Metodología de benchmark de herramientas SAST

PEC 4 – Entrega final

Miguel E. De Vega Martín

Seguridad Empresarial

Nombre Tutor/a de TF
Pau del Canto Rodrigo

Profesor/a responsable de la asignatura
Víctor García Font

Fecha Entrega
09 de Enero de 2024

Universitat Oberta de Catalunya

FICHA DEL TRABAJO FINAL

Título del trabajo:	Application Security Testing Tools
Nombre del autor:	Miguel Enrique de Vega Martín
Nombre del consultor/a:	Pau del Canto Rodrigo
Nombre del PRA:	Víctor García Font
Fecha de entrega (mm/aaaa):	09/01/2024
Titulación o programa:	Máster universitario en Ciberseguridad y Privacidad
Área del Trabajo Final:	Seguridad Empresarial
Idioma del trabajo:	Castellano
Palabras clave	SAST, DAST, Benchmark

Resumen del Trabajo

La evaluación de herramientas para pruebas de seguridad en aplicaciones es esencial cuando las empresas consideran invertir tiempo y esfuerzo en establecer procesos de seguridad en el SDLC. Por ello, es crucial determinar las mejores formas de llevar a cabo un estudio riguroso y una comparación, buscando medir y cotejar de la manera más objetiva posible. Para las organizaciones, es primordial que estas herramientas sean efectivas en la detección de vulnerabilidades, que generen pocos falsos positivos, que ofrezcan interoperabilidad con sistemas de gestión de vulnerabilidades y que proporcionen informes de alta calidad en términos de detección y recomendaciones de mitigación.

El objetivo de este TFM es establecer una metodología de pruebas basándose en otras ya existentes, extendiéndola si es necesario con parámetros de comparación pertinentes para las organizaciones. Adicionalmente, como objetivo secundario, se llevará a cabo una prueba de concepto para obtener métricas sobre aciertos, falsos positivos y falsos negativos, desarrollando un sistema que compare el resultado de una herramienta con lo esperado, alineando ambas en un formato estandarizado como el SARIF.

Abstract

Evaluating tools for security testing in applications is essential when companies consider investing time and effort in establishing security processes within the SDLC. For this reason, it's crucial to determine the best ways to conduct a thorough study and comparison, aiming to measure and contrast as objectively as possible. For organizations, it is paramount that these tools are effective in detecting vulnerabilities, produce few false positives, offer interoperability with vulnerability management

systems, and provide high-quality reports in terms of detection and mitigation recommendations.

The aim of this Master's thesis is to establish a testing methodology based on existing ones, expanding it when necessary with relevant comparison parameters for organizations. Additionally, as a secondary objective, a proof of concept will be carried out to obtain metrics on accuracy, false positives, and false negatives, developing a system that contrasts the tool's outcome with the expected results, aligning both in a standardized format such as SARIF.

Índice

1 Introducción

1.1 Contexto y justificación del Trabajo

En la sociedad actual, la dependencia de la tecnología y el software es cada vez más pronunciada y se mantiene la tendencia. Desde la comunicación cotidiana hasta el funcionamiento de infraestructuras críticas, gran parte de nuestras actividades dependen de sistemas informáticos y aplicaciones.

Asimismo, la rápida evolución de la tecnología y el software implica una introducción constante de nuevas aplicaciones y sistemas en la vida cotidiana.

Ambos elementos interactúan entre si lo que provoca un aumento significativo en las amenazas cibernéticas. Los atacantes utilizan vulnerabilidades en el software para llevar a cabo ciberataques, comprometer equipos, robo de datos y otros delitos cibernéticos.

La seguridad del software, por tanto, es un elemento esencial en esta sociedad, ya que afecta directamente a nuestra dependencia de la tecnología, la protección contra amenazas cibernéticas, la privacidad de los datos y la implementación de nuevos avances tecnológicos. Es un aspecto crítico para mejorar el funcionamiento seguro y efectivo de muchas de las actividades y servicios que damos por sentado en la vida cotidiana.

La seguridad del software abarca todo el Ciclo de Vida de Desarrollo del software (S-SDLC), desde la planificación, el diseño, hasta su puesta en marcha, mantenimiento e incluso puesta fuera del servicio.

Las organizaciones que hacen uso de este software, sin asegurar completamente, se exponen a sufrir problemas de seguridad que pueden ser de mayor o menor riesgo como, por ejemplo, la exposición de información sensible de clientes, cuantiosas multas por incumplimiento de normativas legales, interrupción del servicio, daños reputacionales, etc.…

Entre los procesos que se recoge en el S-SDLC es altamente recomendable la automatización de las pruebas de seguridad en varias fases del Ciclo de Vida del Desarrollo de Software (SDLC). Esta automatización se logra mediante el uso de herramientas específicas de seguridad, realizar pruebas como Static Application Security Testing (SAST), Software Composition Analysis (SCA) y Dynamic Application Security Testing (DAST).

Sin embargo, es importante destacar que las pruebas automatizadas, aunque son vitales, no son suficientes por sí solas. Es por eso por lo que, como parte del proceso del S-SDLC, se recomienda complementar estas pruebas automatizadas con auditorías manuales realizadas por expertos en seguridad. Estos auditores expertos pueden identificar vulnerabilidades que las pruebas automatizadas pueden pasar por alto y

evaluar la seguridad desde una perspectiva integral, teniendo en cuenta la lógica de negocio.

Para la consecución de este objetivo es imperativo realizar un minucioso análisis de las herramientas de automatización de pruebas disponibles en el mercado.
Este, debe cumplir con ciertos requisitos esenciales que permitan su adopción satisfactoria por parte de la organización.

Además de la eficacia inherente de las herramientas, es fundamental que estas faciliten la gestión adecuada de las vulnerabilidades detectadas y presenten un grado de interoperabilidad que fomente la integración efectiva con otros componentes del entorno tecnológico.

Para mitigar estos riesgos, se establecen pruebas de seguridad automáticas de SAST, SCA y DAST (que permitan detectar vulnerabilidades en diferentes contextos de pruebas), para esto se analiza el código fuente y los componentes de terceros que la componen, para una vez que existe un producto mínimo viable establecer pruebas sobre la seguridad durante su ejecución.

Existen diferentes herramientas en el mercado para realizar esta labor, pero cada una ofrece unas características particulares o limitaciones cómo, por ejemplo, las tecnologías que permiten analizar, los falsos positivos y negativos que emiten, interoperabilidad con otras herramientas, etc.…

Estos problemas conllevan pérdidas de tiempo y esfuerzo al contratar o hacer uso de herramientas inadecuadas, problemas que pueden llevar incluso a la preferencia por parte de la organización a no tomar medidas de seguridad en el desarrollo. Para solucionar estos problemas, las organizaciones pueden acudir a benchmarks y/o efectuar pruebas mediante metodologías adecuadas, pudiendo así entender que alcances cubren cada una de las herramientas, además de los puntos fuerte y débiles que presentan. La búsqueda de un sistema de medición de las herramientas de pruebas de seguridad en aplicaciones ha sido objeto de muchos estudios (1), incluso existen artículos donde se presentan las complicaciones para efectuar un correcto benchmark (2).

Finalmente es preciso señalar que, en el contexto del análisis de vulnerabilidades en aplicaciones, se presentan desafíos peculiares. Se debe considerar que se están evaluando implementaciones de código desarrolladas por diversos equipos, lo que puede generar variaciones en los conocimientos y prácticas relacionadas con el Desarrollo Seguro, incluso por la mera rotación interna de los propios equipos. Por último, las restricciones de tiempo y los plazos ajustados pueden dar lugar a la aceptación de software que no cumple con los estándares de seguridad deseados, lo que puede resultar en costos adicionales en el futuro.

Este trabajo estudiará las características de un conjunto de herramientas para que una organización pueda evaluar cual se adapta a sus necesidades, independientemente de su tamaño.

En primer lugar, se procederá a estudiar los puntos fuertes y débiles de cada herramienta bajo unos criterios previamente descritos en una metodología. Para ello, desde el listado descrito en la organización Open Source Foundation for Application Security (OWASP) se hará una selección. Se partirá de este listado debido a que OWASP es un referente en el sector, siendo una comunidad preocupada en la seguridad colectiva y está orientada a la generación de guías, metodologías, frameworks y herramientas que mejoren la seguridad de las Organizaciones y ayuden a los profesionales de la seguridad a llevar a cabo su cometido.

Por último, hay que mencionar que el listado de herramientas será open source, debido a que se trata de hacer una estudio y comparación de herramientas accesibles para cualquier tipo de organización.

1.2 Objetivos del Trabajo

Este Trabajo tienes dos objetivos fundamentales:

1. Definir una metodología propia de evaluación de herramientas de SAST, DAST y SCA, que permita a una organización seleccionar las herramientas más adecuadas a sus necesidades, especificando sus posibles limitaciones de uso y otros criterios adicionales como: usabilidad, efectividad, interoperabilidad, etc....
2. Elaborar una prueba de concepto o PoC (o por sus siglas en inglés de “Proof of Concept”) donde se ponga a prueba la metodología para la evaluación de las herramientas SAST.

Para acotar el alcance de las herramientas a evaluar, se limitará a un conjunto previamente seleccionado y serán de tipo open source.

Los objetivos principales pueden desglosarse en el siguiente itinerario:

1. Definir una metodología de valoración de las herramientas añadiendo parámetros que suelen ser de interés para las organizaciones.
2. Evaluar las herramientas con el método de evaluación propuesto.
3. Evaluar el uso de estas herramientas en procesos de automatización como los ciclos de S-SDLC.
4. Efectuar una prueba de concepto que permita determinar la eficacia (detección de vulnerabilidades, falsos positivos y negativos) en la detección de vulnerabilidades en herramientas SAST, permitiendo medir de manera automatizada uno de los parámetros de la metodología propuesta. Para esta automatización se evaluará el uso del formato estandarizado SARIF (3).

1.3 Impacto en sostenibilidad, ético-social y de diversidad

La Agenda 2030, con sus Objetivos de Desarrollo Sostenible (ODS), establece un marco global para abordar los desafíos multifacéticos que enfrenta el mundo.
Este proyecto busca medir la eficiencia, usabilidad e interoperabilidad de herramientas de seguridad de tipo SAST, DAST y SCA, en adelante nos referiremos a ellas como herramientas de seguridad, especialmente en el contexto del Ciclo de Vida de Desarrollo de Software Seguro (S-SDLC).

Este proyecto presenta un carácter transversal a los ODS. Esto es, tiene relevancia directa e indirecta en varios de estos ODS, que se proceden a enumerar brevemente sin ánimo de ser exhaustivos:

- Educación de Calidad (Usabilidad y ODS 4):
  La usabilidad de las herramientas es esencial para asegurar que los desarrolladores y profesionales de TI, independientemente de su nivel de experiencia, puedan utilizarlas eficazmente. Esto contribuye a una formación y educación de calidad en el ámbito de la ciberseguridad.

- Trabajo Decente y Crecimiento Económico (S-SDLC y ODS 8):
  Las herramientas de seguridad permiten un crecimiento económico más sostenible al reducir los costos asociados a las brechas de seguridad y mejorar la confianza en las soluciones digitales, asegurando el correcto funcionamiento de todos los sistemas críticos, especialmente aquellos relacionados con intercambios de bienes y servicios, así como medios de pago.

- Industria, Innovación e Infraestructura (ODS 9):
  La promoción de infraestructuras resilientes es esencial. Las herramientas de seguridad son fundamentales para mejorar la seguridad que las aplicaciones de software cuenten con el menor número de vulnerabilidades (siendo el objetivo cero vulnerabilidades). Al medir y mejorar la eficiencia y usabilidad de estas herramientas, se contribuye a una infraestructura digital más robusta y confiable a nivel global.

- Interoperabilidad Ciudades y Comunidades Sostenibles (ODS 11):
  Las ciudades inteligentes y las infraestructuras modernas dependen de sistemas interconectados. La interoperabilidad de las herramientas de seguridad es esencial para asegurar que estos sistemas complejos funcionen de manera segura y eficiente.

- Eficiencia Producción y Consumo Responsables (ODS 12):
  Al medir y mejorar la eficiencia de las herramientas de seguridad, se promueve un enfoque más responsable y sostenible en el desarrollo y consumo de software.

- Paz, Justicia e Instituciones Sólidas (ODS 16):
  En un mundo digital, la paz también implica seguridad cibernética. Las herramientas que mejoran la seguridad del software ayudan a prevenir ciberataques y otros delitos cibernéticos, contribuyendo a sociedades más pacíficas y justas. Un estudio que ayuda a las organizaciones a seleccionar las mejores herramientas contribuye directamente a este objetivo.

- Alianzas para Lograr los Objetivos (ODS 17):
  La colaboración global es esencial para abordar desafíos cibernéticos. Un proyecto que evalúa herramientas de seguridad a nivel mundial fomenta la

colaboración y el intercambio de mejores prácticas entre organizaciones, gobiernos y otros actores.

El impacto sobre la sostenibilidad que estas herramientas de seguridad aportan como ventajas son la reducción de residuos digitales y favorecer/asegurar una economía sostenible. Como principal inconveniente el consumo de recursos.

- La reducción de residuos se obtiene mediante la reducción de actualizaciones y parches de emergencia que requieren los sistemas para evitar o paliar y eliminar las brechas de seguridad.

- La disminución de las brechas de seguridad y ataques reduce las pérdidas económicas, incrementa la confianza los usuarios de los distintos sistemas y contribuye a una economía más estable y sostenible.

- El principal coste, es el consumo de recursos como consecuencia de la implementación y prueba de múltiples herramientas, el aumento de la capacidad de los servidores para asegurar los servicios, así como el consumo de energía requerido para efectuar todo lo anterior.

El impacto ético-social de estas herramientas presenta como principal ventaja una mayor protección de los datos personales y por ende una mayor confianza en el mundo digital. Siendo la accesibilidad el principal hándicap.

- Un software más seguro permite una mejor protección de los datos personales y asegurar un principio de los derechos humanos tan importante como es el derecho a la privacidad.

- La seguridad del software fomenta una mayor seguridad en las soluciones digitales y contribuye a una mayor inclusión tanto digital como social.

- El principal problema es la accesibilidad en aquellos casos en el que las herramientas son privativas, no son fácilmente accesibles o el costo de las licencias o requisito de hardware son inasumibles en términos de costes, lo que limitaría su adopción no solo en las organizaciones más pequeñas si no en regiones con menos recursos fomentando desigualdades económicas y de oportunidades.

El impacto en la diversidad de estas herramientas presenta como ventajas fundamentales una mayor inclusión global y una promoción de la diversidad en este ámbito siendo las barrera lingüísticas y culturales los principales retos.

- Estas herramientas contribuyen a la implementación de la Agenda 2030 cuyo objetivo último es la inclusión global.

- El aumento en la diversidad se obtiene alentando a nuevos actores de diferentes orígenes a participar y contribuir en el mundo de la ciberseguridad.

- La existencia de barreras lingüísticas y culturales constituye el principal reto en esta área a la hora de evaluar y recomendar herramientas, en determinados contextos y regiones.

En conclusión, este trabajo evalúa herramientas de Seguridad, que en el contexto de la Agenda 2030 tiene el potencial de contribuir a varios ODS. Al hacerlo, no sólo se mejora la seguridad del software a nivel mundial, sino que también se avanza hacia un mundo más seguro, inclusivo y sostenible.

1.4 Enfoque y método seguido

Este proyecto se compondrá principalmente de dos partes. Inicialmente, será una investigación de la evolución actual de las herramientas de SAST, DAST y SCA, teniendo en cuenta que el campo de la seguridad y tecnología progresa rápidamente y de manera continua hay cambios relevantes que obligan a una nueva revisión. Para esto, se ampliará las metodologías habituales de valoración y se propondrán nuevos parámetros, para medir los puntos fuertes y débiles de cada una de las herramientas (orientado a una matriz de confusión).

