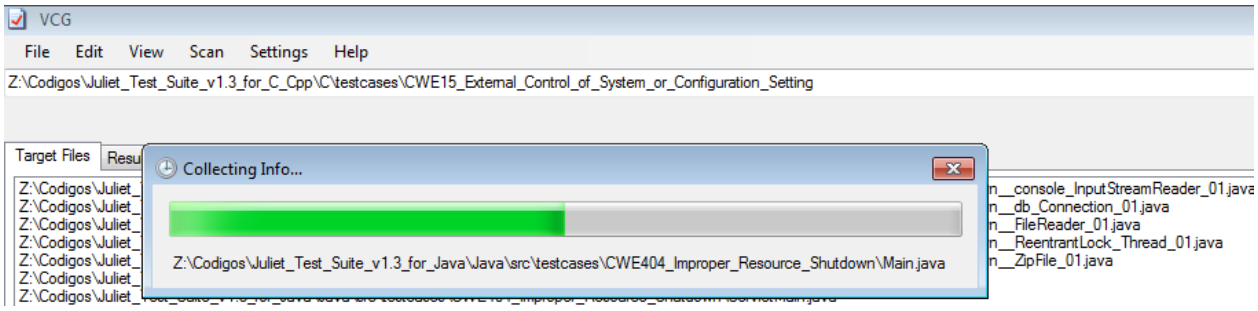

Figura 17 - VCG análisis (V)

5. Es entonces cuando empieza el escaneo y se van rellenando las pestañas *Results* y *Summary Table* con los resultados que se van obteniendo del análisis.

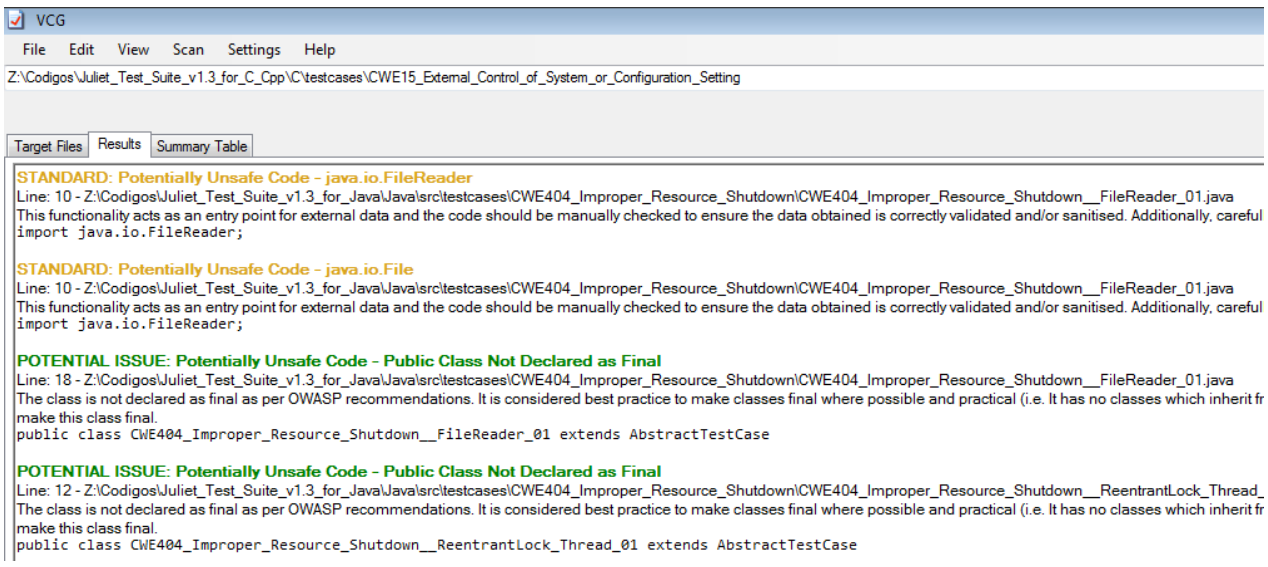

Figura 18 - VCG análisis (VI)

6. Finalmente, se dispone de los resultados detallados en la pestaña *Results* y de un resumen en la pestaña *Summary Table*. Se pueden exportar estos resultados en un fichero con formato .csv seleccionando la opción exportar a CSV en el menú desplegable *File*.

Capítulo V: Presentación de resultados

A lo largo de este capítulo se mostrarán todos los resultados obtenidos de los diferentes análisis realizados, presentados por herramienta, lenguaje y CWE.

Para cada CWE, se comenzará analizando la cantidad de ficheros que ha considerado la herramienta que son maliciosos o que tienen una vulnerabilidad, mostrando también que cantidad de ficheros han pasado desapercibidos. Acto seguido se mostrará un listado de todas los CWE de las vulnerabilidades halladas junto con una gráfica en la que se representa el número de vulnerabilidades de cada uno de los CWE por criticidad. La criticidad será establecida por la herramienta, tomando de forma general los valores: Crítica, Alta, Media y Baja.

Fortify

La primera herramienta a mostrar será aquella que se encuentra dentro del mundo del software privativo. Esta herramienta, de la empresa Microfocus, ha obtenido los siguientes resultados:

Lenguaje analizado: C++

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**

  Durante este escaneo se han analizado 82 ficheros. El listado de ficheros se encuentra en el Anexo A.1.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

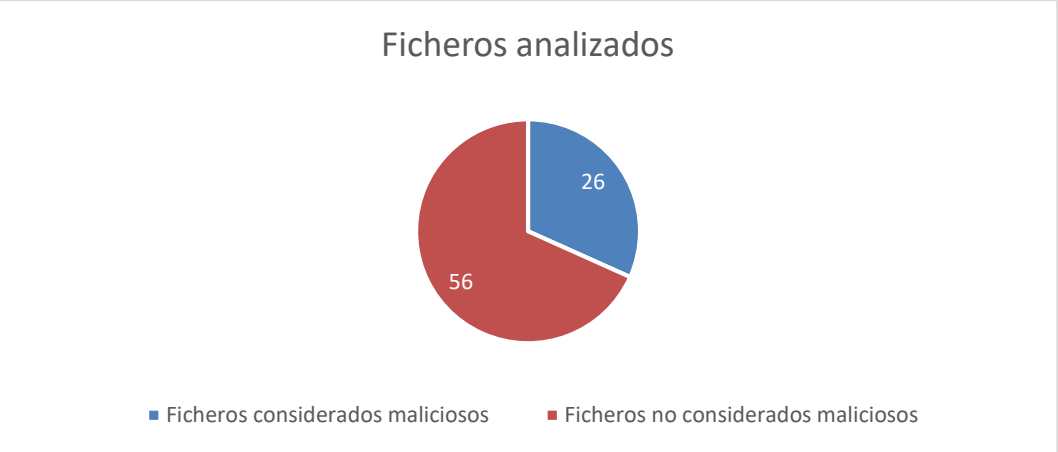

Figura 19 - Ficheros analizados Fortify C++ CWE15

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-15: External Control of System or Configuration Setting

- CWE-170: Improper Null Termination
- CWE-404: Improper Resource Shutdown or Release
- CWE-676: Use of Potentially Dangerous Function

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

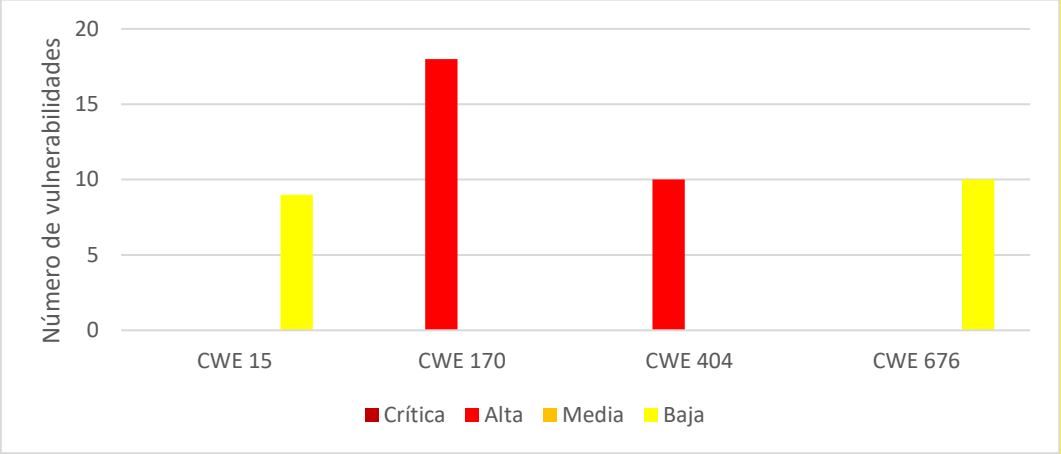

Figura 20 - CWEs por criticidad Fortify C++ CWE15

De estos resultados se obtienen las siguientes métricas:

TP	9	Precision	0,191489362
FP	38	Recall	0,109756098
FN	73	F-measure	0,139534884

Tabla 2 - Medidas obtenidas Fortify C++ CWE15

- **CWE-23: Relative Path Traversal**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 21 - Ficheros analizados Fortify C++ CWE23

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-23: Relative Path Traversal
- CWE-121: Stack-based Buffer Overflow
- CWE-170: Improper Null Termination
- CWE-404: Improper Resource Shutdown or Release
- CWE-457: Use of Uninitialized Variable
- CWE-477: Use of Obsolete Function
- CWE-561: Dead Code

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

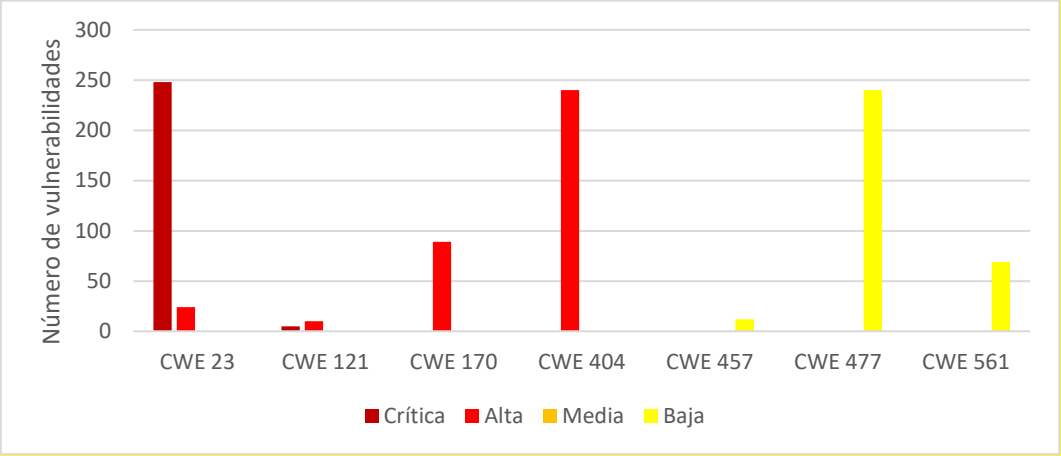

Figura 22 - CWEs por criticidad Fortify C++ CWE23

De estos resultados se obtienen las siguientes métricas:

TP	272	Precision	0,290288154
FP	665	Recall	0,544
FN	228	F-measure	0,378566458

Tabla 3 - Medidas obtenidas Fortify C++ CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 164 ficheros. El listado de ficheros se encuentra en el Anexo A.1.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 23 - Ficheros analizados Fortify C++ CWE259

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-121: Stack-based Buffer Overflow
- CWE-259: Use of Hard-coded Password
- CWE-457: Use of Uninitialized Variable
- CWE-561: Dead Code
- CWE-615: Information Exposure Through Comments
- CWE-676: Use of Potentially Dangerous Function

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

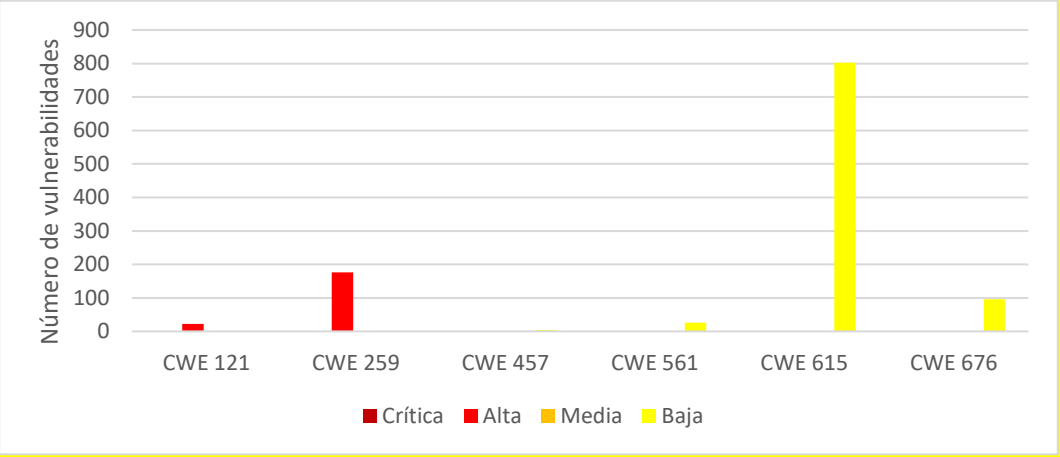

Figura 24 - CWEs por criticidad Fortify C++ CWE259

De estos resultados se obtienen las siguientes métricas:

TP	164	Precision	0,145648313
FP	962	Recall	1

FN	0	F-measure	0,254263566

Tabla 4 - Medidas obtenidas Fortify C++ CWE259

- **CWE-369: Divide By Zero**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 25 - Ficheros analizados Fortify C++ CWE369

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)
- CWE-404: Improper Resource Shutdown or Release
- CWE-457: Use of Uninitialized Variable
- CWE-477: Use of Obsolete Function
- CWE-561: Dead Code
- CWE-563: Assignment to Variable without Use

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

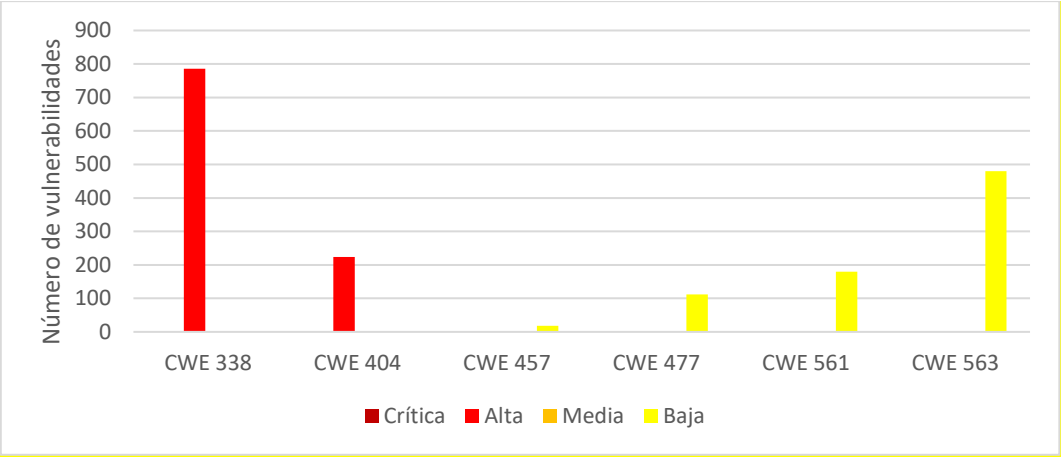

Figura 26 - CWEs por criticidad Fortify C++ CWE369

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	1800	Recall	0
FN	500	F-measure	N/A