Por otro lado, desarrollar una propuesta metodológica, que incluirá en sus parámetros de medición la interoperabilidad y los formatos de salida, así como su capacidad de ser utilizada en automatizaciones de pruebas como, por ejemplo, en sistemas de DevSecOps y/o gestores de vulnerabilidades.

La metodología utilizada en este caso será en su base similar a otras en las que se evalúa los reportes que devuelven las herramientas respecto los que se espera, ya que se lanzan sobre proyectos con vulnerabilidades intencionadas y conocidas, cómo por ejemplo, OWASP Benchmark (4). Se analizarán las metodologías existentes, teniendo en cuenta que los resultados deben ser reproducibles (al menos los parámetros más objetivos). Sin embargo, para incluir otros factores que suelen ser demandados por las organizaciones en este tipo de evaluaciones, se va a ampliar esta metodología en la que se tendrán en cuenta diferentes factores cómo:

- Facilidad de instalación.
- Facilidad de uso.
- Reporte o informe en formatos estandarizados.
- Tasas de efectividad:
    - Verdadero positivo
    - Falso positivo
    - Verdadero negativo
    - Falso negativo

Para este último punto, se tratará de automatizar el proceso de evaluación de la eficiencia mediante el desarrollo de una Prueba de Concepto.

A continuación, en la Ilustración 1 se expone la planificación de la investigación en un diagrama de Gantt.

1.5 Planificación del Trabajo

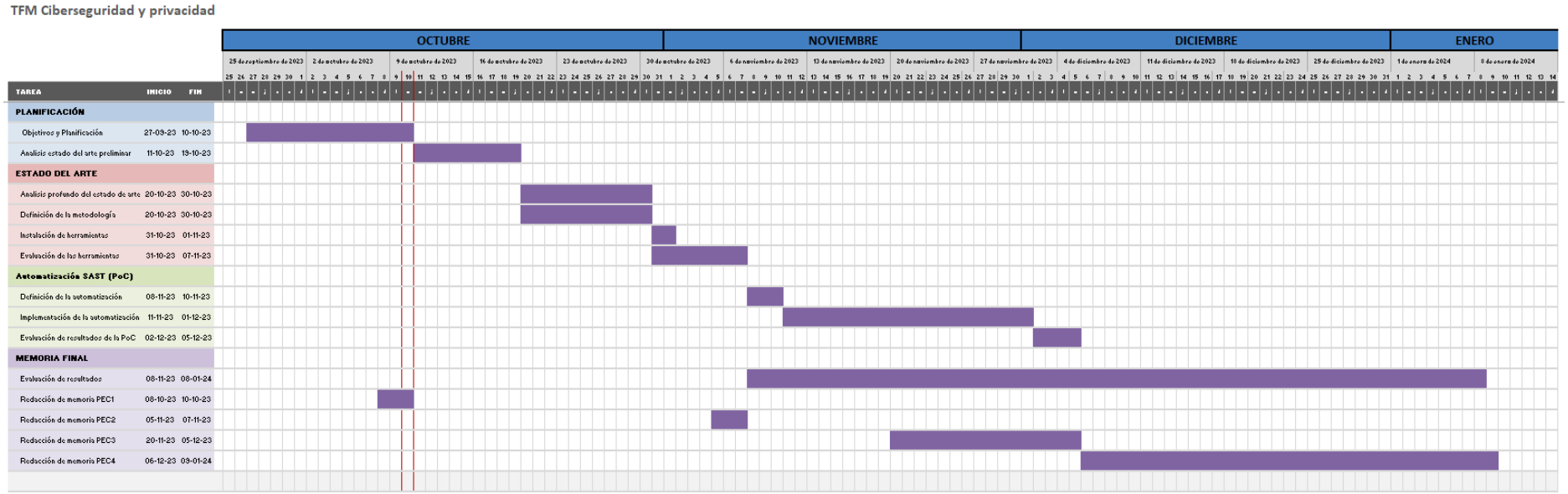

Ilustración 1 - Diagrama de Gantt de la planificación del proyecto.

1.6 Recursos y presupuesto del proyecto

Para llevar a cabo este proyecto será necesario un equipo informático, en principio, no se estima que este equipo necesite recursos especialmente exigentes. Por lo tanto, se hará uso de un ordenador personal de características comunes. El coste estimado del equipo es de aproximadamente 900€.

En la evaluación de las pruebas se utilizará como sistema operativo un Ubuntu 22.04 LTS (GNU/Linux derivación de Debian), con licencia de uso libre. Las herramientas que se van a analizar son de software libre u open source, por lo que no serán necesarias licencias de pago. En caso de ser necesario, se incluirían en el presupuesto o se solicitaría una demo al proveedor.

Sobre la infraestructura necesaria, en obligatorio contar con una conexión a internet, cuyo coste estimado es 59€/mes. Para la creación de copias de seguridad de la memoria se utilizará el especio de la nube que provee la universidad en OneDrive. Para el posible código fuente o similar se utilizará GitHub con repositorio privado (en su versión gratuita).

El presupuesto total del proyecto durante los meses planificados es de 959€.

1.7 Breve sumario de productos obtenidos

Entrega 1 - PEC 1: Definición y Planificación del trabajo final:

Se corresponde con el presente documento que incluye la introducción, objeto de estudio, así como el plan de trabajo que se sigue para la ejecución de este TFM.

Entrega 2 - PEC 2: Estado del Arte:

Se compone de un documento de texto, en el que se procederá a desarrollar los siguientes elementos:

- Investigación de las metodologías de evaluación de herramientas existentes.
- Investigación de las herramientas open source que existen en el mercado que permitan la consecución de los objetivos del trabajo.
- Selección de las herramientas open source más representativas.
- Instalación de las herramientas y análisis inicial.

Entrega 3 - PEC 3: Metodología, diseño, implementación y análisis del modelo de comparación de herramientas:

En esta fase se procederá al desarrollo de la metodología e implementación de esta. Además, se procederá al modelado y evaluación de los resultados obtenidos:

- Desarrollo de la metodología que se aplicará.

- Desarrollo de la automatización de la evaluación de las herramientas de SAST. Se entregarán archivos de código en el formato ZIP.
- Se incluirán los resultados de las herramientas analizadas.

Entrega 4 - PEC 4: Redacción de la memoria:

En esta fase se procederá a la redacción de la memoria final en formato texto. Asimismo, se entregarán los archivos con código de programación de la automatización de la evaluación de las herramientas de SAST.

Entrega 5 - VIDEO EXPLICATIVO:

Documento multimedia en formato video que contiene la presentación formal de este trabajo, así como una copia del material de la presentación en formato de texto y/o cualquier otro documento necesario.

1.8 Breve descripción de los otros capítulos de la memoria

El Trabajo contiene los siguientes capítulos:

Capítulo 2. Estado del arte:

En primer lugar, se efectuar un análisis preliminar de las herramientas open source existentes en el mercado. El objetivo es identificar el listado de las que desde la documentación presenten mejores condiciones de cumplir con los requisitos que se establecerán en la metodología.

Posteriormente se procederá a buscar en la bibliografía los artículos, caso de uso y estudios sobre la evaluación de herramientas de aseguramiento del software (en el contexto de las pruebas SAST, DAST y SCA). En esta fase el objetivo es obtener toda la información posible sobre casos similares y que soluciones se han dado a los problemas previamente identificados de modo que se establezca la metodología a emplear más eficiente para el correcto desarrollo de la tarea de evaluación.

Capítulo 3. Metodología:

En el que se procederá a definir la metodología, materiales y tareas que se llevaran a cabo para la propuesta de solución. Para esto, una vez conocidas las metodologías existentes se hará uso de la más optima, ampliándola en caso necesario.

Capítulo 4. Automatización de la evaluación de herramientas SAST (PoC):

En esta parte se procederá a la definición e implementación de la automatización.

Capítulo 5. Análisis de los resultados:

En esta parte se procederá a analizar los avances efectuados y los resultados obtenidos.

Capítulo 6. Conclusiones y trabajos futuros:

Se abordarán las conclusiones finales y se propondrán líneas para futuras mejoras de este proyecto.

1.9 Análisis de riesgos

En este apartado se incluirán algunos de los riesgos potenciales que pueden surgir durante el desarrollo del proyecto, que pueden impactar en el alcance y plazos de la planificación, enfoque u objetivos marcados:

- Que la lista de herramientas pueda ser excesiva. (RIESGO MEDIO).
    - Planteamiento para mitigar el riesgo: se seleccionará las herramientas que mejor evaluadas o consideradas estén en la comunidad de seguridad, pudiéndose ampliar o reducir según se desarrolle el proyecto.
- Que la interoperabilidad para la generación de la prueba de concepto no sea concluyente. (RIESGO ALTO)
    - Para automatizar la medición de la efectividad de las herramientas de SAST hay que analizar cómo homogeneizar los formatos de las herramientas para simplificar el desarrollo de la prueba de concepto.

2 Estado del arte

2.1 Introducción

En el contexto de la evaluación de herramientas de SAST, se han desarrollado proyectos de benchmark con el propósito de medir su efectividad. La correcta comprensión del funcionamiento de estos benchmarks es crucial, ya que las estrategias de detección pueden variar entre las herramientas de seguridad. Por lo tanto, es esencial considerar el enfoque de diseño de otras pruebas de benchmark al buscar aquellas que sean replicables y respaldadas por una metodología pública y auditable (debe permitir que pueda recrearse). Esto permite ofrecer veracidad de los resultados, excluyendo aquellos que carecen de un respaldo sólido en su información final.

Existen actualmente organizaciones que han creado desarrollos para hacer pruebas de benchmark como, por ejemplo, OWASP con el OWASP Benchmark. En ese caso, esta herramienta de evaluación solo permite evaluar herramientas de SAST (limitado a solo Java). Por otro lado, desde hace pocos años, se ha estandarizado el formato de reporte de las herramientas SAST a un formato llamado SARIF (del que analizaremos puntos fuertes y débiles).

En la comunidad de DevOps, se han creado enfoques interesantes, como el framework en desarrollo de GitLab, que, lamentablemente, solo admite ciertos lenguajes de programación, excluyendo lenguajes populares como C/C++. Sin embargo, existen investigaciones valiosas, como el artículo "Benchmarking Static Analysis Tools for C", (5) ", que evalúa 11 herramientas en aplicaciones C/C++.

Para una visión completa de las opciones disponibles, es esencial realizar una revisión exhaustiva del estado actual de la investigación, ya que el sector y la academia han introducido novedades relevantes que deben considerarse en el proyecto en desarrollo.

2.2 Benckmarks de SAST

Los benchmark en general se basan en la comparación del reporte emitido por la herramienta de seguridad respecto a un reporte con los resultados esperados que previamente ha sido creado, con un conjunto de casos de pruebas que permiten entender un poco mejor cómo opera la herramienta.

Generalmente, el flujo de funcionamiento del benchmark es:

1. Se lanza la herramienta de seguridad a evaluar.
    a. Puede requerirse preconfigurar el benchmark, ya sea para lanzar un script que precompile los casos de pruebas o aplicaciones vulnerables, o tareas similares dependientes del lenguaje de programación o la tecnología a evaluar de la herramienta, etc...

2. Se obtiene el reporte. Este reporte puede requerir una normalización, ya que cada herramienta suele reportar de manera personalizada. Se pueden seguir dos estrategias principalmente:
    a. Normalización del reporte de la herramienta de seguridad.
    b. Uso de algún formato de reporte estandarizado.
3. Comparación de resultados: en este punto, se compara el resultado de la herramienta (procesado si en necesario) y el reporte esperado. En esta etapa se recoge la información en bruto.
4. En este punto, la herramienta de benchmark determina la efectividad de la herramienta de seguridad analizando los positivos reales, los falsos positivos y negativos. Pueden determinarse otros factores durante la ejecución que puedan ser de interesa como, por ejemplo, el tiempo de análisis, recursos consumidos, etc…

Se realiza este proceso con cada herramienta de seguridad SAST a evaluar de un listado y se comparan los resultados obtenidos entre las herramientas.

En la Ilustración 2 se puede ver un diagrama del proceso de evaluación y obtención de datos para una herramienta.

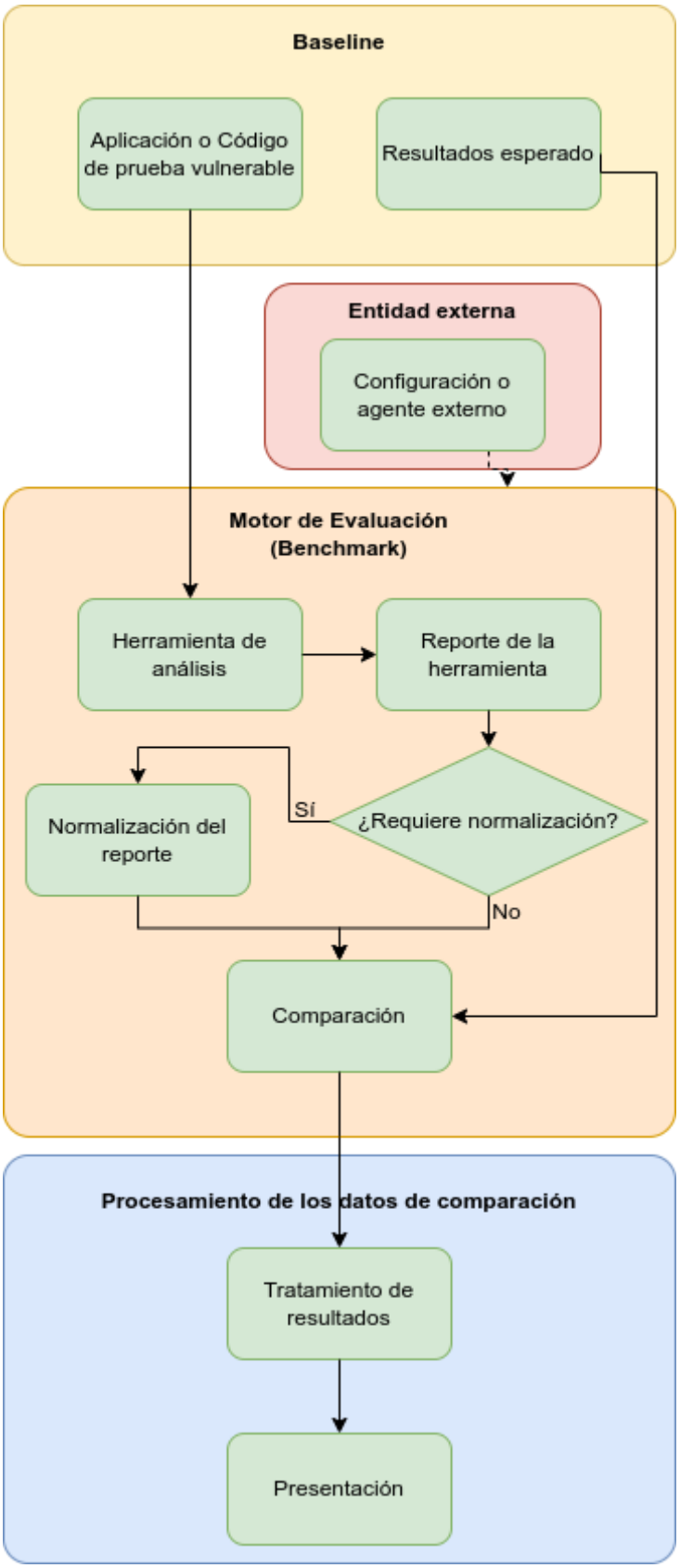

Ilustración 2 - Diagrama de benchmark genérico. Elaboración propia.