Tabla 5 - Medidas obtenidas Fortify C++ CWE369

- **CWE-404: Improper Resource Shutdown or Release**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 27 - Ficheros analizados Fortify C++ CWE404

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-404: Improper Resource Shutdown or Release
  - CWE-457: Use of Uninitialized Variable
  - CWE-561: Dead Code

- CWE-563: Assignment to Variable without Use

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

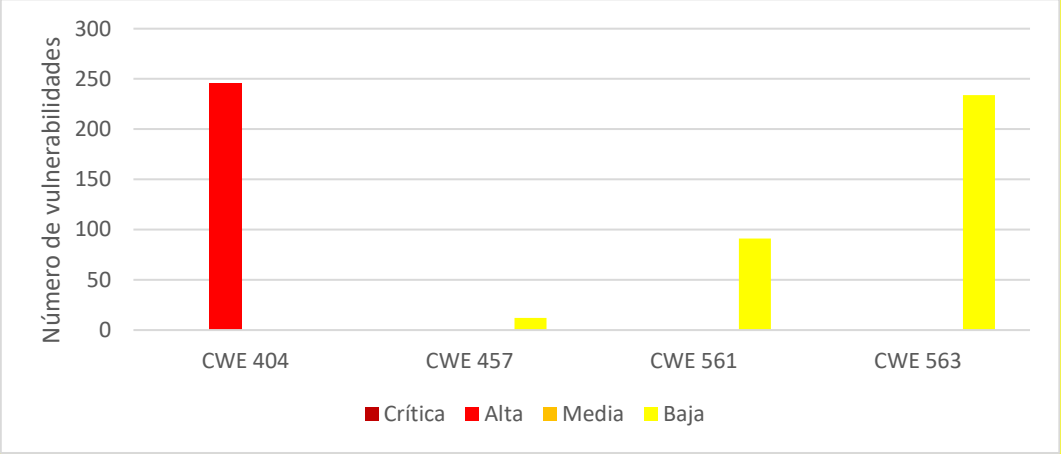

Figura 28 - CWEs por criticidad Fortify C++ CWE404

De estos resultados se obtienen las siguientes métricas:

TP	246	Precision	0,421955403
FP	337	Recall	0,492
FN	254	F-measure	0,454293629

Tabla 6 - Medidas obtenidas Fortify C++ CWE404

Lenguaje analizado: Java

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 29 - Ficheros analizados Fortify Java CWE15

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-15: External Control of System or Configuration Setting e
- CWE-246: J2EE Bad Practices: Direct Use of Sockets
- CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
- CWE-397: Declaration of Throws for Generic Exception
- CWE-474: Use of Function with Inconsistent Implementations
- CWE-489: Leftover Debug Code
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-497: Exposure of System Data to an Unauthorized Control Sphere
- CWE-502: Deserialization of Untrusted Data
- CWE-561: Dead Code
- CWE-730: Denial of Service

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

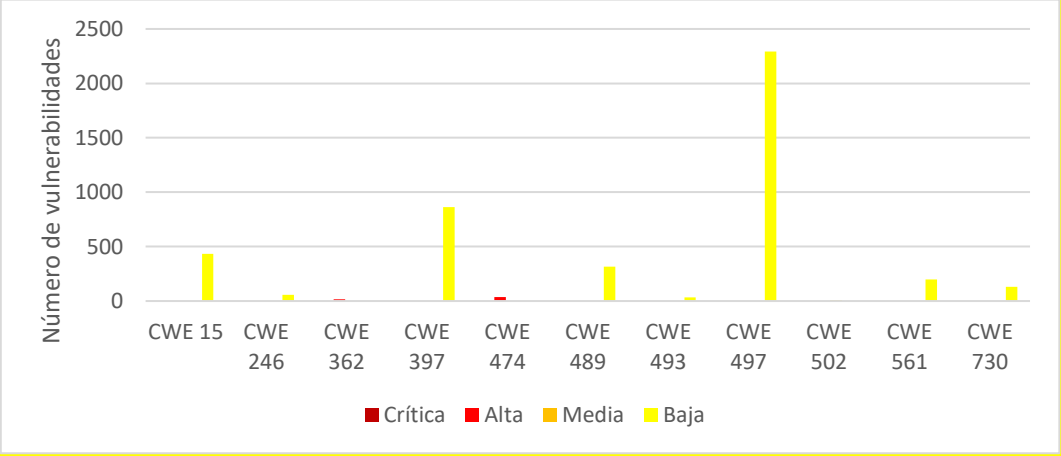

Figura 30 - CWEs por criticidad Fortify Java CWE15

De estos resultados se obtienen las siguientes métricas:

TP	434	Precision	0,099086758
FP	3946	Recall	0,868
FN	66	F-measure	0,177868852

Tabla 7 - Medidas obtenidas Fortify Java CWE15

- **CWE-23: Relative Path Traversal**

Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 31 - Ficheros analizados Fortify Java CWE23

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-23: Relative Path Traversal
- CWE-246: J2EE Bad Practices: Direct Use of Sockets
- CWE-253: Incorrect Check of Function Return Value
- CWE-397: Declaration of Throws for Generic Exception
- CWE-404: Improper Resource Shutdown or Release
- CWE-474: Use of Function with Inconsistent Implementations
- CWE-476: NULL Pointer Dereference
- CWE-489: Leftover Debug Code
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-497: Exposure of System Data to an Unauthorized Control Sphere
- CWE-502: Deserialization of Untrusted Data
- CWE-561: Dead Code
- CWE-563: Assignment to Variable without Use
- CWE-730: Denial of Service

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

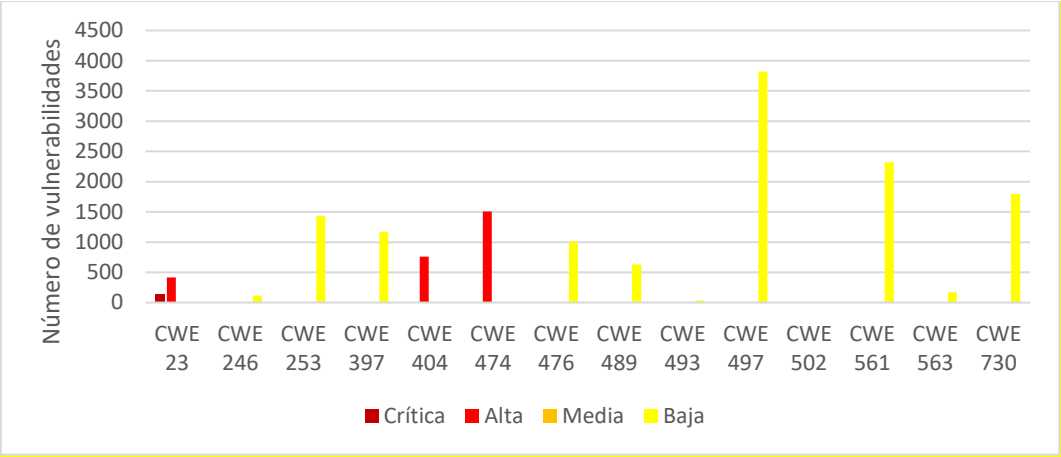

Figura 32 - Ficheros analizados Fortify Java CWE23

De estos resultados se obtienen las siguientes métricas:

TP	500	Precision	0,03262217
FP	14827	Recall	1
FN	0	F-measure	0,063183168

Tabla 8 - Medidas obtenidas Fortify Java CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 180 ficheros. El listado de ficheros se encuentra en el Anexo A.2.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 33 - Ficheros analizados Fortify Java CWE259

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-245: J2EE Bad Practices: Direct Management of Connections
  - CWE-256: Unprotected Storage of Credentials
  - CWE-259: Use of Hard-coded Password

- CWE-397: Declaration of Throws for Generic Exception
- CWE-489: Leftover Debug Code
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-497: Exposure of System Data to an Unauthorized Control Sphere
- CWE-561: Dead Code
- CWE-615: Information Exposure Through Comments
- CWE-730: Denial of Service

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

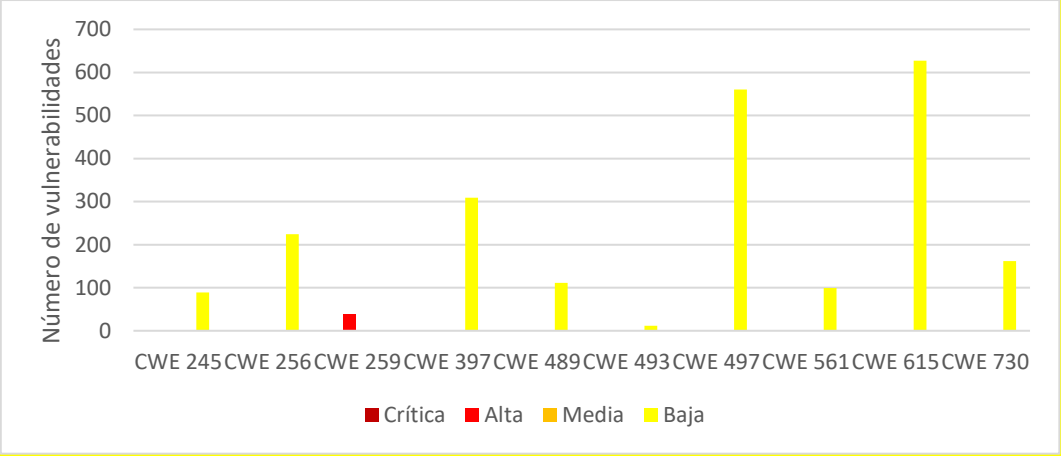

Figura 34 - CWEs por criticidad Fortify Java CWE259

De estos resultados se obtienen las siguientes métricas:

TP	39	Precision	0,017473118
FP	2193	Recall	0,216666667
FN	141	F-measure	0,032338308

Tabla 9 - Medidas obtenidas Fortify Java CWE259

- **CWE-369: Divide By Zero**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 35 - Ficheros analizados Fortify Java CWE369

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-246: J2EE Bad Practices: Direct Use of Sockets
- CWE-397: Declaration of Throws for Generic Exception
- CWE-474: Use of Function with Inconsistent Implementations
- CWE-489: Leftover Debug Code
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-497: Exposure of System Data to an Unauthorized Control Sphere
- CWE-502: Deserialization of Untrusted Data
- CWE-561: Dead Code
- CWE-563: Assignment to Variable without Use
- CWE-730: Denial of Service

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

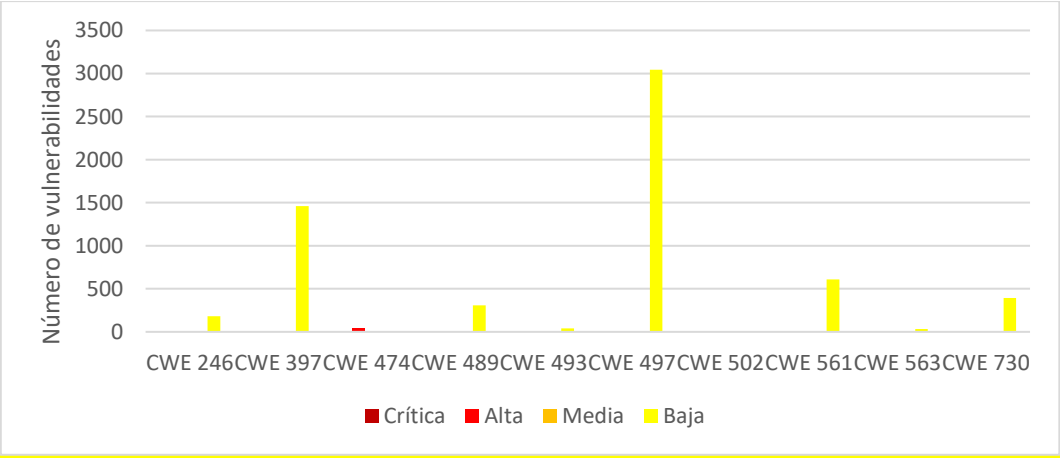

Figura 36 - CWEs por criticidad Fortify Java CWE369

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	6107	Recall	0
FN	500	F-measure	N/A

Tabla 10 - Medidas obtenidas Fortify Java CWE369

- **CWE-404: Improper Resource Shutdown or Release**

  Durante este escaneo se han analizado 5 ficheros. El listado de ficheros se encuentra en el Anexo A.2.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 37 - Ficheros analizados Fortify Java CWE404

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-397: Declaration of Throws for Generic Exception
- CWE-404: Improper Resource Shutdown or Release
- CWE-474: Use of Function with Inconsistent Implementations
- CWE-476: NULL Pointer Dereference
- CWE-489: Leftover Debug Code
- CWE-497: Exposure of System Data to an Unauthorized Control Sphere
- CWE-563: Assignment to Variable without Use
- CWE-690: Unchecked Return Value to NULL Pointer Dereference
- CWE-730: Denial of Service

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

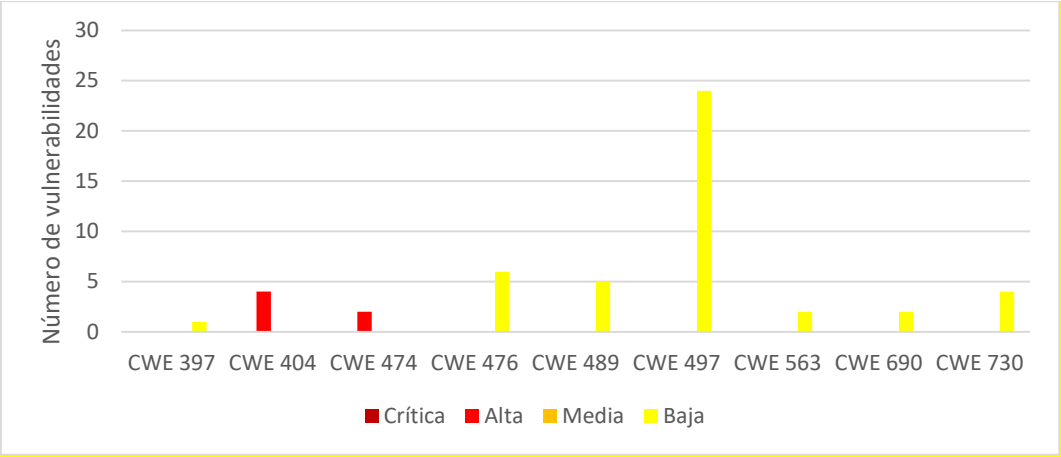

Figura 38 - CWEs por criticidad Fortify Java CWE404

De estos resultados se obtienen las siguientes métricas:

TP	4	Precision	0,08
FP	46	Recall	0,8
FN	1	F-measure	0,145454545

Tabla 11 - Medidas obtenidas Fortify Java CWE404

Lenguaje analizado: PHP

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**

  Durante este escaneo se han analizado 499 ficheros. El listado de ficheros se encuentra en el Anexo A.3.1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 39 - Ficheros analizados Fortify PHP CWE79

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

Figura 40 - CWEs por criticidad Fortify PHP CWE79

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	499	Recall	0
FN	499	F-measure	N/A