2.2.1 OWASP Benckmark Java

El OWASP Benchmark es un proyecto de OWASP que proporciona un conjunto de pruebas diseñadas para evaluar la efectividad de las herramientas SAST en la identificación de vulnerabilidades de seguridad para el lenguaje de programación Java (exclusivamente).

Características principales de este benchmark:

- Ventajas:
  - Dispone de un proyecto en Github, escrito en Java, funcional que puede utilizarse libremente.
  - Contribución a la comunidad: El proyecto es open source y está abierto a la contribución de la comunidad (y apoyo de OWASP), lo que significa que cualquier persona puede contribuir a mejorar el benchmark o agregar nuevas pruebas lo que mejora la medición de su eficacia al tener más test que valorar.
- Desventajas:
  - Limitaciones del OWASP Benchmark: La principal limitación es que está diseñado exclusivamente para el lenguaje de programación Java.
  - Por otro lado, las pruebas (que determinan el alcance y calidad de la evaluación) están muy orientadas al OWASP Top 10 de Web.

2.2.2 Benchmark Gitlab framework

Es un proyecto open source creado por Gitlab para medir la eficiencia de las herramientas de seguridad SAST en su plataforma.

Características principales de este benchmark:

- Ventajas:
  - Proyecto en Gitlab, escrito en Go, funcional que puede utilizarse libremente en la plataforma de Gitlab.
  - Generación de informes de resultados: A partir de la comparación de los resultados de la herramienta con los esperados, se calculan métricas como la precisión, para esto se compra el archivo generado por la herramienta durante el análisis y el archivo “gl-sast-report.json”, generando un archivo de tipo diff llamado “gl-sast-diff-report.json” que contiene la discrepancia de reporte. Una vez finalizado el paso anterior, se presentan estas discrepancias en un formato visual en la plataforma de Gitlab.
- Desventajas:
  - Normalización no estándar: Aunque normaliza los informes para su comparación, la normalización no sigue un estándar de la industria, lo que puede limitar su aplicabilidad en entornos que requieren estándares específicos.
  - Dependencia de la infraestructura de Gitlab: Al estar vinculado a Gitlab, puede no ser adecuado para organizaciones que utilizan otras plataformas de desarrollo y alojamiento.

2.3 Beneficios y deficiencias de los benchmark de SAST

La comparación de herramientas SAST no es posible únicamente con listas, conjuntos de pruebas y benchmarks (2). Comparar herramientas SAST es una tarea compleja debido a la diversidad de vulnerabilidades no conocidas. Aunque los benchmarks, que implican pruebas con conjuntos de vulnerabilidades comunes, como OWASP Top 10, SANS-25, son útiles, pero tienen limitaciones. Estos benchmarks a menudo representan un conjunto limitado de vulnerabilidades, con ejemplos específicos de implementaciones. También pueden estar desactualizados y centrados en lenguajes de programación, frameworks y tecnologías específicas. Por lo tanto, no son perfectos a la hora de evaluar herramientas SAST en entornos del mundo real.

Sin embargo, a pesar de estas limitaciones, los benchmarks educan a los desarrolladores y aumentan la conciencia sobre cuestiones de seguridad comunes. A pesar de que comparar herramientas SAST con listas y benchmarks no es un proceso simple, sigue siendo valioso para identificar vulnerabilidades y fortalecer la seguridad de las aplicaciones.

2.3.1 Diferencia entre los analizadores SAST

Cabe desatacar que las herramientas SAST puede analizar el código fuente, pero también hay herramientas que realizan en análisis sobre las versiones compiladas. Para este caso, es conveniente hacer distinción entre las piezas de software a analizar cuyo compilado es de tipo binario (compilaciones como en C/C++) y tipo byte-code (compilaciones como Java). Este factor puede complicar los análisis, ya que la compilación previa es un requisito sine qua non para obtener resultados, lo que hace la herramienta menos usable contra una que solo requiere el código fuente.

2.4 Estandarización de pruebas SAST - SAMATE

El proyecto SAMATE (Software Assurance Metrics And Tool Evaluation) (6) desarrollado por el NIST es una respuesta al desafío constante de mejorar la calidad y seguridad del software. SAMATE se enfoca en proporcionar métricas y evaluaciones objetivas para herramientas de análisis de seguridad del software, incluyendo análisis estático (SAST), análisis dinámico (DAST) y otras técnicas de seguridad. Estas herramientas son fundamentales para identificar y mitigar vulnerabilidades en el software, y se centra en proporcionar un conjunto de métricas y evaluaciones objetivas para las herramientas de análisis de seguridad del software. A través de SAMATE, el NIST busca establecer una base sólida para comparar y contrastar estas herramientas, poniendo a disposición de desarrolladores, investigadores y profesionales de la seguridad la información necesaria para tomar decisiones informadas y, en última instancia, mejorar la seguridad del software.

El proyecto SAMATE se esfuerza por ser un recurso integral, completo y confiable para la comunidad de seguridad del software, ofreciendo guías, informes técnicos y casos de

prueba, destacando el SARD (Software Assurance Reference Dataset) (7) para pruebas de SAST, incluyendo set de pruebas propio e incluyendo piezas de software libre en versiones concretas, en los cuales se encontró un conjunto de vulnerabilidades que ahora sirven para mejorar las herramienta de análisis de SAST. Por tanto, en estos sets de pruebas podemos encontrar conjuntos de pruebas intencionadamente vulnerables que pueden no ser ejemplos de código real, sino simplemente código vulnerable "testigo" y otro conjunto de pruebas extraídas de proyectos reales. Esta perspectiva es interesante, ya que permite tener una perspectiva mixta a la hora de efectuar las pruebas de benchmark.

La colaboración activa con proveedores de software y eventos como el SATE (Static Analysis Tool Exposition) (8) fomenta el crecimiento y la evolución del campo de la seguridad del software. Ya que en estas concentraciones se efectúan retos de analizar las herramientas de diferentes proveedores.

Ventajas:

- Ofrece métricas y evaluaciones objetivas para herramientas de análisis de seguridad.
- Proporciona una amplia gama de recursos, incluyendo casos de prueba y guías, esto es, es un recurso integral.
- Facilita la colaboración con la comunidad y proveedores de software, fomentando el crecimiento y la evolución en seguridad del software.
- Respaldo de una Organización oficial como el NIST.

Desventajas:

- Limitación de alcance: SAMATE puede estar enfocado principalmente en el análisis estático de seguridad (SAST) y no cubrir con el mismo nivel de profundidad otras áreas de seguridad del software, como el análisis dinámico (DAST), la seguridad en la capa de red o aspectos de seguridad más amplios.
- Relevancia de los casos de prueba: La eficacia de los casos de prueba proporcionados por SAMATE puede variar dependiendo de la evolución de las amenazas y las tendencias en seguridad del software. Los casos de prueba pueden no abordar vulnerabilidades emergentes o avanzadas.
- Complejidad y tiempo: La configuración y ejecución de pruebas en proyectos de benchmark como SAMATE pueden requerir tiempo y esfuerzos considerables, lo que podría ser una desventaja para organizaciones con recursos limitados.
- Necesidad de actualización: A medida que las tecnologías y las prácticas de desarrollo evolucionan, los criterios y métricas utilizados por SAMATE pueden volverse obsoletos, lo que requiere actualizaciones regulares para mantener la relevancia.
- Dependencia de la comunidad y colaboración: El éxito de proyectos como SAMATE a menudo depende de la colaboración activa de la comunidad de seguridad del software y los proveedores de herramientas. La falta de participación o colaboración puede limitar la utilidad de la iniciativa.

- Incompatibilidad con ciertos entornos: Podría no ser adecuado para entornos de desarrollo específicos o para organizaciones que utilizan herramientas de análisis personalizadas o propias que no se ajustan al marco de referencia de SAMATE.

2.4.1 Metodología

La metodología de SAMATE se ha diseñado teniendo en una serie de pasos y procedimientos rigurosos diseñados para evaluar de la manera más completa posible las herramientas de análisis de seguridad del software, como las herramientas SAST en este caso (intentando simplificar el proceso). La metodología consta de las siguientes etapas:

2.4.1.1 Definición de Criterios de Evaluación

Este proceso define los criterios de evaluación, siendo los principales:

- La precisión se refiere a la capacidad de la herramienta para identificar correctamente las vulnerabilidades con el mínimo posible de falsos positivos, siendo cero el valor deseable.
- La cobertura implica la capacidad de la herramienta para analizar todas las partes relevantes del código en busca de vulnerabilidades. Normalmente las herramientas suelen estar acotadas a vulnerables más típicas como, por ejemplo, el Top 10 de OWASP (web).
- La eficiencia se refiere a la rapidez con que la herramienta puede realizar el análisis y proporcionar resultados. Es habitual que las herramientas de DAST requieran más tiempo que las de SAST cuando la web tiene cierta envergadura.
- La facilidad de uso evalúa cuán sencillo es para los usuarios operar la herramienta. Este punto es ciertamente subjetivo, pero es interesante de cara a las Organizaciones que tienen que implementar las herramientas, operarlas e incluso conectarlas con otras soluciones.

2.4.1.2 Creación de casos de pruebas

Se han creado una base de datos de un conjunto de casos de prueba (SARD (7)) que serán utilizados para evaluar las herramientas. Estos casos de prueba se diseñan para cubrir una amplia gama de vulnerabilidades y escenarios de seguridad comunes, y se utilizan para evaluar cómo las herramientas SAST identifican y responden a estos problemas. Los casos de prueba están documentados extensamente, con información sobre la vulnerabilidad específica que se está probando, la categoría de la vulnerabilidad, y el resultado esperado, es decir, si se trata de un positivo verdadero o un falso positivo.

2.4.1.3 Comparación de resultados

Por último, esta metodología permite valorarla seguridad de la aplicación de las herramientas SAST a los casos de prueba de referencia y la evaluación de los resultados obtenidos. Este paso implica comparar los resultados proporcionados por la

herramienta con los resultados esperados documentados en los casos de prueba de referencia. Esta comparación permite determinar la precisión, cobertura, y en cierta medida, la eficiencia y facilidad de uso de la herramienta en un entorno controlado.

2.4.1.4 Generación de informes

Los resultados de estas evaluaciones son luego compilados en informes detallados que proporcionan una visión clara y lo más objetiva posible de las fortalezas y debilidades de cada herramienta SAST evaluada, facilitando la toma de decisiones informadas por parte de los desarrolladores y profesionales de la seguridad del software.

2.5 Formato SARIF

El formato SARIF (3), se ha establecido como un estándar abierto orientado a las pruebas de análisis estático de seguridad en aplicaciones. Su desarrollo fue llevado a cabo por el grupo de trabajo de análisis estático de la “Organization for the Advancement of Structured Information Standards” (OASIS) con el objetivo principal de crear un formato común para compartir resultados de análisis estático entre diversas herramientas y plataformas.

Una de las principales causas para la creación de SARIF fue la gran diversidad de formatos de salida propietarios (y open source) utilizados por diferentes herramientas de análisis estático, que obstaculizaban la interoperabilidad y la integración de los resultados de análisis. Con la introducción de SARIF, se ha iniciado un camino con el objetivo de facilitar la integración de herramientas de análisis estático en procesos de desarrollo de software más amplios, permitiendo una comprensión más clara y profunda de los resultados de análisis.

SARIF define una serie de parámetros fundamentales para representar los resultados de un análisis estático:

- Resultado: Es la representación individual de un hallazgo de análisis estático que incluye una ubicación, un mensaje descriptivo y, en algunos casos, una solución recomendada.
- Ubicación: Define el lugar específico en el código fuente donde se ha identificado un problema. Consiste en un archivo y una región dentro de ese archivo.
- Mensaje: Es una descripción textual del problema identificado.
- Solución: Representa una acción sugerida para resolver el problema identificado.
- Herramienta: Es la representación de la herramienta de análisis estático que generó el resultado, e incluye el nombre y la versión de la herramienta.
- Reglas: En cada análisis se vuelca el conjunto de reglas utilizado durante el análisis.

SARIF ha sido diseñado con el objetivo de permitir la flexibilidad, permitiendo a las herramientas agregar campos y entidades adicionales según sea necesario. Además,

es compatible con la internacionalización, lo que facilita la traducción de mensajes a diferentes idiomas.

Para que un informe de una herramienta SAST sea completo en formato SARIF, debe incluir los siguientes parámetros clave:

- Versión de SARIF: La versión del formato SARIF utilizado para generar el informe.
- Schema: La URL del esquema JSON que define el formato del informe.
- Herramienta: Información detallada de la herramienta que generó el informe.
- Resultados: Los resultados individuales del análisis, incluidas las ubicaciones, mensajes y soluciones recomendadas.
- Ubicaciones: Las ubicaciones específicas de los resultados en el código fuente.
- Configuración de la herramienta: Los parámetros y configuraciones utilizados para ejecutar la herramienta.
- Ejecuciones: Las ejecuciones individuales de la herramienta, que incluyen resultados, configuración de la herramienta, y otras informaciones relevantes como, por ejemplo, el set de reglas utilizado.

Con el tiempo, SARIF ha evolucionado más allá de su uso original en el análisis estático de seguridad y se está considerando como un formato de informe para otras pruebas de seguridad en aplicaciones, como el Análisis de Composición de Software (SCA) y el Análisis Dinámico de Seguridad de Aplicaciones (DAST). Esta extensión del uso de SARIF refleja su utilidad y relevancia en el ámbito de la seguridad de las aplicaciones, donde la necesidad de un formato común de informe es esencial para una integración y análisis efectivos de los resultados de las pruebas de seguridad. Con la creciente complejidad del panorama de la ciberseguridad, la adopción de SARIF en estas otras áreas de prueba de seguridad es viable gracias a la flexibilidad del formato, permitiendo estandarizar los reportes de las herramientas de seguridad de aplicaciones a la interoperabilidad, integridad de la información y coherencia de resultados.

2.6 Otros benckmarks para DAST y SCA

Aunque excede los objetivos de este documento, resulta interesante destacar la existencia de benchmarks en otros ámbitos de pruebas de seguridad de aplicaciones, como DAST y SCA. La adaptación de SARIF a estas áreas aún en desarrollo promete ser un formato interesante para futuras investigaciones y trabajos académicos. Ejemplos como la adopción de SARIF por OWASP ZAP (9). indican un avance en esta dirección. Por lo tanto, un objetivo valioso para futuras investigaciones es explorar a fondo cómo SARIF puede aplicarse y mejorar las pruebas DAST y SCA, contribuyendo a la estandarización y uniformidad en los informes de seguridad de aplicaciones.

2.6.1 Benchmark DAST

Existe un proyecto que llamado WAVSEP (Web Application Vulnerability Scanner Evaluation Project), el cual es un conjunto de pruebas diseñado para evaluar la

efectividad de los escáneres de vulnerabilidades de aplicaciones web (DAST). Sus características principales son:

- Evaluación de escáneres de vulnerabilidades: Proporciona un conjunto de pruebas específicamente diseñado para evaluar la capacidad de los escáneres de vulnerabilidades para identificar y reportar vulnerabilidades de seguridad comunes en aplicaciones web, como Cross-Site Scripting (XSS), SQL Injection, y más.
- Comparación de herramientas: de manera similar a el OWASP Benchmark (teniendo en cuenta que tienen alcances diferentes), WAVSEP permite comparar diferentes escáneres de vulnerabilidades de aplicaciones web y técnicas de análisis de seguridad.
- Entorno de prueba controlado: WAVSEP proporciona un entorno de prueba controlado que contiene vulnerabilidades de seguridad específicas. Esto permite comparar los resultados de una herramienta contra otra.
- Resultados objetivos: Al proporcionar resultados objetivos, WAVSEP ayuda a los usuarios a comprender las fortalezas y debilidades de las herramientas y técnicas analizadas.
- Contribución a la comunidad: El proyecto es open source y está abierto a la contribución de la comunidad, lo que significa que cualquier persona puede contribuir a mejorar WAVSEP o agregar nuevas pruebas.
- Limitaciones: A pesar de ser una herramienta valiosa, WAVSEP tiene algunas limitaciones. La principal es que se centra exclusivamente en los escáneres de vulnerabilidades de aplicaciones web, por lo que no es aplicable para evaluar herramientas que analizan otros tipos de aplicaciones. Además, el proyecto lleva tiempo sin actualizarse, la última versión reléase data del 12 de Junio de 2015 y el último commit[1].

2.6.2 Benchmark SCA

Para este tipo de evaluaciones no existe un estándar general, por tanto, se puede complicar más que en el caso de las pruebas de tipo SAST y DAST. Esto se debe que las pruebas SAST y DAST las comparaciones se realizan bajo resultados esperados estáticos. Sin embargo, en el caso de las herramientas de SCA en contexto de aplicaciones, corresponde con el análisis de dependencias de terceros, que se compara con una base de datos que está en continua evolución debido a que siguen apareciendo CVE en los desarrollos de terceros.

No obstante, existen artículos interesantes sobre con las características que debe tener el benchmark de SCA en los que se establecen una cierta metodología de cómo se puede abordar este caso. Hay un artículo (10) de un colaborador de la organización de Linux Foundation[2] que detalla las métricas y características que se deben tener en

[1] El último commit data del 26 de Febrero de 2014 es (revisado en 22 de Octubre de 2023): https://github.com/sectooladdict/wavsep/commit/679a8fbbfc2c3ea928fb7295f8f749873f10e3bf

[2] Se resume el capítulo 12 del informe “source compliance in the enterprise” (11).

cuenta (siendo similares a las necesidades del resto de bechmarks de otros tipos de análisis), pudiéndose utilizar como punto de partida para el desarrollo de benchmarks de SCA.

2.7 Casos de uso

Existen varios casos en los que los proveedores de herramientas de internet han hecho uso de benchmark para publicitar la calidad de sus herramientas. Uno de ellos es Hdiv, un proveedor de herramienta de IAST que permitía detectar las vulnerabilidades en el código fuente antes y durante el uso de una aplicación, además de detectar dependencias de terceros vulnerables. Esta empresa fue recientemente ha sido adquirida por otra empresa del sector (DataDog). En su página web (al ya no estar disponible por la compra se ha recuperado desde https://web.archive.org) se podían encontrar los resultados obtenidos con el test de pruebas del Benchmark de OWASP (12), que se pueden observar en las en la Ilustración 3, donde se muestra un gráfico con los resultados por tipo de vulnerabilidad, en la Ilustración 4 se muestra una tabla con el número de tests que se han efectuado.

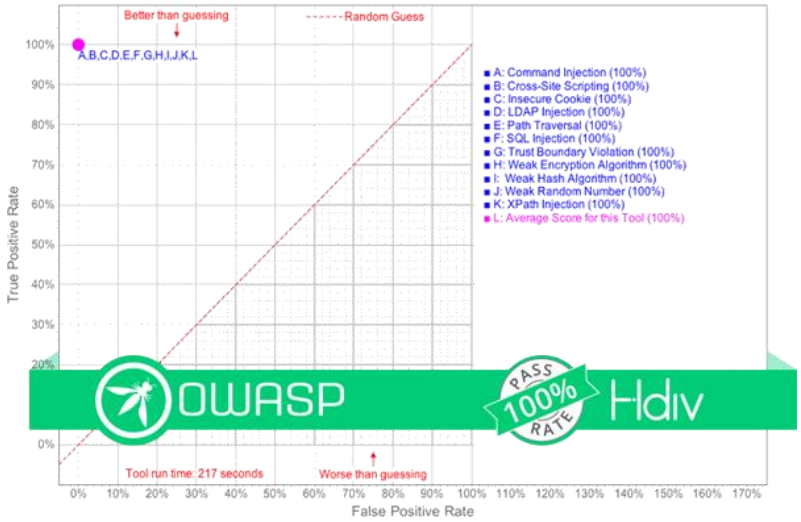

Ilustración 3 - Gráfico de resultados de Hdiv en el benchmark de OWASP. Imagen recuperada de https://web.archive.org/web/20220625103146/https://hdivsecurity.com/owasp-benchmark

.

Category	CWE #	TP	FN	TN	FP	Total	TPR	FPR	Score
Command Injection	78	126	0	125	0	251	100,00%	0,00%	100,00%
Cross-Site Scripting	79	246	0	209	0	455	100,00%	0,00%	100,00%
Insecure Cookie	614	36	0	31	0	67	100,00%	0,00%	100,00%
LDAP Injection	90	27	0	32	0	59	100,00%	0,00%	100,00%
Path Traversal	22	133	0	135	0	268	100,00%	0,00%	100,00%
SQL Injection	89	272	0	232	0	504	100,00%	0,00%	100,00%
Trust Boundary Violation	501	83	0	43	0	126	100,00%	0,00%	100,00%
Weak Encryption Algorithm	327	130	0	116	0	246	100,00%	0,00%	100,00%
Weak Hash Algorithm	328	129	0	107	0	236	100,00%	0,00%	100,00%
Weak Random Number	330	218	0	275	0	493	100,00%	0,00%	100,00%
XPath Injection	643	15	0	20	0	35	100,00%	0,00%	100,00%
Totals		**1415**	**0**	**1325**	**0**	**2740**			
Overall Results							**100,00%**	**0,00%**	**100,00%**

Ilustración 4 - Tabla de resultados de Hdiv en el benchmark de OWASP por tipo de vulnerabilidad. Imagen recuperada de https://web.archive.org/web/20220625103146/https://hdivsecurity.com/owasp-benchmark

.

Otro benchmark de interés, da lugar en las jornadas de SATE (del NIST) donde los proveedores ponen a prueba sus herramientas participando en un evento que hace uso de la metodología de SAMATE. En la Ilustración 5, podemos ver un ejemplo de los resultados de las herramientas en las pruebas de Java en SATE V (13):

Test Case	Openfire	JSPWiki
# Participants	6	6
Tool L	13 568	2165
Tool M	87 631	19 734
Tool N	1144	753
Tool O	950	186
Tool P	1863	97
Tool Q	573	90
Total	**105 729**	**23 025**
# Selected	**180**	**180**

Ilustración 5 - Tabla de resultados de herramientas antes los test suite de SARD en SATE V para los test case de Java.

3 Metodología

La metodología que se propone en la investigación es similar a las utilizadas en cualquier sistema de benchmark. Para poder realizar este proceso se debe contar un set de pruebas que haga la función de “Testigo” con un conjunto de resultados esperados (conocidos previamente). Una vez que tenemos un conjunto de pruebas con resultados conocidos podemos lanzar contra dicho conjunto un análisis de cada herramienta de seguridad. Una vez se genera el reporte de la herramienta que comparar, podemos validar los resultados y presentar un diagrama de confusión donde medir la fiabilidad de la herramienta.

El diagrama inicial de la solución sería:

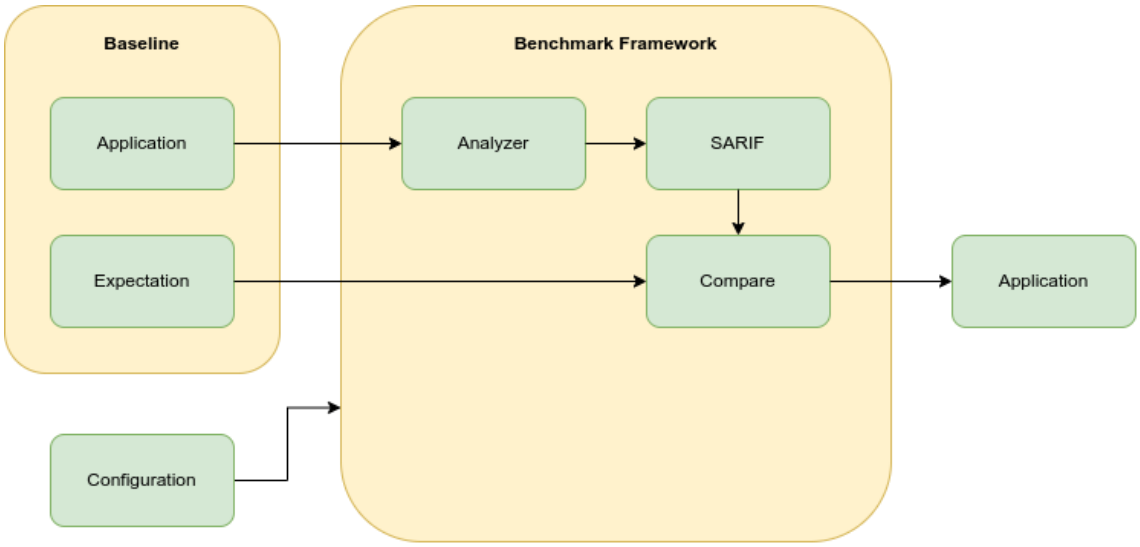

Ilustración 6 - Diagrama de funcionamiento del Benchmark de la PoC. Elaboración propia.

El flujo requiere una serie de etapas:

1. Baseline: en esta etapa la herramienta de seguridad debe lanzarse para analizar el contexto a medir de la aplicación (SAST, DAST o SCA).
2. Configuración: es recomendable que exista cierta configuración de la herramienta de benchmark para limitar contextos (hacer sólo SAST/DAST/SCA).
3. Benchmark: en esta etapa se hace el análisis por parte de la herramienta de seguridad a la aplicación (según la prueba, puede requerir pasos previos como la compilación), para después efectuar obtener los datos del reporte de la herramienta. Por último, se extrae del archivo que acompaña a la prueba (el archivo con los resultados esperados) los resultados correctos y se comparan con lo reportado por la herramienta de seguridad. En esta etapa se comprueba que ha detectado la herramienta bajo el criterio de comparación del benchmark.
4. Resultados: En esta etapa se realizar una analítica de los datos obtenidos. Habitualmente se genera una matriz de confusión donde se detectan los Verdaderos Positivos (TP), Falsos Positivos (FP), Verdaderos Negativos (TN), Falsos Negativos (FN).

Por tanto, para efectuar un benchmark se necesita:

- Un conjunto de pruebas confiable que contenga información de los resultados.
- Criterio de los resultados.
- Selección de herramientas de seguridad.
- Proceso de comparación y verificación.

Una vez establecidos estos requisitos (que se verán a continuación), se puede proceder a la implementación de la PoC.

3.1 Conjunto de pruebas

Tras el estudio de los diferentes enfoques, se creará una prueba de concepto siguiendo un procedimiento similar a los benchmark existentes, con la diferencia en el uso de un formato estandarizado en el proceso de comparación, donde se utilizará el formato de SARIF.

La base de datos SARD de SAMATE del NIST como fuente de set de pruebas confiable, donde podemos encontrar varios sets de pruebas por lenguajes de programación y grupos de vulnerabilidades. Para esto, se analizará y se seleccionará una variedad de casos de prueba que incluir en las pruebas de detección de vulnerabilidades por parte de las herramientas de SAST y contrastarlas con las que reporta la base de datos.

La información recopilada de la revisión bibliográfica respalda la creación de un desarrollo orientado a automatizar la evaluación de herramientas de SAST. Esta herramienta se basará en el análisis de un conjunto de herramientas de análisis seleccionadas en función de los hallazgos de la investigación previa.

Para la prueba de concepto de esta automatización, se analizará un listado de herramientas de análisis recogida en este capítulo, donde se procederá a determinar las más interesantes para la PoC.

3.2 Herramientas de análisis de seguridad SAST, DAST, SCA

La prueba de concepto (PoC) debe probarse con un conjunto de herramientas de análisis, para esto, a continuación, se presenta la Tabla 1 que recopila las principales herramientas de análisis de seguridad disponibles en el mercado (open source o privativas) para cada una de estas categorías, SAST, DAST y SCA, destacando las tecnologías que son capaces de analizar cada herramienta.

Esta información se proporciona con el objetivo de ofrecer un panorama integral que facilite la comprensión de las distintas opciones disponibles para los desarrolladores y profesionales de la seguridad informática en su búsqueda de soluciones efectivas para proteger sus aplicaciones y sistemas. Esta tabla recoge otro aspecto fundamental como,

por ejemplo, si soporta el reporte del análisis en el formato exigido (SARIF) en la metodología que estamos exponiendo.

	SAST						SCA								DAST			
	CodeQL	SonarQube	Semgrep	Fortify	Checkmarx	SpotBugs	WhiteSource	Black Duck	Snyk	Dependabot	FOSSA	Dependency-check	Anchore	Trivy	OWASP ZAP	Burp Suite	AppScan	Acunetix
SoportyeSARIF	✓	✓	✓	✓	✓	✓			✓			✓	✓	✓	✓			
Tipo Licencia	Privativa	LGPL (Software Libre)	LGPL (Software Libre)	Privativa	Privativa	LGPL (Software Libre)	Privativa	Privativa	Privativa	MIT (Software Libre)	AGPL-3.0 (Software Libre)	Apache 2.0 (Software Libre)	Apache 2.0 (Software Libre)	Apache 2.0 (Software Libre)	Apache 2.0 (Software Libre)	Privativa	Privativa	Privativa
.NET				✓	✓					✓		✓						
ABAP				✓	✓													
Apex					✓													
Aplicaciones web															✓	✓	✓	✓
C#	✓	✓		✓			✓	✓	✓		✓							
C++	✓		✓	✓			✓	✓	✓		✓							
Cobol				✓	✓													
Contenedores Docker													✓	✓				
CSS		✓																
CVE de paquetes del SO													✓	✓				
Docker			✓						✓		✓							
Elm										✓								
Elixir										✓								
Go	✓		✓	✓	✓		✓	✓	✓	✓	✓							
Groovy				✓	✓		✓	✓	✓		✓							
HTML		✓	✓		✓		✓	✓	✓		✓							
Java	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
JavaScript	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓							
JSP					✓													
Kotlin		✓	✓	✓	✓		✓	✓	✓		✓							
Kubernetes									✓		✓			✓				
Librerías de Maven													✓	✓				
NPM														✓				
Objective-C				✓	✓		✓	✓	✓		✓							
PHP		✓	✓	✓	✓		✓	✓		✓	✓							
Pip y otros gestores														✓				
PL/SQL				✓	✓													
Python	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓							
Ruby		✓	✓	✓	✓		✓	✓	✓	✓	✓							
Rust										✓								
Scala				✓	✓		✓	✓	✓		✓							
Swift		✓	✓	✓	✓		✓	✓	✓	✓	✓							
Terraform									✓		✓							
T-SQL				✓	✓													
TypeScript	✓	✓	✓	✓	✓		✓	✓	✓		✓							
VB6				✓	✓													
XML		✓			✓		✓	✓	✓		✓							
YAML			✓															

Tabla 1 - Tabla de herramientas de SAST, DAST y SCA. Elaboración propia.

3.3 Criterio de los resultados

Para este punto, se ha determinado mantener el mismo criterio sobre la clasificación del reporte de las herramientas que los habituales del sector. Este criterio nos debe permitir que crear métricas de por cada paquete, CWE y por el OWASP Top10.

El criterio utilizado es el siguiente:

- TP: La herramienta SI ha reportado una vulnerabilidad que SI existe en el reporte SARIF del paquete de SARD.
- FP: La herramienta SI ha reportado una vulnerabilidad que NO existe en el reporte SARIF del paquete de SARD.
- FN: La herramienta NO ha reportado una vulnerabilidad que SI existe en el reporte SARIF del paquete de SARD.
- TN: La herramienta NO ha reportado una vulnerabilidad que NO existe en el reporte SARIF del paquete de SARD.