Tabla 12 - Medidas obtenidas Fortify PHP CWE79

- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

  Durante este escaneo se han analizado 228 ficheros. El listado de ficheros se encuentra en el Anexo A.3.2 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 41 - Ficheros analizados Fortify PHP CWE89

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
- CWE-259: Use of Hard-coded Password

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

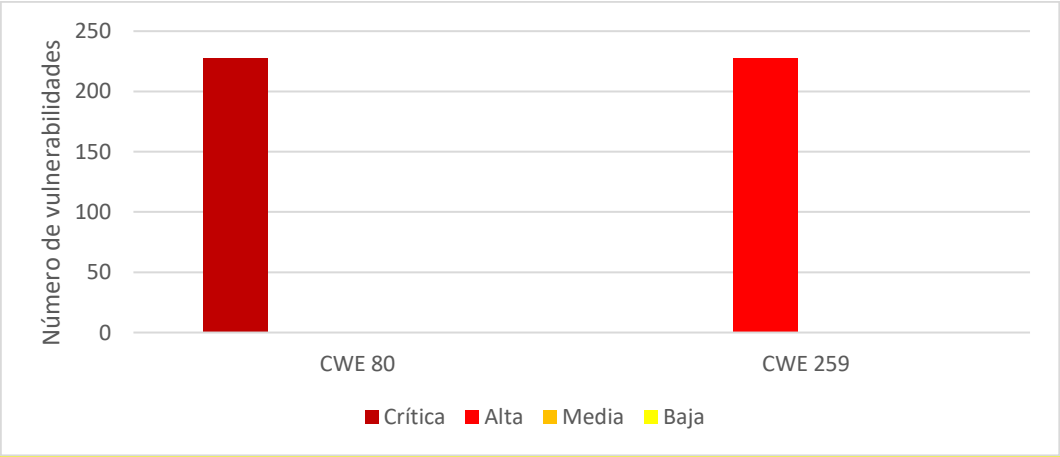

Figura 42 - CWEs por criticidad Fortify PHP CWE89

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	456	Recall	0
FN	228	F-measure	N/A

Tabla 13 - Medidas obtenidas Fortify PHP CWE89

- **CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.3 CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 43 - Ficheros analizados Fortify PHP CWE90

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

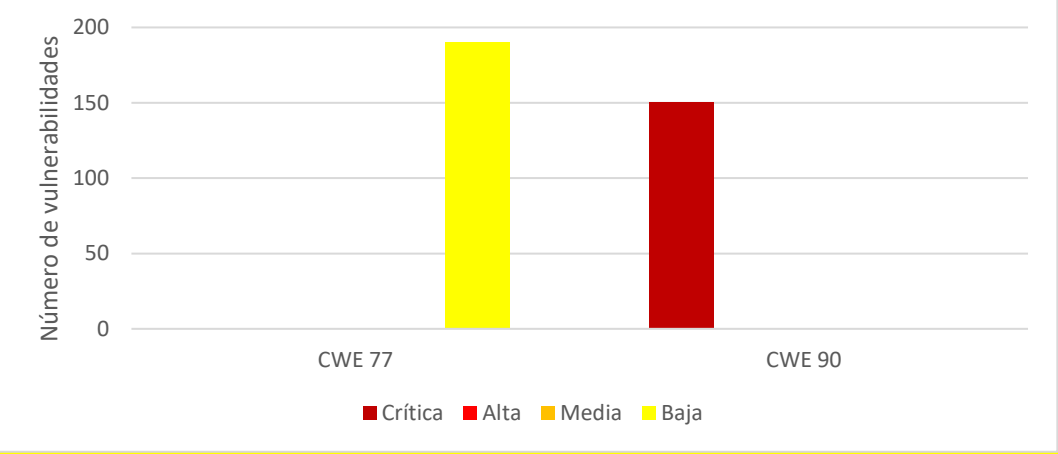

Figura 44 - CWEs por criticidad Fortify PHP CWE90

De estos resultados se obtienen las siguientes métricas:

TP	150	Precision	0,441176471
FP	190	Recall	0,449101796
FN	184	F-measure	0,445103858

Tabla 14 - Medidas obtenidas Fortify PHP CWE90

- **CWE-91: XML Injection (aka Blind XPath Injection)**

  Durante este escaneo se han analizado 266 ficheros. El listado de ficheros se encuentra en el Anexo A.3.4 CWE-91: XML Injection (aka Blind XPath Injection). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 45 - Ficheros analizados Fortify PHP CWE91

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

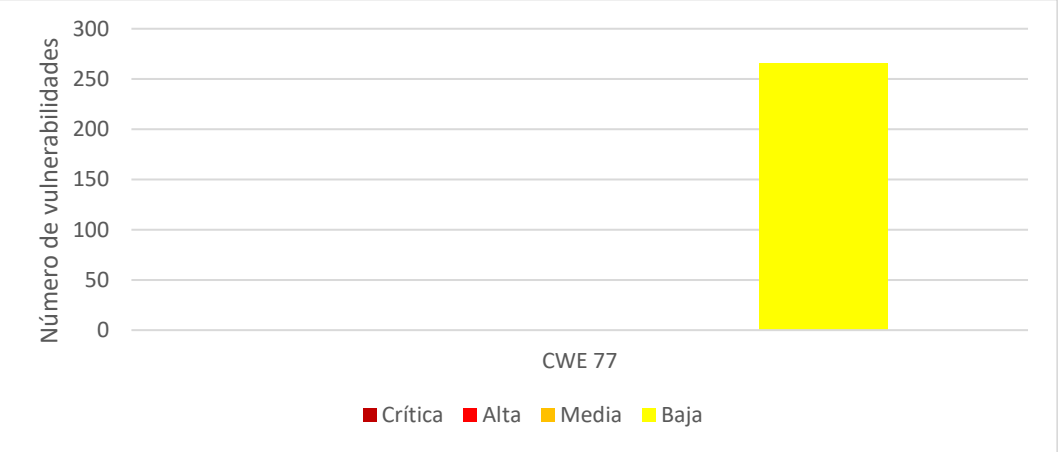

Figura 46 - CWEs por criticidad Fortify PHP CWE91

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	266	Recall	0
FN	266	F-measure	N/A

Tabla 15 - Medidas obtenidas Fortify PHP CWE91

- **CWE-601: URL Redirection to Untrusted Site ('Open Redirect')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.5 CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 47 - Ficheros analizados Fortify PHP CWE601

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
  - CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
  - CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

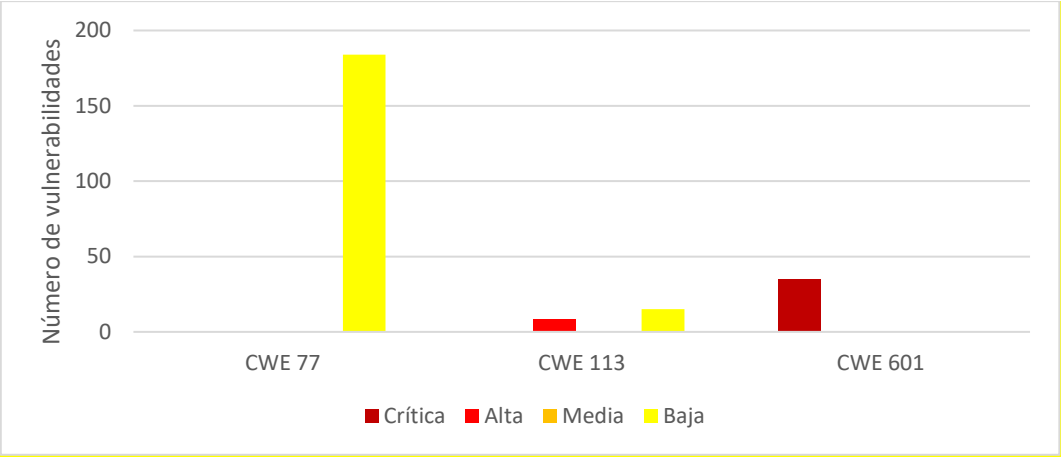

Figura 48 - CWEs por criticidad Fortify PHP CWE601

De estos resultados se obtienen las siguientes métricas:

TP	35	Precision	0,144628099
FP	207	Recall	0,104790419
FN	299	F-measure	0,121527778

Tabla 16 - Medidas obtenidas Fortify PHP CWE601

SonarQube

La segunda herramienta a mostrar dispone de una versión de pago, pero la versión utilizada ha sido la versión *Community*, así como sus *plugins*, los cuales son *opensource* también:

Lenguaje analizado: C++

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**

  Durante este escaneo se han analizado 82 ficheros. El listado de ficheros se encuentra en el Anexo A.1.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 49 - Ficheros analizados SonarQube C++ CWE15

- **CWE-23: Relative Path Traversal**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 50 - Ficheros analizados SonarQube C++ CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 164 ficheros. El listado de ficheros se encuentra en el Anexo A.1.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 51 - Ficheros analizados SonarQube C++ CWE259

- **CWE-369: Divide By Zero**

Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 52 - Ficheros analizados SonarQube C++ CWE369

- **CWE-404: Improper Resource Shutdown or Release**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 53 - Ficheros analizados SonarQube C++ CWE404

Como se puede observar, SonarQube no ha considerado ningún fichero del lenguaje C++ como malicioso. El plugin utilizado para analizar C++ solo ha detectado resultados del tipo *Code Smell* los cuales dan pautas no relacionadas con seguridad (como escribir el nombre del fichero analizado, por ejemplo).

Lenguaje analizado: Java

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

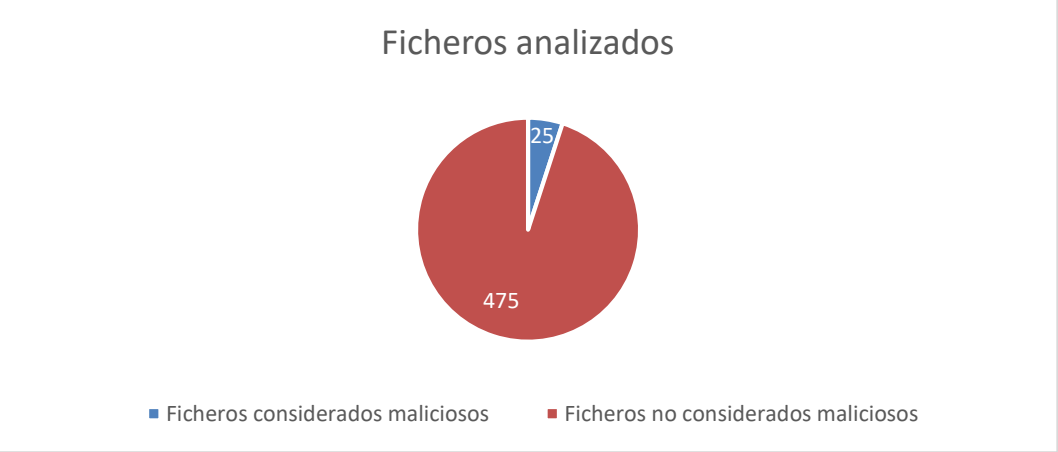

Figura 54 - Ficheros analizados SonarQube Java CWE15

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-493: Critical Public Variable Without Final Modifier
- CWE-500: Public Static Field Not Marked Final

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

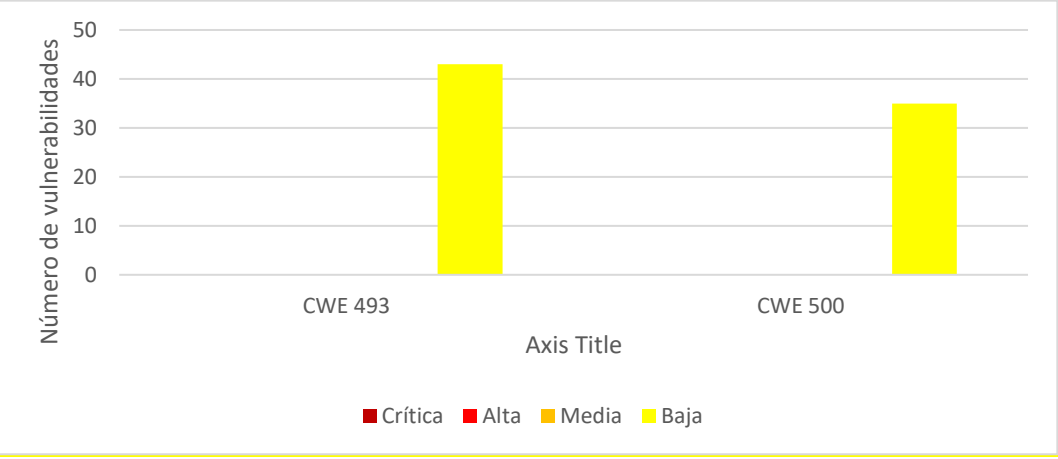

Figura 55 - CWEs por criticidad SonarQube Java CWE15

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	78	Recall	0

FN	500	F-measure	N/A

Tabla 17 - Medidas obtenidas SonarQube Java CWE15

- **CWE-23: Relative Path Traversal**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

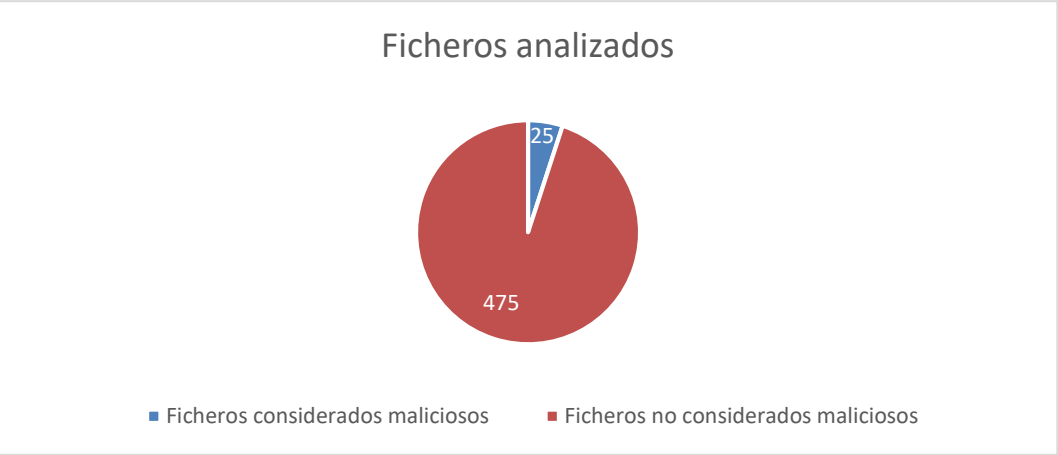

Figura 56 - Ficheros analizados SonarQube Java CWE23

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-493: Critical Public Variable Without Final Modifier
- CWE-500: Public Static Field Not Marked Final