3.4 Proceso de comparación y verificación

Para efectuar la comparación, se extraerá de los resultados esperados el CWE correspondiente a la vulnerabilidad y el archivo en el que se ubica la misma. En principio, no se efectuará una coincidencia a nivel de línea de código debido a que, según la herramienta, se comprobado que la vulnerabilidad en algunos casos se detecta donde introduce el factor que hace vulnerable al código y en otras se reporta donde se ejecuta la instrucción.

Debido a esto, se limita la comparación y verificación a que exista la vulnerabilidad esperada en el fichero donde se reporta. Por ejemplo, si un archivo tiene 3 vulnerabilidades de CWE-89 (SQL injection) y la herramienta emite 3 reportes de ese CWE en ese archivo, se considera que es correcto.

Este criterio puede adaptarse según se implemente la PoC, una vez que se hayan analizado los conjuntos de pruebas u otros detalles que surjan.

4 Automatización de la evaluación herramientas SAST (PoC)

Para la realización de le presente prueba de concepto (PoC) se ha empleado un equipo informático de uso doméstico y que presenta unas especificaciones moderadas para los estándares habituales para este tipo de actividades. Es relevante especificar el dispositivo que se ha utilizado debido a que ha sido un factor limitante durante la investigación. El hardware utilizado consistió en un dispositivo:

MSI Apache Pro GE62 con las siguientes características:

- Procesador: Intel i7-7700HQ (4 núcleos físicos / 8 hilos lógicos).
- Gráfica: NVIDIA GTX 1050 Ti.
- RAM: 32 GB (SDDR4) + 33 GB (SWAP).
- Discos:
    - Samsung MZNLN256HMHQ (256GB).
    - Crucial CT2000MX500SSD1 (2TB).
- Sistema Operativo: Ubuntu 22.04 LTS.
- IDE: PyCharm 2022.3.

4.1 Alcance inicial de la PoC

Para poner a prueba la propuesta de esta metodología se va a llevar a cabo una Prueba de Concepto (PoC). Debido a la necesidad de ajustarse a un periodo de tiempo se va a establecer el siguiente alcance inicial:

- Lenguajes de programación: la investigación se centrará en Java, al ser el lenguaje más común dentro de las organizaciones y empresas. En caso necesario se ampliará a alguno otro de interés.
- Tipo de análisis: se efectuará la PoC solamente para las revisiones de código fuente (SAST), quedando excluidas las pruebas dinámicas (DAST) y de revisión de componentes de terceros (SCA), que se proponen como futuros trabajos.
- Herramientas SAST evaluadas en la PoC:
    - **Semgrep**: esta herramienta será la primera a poner a prueba ya que no requiere compilar el código fuente. Además, esta herramienta permite el análisis para varias tecnologías
    - **Spotbugs**: esta herramienta será la segunda a probar, ya que requiere la compilación del proyecto y es un paso adicional. Es necesario aclarar que esta herramienta sólo hace análisis para compilaciones de Java.
- Conjunto de pruebas: se analizará y escogerá dentro de los proyectos de SARD aquellos que contienen información suficiente para llevar a cabo la PoC.

4.2 Fases de la PoC

Al tratarse de un desarrollo desde cero, sin mucha información previa, se ha establecido una serie de fases en las que se van implementando funcionalidades, siendo:

- Fase 1 - descarga de tests: en esta tarea el script hará la descarga automatizada de cada test del conjunto de tests de pruebas.
- Fase 2 - descompresión de archivos, donde se ubica el código fuente del proyecto a analizar.
- Fase 3 - extracción de archivo SARIF del test case que servirá como fuente para comprar y validar el reporte de la herramienta de seguridad.
- Fase 4 - análisis de test con la herramienta de seguridad (inicialmente Semgrep).
    - Fase 4.1 - compilación del test: para las herramientas de seguridad que como prerrequisito necesiten el proyecto de test compilado, se debe efectuar la tarea.
- Fase 5 - extracción de archivo SARIF de la herramienta de seguridad: una vez que el análisis se ha llevado a cabo con éxito, se obtendrá la información necesaria para la comparación.
- Fase 6 - comparación de los datos de ambos archivos SARIF: en este punto ya se puede efectuar una comparación bajo los criterios establecidos.
- Fase 7 - obtención y análisis de resultados: en este punto obtenemos la información ya facilitada para comprar con otras herramientas de seguridad.

Estas implementaciones se pueden identificar dentro del código fuente en los comentarios.

4.3 Diseño inicial

Para el diseño inicial de la PoC se han implementado dos desarrollos diferenciados especializados en una actividad diferente cada uno:

1. Extractor de datos (ETL): Permite efectuar la extracción, tratamiento y almacenamiento de los datos orientado a su posterior tratamiento analítico. Este módulo ha sido implementado utilizando Python dentro de un proyecto de PyCharm.
2. Analizador: Orientado a la explotación de los datos obtenidos en la fase anterior, pudiéndose hacer comparaciones y analítica de datos desde datos almacenados. Este módulo se ha implementado en diferentes archivos de Jupyter con el objetivo de hacer un rápido estudio de los resultados.

4.4 Implementación de la PoC

4.4.1 ETL de datos

El desarrollo para la extracción, tratamiento y almacenamiento de los datos está formado por una serie de scripts dentro del proyecto de Pycharm. En este proyecto, se ha guardado una copia de los identificadores y URL de cada Test Suite, que contienen los diferentes sets de pruebas clasificadas por lenguaje. De esta forma, se efectúa la descarga de cada uno de los Tests Suites de SARD de manera automatizada, extrayéndose en una carpeta y extrayendo la información de los archivos JSON (SARIF) de cada uno de los paquetes. Cada suite case está formado por un conjunto de módulos donde se encuentra el código vulnerable. Este paso nos permite disponer del código

fuente, información de compilación y ejecución de los proyectos, es decir, de cada test suite. Además, se adjunta el archivo SARIF que se procesa para conocer los TP y FP. Según el tipo de suite case se tratan unos objetivos u otros, es decir, hay suite case orientados a determinar los TP o vulnerabilidades existentes que, en principio, deben ser detectadas por la herramienta y otros case suite orientados a determinar los FP o vulnerabilidades reportadas por la herramienta que en realidad no son vulnerabilidades existentes (la herramienta emite falsas alarmas).

Tras la descarga de los tests suites, se procede a lanzar los análisis de las herramientas SAST, generándose un reporte JSON (SARIF) de la herramienta. Este es uno de los puntos más relevantes, ya que según el tipo de herramienta de SAST el análisis puede ser sobre el código fuente o sobre los objetos compilados como, por ejemplo, el byte-code. Para efectuar este paso, hay que automatizar que se lance el análisis de la herramienta en cada Test Suite y se guarde el archivo, que se guarda con el nombre de la herramienta más la indicación del formato, por ejemplo, para la herramienta Semgrep se almacena como “semgrep_sarif.json”. Esto nos permite verificar si existe el fichero y evitar que se vuelva a ejecutar el análisis (optimización para minimizar el tiempo de ejecución).

Por último, para cada uno de estos archivos SARIF se obtiene la siguiente información, la cual se guarda en diferentes tablas dentro de una base de datos (SQLite) para posteriores análisis de resultados. Inicialmente se guardaba la información en archivos CSV (sigue la implementación en el código) pero se ha optado por el uso de una base de datos ligera para poder efectuar consultas, filtrar datos y en generar facilitar la labor de obtener datos de entrada para el módulo de análisis. El modelo de datos es el siguiente:

1. Tabla de OWASP: en esta tabla se hace una relación de CWE por cada una de las 10 categorías del Top 10. Esta tabla permite hacer una agrupación de los CWE por cada categoría de OWASP. Debido a que, en principio, existen muchas vulnerabilidades de CWE en los repositorios de SARD, se ha guardado la relación para automatizarlo en la fase de análisis.
2. Tabla de todos los proyectos: se guarda una relación de los proyectos y su identificador en SARD. Esta tabla permite guardar en base de datos todos los proyectos de SARD para mantener la trazabilidad. De esta manera en la base de datos esta la información necesaria para replicar las pruebas.
3. Tabla SARIF: se guarda la información obtenida del tratamiento de los datos. En esta tabla se guarda la información necesaria de cada uno de los proyectos (Tests Suite), sirviéndonos posteriormente como información para hacer la búsqueda de coincidencias). En esta tabla se guardan los campos del archivo SARIF del Test Suite:
    a. Package: nombre del paquete.
    b. Identifier: nombre del módulo del paquete.
    c. Type: tipo de reporte. En este caso, aplica “source code”.
    d. Language: Lenguaje de programación.
    e. State: tipo de valor del reporte. Este parámetro es fundamental, ya que especifica si se trata de una vulnerabilidad, un falso positivo o mixto.

Estos estados se representan respectivamente como: “bad”, “good” y “mixed” (se detalla más los siguientes apartados).

f. RuleId: identificador del CWE.
g. URI: ruta del archivo a analizar.
h. StartLine: linea de inicio de la vulnerabilidad.
i. filter_bad_or_good: campo adicional calculado a partir de un filtro en el que se verifica en la ruta del archivo si se trata de una vulnerabilidad o de un falso positivo para los casos de tipo mixed.

4. Tabla SEMGREP: Esta tabla guarda la información de cada uno de los análisis de la herramienta. La información que se almacena es:
    a. URI: ruta del archivo a analizar (incluye en nombre del módulo en la ruta).
    b. StartColumn: columna en la que se encuentra, dentro de la línea.
    c. StartLine: línea de inicio de la vulnerabilidad.
    d. RuleId: Identificador de la regla del analizador de la herramienta (no coincide con el CWE).
    e. Codigo_CWE: Identificador de la vulnerabilidad del CWE, con descripción incluida.
    f. Codigo_CWE_corto: solo el identificador del tipo de vulnerabilidad en el CWE.
    g. Package: nombre del paquete.
5. Tabla SEMGREP_FULL_RULES: En esta tabla se almacenan todas las reglas utilizadas en el escaner o análisis de Semgrep. En cada uno de los ficheros de SARIF de Semgrep, en uno de los campos se guardan todas las reglas utilizadas en el análisis. En este caso, se almacena esta información para poder comparar más adelante el alcance de la herramienta (los CWE que se detectan con la herramienta) en comparación con los CWE existentes en los Tests Suite.
    a. RULE_ID: este campo corresponde al identificador de cada regla y es único para cada una.
    b. CWE: Corresponde al identificador de la vulnerabilidad que abarca. En este caso viene con el título del CWE.
    c. CWE_CODE: Este campo es sólo el código del CWE, sin títulos u otra información.

En las tablas de SARIF y SEMGREP, se almacena en una misma tabla todos los proyectos (Test Suite) y los módulos que los componen. Para diferenciarlos correctamente, se almacena cada registro indicando su procedencia. Al hacerse uso de una base de datos, se puede filtrar de manera sencilla.

Para facilitar aún más la labor de análisis desde el ETL, se ha implementado el criterio que se va a aplicar y se ha procedido a revisar la documentación de SARD para analizar cómo se reporta las vulnerabilidades. Esta información se añade a las tablas de la base de datos en

En este caso, revisando la documentación se ha visto que uno de los parámetros de los reportes es el llamado “state”, que tiene los siguientes valores:

- good: este valor determina que la vulnerabilidad del reporte del archivo de SARIF se trata de una falsa alarma o falso positivo.

- bad: este valor determina que la vulnerabilidad del reporte del archivo de SARIF se trata de una alarma real o verdadero positivo.
- mixed: este valor no determina que la vulnerabilidad del reporte del archivo de SARIF se trate de una falsa alarma o no, sino que puede ser de cualquier tipo.

Este campo, permite realizar una primera valoración de lo que se puede obtener de los diferentes paquetes, ya que en la generalidad de los paquetes se encuentran debidamente identificados (con “bad” o “good”).

Sin embargo, en el caso de los proyectos de Java, el número de vulnerabilidades clasificadas como “bad” o “good” no son muy altas. Realmente son muchos los proyectos de Java que tienen un número alto de vulnerabilidades son los proyectos que identifican las vulnerabilidades como “mixed” y esto complica la etapa de análisis. Para solventar este problema, se ha revisado la documentación en la que se ha encontrado que una posible manera de clasificar las vulnerabilidades. En los casos de cada paquete, el código fuente tiene en su nombre de archivo las cadenas “bad” y “good” que permiten desambiguar las vulnerabilidades. No obstante, durante la evaluación inicial, se ha detectado que no todos los archivos tienen estas cadenas por lo que la posible reclasificación solo se puede llevar a cabo a un subconjunto determinado de las vulnerabilidades. Se puede ver extractos de la documentación consultada en las Ilustración 7 y Ilustración 8.

Ejemplos de nombres de test (15) en los que algunos están identificados en los nombre pero muchos otros no:

- Java test case CWE476_NULL_Pointer_Dereference__int_array_01 consists of one file:
  - CWE476_NULL_Pointer_Dereference__int_array_01.java
- Java test case CWE476_NULL_Pointer_Dereference__int_array_22 consists of two files:
  - CWE476_NULL_Pointer_Dereference__int_array_22a.java
  - CWE476_NULL_Pointer_Dereference__int_array_22b.java
- Java test case CWE476_NULL_Pointer_Dereference__Integer_54 consists of five files:
  - CWE476_NULL_Pointer_Dereference__Integer_54a.java
  - CWE476_NULL_Pointer_Dereference__Integer_54b.java
  - CWE476_NULL_Pointer_Dereference__Integer_54c.java
  - CWE476_NULL_Pointer_Dereference__Integer_54d.java
  - CWE476_NULL_Pointer_Dereference__Integer_54e.java
- Java test case CWE563_Unused_Variable__unused_public_member_variable_01 consists of two files:
  - CWE563_Unused_Variable__unused_public_member_variable_01_bad.java
  - CWE563_Unused_Variable__unused_public_member_variable_01_good1.java

Bad File for Class-Based Flaws

In a test case for a class-based flaw, the bad class:

- Is located in a file that ends in _bad (before the extension). For example, "CWE581_Object_ Model_Violation__hashCode_01_bad.java."
- Contains a required bad method with a signature like the bad method in a test case for a non-class-based flaw. This method makes use of the bad class for this test case to exercise the flaw being tested.
    - This is a public method in the class.
- The class is based on the file name (as required by the Java compiler). The class inherits AbstractTestCaseClassIssueBad.
- Has a main method that calls the bad method. Like the main methods in test cases for non-class-based flaws, this method is only used for testing or building separate archives for the test case.

Good File for Class-Based Flaws

In a test case for a class-based flaw, the good class:

- Is located in a file that ends in "_good1" (before the extension). For example, "CWE581_Object_ Model_Violation__hashCode_01_good1.java." Future versions of the test cases may include additional good files with names containing "_good2," "_good3," etc.
- Contains a required primary good method named "good" with a signature like the good method in a test case for a non-class-based flaw. Like the primary good method in a test case for a non-class-based flaw, this method only calls the secondary good method in this file.
    - This is a public method in the class.

Ilustración 7 - Captura de la guía de uso de "Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf" con el clasificado de los archivos de código fuente mediante etiquetas. Fuente SARD SAMATE.

3.4.1 Test Case File Names

Test case files are named with the following parts in order:

Part	Description	Optional/Mandatory
"CWE"	String Literal	Mandatory
CWE ID	Numerical identifier for the CWE entry associated with this test case, such as "36"	Mandatory
"_"	String Literal	Mandatory
Shortened CWE entry name	A potentially shortened version of the CWE entry name, with underscores between words, such as "Absolute_Path_Traversal"	Mandatory
"__" (two underscores)	String Literal	Mandatory
Functional Variant Name	A word or short phrase describing this particular variant of the issue, such as "fromConsole." This item is described further in Section 3.2 above.	Mandatory
"_"	String Literal	Mandatory
Flow Variant	A two digit integer value describing the type of complexity of the test case, such as "01," "02," or "61." This item is described further in Section 3.3 above.	Mandatory
Sub-file Identifier	A string that identifies this file in a test case consisting of multiple files, such as "a," "b," "_bad," "_good1." This item is described further in Section 3.4.2 below.	Optional
"."	String Literal	Mandatory
Language identifier / file extension	String Literal "java"	Mandatory

Table 3 – Test Case File Name Components

Ilustración 8 - Captura de la guía de uso de "Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf". Fuente SARD SAMATE.

Esta información es relevante, ya que condiciona los proyectos de Test Suite que se utilizarán como muestra. En el apartado de “Análisis de los Tests Suite” se detalla de manera más extensa.

4.4.1.1 Criterio final para los resultados

El criterio para determinar la coincidencia de los análisis con los datos esperados es una ampliación al definido inicialmente, que se ha basado en efectuar una comparación con en la que ambos deben coincidir en:

- El paquete del que procede.
- Paquete de la ruta del archivo (lo que implica en modulo concreto de manera implícita).
- Lenguaje de programación (la coincidencia tiene que ser exacta a nivel de archivo y por tanto de extensión).
- Vulnerabilidad dentro del CWE: la vulnerabilidad debe ser la misma. En este punto, se puede mejorar el filtrado de coincidencias, ya que la herramienta puede reportar un CWE diferente al esperado, pero pertenecer a la misma familia o niveles realmente cercanos de CWE.
- State: si se trata de una vulnerabilidad real TP (state=bad) o en un Falso positivo FP (state=good). Este parámetro de clave para el cálculo de TP, FP, FN y TN. Para los casos “mixed” se ha tratado de identificar adicionalmente buscando en el nombre de cada archivo si contiene la cadena “bad” y “good”.

Los paquetes que permiten determinar los TP y FP son los que tienen state= bad, porque el reporte de SARIF esperado (el que acompaña al test suite) solo incluye las vulnerabilidades que existen o TP, no recoge falsos positivos (todo lo que hay en el paquete son del tipo state=bad).

Ejemplo: si en el SARIF existe en un CWE-X (en un paquete, modulo y archivo) 3 vulnerabilidades (bad), supongamos los siguientes casos:

- La herramienta para ese CWE-X reporta 2, estas son 2 TP y ha dejado una sin reportar es 1 FN.
- La herramienta reporta para ese CWE-X las 3, que son las 3 que hay realmente, por tanto, son 3 TP.
- La herramienta reporta para ese CWE-X más de 3, por ejemplo 6, son 3 TP y 3 FP.

Los paquetes que permiten determinar los FP y TN son los que tienen state= good, porque el reporte de sarif (el que acompaña al test suite) solo incluye las vulnerabilidades que realmente no son vulnerabilidades o FP, es decir, no tiene una vulnerabilidad real, sino que el objetivo es que la herramienta emita una falsa alarma o un reporte de FP (todo lo que hay en el paquete es state=good).

Ejemplo: si en el SARIF existe en un CWE-X (en un paquete, modulo y archivo) 3 falsas vulnerabilidades (good), supongamos los siguientes casos:

- La herramienta para ese CWE-X reporta 2 son 2 FP y ha dejado una sin reportar es 1 TN.
- La herramienta reporta para ese CWE-X las 3, son 3 FP.
- La herramienta reporta para ese CWE-X más de 3, por ejemplo 5, son 3 FP + 5 FP (los que saque demás), ya que todo lo que este reportando son FP.

Para esta etapa de la investigación, se ha creado un script adicional para mantener los dos contextos de la investigación separados.

4.4.1.2 Sobre la implementación técnica

Se ha creado un proyecto Python que permite hacer la prueba de manera automática. La estructura del módulo de extracción es:

```
engine-sarif
├── log
├── resources
│   ├── bruto
│   ├── descargas_test_suites
│   ├── doc
│   ├── procesado
│   ├── tools
│   └── unzips_test_suites
├── src
│   ├── config
│   │   ├── config.py
│   │   └── __init__.py
│   ├── dblite.py
│   ├── __init__.py
│   ├── main.py
│   └── utils
│       ├── __init__.py
│       ├── statics.py
│       └── utils.py
└── venv
```

Ilustración 9 - Estructura del módulo de extracción de datos.

Las funciones principales son:

- resources: en esta carpeta se ubican los recursos que se irán descargando y utilizando, donde:
  - descarga_test_suites: ubicación de los archivos comprimidos de los test suites de SARD.
  - procesado: en esta ubicación se guarda información para depuración.
  - tools: en esta ubicación se descargarán, cuando se implemente, las herramientas a analizar. Es interesante mantener lo más automatizado el proceso.
  - unzips_test_suites: es esta ubicación de descargan los tests suites, donde se obtienen los archivos a analizar y el archivo sarif con los resultados esperados. Una vez analizados, se guarda el archivo sarif de la herramienta (con su nombre para identificarla) y no se vuelve a analizar si existe, a menos que se fuerce.
- src: en esta ubicación se concentra el código fuente:
  - config: este paquete contiene la información necesaria para la configuración de la herramienta.
  - utils: este paquete contiene métodos y funciones que se utilizan en varias ocasiones, con el objetivo de minimizar el código duplicado.

- sqlite: este archivo contiene la configuración del CRUD de la base de datos donde se vuelca la información más relevante.
- main: este archivo es el archivo ejecutable que corre el módulo de extracción

Finalmente, para el análisis de datos se desarrollado un archivo de Jupyter que permita desarrollar y visualizar fácilmente los datos que se obtienen de la base de datos (cargados por ahora como archivos “csv”). Este script permite obtener las métricas que se verán en los siguientes capítulos.

4.4.1.3 Problemas detectados y soluciones

La fase de desarrollo se ha visto ralentizada notablemente un consumo de tiempo excesivo en las siguientes tareas:

- Tiempo muy elevado para el parseo de los archivos JSON (SARIF). Los archivos pesan cientos de MB, lo que requiere muchísimo tiempo para su parseo (para paquetes grandes los tests se han demorado incluso horas). A su vez, se ven incrementado por la aparición de errores varios de ejecución tales como nombres de archivos que ocasionaban error de exceso de longitud de la ruta del archivo y otros de esta naturaleza.
- Tiempos de ejecución de las pruebas excesivamente altos y que obedecen fundamentalmente a dos causas:
  - La ejecución de las pruebas en las herramientas de revisión de código fuente, ya que es proporcional el tiempo de análisis al número de ficheros a analizar.
  - La ejecución de la compilación de los proyectos para las herramientas de revisión de byte-code. En el caso de los proyectos que requieres ser compilados se incrementa notablemente el tiempo. Este factor está íntimamente ligado a dos factores principales:
    - La compilación del código fuente del propio proyecto.
    - Descarga de las dependencias de terceros.
- Los sets de pruebas de SARD emplean ejemplos de código con vulnerabilidades conocidas que en principio vienen acompañados de un reporte en formato SARIF para contrastar los datos con la herramienta, es decir, especifica si se trata de un TP o un FP. Sin embargo, se ha detectado que en algunos casos no es así o hay matices en el reporte que complican la clasificación de la vulnerabilidad como TP o FP. En la fase de análisis se han encontrado una serie de incidencias que han contribuido a ralentizar esta fase de la investigación, así como a rebajar el alcance y expectativas del PoC y que se procede a pormenorizar a continuación:
  - El formato SARIF es un estándar, pero se ha detectado que las herramientas no reportan las vulnerabilidades con una compatibilidad del 100%. Existen casos en que herramientas como, por ejemplo, Semgrep describe los CWE de las vulnerabilidades en áreas del formato más relacionadas con los metadatos (nodo de “tag”).

  - En el caso de la herramienta de Semgrep, esta información se encuentra en el nodo tag y no se encuentra en los nodos esperados.
  - Los CWE de las herramientas y el reporte esperado puede diferir aun tratándose de la misma vulnerabilidad. Es decir, el nivel del tipo de vulnerabilidad del CWE puede ser del mismo tipo, pero de niveles diferentes. Este punto hay que tenerlo en cuenta en futuras investigaciones.
  - Los tests suite y tests case, tienen un conjunto de vulnerabilidades clasificadas según el lenguaje pueden ser limitados para el objetivo de la investigación.
  - El alcance de cada uno de los test suite ha sido otro factor determinante en esta fase. Los tests cases contienen conteniendo un número de casos discreto y contemplando un conjunto de vulnerabilidades concreto y especifico, esto llegar a ser muy poco representativo y exhaustivo sobre el Top10 de OWASP, ya que no se abarcan todos los capítulos de OWASP. Por ejemplo, para PHP existe un paquete para las vulnerabilidades de SQL injection y XSS exclusivamente, que en el Top 10 de OWASP forman parte de 1 ó 2 puntos como máximo.
- Hay que mencionar otro limitante derivados de la tipología de los paquetes que los forman pudiéndose distinguir dos tipos generales:
  - Proyectos reales: Consistente en paquetes de proyectos de open source en los que se conocen vulnerabilidades y/o falsos positivos. Estos paquetes no cuentan con muchas vulnerabilidades, por lo que la volumetría puede resultar baja desde una perspectiva más analítica.
  - Proyectos de tests vulnerables: Estos paquetes son proyectos que se han creado con muchos casos de vulnerabilidades, donde o existen muchos tests para un tipo de vulnerabilidades o muchos tests para un conjunto concreto de tipos de vulnerabilidades. Inicialmente estos paquetes constituyen la configuración ideal para este estudio. Sin embargo, algunos de los reportes de vulnerabilidades de los paquetes de SARD son inconcluyentes o ambiguos (este problema se concentra sobre todo en estos paquetes), es decir, hacen uso de una clasificación de la vulnerabilidad no especifico. Esto complica el análisis de los reportes, debido que obliga a buscar maneras alternativas para determinar la clasificación, complicando el desarrollo de las pruebas y siendo una de las causas principales para rebajar el alcance de la investigación tal como se concebía inicialmente, junto a la que se procede a comentar a continuación.
- Las herramientas de análisis no tienen reglas de detección para todas las vulnerabilidades. Es decir, el subconjunto de vulnerabilidades que pueden analizar no tiene por qué coincidir con el de los paquetes o test suites de SARD. Esto puede producir que haya test suites que se analicen y los resultados obtenidos sean cercanos a cero o cero en todos sus puntos. Hay que tener en cuenta que puede analizar las herramientas de SAST para determinar la viabilidad del análisis y los resultados. Si una herramienta de SAST solo puede analizar un conjunto de CWE y no están en los Tests Suites, no podemos realizar un benchmark sobre esas vulnerabilidades.

- Los proyectos de SARD están preparados para ser analizados tanto en pruebas SAST, DAST y SCA. Esto significa que, en principio, tienen la información necesaria para ser compilados y en determinados casos, ejecutados. Para esto, contienen:
    - Makefile: Archivo que permite mediante un comando compilar el proyecto. Este archivo es clave para su construcción, ya que permite de una manera simplificada compilar los Test Suites para ser analizados por las herramientas de SAST que requieren el software compilado.
    - Dockerfile: Archivo que permite crear un contenedor para correrlo. Este otro elemento puede tener utilidad a futuro, ya que en caso de que no sea sencillo automatizar la compilación de los proyectos, puede ser interesante generar el contenedor de Docker para realizar las operaciones de compilación y posteriormente extraer la información necesaria.

Sin embargo, la automatización de estos procesos puede no ser tan sencilla como puede parecer. En el caso de incluir la herramienta de pruebas de SAST para Java, esta requiere que se compile el proyecto de Java debido a que efectúa un análisis de los "byte-code" (y no del código fuente) del proyecto. Pero durante el análisis para automatizar este proceso, se ha visto que hay versiones diferentes de Java y del constructor "Ant". Para poder automatizar este proceso, es probable que se deban hacer implementaciones que analicen la versiones de estos y permitan compilar y analizar los proyectos. Por otro lado, se han efectuado pruebas en las que incluso utilizando las versiones especificadas daba errores de compilado del proyecto.

Para resolver los problemas comentados se ha procedido a efectuar las siguientes optimizaciones:

- Guardado de los archivos JSON generados por las herramientas para sólo calcular los proyectos nuevos.
- Volcado parcial de la información necesaria para la etapa de análisis de las herramientas SAST a una pequeña base de datos de tipo SQLite. Con esto evitamos reanalizar en cada ejecución cada proyecto de SARD, ya que cada herramienta de SAST añadirá un tiempo considerable a las pruebas. Guardando esta información, podemos procesarla en la etapa de comparación directamente con los resultados del SARIF del proyecto de SARD.
- Volcado parcial de la información procedente de los archivos de SARIF de cada uno de los proyectos de SARD a la base de datos SQLite. Este procesamiento permite tener los datos con los que hay que contrastar los resultados de las herramientas de SAST. Con esta optimización, podemos comparar los resultados de cada herramienta directamente.
- Creación de una memoria SWAP virtual en modo archivo de respaldo de la memoria RAM física. Esta optimización se hace necesaria para varias etapas:
    - En el análisis de código fuente y/o byte-code, las herramientas de SAST se hacen demandantes de los recursos del equipo, llegando a consumir muchos recursos ya que suelen trabajar en modo multi-thread. En el caso de Semgrep, hace uso de unos 4-6 hilos para efectuar el análisis. Por otro lado, la memoria RAM utilizada puede ser más de la esperada, pero

al crear ficheros de reporte de tipo SARIF (que suelen ser pesados) hay mucho consumo de memoria.
    - En la etapa de análisis hay que efectuar una carga de archivos pesados en memoria y procesarlos. El trabajo con archivos de texto grandes aumenta el consumo de memoria y junto con otros procesos pueden producir fallos en la ejecución.
- Se ha establecido los procesos de ejecución de los escáneres con prioridad “muy alta” para evitar el cierre por parte del sistema operativo ante otros procesos.
- Uso de versiones de Java y Ant para construir los proyectos que cubran la mayor cantidad de paquetes. La versión de Java 8 y Ant 1.7.3 tienen alta compatibilidad con la mayoría de los proyectos.
- Durante la ejecución, se hacen descargas de los archivos zip que contienen cada uno Test Suite de SARD. Debido al peso de los proyectos, se ha optimizado el borrado de cada paquete zip una vez descomprimido, para limitar el consumo de la memoria del disco.

4.4.2 Análisis de datos

En este módulo de la PoC se ha creado una serie de scripts en formato Jupyter, ya que la explotación de los datos se hace mediante librerías típicas de ciencia de datos cómo:

- Pandas: librería para la conversión de la salida del ETL en DataFrames que permitan trabajar con altos volúmenes de datos.
- MatPlot: librería para la representación de la información y realización de gráficos.

En esta fase, se hace uso de la información que se ha almacenado en la base de datos y se aplica el criterio anteriormente detallado (apartado 3.3 y 3.4). Mediante los scripts se ha podido realizar las comparaciones de las dos tablas donde se encuentras las vulnerabilidades detectadas por la herramienta de seguridad, Semgrep, en la tabla “SEMGREP” y los resultados esperados de SARD en la tabla “SARIF”. Para realizar la comparación efectúan operaciones con los dataframes siguientes se aplica el criterio definido en el apartado 4.4.1.1.

Ahora con los dos dataframes se puede efectuar la operación que permita encontrar la coincidencia, en este caso, lo que interesa encontrar es la interferencia de los dos conjuntos.