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

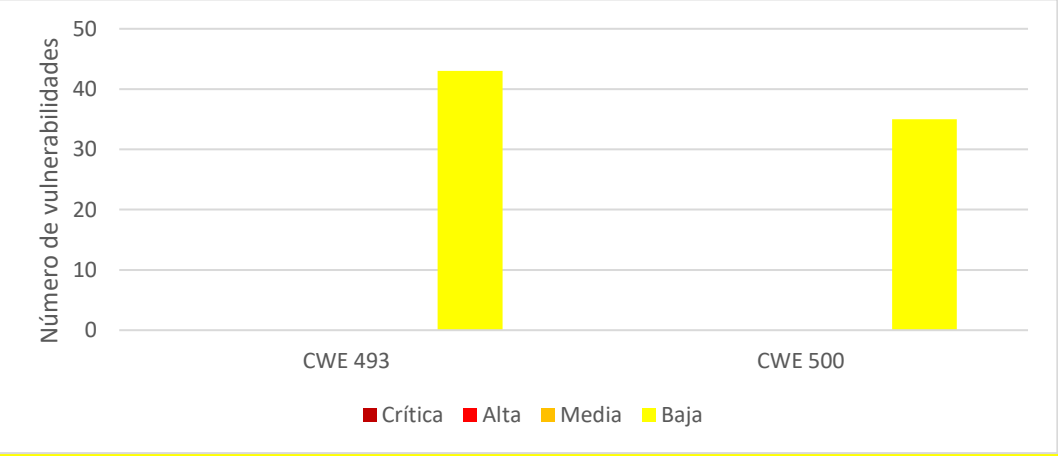

Figura 57 - CWEs por criticidad SonarQube Java CWE23

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	78	Recall	0
FN	500	F-measure	N/A

Tabla 18 - Medidas obtenidas SonarQube Java CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 180 ficheros. El listado de ficheros se encuentra en el Anexo A.2.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 58 - Ficheros analizados SonarQube Java CWE259

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-259: Use of Hard-coded Password
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-500: Public Static Field Not Marked Final

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

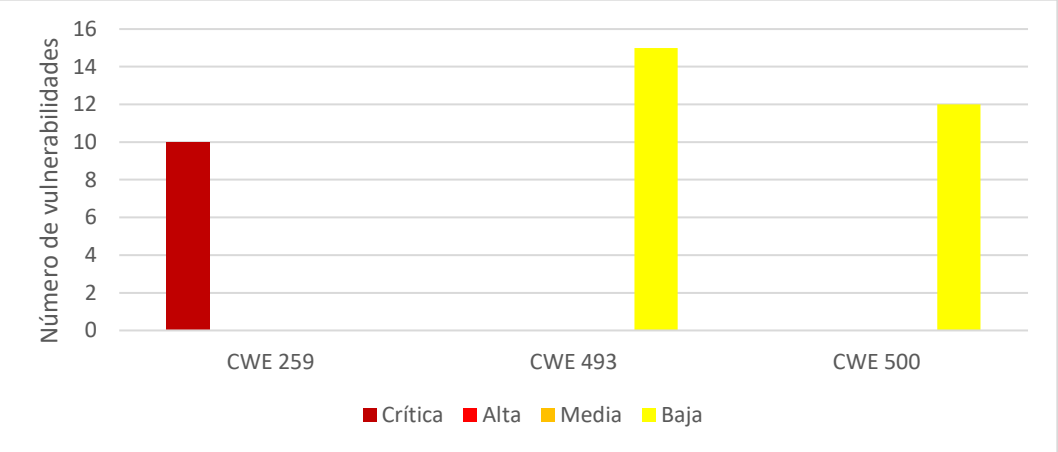

Figura 59 - CWEs por criticidad SonarQube Java CWE259

De estos resultados se obtienen las siguientes métricas:

TP	10	Precision	0,27027027
FP	27	Recall	0,055555556
FN	170	F-measure	0,092165899

Tabla 19 - Medidas obtenidas SonarQube Java CWE259

- **CWE-369: Divide By Zero**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 60 - Ficheros analizados SonarQube Java CWE369

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-493: Critical Public Variable Without Final Modifier
  - CWE-500: Public Static Field Not Marked Final

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

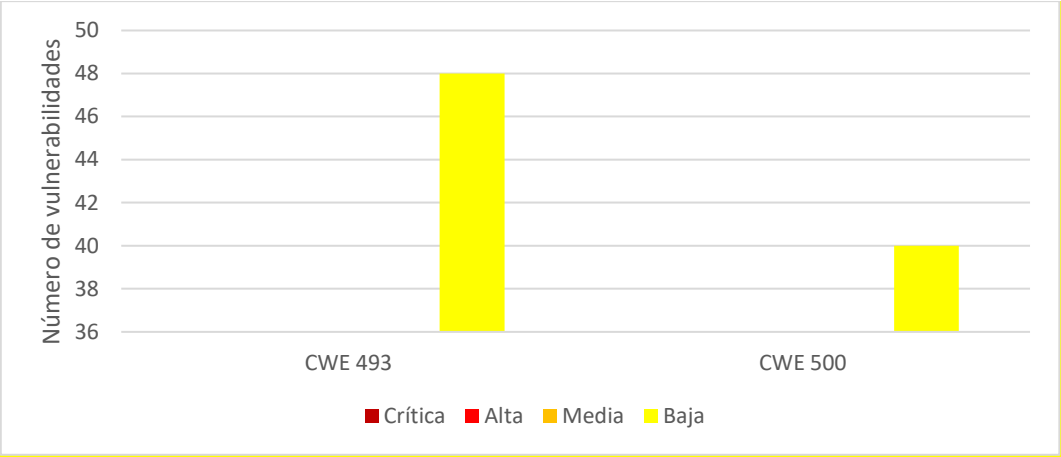

Figura 61 - CWEs por criticidad SonarQube Java CWE369

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	88	Recall	0
FN	500	F-measure	N/A

Tabla 20 - Medidas obtenidas SonarQube Java CWE369

- **CWE-404: Improper Resource Shutdown or Release**

  Durante este escaneo se han analizado 5 ficheros. El listado de ficheros se encuentra en el Anexo A.2.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 62 - Ficheros analizados SonarQube Java CWE404

Como se puede observar en la figura anterior (figura 62), SonarQube no ha considerado ningún fichero como malicioso para el análisis de este CWE en Java.

Lenguaje analizado: PHP

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**

  Durante este escaneo se han analizado 499 ficheros. El listado de ficheros se encuentra en el Anexo A.3.1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 63 - Ficheros analizados SonarQube PHP CWE79

- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

  Durante este escaneo se han analizado 228 ficheros. El listado de ficheros se encuentra en el Anexo A.3.2 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 64 - Ficheros analizados SonarQube PHP CWE89

- **CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.3 CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

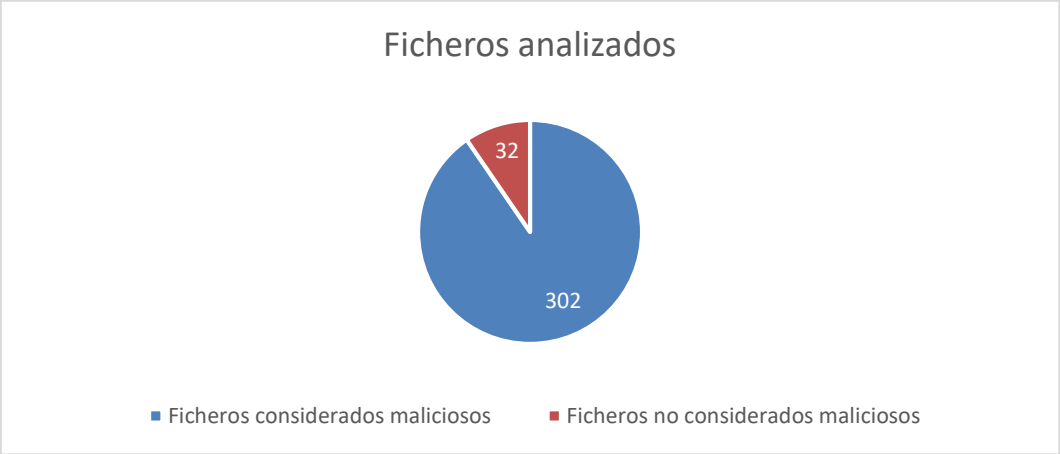

Figura 65 - Ficheros analizados SonarQube PHP CWE90

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-521: Weak Password Requirements

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

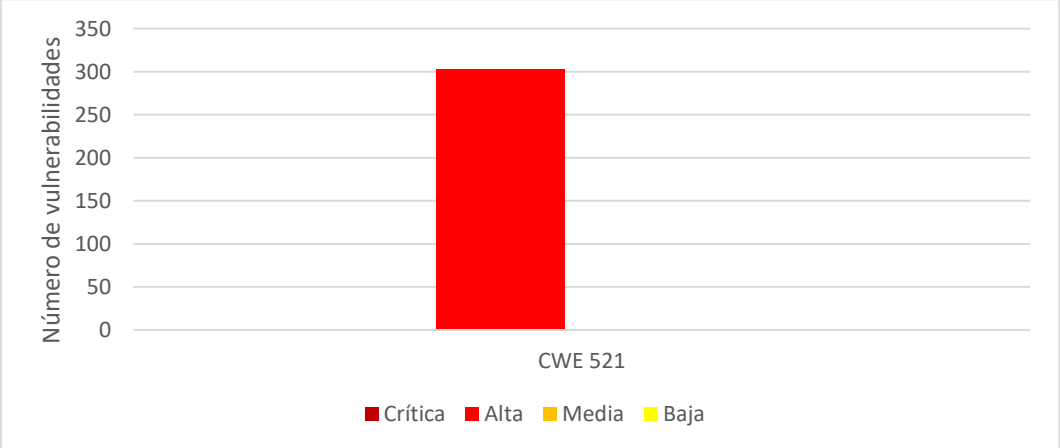

Figura 66 - CWEs por criticidad SonarQube PHP CWE90

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0

FP	302	Recall	0
FN	334	F-measure	N/A

Tabla 21 - Medidas obtenidas SonarQube PHP CWE90

- **CWE-91: XML Injection (aka Blind XPath Injection)**

  Durante este escaneo se han analizado 266 ficheros. El listado de ficheros se encuentra en el Anexo A.3.4 CWE-91: XML Injection (aka Blind XPath Injection). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 67 - Ficheros analizados SonarQube PHP CWE91

- **CWE-601: URL Redirection to Untrusted Site ('Open Redirect')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.5 CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 68 - Ficheros analizados SonarQube PHP CWE601

Como se puede observar a partir de las figuras 63-68, SonarQube ha conseguido detectar, a partir de su plugin para PHP, únicamente vulnerabilidades en uno de los cinco conjuntos de

muestra utilizados para analizar este lenguaje. Para el resto de conjuntos de muestra ha detectado *Code Smells*, pero no vulnerabilidades de seguridad.

VisualCodeGreeper

La última herramienta a analizar será VisualCodeGreeper. Esta herramienta está fundamentalmente pensada para escaneos de equipos pequeños debido a su poca escalabilidad:

Lenguaje analizado: C++

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**
  Durante este escaneo se han analizado 82 ficheros. El listado de ficheros se encuentra en el Anexo A.1.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

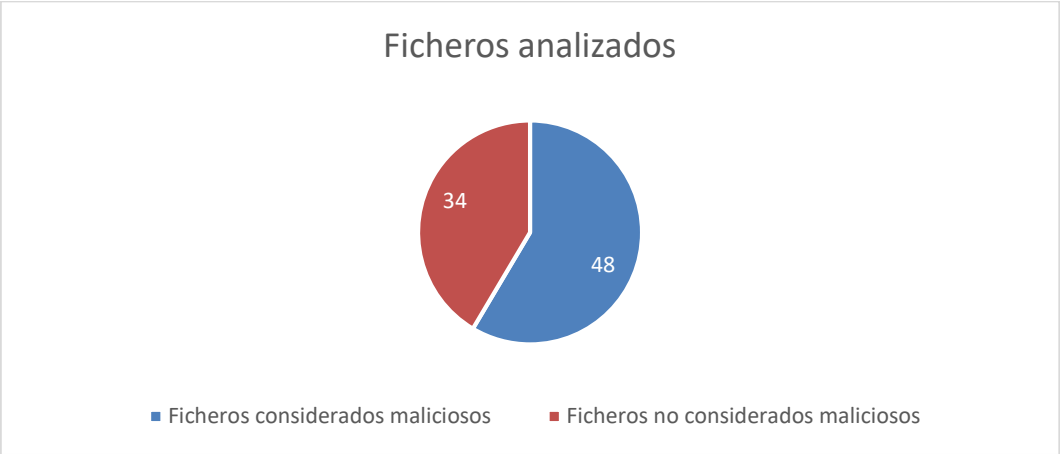

Figura 69 - Ficheros analizados VisualCodeGreeper C++ CWE15

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-121: Stack-based Buffer Overflow

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

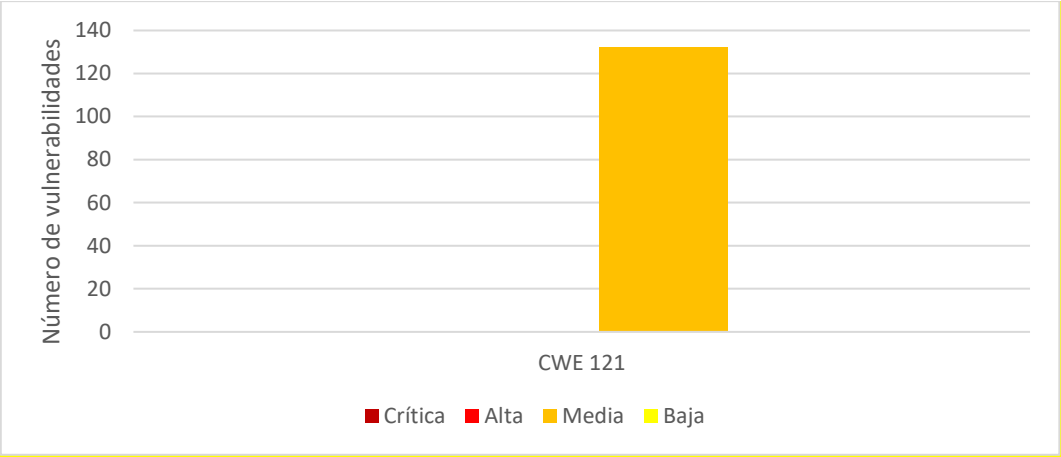

Figura 70 - CWEs por criticidad VisualCodeGreeper C++ CWE15

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	132	Recall	0
FN	82	F-measure	N/A

Tabla 22 - Medidas obtenidas VisualCodeGreeper C++ CWE15

- **CWE-23: Relative Path Traversal**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

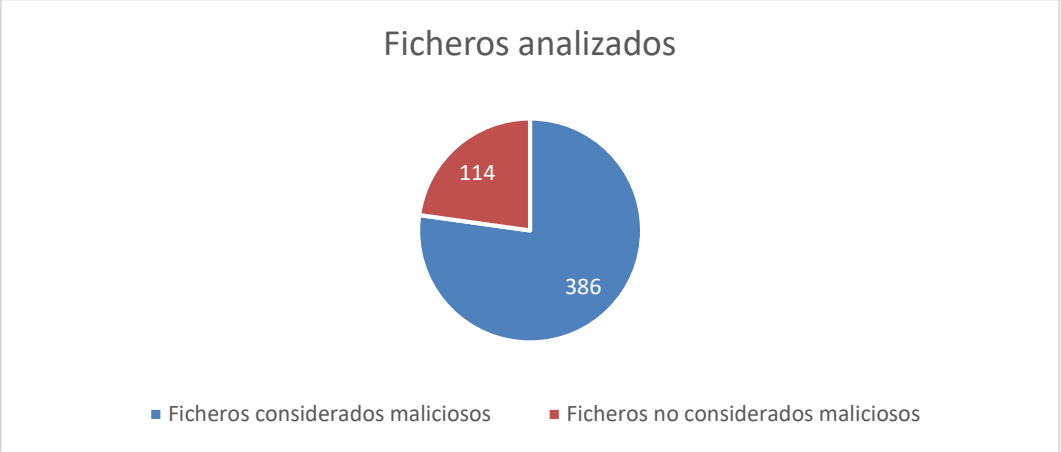

Figura 71 - Ficheros analizados VisualCodeGreeper C++ CWE23

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-23: Relative Path Traversal
  - CWE-121: Stack-based Buffer Overflow
  - CWE-190: Integer Overflow or Wraparound