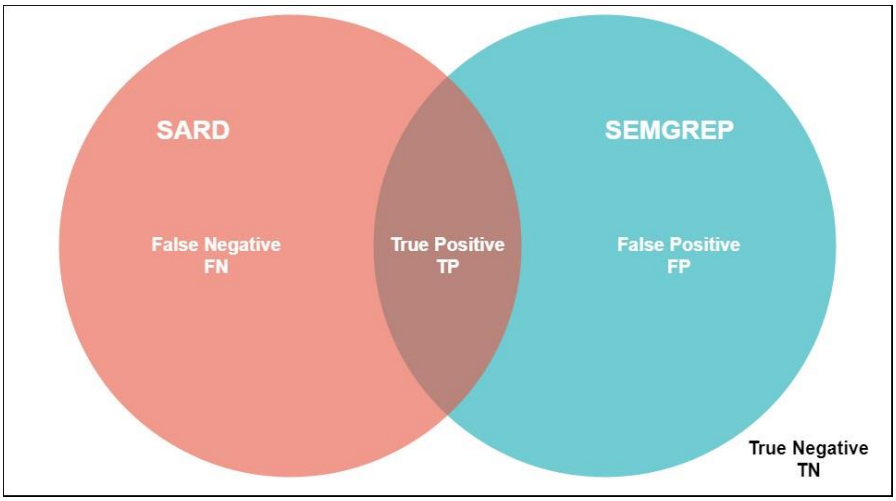

Ilustración 10 - Diagrama de Venn del planteamiento del criterio a los dataframes, para la herramienta Semgrep. Elaboración propia.

Cómo se puede ver en la Ilustración 10, se identifican las diferentes áreas que forman la matriz de confusión. Este criterio se aplica por CWE esperado de cada Test Suite de SARD. Las áreas del diagrama corresponden a:

- True Positive: corresponde a las vulnerabilidades reales correctamente detectadas. Para este caso se hace uso de la función de coincidencia de los dataframes donde el dataframe de SARD especifica explícitamente que es vulnerable.
- False Positive: corresponde a vulnerabilidades reportadas por la herramienta que no son vulnerabilidades reales. Para este caso se hace uso de la función de coincidencia de los dataframes donde el dataframe de SARD especifica explícitamente que no es vulnerable y es un falso positivo.
- False Negative: corresponde a vulnerabilidades no reportadas por la herramienta que son vulnerabilidades reales. Para este caso se hace uso de la función de coincidencia de los dataframes donde el dataframe de SARD especifica explícitamente que no es vulnerable y es un falso positivo.
- True Negative: corresponde a vulnerabilidades no reportadas por la herramienta que no son vulnerabilidades reales. Para este caso se hace uso de la función de coincidencia de los dataframes donde el dataframe de SARD especifica explícitamente que no es vulnerable y es un falso positivo. Es decir, la herramienta no ha reportado un falso positivo.

Por otro lado, en los siguientes apartados se analiza el contenido de los Tests Case de SARD para conocer la diversidad de vulnerabilidades y de pruebas que presentan con el fin de conocer el alcance de los mismos.

Finalmente, en el capítulo de “Análisis de los resultados” se presenta la información final sobre los lenguajes de programación analizados.

4.5 Análisis de los Tests Suite de SARD

La presente investigación se propone llevar a cabo una evaluación más detallada del SARD Suite proporcionado por el proyecto SAMATE. Este conjunto de pruebas y casos de prueba se ha concebido con el objetivo de evaluar la seguridad del software, centrándose en la identificación de vulnerabilidades comunes. El repositorio se encuentra accesible en la dirección https://samate.nist.gov/SARD/test-suites , albergando un buen número de tests suites o test cases que abarcan cinco tecnologías clave: C++, Java, PHP, C#, y C.

Para lograr una descripción eficiente del contenido del repositorio, se presenta en la Ilustración 11 una representación gráfica que ilustra los principales datos. Cómo se puede observar, se destacan las cinco tecnologías respaldadas por el repositorio, a saber, C++, Java, PHP, C#, y C. Un análisis rápido revela un claro predominio del lenguaje C++, cuyo número de tests suite supera en conjunto a los otros cuatro. Este hecho sugiere la relevancia y enfoque especializado en esta tecnología específica en el contexto de las evaluaciones de seguridad de software.

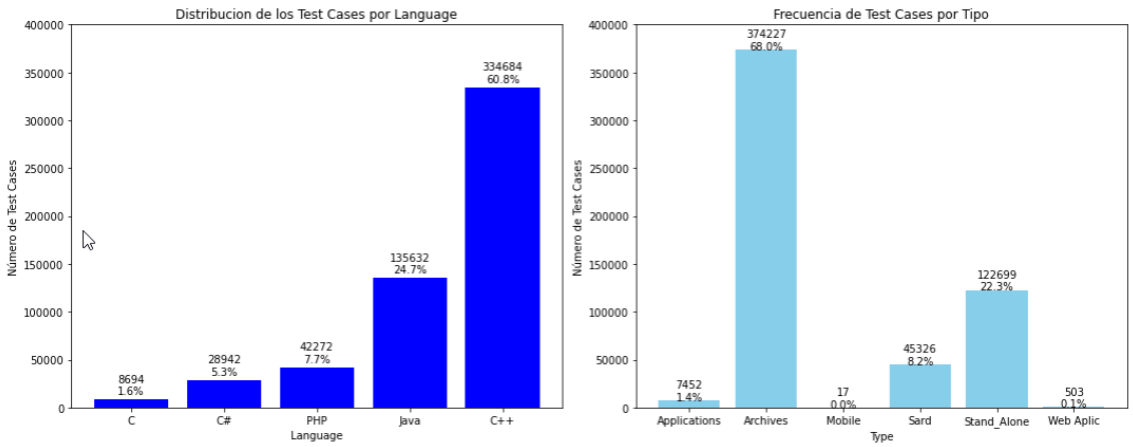

Ilustración 11 - Gráficas de distribución de las vulnerabilidades en los Test Suite y por tipo de aplicación.

En imagen de la derecha de la Ilustración 11, se clasifican los tests suites en cinco categorías principales:

1. Standalone Applications: Este grupo se enfoca en programas de software que operan de manera independiente, sin depender de otro software para su ejecución. Las pruebas buscan identificar vulnerabilidades específicas de estas aplicaciones, como posibles problemas de seguridad en el manejo de datos locales, accesos no autorizados y debilidades en la implementación del código.

2. Mobile Applications: Se centra en la evaluación de aplicaciones diseñadas para dispositivos móviles, como teléfonos inteligentes y tabletas. La atención se dirige hacia la identificación de vulnerabilidades específicas de las aplicaciones móviles, abordando temas como la gestión de datos sensibles, ataques de intercepción y manipulación de datos, así como debilidades en la autenticación y autorización.

3. Stand-alone Applications: Esta categoría se refiere a aplicaciones independientes que no requieren de otros programas para su ejecución. Las pruebas se centran en la evaluación de vulnerabilidades específicas de este tipo de aplicaciones, incluyendo problemas de seguridad en la entrada de datos, manipulación de archivos locales y la gestión adecuada de recursos del sistema.

4. Stand-alone Suites: Este grupo engloba conjuntos de aplicaciones independientes diseñadas para trabajar en conjunto como una suite o conjunto de herramientas. Las pruebas buscan identificar problemas de seguridad en la interacción entre las aplicaciones dentro de la suite, así como en la forma en que comparten datos y recursos.

5. Web Applications: Se centra en evaluar aplicaciones que funcionan a través de navegadores web, abordando vulnerabilidades específicas relacionadas con la seguridad web, como ataques de inyección (por ejemplo, SQL injection, XSS), gestión de sesiones, control de acceso y otros riesgos asociados con el desarrollo de aplicaciones web.

6. SARD Suites: Esta categoría hace referencia al conjunto general de pruebas y casos de prueba proporcionados por el Software Assurance Reference Dataset. Dichas pruebas abarcan diversas categorías y tecnologías, diseñadas para evaluar la seguridad del software en diversos contextos, incluyendo aplicaciones standalone, web y móviles.

En este gráfico, se evidencia que las categorías predominantes son "Archives" con un 68% del total y "Stand Alone" con un 22%. Este hallazgo proporciona una visión inicial sobre las áreas de enfoque en la evaluación de seguridad de software, donde la manipulación de archivos y la autonomía de las aplicaciones se consideran críticas.

A continuación, se hace un análisis de cada una de las tecnologías para conocer que ofrecen.

4.5.1 Distribución de vulnerabilidades por lenguaje de programación y su categoría

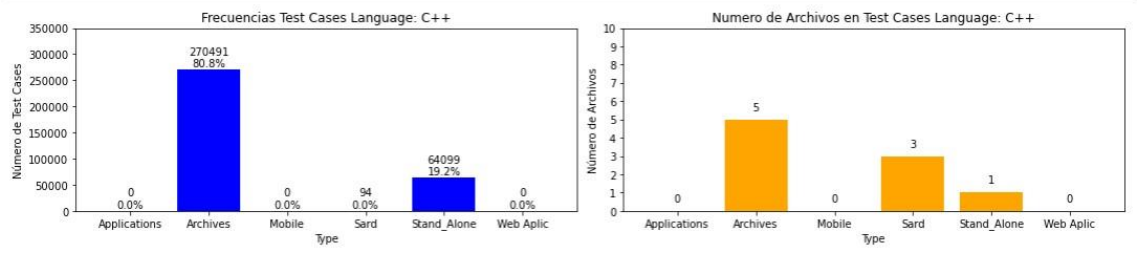

Ilustración 12 -Diagrama de distribución de C++ de vulnerabilidades por tipo de archivo de Test Suite y distribución de los archivos existentes por cada tipo de Test Suite. Elaboración propia.

Cómo se ve en la Ilustración 12, en el caso del lenguaje C++ se destaca como el de mayor número de tests cases, distribuidos en tres categorías con 9 archivos, 5 en "Archives," 3 en "SARD," y 1 en "Stand Alone."

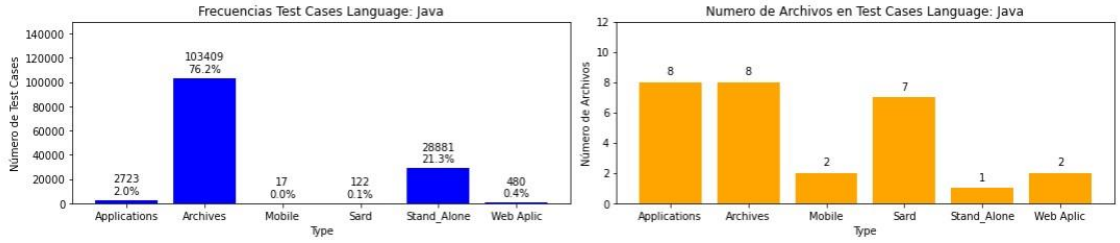

Ilustración 13 - Diagrama de distribución de Java de vulnerabilidades por tipo de archivo de Test Suite y distribución de los archivos existentes por cada tipo de Test Suite. Elaboración propia.

Cómo se puede observar en la Ilustración 13, para el caso de Java sigue en importancia, presentando el mayor número de archivos con un total de 28, abarcando todas las categorías. Este resultado posiciona a Java como el test case más representativo en el contexto del SARD Suite. No obstante, es fundamental profundizar en un análisis posterior para comprender la naturaleza y diversidad de las vulnerabilidades identificadas en este lenguaje.

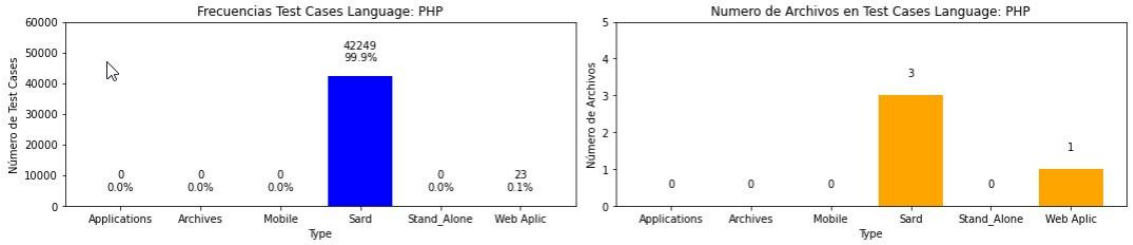

Ilustración 14 - Diagrama de distribución de PHP de vulnerabilidades por tipo de archivo de Test Suite y distribución de los archivos existentes por cada tipo de Test Suite. Elaboración propia.

Tal como se ve Ilustración 14, la proporción de tests existentes para este lenguaje de programación es inferior a los anteriores lenguajes de programación. No obstante, parece que el número de vulnerabilidades es suficientemente interesante. Además, este lenguaje de programación es relevante en el mundo de IT, ya que se trata de un lenguaje de programación ampliamente extendido en muchas aplicaciones Web.

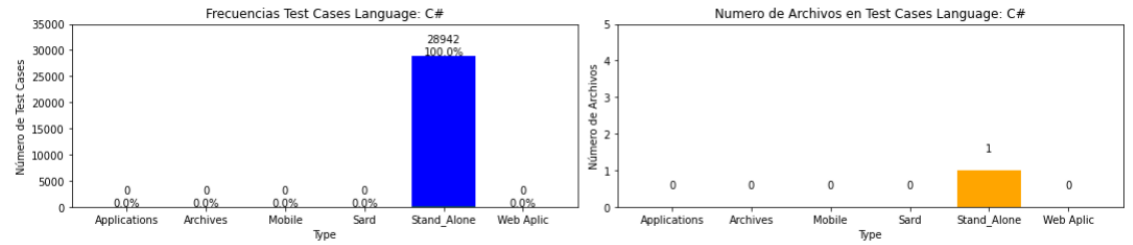

Ilustración 15 - Diagrama de distribución de C# de vulnerabilidades por tipo de archivo de Test Suite y distribución de los archivos existentes por cada tipo de Test Suite. Elaboración propia.

En la Ilustración 15, se puede apreciar como el volumen de vulnerabilidades es inferior a los casos anteriores. No obstante, el volumen sigue siendo relevante para efectuar pruebas.

Cabe destacar que, el número de tests case para el resto de los lenguajes, como PHP y C#, disminuye ostensiblemente. En un esfuerzo por contextualizar aún más la evaluación del SARD Suite de SAMATE, es interesante vincular estos resultados con el OWASP Top 10 de 2021.

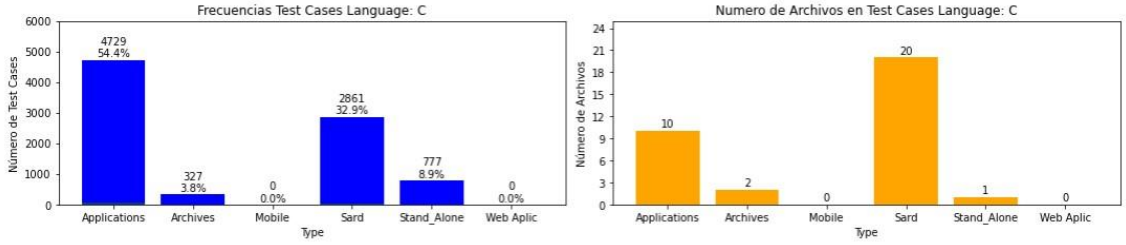

Ilustración 16 - Diagrama de distribución de C de vulnerabilidades por tipo de archivo de Test Suite y distribución de los archivos existentes por cada tipo de Test Suite. Elaboración propia.

Por último, se puede apreciar el caso de C en la Ilustración 16, donde se puede apreciar un volumen de vulnerabilidades mucho más bajo que en el resto de los casos. No obstante, el volumen es suficiente para realizar pruebas. Este lenguaje no es muy utilizado en aplicaciones Web, pero si como librerías o desarrollos más orientados a aplicaciones de escritorio o similares, tal como se puede observar en la distribución de los casos por tipo de lenguaje (gráfico derecho).