- CWE-710: Improper Adherence to Coding Standards

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

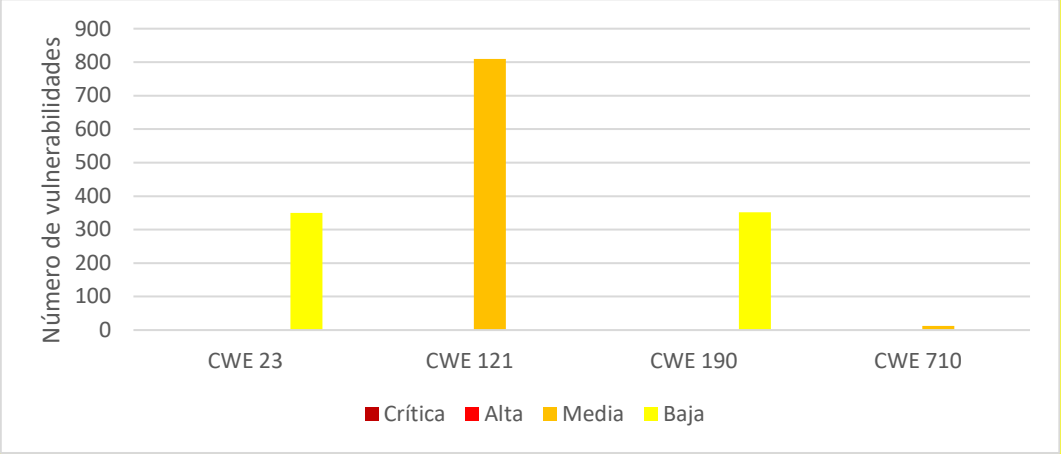

Figura 72 - CWEs por criticidad VisualCodeGreeper C++ CWE23

De estos resultados se obtienen las siguientes métricas:

TP	350	Precision	0,229658793
FP	1174	Recall	0,7
FN	150	F-measure	0,345849802

Tabla 23 - Medidas obtenidas VisualCodeGreeper C++ CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 164 ficheros. El listado de ficheros se encuentra en el Anexo A.1.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

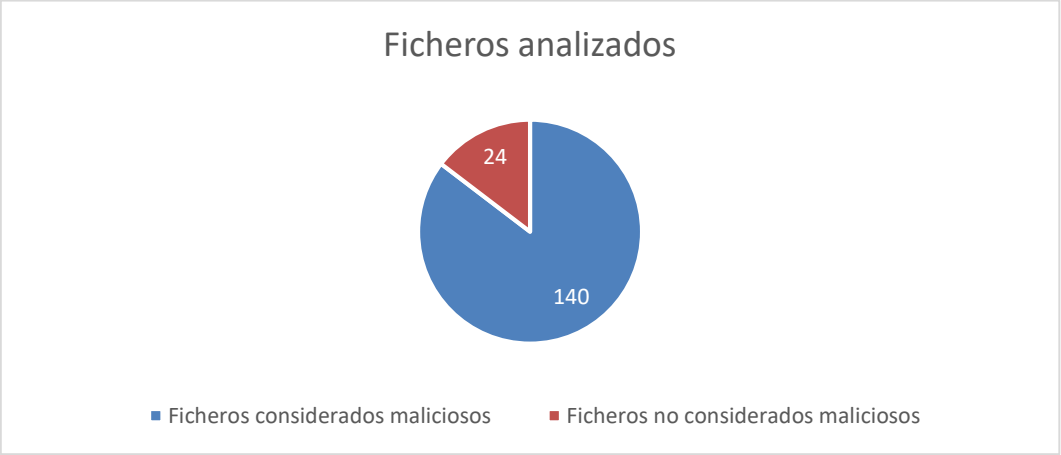

Figura 73 - Ficheros analizados VisualCodeGreeper C++ CWE259

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-121: Stack-based Buffer Overflow
- CWE-190: Integer Overflow or Wraparound
- CWE-615: Information Exposure Through Comments
- CWE-710: Improper Adherence to Coding Standards

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

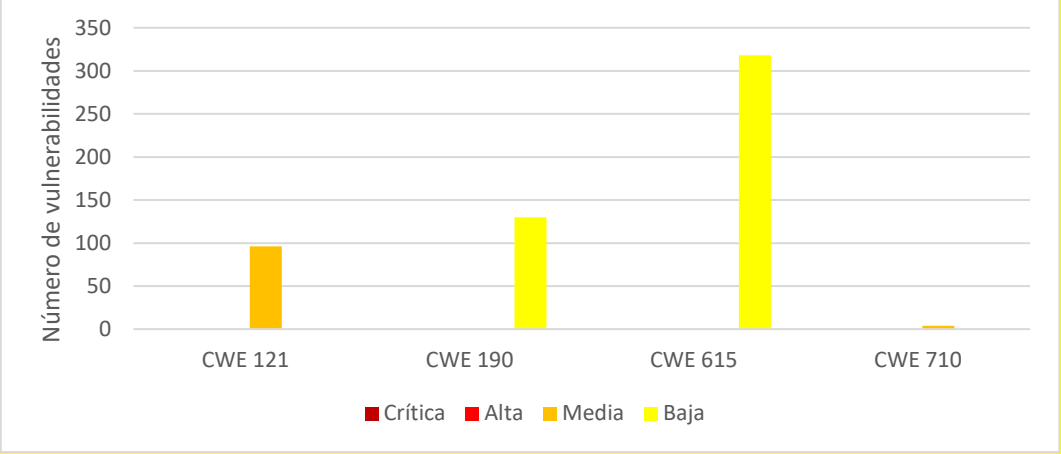

Figura 74 - CWEs por criticidad VisualCodeGreeper C++ CWE259

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	548	Recall	0
FN	164	F-measure	N/A

Tabla 24 - Medidas obtenidas VisualCodeGreeper C++ CWE259

- **CWE-369: Divide By Zero**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

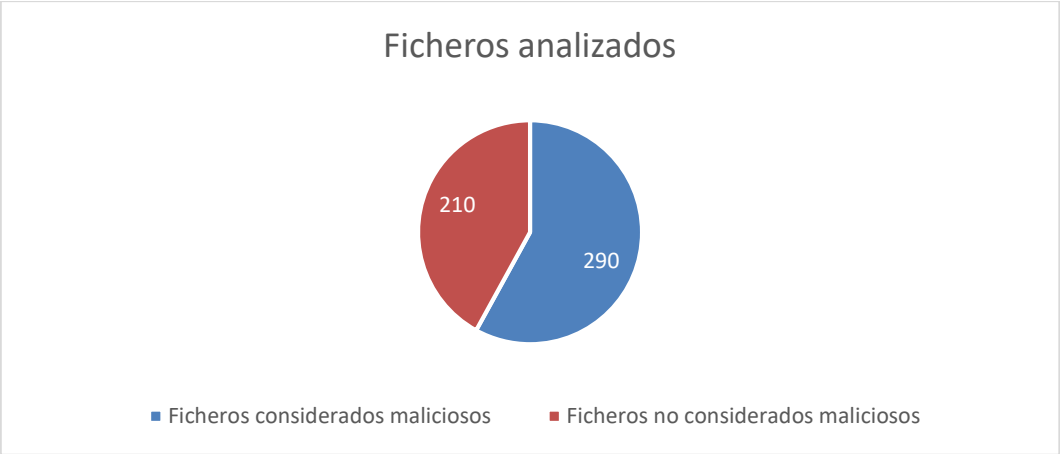

Figura 75 - Ficheros analizados VisualCodeGreeper C++ CWE369

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-121: Stack-based Buffer Overflow
- CWE-615: Information Exposure Through Comments
- CWE-710: Improper Adherence to Coding Standards

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

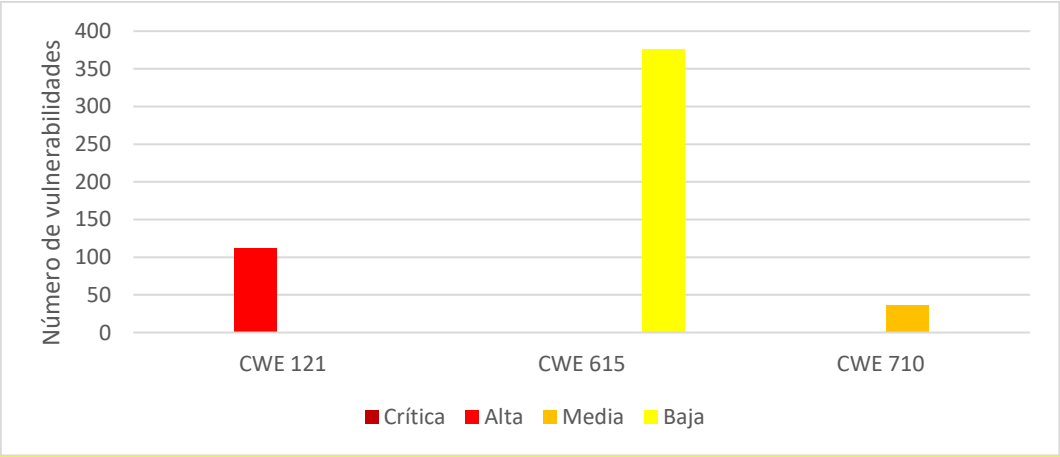

Figura 76 - CWEs por criticidad VisualCodeGreeper C++ CWE369

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	525	Recall	0
FN	500	F-measure	N/A

Tabla 25 - Medidas obtenidas VisualCodeGreeper C++ CWE369

- **CWE-404: Improper Resource Shutdown or Release**

Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.1.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

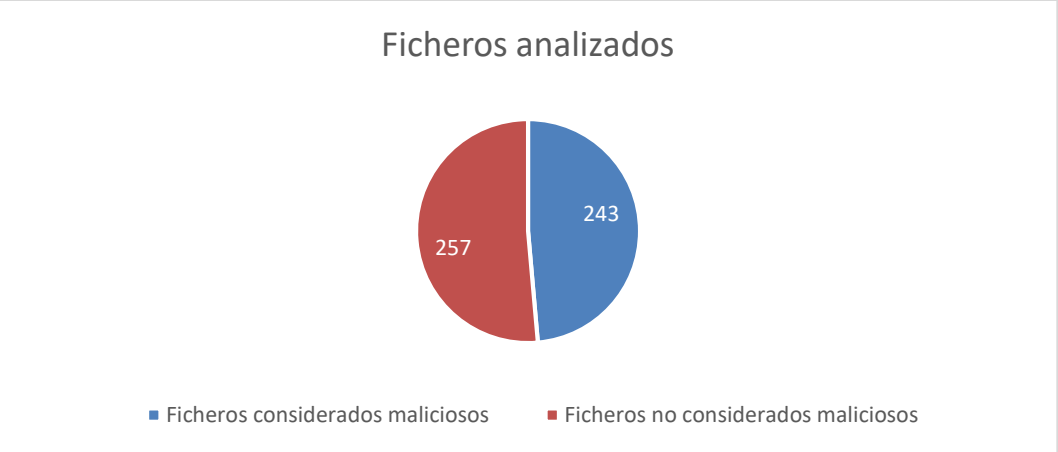

Figura 77 - Ficheros analizados VisualCodeGreeper C++ CWE404

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-23: Relative Path Traversal
- CWE-710: Improper Adherence to Coding Standards

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

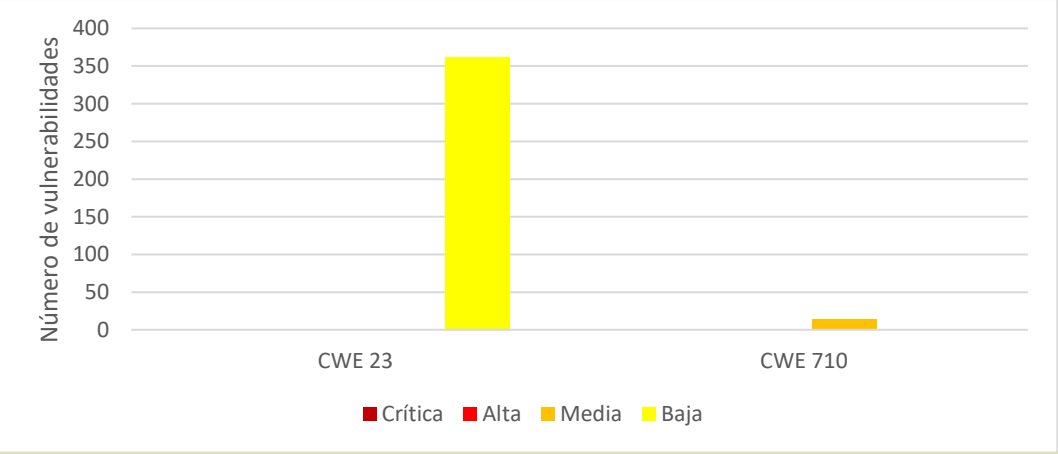

Figura 78 - CWEs por criticidad VisualCodeGreeper C++ CWE404

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	376	Recall	0
FN	500	F-measure	N/A

Tabla 26 - Medidas obtenidas VisualCodeGreeper C++ CWE404

Lenguaje analizado: Java

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-15: External Control of System or Configuration Setting**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.1 CWE-15: External Control of System or Configuration Setting. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 79 - Ficheros analizados VisualCodeGreeper Java CWE15

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-20: Improper Input Validation
- CWE-112: Missing XML Validation
- CWE-190: Integer Overflow or Wraparound
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-495: Private Data Structure Returned From A Public Method
- CWE-615: Information Exposure Through Comments

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

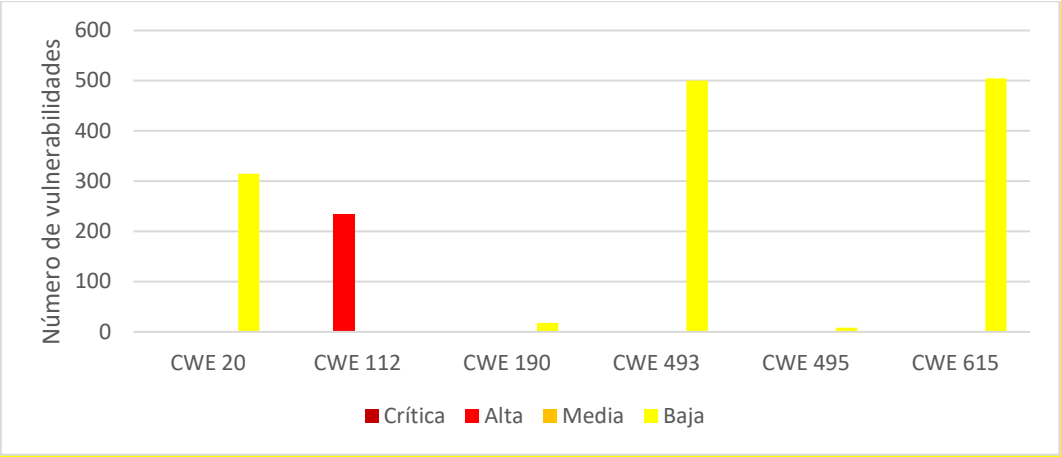

Figura 80 - CWEs por criticidad VisualCodeGreeper Java CWE15

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	1580	Recall	0
FN	500	F-measure	N/A

Tabla 27 - Medidas obtenidas VisualCodeGreeper Java CWE15

- **CWE-23: Relative Path Traversal**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.2 CWE-23: Relative Path Traversal. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 81 - Ficheros analizados VisualCodeGreeper Java CWE23

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-20: Improper Input Validation
  - CWE-112: Missing XML Validation
  - CWE-190: Integer Overflow or Wraparound

- CWE-493: Critical Public Variable Without Final Modifier
- CWE-495: Private Data Structure Returned From A Public Method
- CWE-615: Information Exposure Through Comments

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

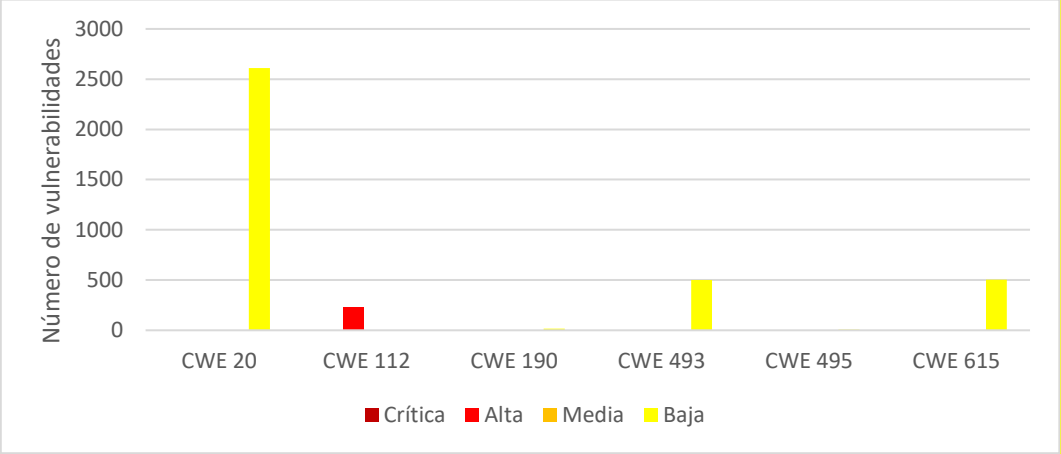

Figura 82 - CWEs por criticidad VisualCodeGreeper Java CWE233

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	3875	Recall	0
FN	500	F-measure	N/A

Tabla 28 - Medidas obtenidas VisualCodeGreeper Java CWE23

- **CWE-259: Use of Hard-coded Password**

  Durante este escaneo se han analizado 180 ficheros. El listado de ficheros se encuentra en el Anexo A.2.3 CWE-259: Use of Hard-coded Password. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 83 - Ficheros analizados VisualCodeGreeper Java CWE259

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-20: Improper Input Validation
- CWE-190: Integer Overflow or Wraparound
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-495: Private Data Structure Returned From A Public Method
- CWE-615: Information Exposure Through Comments

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

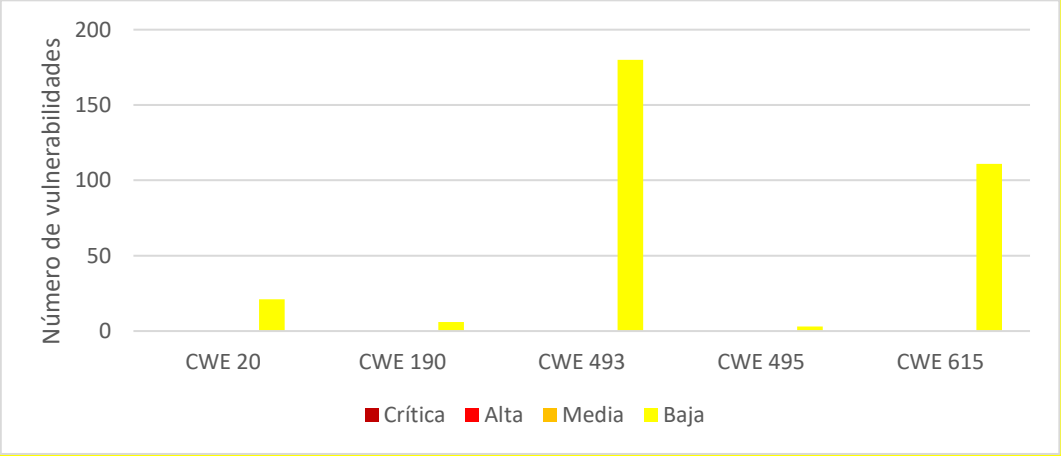

Figura 84 - CWEs por criticidad VisualCodeGreeper Java CWE259

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	321	Recall	0
FN	180	F-measure	N/A

Tabla 29 - Medidas obtenidas VisualCodeGreeper Java CWE259

- **CWE-369: Divide By Zero**

  Durante este escaneo se han analizado 500 ficheros. El listado de ficheros se encuentra en el Anexo A.2.4 CWE-369: Divide By Zero. A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 85 - Ficheros analizados VisualCodeGreeper Java CWE369

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-20: Improper Input Validation
- CWE-190: Integer Overflow or Wraparound
- CWE-493: Critical Public Variable Without Final Modifier
- CWE-495: Private Data Structure Returned From A Public Method
- CWE-615: Information Exposure Through Comments

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

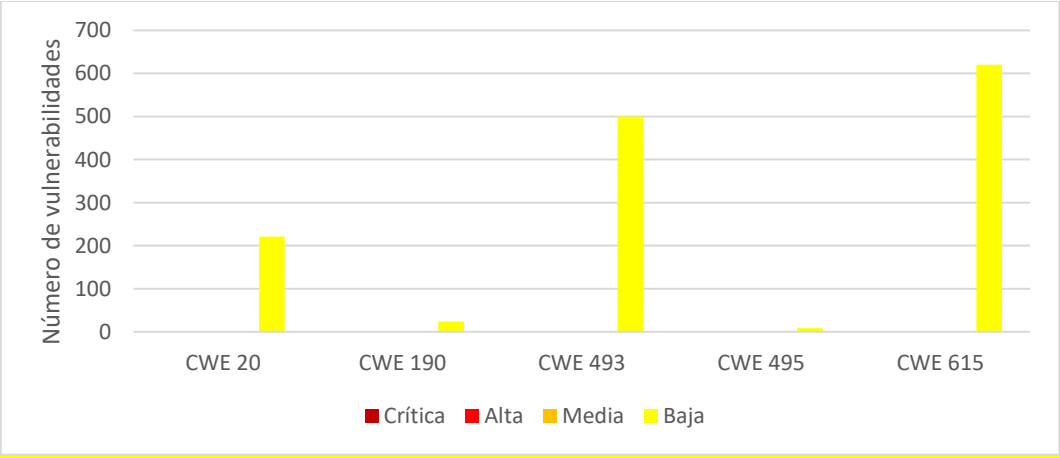

Figura 86 - CWEs por criticidad VisualCodeGreeper Java CWE369

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	1373	Recall	0
FN	500	F-measure	N/A

Tabla 30 - Medidas obtenidas VisualCodeGreeper Java CWE369

- **CWE-404: Improper Resource Shutdown or Release**

  Durante este escaneo se han analizado 5 ficheros. El listado de ficheros se encuentra en el Anexo A.2.5 CWE-404: Improper Resource Shutdown or Release. A continuación, se muestran los resultados obtenidos de forma gráfica.

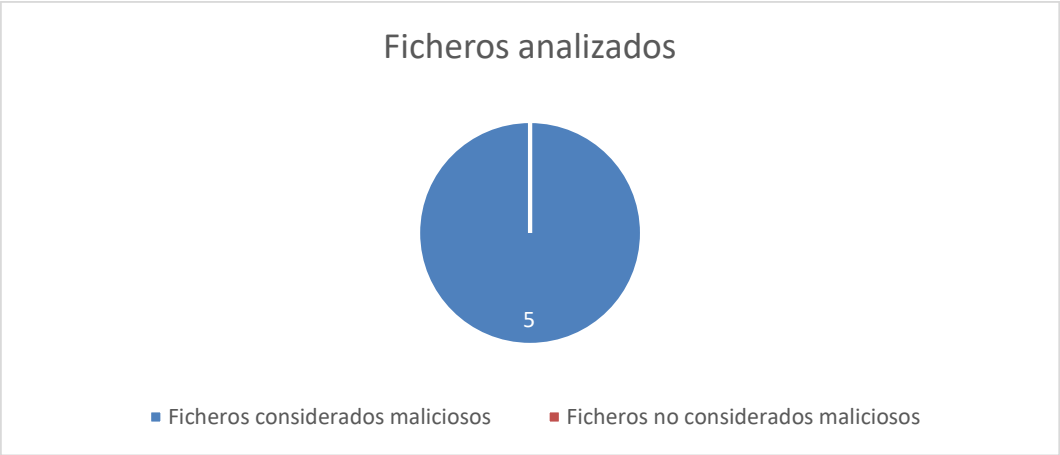

Figura 87 - Ficheros analizados VisualCodeGreeper Java CWE404

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-20: Improper Input Validation
  - CWE-190: Integer Overflow or Wraparound
  - CWE-493: Critical Public Variable Without Final Modifier

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

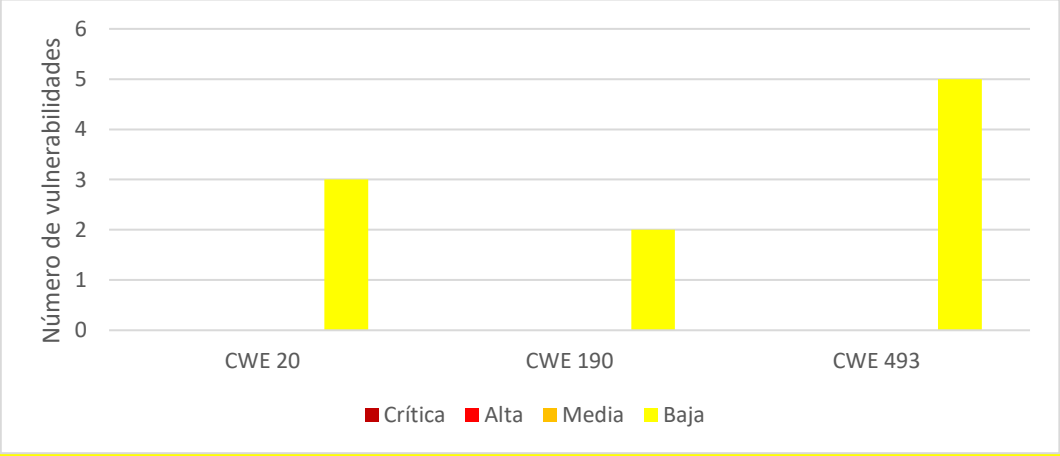

Figura 88 - CWEs por criticidad VisualCodeGreeper Java CWE404

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	10	Recall	0
FN	5	F-measure	N/A

Tabla 31 - Medidas obtenidas VisualCodeGreeper Java CWE404

Lenguaje analizado: PHP

Para este lenguaje se han analizado 5 CWEs diferentes, obteniendo los siguientes resultados:

- **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**

  Durante este escaneo se han analizado 499 ficheros. El listado de ficheros se encuentra en el Anexo A.3.1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 89 - Ficheros analizados VisualCodeGreeper PHP CWE79

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-94: Improper Control of Generation of Code ('Code Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

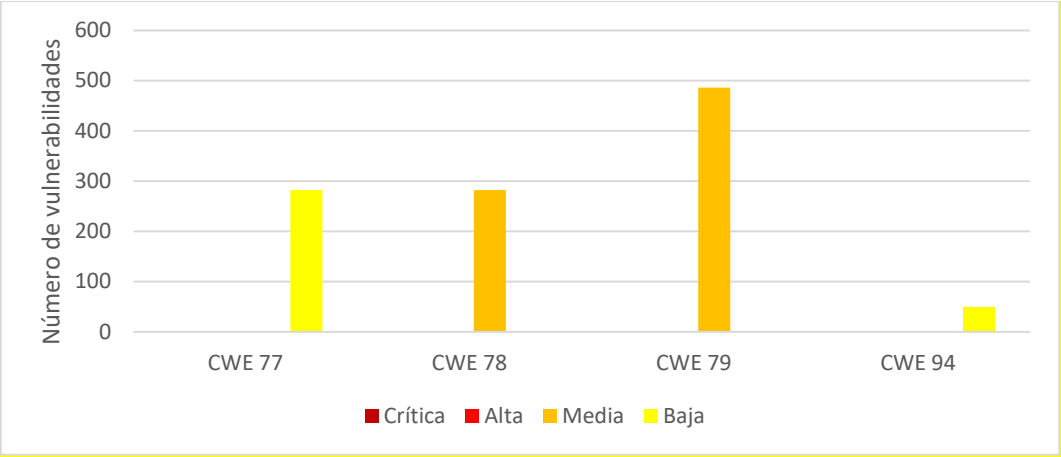

Figura 90 - CWEs por criticidad VisualCodeGreeper PHP CWE79

De estos resultados se obtienen las siguientes métricas:

TP	486	Precision	0,441818182
FP	614	Recall	0,973947896
FN	13	F-measure	0,607879925

Tabla 32 - Medidas obtenidas VisualCodeGreeper PHP CWE79

- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

  Durante este escaneo se han analizado 228 ficheros. El listado de ficheros se encuentra en el Anexo A.3.2 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 91 - Ficheros analizados VisualCodeGreeper PHP CWE89

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-94: Improper Control of Generation of Code ('Code Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

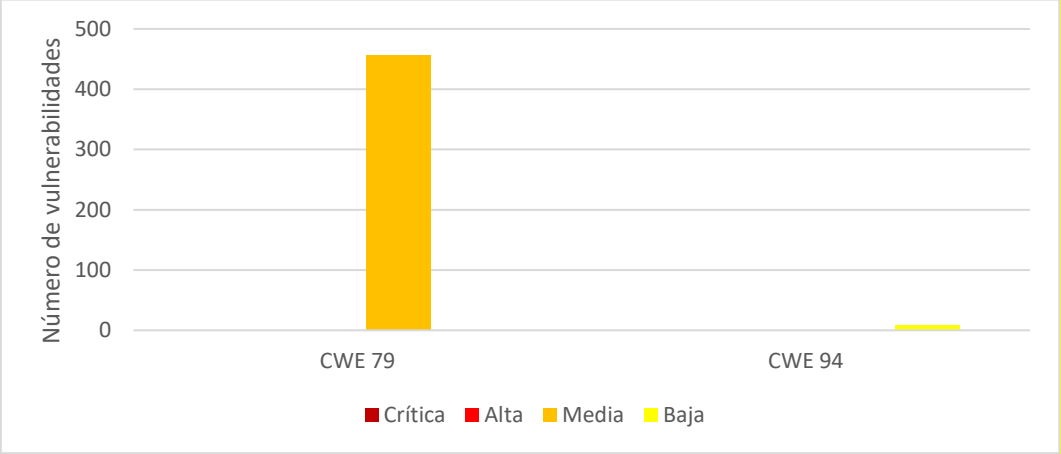

Figura 92 - CWEs por criticidad VisualCodeGreeper PHP CWE89

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	464	Recall	0
FN	228	F-measure	N/A

Tabla 33 - Medidas obtenidas VisualCodeGreeper PHP CWE89

- **CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.3 CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection'). A continuación, se muestran los resultados obtenidos de forma gráfica.

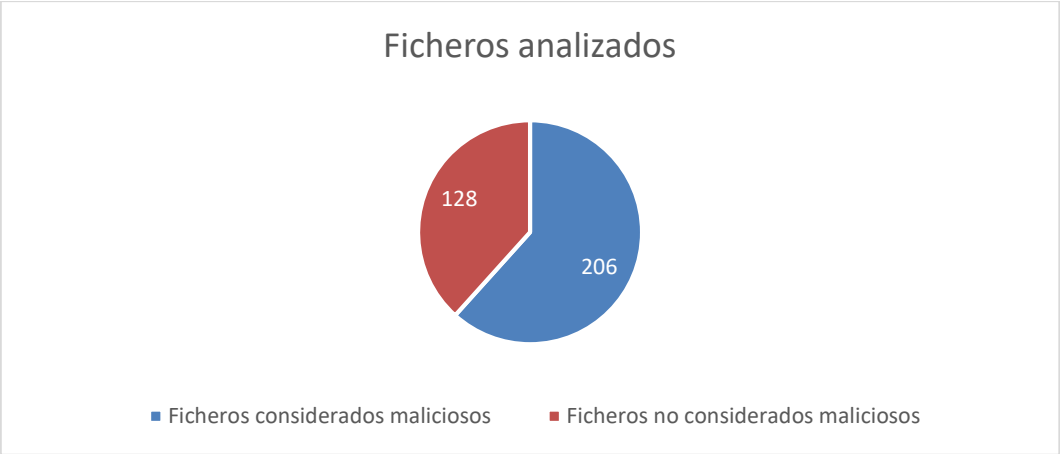

Figura 93 - Ficheros analizados VisualCodeGreeper PHP CWE90

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-94: Improper Control of Generation of Code ('Code Injection')
- CWE-377: Insecure Temporary File

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

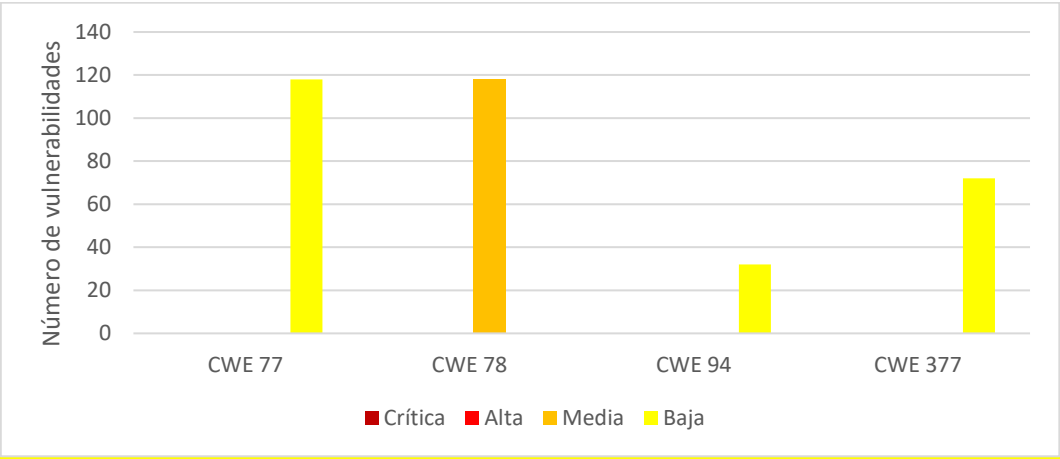

Figura 94 - CWEs por criticidad VisualCodeGreeper PHP CWE90

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	340	Recall	0

FN	334	F-measure	N/A

Tabla 34 - Medidas obtenidas VisualCodeGreeper PHP CWE90

- **CWE-91: XML Injection (aka Blind XPath Injection)**

  Durante este escaneo se han analizado 266 ficheros. El listado de ficheros se encuentra en el Anexo A.3.4 CWE-91: XML Injection (aka Blind XPath Injection). A continuación, se muestran los resultados obtenidos de forma gráfica.

Figura 95 - Ficheros analizados VisualCodeGreeper PHP CWE91

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

- CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')
- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-94: Improper Control of Generation of Code ('Code Injection')

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

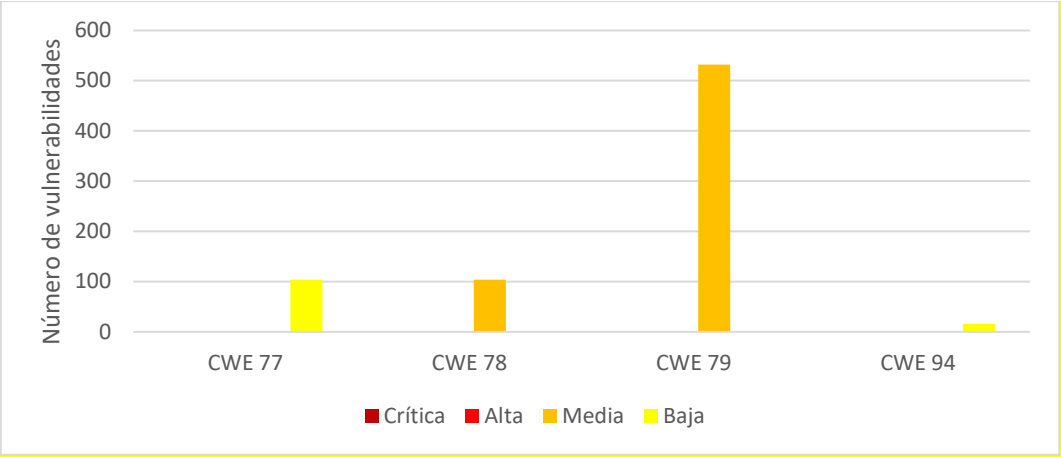

Figura 96 - CWEs por criticidad VisualCodeGreeper PHP CWE91

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	756	Recall	0
FN	266	F-measure	N/A

Tabla 35 - Medidas obtenidas VisualCodeGreeper PHP CWE91

- **CWE-601: URL Redirection to Untrusted Site ('Open Redirect')**

  Durante este escaneo se han analizado 334 ficheros. El listado de ficheros se encuentra en el Anexo A.3.5 CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). A continuación, se muestran los resultados obtenidos de forma gráfica.

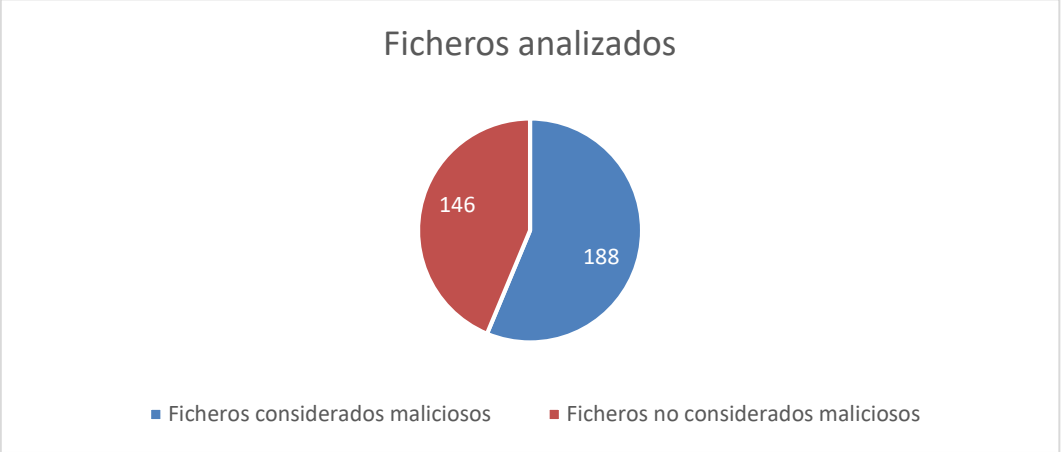

Figura 97 - Ficheros analizados VisualCodeGreeper PHP CWE601

Dentro del conjunto de vulnerabilidades encontradas, se han detectado los siguientes CWE:

  - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')

- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-94: Improper Control of Generation of Code ('Code Injection')
- CWE-377: Insecure Temporary File

A continuación, se representa la cantidad de vulnerabilidades obtenida de cada uno de los CWE por criticidad:

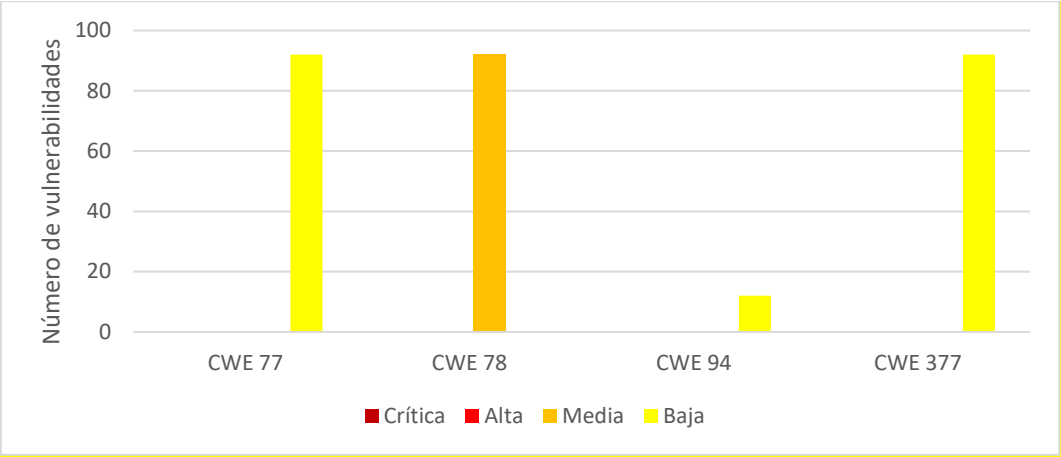

Figura 98 - CWEs por criticidad VisualCodeGreeper PHP CWE6011

De estos resultados se obtienen las siguientes métricas:

TP	0	Precision	0
FP	288	Recall	0
FN	334	F-measure	N/A

Tabla 36 - Medidas obtenidas VisualCodeGreeper PHP CWE601

Capítulo VI: Discusión de resultados

A lo largo de este capítulo, se presentará una discusión de los resultados mostrados en el capítulo anterior en la que se realizarán las comparativas más destacables de los resultados obtenidos, así como ventajas e inconvenientes experimentados durante el uso de las diferentes herramientas.

Una vez realizados los diferentes análisis, se pueden hacer varias comparativas entre las diferentes herramientas, así como entre los diferentes lenguajes:

- Comparativa entre la precisión y el *recall* medio de cada herramienta.
- Comparativa entre C++ y Java.
- Resultados positivos por CWE.

A continuación, se muestra cada una de ellas:

- **Precisión y Recall medios**: A continuación, se muestra una comparativa de las métricas analizadas para cada una de las herramientas.

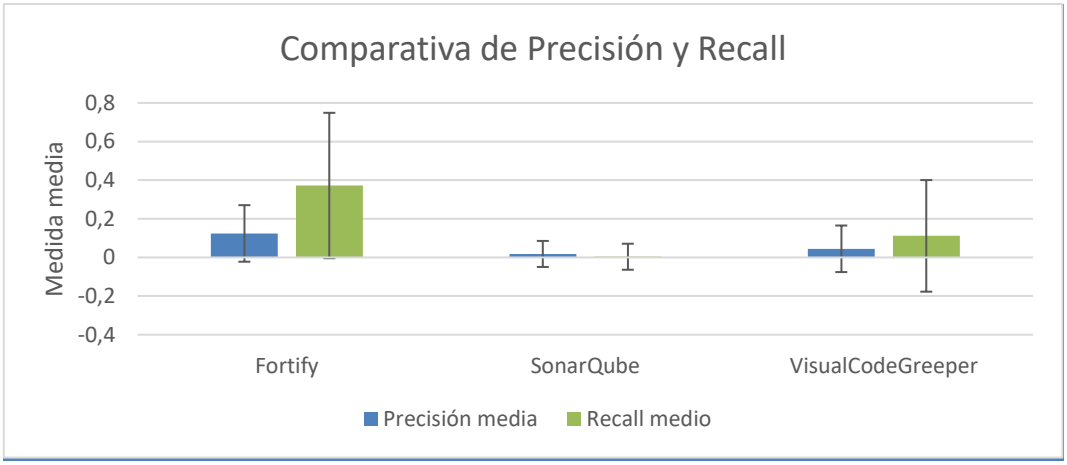

Figura 99 – Comparativa de Precisión y Recall medios de herramientas

De la gráfica anterior se puede deducir que, en los escenarios propuestos, Fortify es la herramienta con resultados de detección superiores, en términos de Precisión y *Recall*, mientras que SonarQube, en su versión *Community*, proporciona resultados cercanos a 0.

- **Comparativa de herramientas C++ vs Java**: Dado que tanto para Java como para C++ se han analizado los mismos CWE, es posible determinar en qué situaciones es mejor una herramienta u otra. A continuación, se muestra la comparativa en cuanto a precisión:

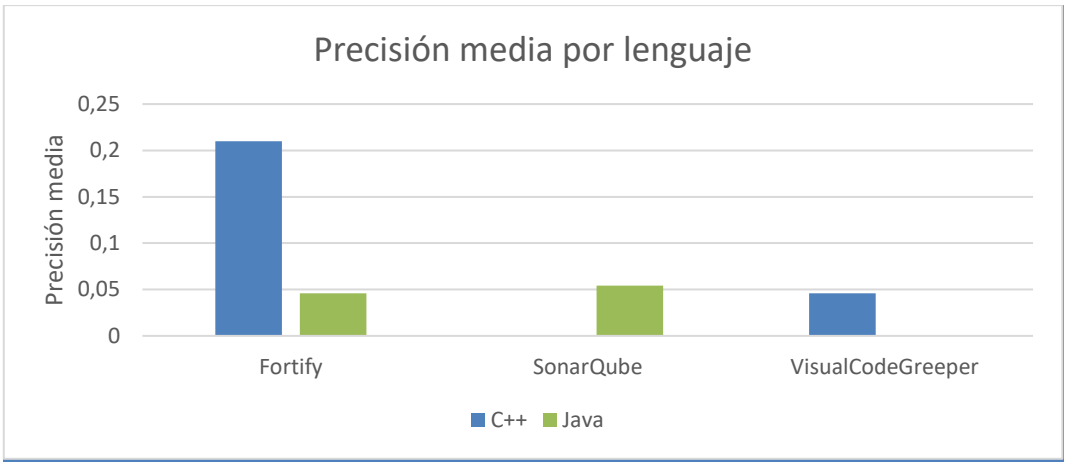

Figura 100 – Comparativa de Precisión media en Java y C++ de herramientas

Como se puede observar, a la hora de analizar C++, Fortify es la alternativa con mayor precisión en sus resultados, mientras que el caso de Java, es SonarQube (con el *plugin* FindSecBugs) la que ofrece resultados más precisos. En caso de disponer únicamente de herramientas de software libre, las alternativas evidentes serían: VCG para C++ y SonarQube para Java.

Realmente, si luego se analiza la variable *Recall* en Java para Fortify y SonarQube, se obtiene la siguiente gráfica:

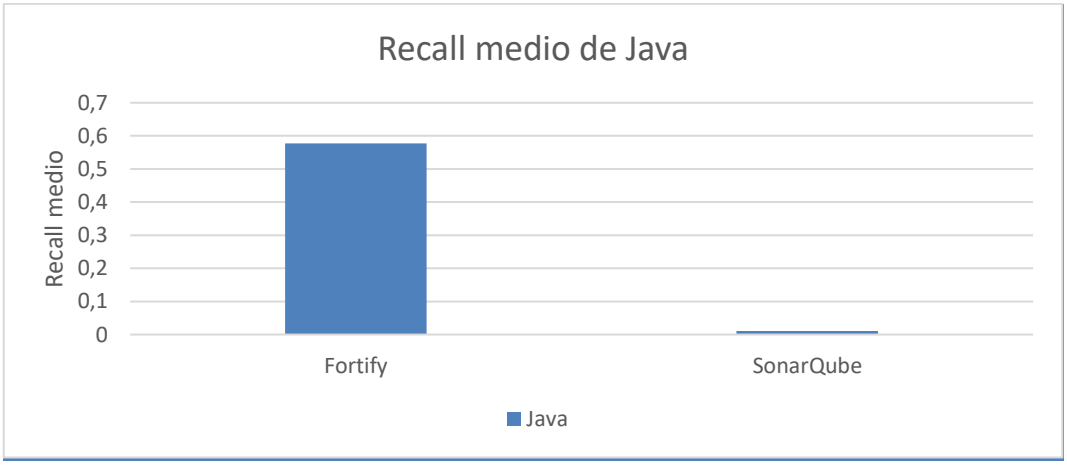

Figura 101 – Comparativa de Recall medio en Java de Fortify vs SonarQube

Como se puede observar, aunque Fortify es menos preciso, obtiene mayor ratio de verdaderos positivos (TP), lo cual lleva a la siguiente conclusión: Fortify proporciona gran cantidad de Falsos positivos, lo cual hace que disminuya su precisión. Por otro lado, encuentra más positivos que SonarQube, lo cual hace que sea más fiable.

- **Resultados positivos de seguridad obtenidos por CWE**: En la siguiente imagen, se muestra la cantidad de análisis realizados en los que la herramienta ha detectado al menos 1 vulnerabilidad del CWE indicado.

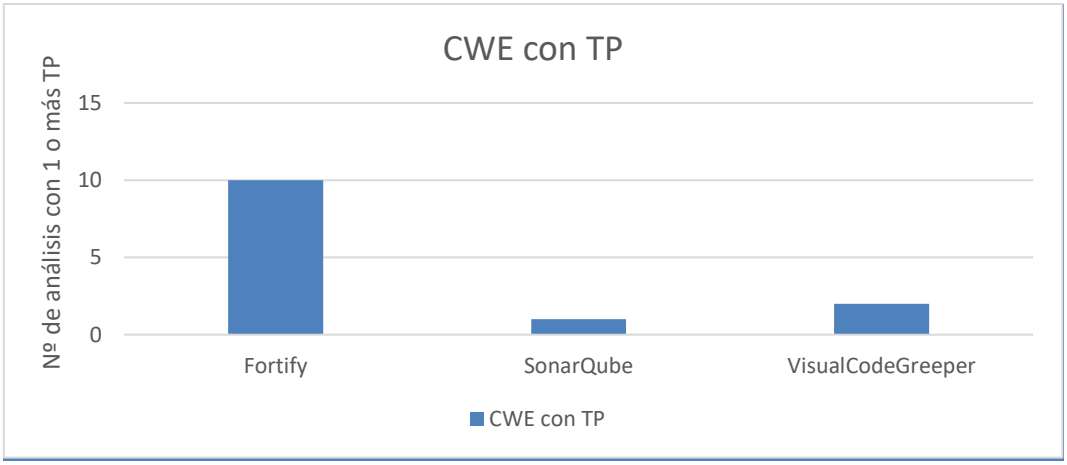

Figura 102 – Comparativa de detección del CWE

Tras analizar los resultados obtenidos, se pueden obtener, de forma clara, las siguientes conclusiones:

- Fortify es el software con mejores resultados en general, tanto en precisión como en *recall*.
- SonarQube, en su versión *Community*, es el software de los 3 que menos resultados ha obtenido, obteniendo resultados únicamente con su plugin de Java.
- VisualCodeGreeper, aunque es el software menos escalable de los 3, ha sacado resultados concisos. Es una herramienta que aún necesita mayor madurez, pero que en equipos pequeños podría funcionar.

Otras conclusiones también obtenidas a lo largo del experimento y que no se ven a través de los resultados obtenidos son:

- Es posible que el repositorio Juliet (utilizado para los lenguajes C y Java) posea más vulnerabilidades de las que indica su documentación, la cual dice que únicamente se debe obtener 1 resultado por fichero. Comprobando los resultados obtenidos por las herramientas se han encontrado nuevas vulnerabilidades reales en los ficheros, además de las que indica la documentación de Juliet. Se ha contactado con SAMATE para comentar este hecho y que se actualice para la siguiente versión.
- PHP ha sido el lenguaje con resultados de detección inferiores, aunque tenía los CWE más comunes, las herramientas han confundido el CWE en la mayoría de los casos.

Capítulo VII: Conclusiones y líneas de trabajo futuro

Como se ha observado durante el experimento, la respuesta a la pregunta realizada inicialmente no es sencilla. El mundo de las herramientas SAST está lleno de posibilidades y no existe una solución trivial a la hora de realizar un análisis de código.

Durante el desarrollo de este experimento se ha:

- Realizado una introducción al mundo SAST, explicando conceptos básicos del análisis estático de código y explicando las principales problemáticas que tiene SAST en la actualidad.
- Llevado a cabo un estudio del estado del arte de las herramientas SAST, así como de las métricas utilizadas en técnicas de *benchmarking* de herramientas de análisis de vulnerabilidades, permitiendo obtener un conjunto de herramientas, un conjunto de muestras y un conjunto de métricas a utilizar para la realización de un piloto.
- Implementado un entorno de laboratorio en el que se han desplegado cada una de las herramientas estudiadas con el fin de analizar las muestras de los CWE seleccionados para cada uno de los lenguajes.
- Desarrollado un análisis de los resultados obtenidos del entorno con el fin de recoger un conjunto de conclusiones del experimento. Dichas conclusiones han sido presentadas con su debida justificación basada en los resultados.

Este trabajo pretende aportar una aproximación inicial al estudio del *benchmarking* de herramientas SAST. Como conclusión, en base a los resultados obtenidos, se puede determinar que el mundo del software libre está aún lejos de alcanzar la variedad de lenguajes que cubre el software privativo, permitiendo este último mayores capacidades no solo de resultados, sino también a nivel de arquitectura.

Por tanto, la pregunta inicial que se realizaba al comienzo de este estudio queda resuelta, aún no se puede confiar al 100% en los resultados obtenidos por las herramientas SAST. Aunque, dentro de lo que se encuentra en la actualidad, el mundo privativo ofrece mejores soluciones.

Otros trabajos futuros que podrían hacerse a partir de este estudio serían:

- Creación de una distribución de laboratorio al estilo Kali Linux orientada al análisis SAST.
- Desarrollo de una herramienta/librería de *benchmarking* para aplicaciones SAST.
- Ampliar el número de vulnerabilidades.
- Ampliar el número de lenguajes.
- Ampliar el número de herramientas.

- Realizar pruebas con aplicaciones reales que ya han sido analizadas, como es el caso de Asterisk, cuyas vulnerabilidades están definidas dentro del repositorio SARD de SAMATE.

Referencias bibliográficas

Antunes, N., & Vieira, M. (2010). Benchmarking Vulnerability Detection Tools for Web Services. *IEEE International Conference on Web Services*, (págs. 203-210). Coimbra.

Bermejo Higuera, J. R. (2014). *Metodología de evaluación de herramientas de análisis automático de seguridad de aplicaciones web para su adaptación en el ciclo de vida de desarrollo.* Universidad Nacional de Educación a Distancia.

Checkmarx Ltd. (2019). *Checkmarx*. Obtenido de Checkmarx – Application Security Testing and Static Code Analysis: https://www.checkmarx.com/

Common Weakness Enumeration. (20 de Junio de 2019). *CWE List Version 3.3*. Obtenido de Common Weakness Enumeration: https://cwe.mitre.org/data/

Deylami, H. M., Ardekani, I. T., Muniyandi, R. C., & Sarrafzadeh, A. (2015). Effects of Software Security on Software Development Life Cycle and. *International Journal of Computational Intelligence and Information Security*, 4-12.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*(27), 861–874.

Gartner. (2019). *Static Application Security Testing (SAST)*. Obtenido de Gartner - IT Glossary: https://www.gartner.com/it-glossary/static-application-security-testing-sast/

German, A. (2003). Software Static Code Analysis Lessons Learned. *Crosstalk*, 13-17.

HPE. (November de 2015). Obtenido de HP Fortify Source Code Analyzer: https://community.microfocus.com/dcvta86296/attachments/dcvta86296/software-security-center-v440/26/1/HP_Fortify_SCA_Install_Guide_4.40.pdf

McGraw, G. (2005). *Software Security: Building Security In.* Massachusetts: Addison Wesley Professional.

Microfocus S.L. (2019). *Microfocus*. Obtenido de Fortify Static Code Analysis Tool: Static Application Security Testing | Microfocus: https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

National Institute of Standards and Technology. (s.f.). *Introduction to SAMATE*. Obtenido de National Institute of Standards and Technology: https://samate.nist.gov/index.php/Introduction_to_SAMATE.html

National Institute of Standards and Technology. (s.f.). *Test Suites*. Obtenido de National Institute of Standards and Technology: https://samate.nist.gov/SARD/testsuite.php

NCC Group. (3 de Mayo de 2016). *VisualCodeGrepper - Code security scanning tool.* Obtenido de GitHub - nccgroup/VCG: VisualCodeGrepper - Code security scanning tool.: https://github.com/nccgroup/VCG

npdunn. (26 de 11 de 2016). *Sourceforge*. Obtenido de VisualCodeGrepper V2.1.0: https://sourceforge.net/projects/visualcodegrepp/

Ospina Delgado, J. P. (2015). *Análisis de seguridad y calidad de aplicaciones (Sonarqube).* Manizales: Universidad Oberta de Catalunya.

OWASP. (6 de Mayo de 2009). *OWASP*. Obtenido de Dynamic Analysis - OWASP: https://www.owasp.org/index.php/Dynamic_Analysis

OWASP. (3 de Agosto de 2016). *OWASP*. Obtenido de Security by Design Principles - OWASP: https://www.owasp.org/index.php/Security_by_Design_Principles

OWASP. (2 de Junio de 2019). *OWASP Top Ten Project*. Obtenido de OWASP: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

OWASP. (13 de Mayo de 2019). *Static Code Analysis*. Obtenido de OWASP: https://www.owasp.org/index.php/Static_Code_Analysis

SonarSource. (2019). *SonarQube*. Obtenido de Code Quality and Security | SonarQube: https://www.sonarqube.org/

SonarSource. (19 de June de 2019). *SonarQube*. Obtenido de SonarQube 7.8 | SonarQube: https://www.sonarqube.org/sonarqube-7-8/

SonarSource. (2019). *SonarQube*. Obtenido de SonarScanner | SonarQube Docs: https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/

Sotirov, A. I. (2005). *Automatic Vulnerability Detection Using Static Source Code Analysis.* Tuscaloosa: University of Alabama Press.

Veracode. (2019). *Veracode*. Obtenido de Use Veracode to secure the applications you build, buy, & manage: https://www.veracode.com/

Wögerer, W. (2005). *A Survey of Static Program Analysis Techniques.* Viena: Technische Universität Wien.

Anexo A: Listados de ficheros por lenguaje

En este apartado se indicarán los ficheros utilizados durante los análisis, con el fin de permitir la reproducción del experimento.

Dada la volumetría de estos documentos, se ha optado por incluir estos listados en ficheros disponibles de forma pública en internet (en modo solo lectura), ya que solo son necesarios en caso de querer reproducir el experimento.

A.1 C++

A.1.1 CWE-15: External Control of System or Configuration Setting

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/13Trh44L_6fLt68RCEAPCgde2ahZMQ6lw9HQKbgEjKgI/edit?usp=sharing

A.1.2 CWE-23: Relative Path Traversal

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1NxYfkIflqH3S5ODxXZ0cUHYlpL9rEpXPasPv_iJDzvo/edit?usp=sharing

A.1.3 CWE-259: Use of Hard-coded Password

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1B6sxIHKTk9ND_gZph-oRGscslt7NAtfjC-gph1pAkjY/edit?usp=sharing

A.1.4 CWE-369: Divide By Zero

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/14ZGEvpFZB4CywUx5cDoV0sVeuLNsSovwm6eXf5e0w-4/edit?usp=sharing

A.1.5 CWE-404: Improper Resource Shutdown or Release

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1O6ctjwe1Oa5kDmfuuGrrODUn1UpjecIqQ53T2zQdGsU/edit?usp=sharing

A.2 Java

A.2.1 CWE-15: External Control of System or Configuration Setting

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1oQ8pOknPcVwlJeZbOfKESWUFSE82g_N972Z3iaKJHOc/edit?usp=sharing

A.2.2 CWE-23: Relative Path Traversal

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1qsr1mLVcJOKNhrAQ39KKdRs-vz7jFzl5oBh0KDsJCNs/edit?usp=sharing

A.2.3 CWE-259: Use of Hard-coded Password

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1Z_FZUDRrCPwoHhvLgroT3uhS5fuZUrsnvGp2MyyOBIY/edit?usp=sharing

A.2.4 CWE-369: Divide By Zero

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1s_p8ZqXmR1GIEHPyRZbNycToC3XGPxCGtreEqciMrEo/edit?usp=sharing

A.2.5 CWE-404: Improper Resource Shutdown or Release

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1-gAz-kEBLYrID8ybhDw9Uz3Wu2AjfEGM-4YyTz_uOq8/edit?usp=sharing

A.3 PHP

A.3.1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1-p8m2-g1T5rq5yimOuydB-y62VmkQz6wy_aCo-Fzp-8/edit?usp=sharing

A.3.2 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1nDuin-EHkRChSmloyQtbkKAX7oPb0PH5oNTw52MUr6c/edit?usp=sharing

A.3.3 CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1uGbxeFOb8iMj-iThY48mKX5XLj9iIOC5WI926Lyj160/edit?usp=sharing

A.3.4 CWE-91: XML Injection (aka Blind XPath Injection)

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1BD-B-OumtP_ZRxf0QhMdgjD4JpklwYhBSzoGxbp8Fuw/edit?usp=sharing

A.3.5 CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

El listado de ficheros analizados se encuentra disponible en el siguiente enlace:

https://docs.google.com/document/d/1PJgFOlr071KK3PDCKNsfHMKVZgNlXbDIwgiabEtNkk4/edit?usp=sharing