Finalmente, se ha revisado el contenido de cada categoría para tratar de relacionarlo con las categorías de vulnerabilidades que contienen.

4.5.2 Contenido de los Tests Suites por categoría

La categoría “Standalone Applications” del SARD Suite, vinculada con "Security Misconfiguration" y "Insecure Direct Object References" en el OWASP Top 10, se enfoca en evaluar aplicaciones autónomas. Estas pruebas son esenciales para identificar y corregir configuraciones inseguras y referencias directas a objetos, reduciendo así los riesgos de accesos no autorizados o manipulación indebida de datos. Ejemplos notables en esta categoría incluyen IARPA STONESOUP en Asterisk y Wireshark.

Las pruebas específicas en Mobile Applications del SARD Suite se alinean con "Insecure Data Storage," "Broken Authentication," y "Security Misconfiguration" en el OWASP Top 10. Abordan vulnerabilidades como el almacenamiento inseguro de datos y problemas de autenticación en aplicaciones móviles. Destacan ejemplos como IARPA STONESOUP en Wireshark y Open Fire.

Similar a la categoría anterior, “Stand-alone Applications” del SARD Suite se relaciona con "Security Misconfiguration" e "Insecure Direct Object References (IDOR)" en el OWASP Top 10. Estas pruebas se centran en vulnerabilidades de aplicaciones

independientes, específicamente en la corrección de configuraciones inseguras y referencias directas a objetos. Ejemplos notables abarcan Apache JMeter y Apache Lenya.

El análisis de Stand-alone Suites en el SARD Suite se relaciona con "Security Misconfiguration" e "Insecure Direct Object References (IDOR)" en el OWASP Top 10. Además de abordar configuraciones inseguras y referencias directas a objetos, se centra en la interacción entre aplicaciones en la suite. Este enfoque contribuye a mitigar amenazas asociadas con configuraciones erróneas y referencias de objetos inseguras. Notables ejemplos incluyen Juliet C# y IARPA STONESOUP en Virtual Machine.

La categoría Web Applications del SARD Suite se superpone directamente con varias categorías del OWASP Top 10, como "Injection," "Cross-Site Scripting (XSS)," "Session Management," y "Access Control." Identifica y evalúa vulnerabilidades específicas relacionadas con la seguridad web, como inyecciones SQL y scripting entre sitios. Ejemplos notables abarcan WordPress, JSP Wiki y Apache-tomcat.

La categoría general de SARD Suites abarca diversas tecnologías y contextos, integrando resultados de pruebas en distintas categorías del OWASP Top 10. Esta amplitud permite una visión integral de vulnerabilidades en aplicaciones standalone, web y móviles, fortaleciendo la postura de seguridad del software de manera coordinada. Ejemplos notables incluyen ITC-Benchmarks y el conjunto de pruebas PHP Vulnerability Test Suite.

Para efectuar este análisis se ha tenido que completar una gran parte de la implementación del ETL, por lo que no se ha podido realizar en un periodo más temprano de la investigación, pero los resultados obtenidos son interesantes para entender el progreso de la investigación y ofrecer recomendaciones para otras investigaciones.

4.6 Herramientas evaluadas

4.6.1 Semgrep

Actualmente, se ha efectuado la prueba de concepto con la herramienta “Semgrep”. Se ha escogido por esta herramienta debido a que el análisis lo efectúa haciendo revisión del código fuente sin necesidad de compilarlo y tiene, soporte para diferentes lenguajes de programación: C/C++, Java, C#, PHP. Esto permite hacer una primera versión de la PoC con los mínimos requisitos iniciales, obteniendo datos de manera más ágil. En el caso de herramientas como Spotbugs, son analizadores de byte-code, lo que implicaría una compilación previa, lo que puede complicar una primera recolección de datos y se ha dejado como siguientes pasos.

Por otro lado, cabe desatacar que esta herramienta posee doble licenciamiento, es decir, hay una sección de uso libre (versión community) y otra privativo (versión PRO).

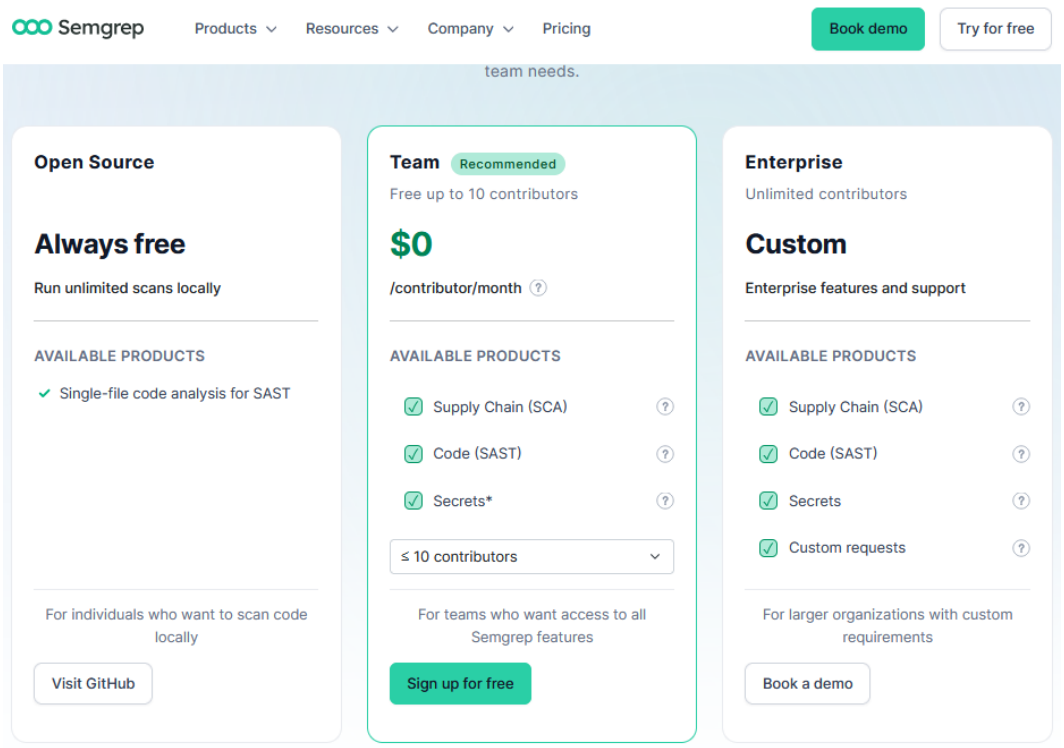

Ilustración 17 - Captura de plan de precios de Semgrep durante la investigación. Imagen obtenida desde documentación oficial[3].

Inicialmente, al instalar Semgrep, el set de reglas de detección de vulnerabilidades está limitado, ya que es el funcionamiento sin licenciamiento. Para las pruebas, se ha registrado una cuenta de manera gratuita que permite el uso de la herramienta con una licencia limitada (gratuita) de hasta 10 desarrolladores (puede verse en la Ilustración 17). Para las pruebas, era suficiente y habilitaba el set completo de reglas de detección.

Lenguaje	Reglas community	Reglas PRO
C#	35	62
Java	112	172
PHP	29	47
C	7	7
C++	0	0

Tabla 2 - Listado de reglas de Semgrep por lenguaje de programación (en las cuales algunas cubren los mismos CWE). Elaboración propia desde la documentación oficial[4].

Cómo se puede ver en la Tabla 2, para algunos lenguajes la cobertura de las reglas (por tanto, el alcance del análisis) puede suponer hasta un incremento de un 50% aproximadamente. Para otros lenguajes, como C/C++, no es una herramienta de SAST de gran cobertura. Además, cabe destacar el número de reglas de seguridad no es directamente proporcional a los CWE o vulnerabilidades que cubre Semgrep, ya que hay casos en el que un subconjunto de las reglas comprueba el mismo CWE.

[3] https://semgrep.dev/pricing/
[4] https://semgrep.dev/p/java

Lenguajes evaluados

Esta herramienta permite el análisis de algunos de los lenguajes del set de pruebas de SARD, en las que en varias la herramienta cuenta con un set de reglas de detección de vulnerabilidades estable. Sin embargo, hay otros lenguajes cuyo análisis está en modo experimental durante la PoC, por lo que los resultados pueden ser inferiores a los esperados por el alcance limitado de las reglas de análisis (véase la Tabla 2).

Lenguajes	Semgrep OSS Engine	Semgrep Pro Engine (cross-function)	Semgrep Pro Engine (cross-file)
C#	GA	GA	GA
Java	GA	GA	GA
PHP	GA	GA	--
C	Experimental	--	--
C++	Experimental	--	--

Tabla 3 - Listado con el estado de desarrollo y análisis de las reglas de Semgrep. Obtenido de la documentación oficial (14).

Cómo se puede observar en las dos tablas anteriores, C y C++ no tienen unas reglas y/o alcance como para ser evaluados, además de tratarse de tecnologías aun en formato de análisis “experimental”.

Conjunto de vulnerabilidades detectables desde Semgrep

Hay que tener en cuenta que esta herramienta funciona evaluando el código fuente directamente mediante el uso de patrones o coincidencias. Por tanto, como todas las herramientas, tienen un alcance limitado al conjunto de reglas que utiliza en el análisis y que suelen estar limitadas a subconjuntos de CWE concretos (se puede consultar en la tabla de reglas de Semgrep donde se vuelcan todas las reglas y su CWE).

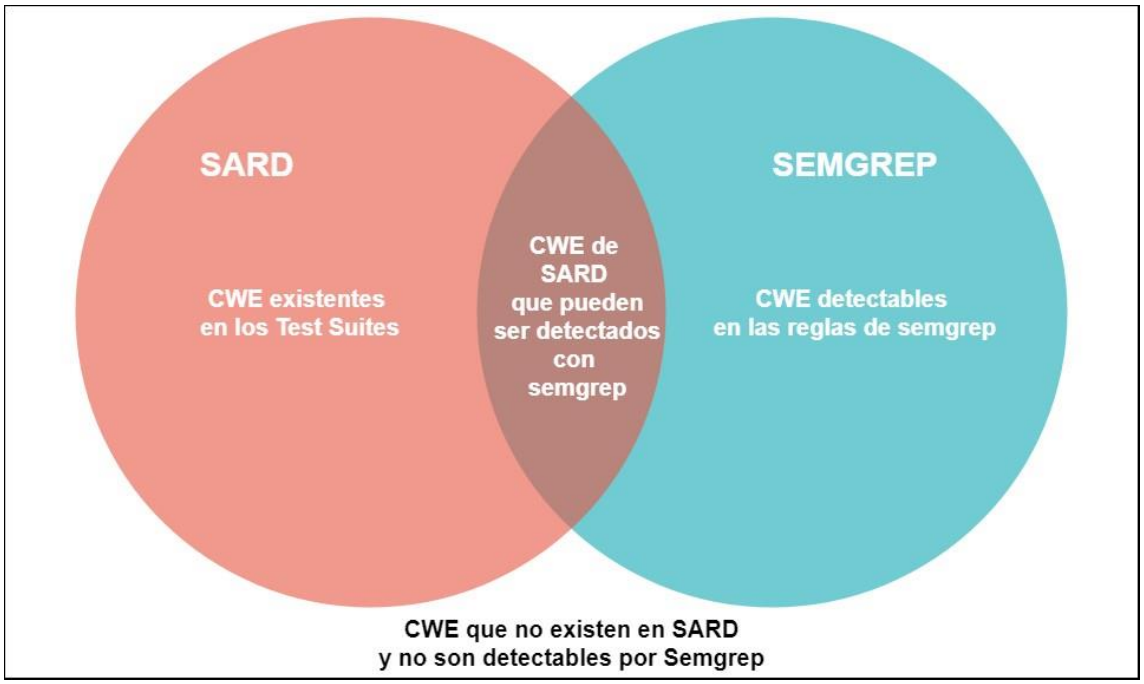

Ilustración 18 - Diagrama de Venn del alcance de los Tests Suite de SARD y el alcance de las reglas de Semgrep. Elaboración propia.

En cualquier set de pruebas hay un conjunto de vulnerabilidades que se pueden detectar y en las herramientas de análisis hay otro conjunto de vulnerabilidades detectables por la misma. Cómo se puede observar en la Ilustración 18, en el caso de SARD y Semgrep hay que tener en cuenta que las áreas pueden ser de tamaños muy diferentes. En este caso, lo ideal es que las áreas se superpongan y la interferencia sea máxima (además de muy reducida el área exterior correspondiente a las vulnerabilidades que no están en los tests).

Paquetes seleccionados

Debido a todo lo anteriormente comentado, se ha procedido a establecer una clasificación de los paquetes que tienen mayor interés a nivel de detecciones de las herramientas.

En los siguientes capítulos, dividido para cada lenguaje, se detallan los paquetes utilizados y su justificación.

Limitaciones y optimizaciones

Es importante reseñar otro factor limitante a la hora de obtener resultados con la herramienta de Semgrep, puesto que en determinados tests suite se han detectado casos particulares. Uno de ellos, es que se emite como reporte “java” archivos con extensión “jsp”, pero en determinados casos hay que especificarle a la herramienta de seguridad el tipo de lenguaje especifico que se va a analizar, quedando fuera otros. Por ejemplo, en Semgrep podemos analizar solo “java” con la directiva “p/java”, pero esto excluye a estos archivos que SARD identifica como “java” pero en realidad no lo son. Por tanto, se recomienda lanzar el análisis en modo “auto”, donde primero detecta los

tipos de archivos a analizar y después los escanea (con el consecuente sobre coste computacional ya que el modo auto activará las reglas de todos los posibles lenguajes de programación u similares detectados como, por ejemplo, las reglas de Docker donde se analizan los “yaml” de generación de las imágenes).

Finalmente, se debe valorar la implementación de la PoC como una toma de contacto para futuros trabajos y se aconseja seguir las indicaciones, así como tener en cuenta los problemas detectados.

4.6.2 Spotbugs

Para esta herramienta se ha implementado el código correspondiente a la compilación de cada uno de los módulos de cada proyecto (o Test Suite). Para esta tarea se ha automatizado el proceso de descarga de la herramienta junto con dos plugins muy útiles para extender las reglas que vienen por defecto, añadiendo un mayor alcance de seguridad. Los dos plugins son:

- Findsecbugs
- Fb-contrib.

Sin embargo, tras revisar los registros del script de ETL, se ha evidenciado que no es una tarea trivial la compilación para ser automatizada desde un Script de Python (de manera genérica).

En la documentación y recursos de cada uno de los Tests Suite hay indicaciones para efectuar la compilación, pero tras la revisión de diferentes casos, se ha podido verificar que las especificaciones a veces son muy diferentes o no retrocompatibles (versiones muy diferentes de Java, Ant u otras tecnologías necesarias).

Debido a estas dificultades y sobre todo el plazo de la investigación, no se ha implementado esta herramienta en la PoC.

Para futuras investigaciones, puede ser interesante el uso de soluciones como Jenkins para la integración herramientas al benchmark. Se adiciona una propuesta en el apartado “Futuros trabajos”.

5 Análisis de los resultados

Con el fin de limitar el alcance de las pruebas, se ha determinado efectuar las pruebas sobre un conjunto de lenguajes limitado. Para la investigación, se ha seleccionado:

- Java: ampliamente utilizado en cualquier organización. Muchos productos hacen uso de este lenguaje de programación como, por ejemplo, AutoFirma[5].
- PHP: ampliamente utilizados en blogs, e-commerce, etc… Por ejemplo, productos como Moodle y WordPress, están desarrollados con este lenguaje de programación. En la Ilustración 19 se evidencia que WordPress presenta una cuota de mercado en internet lo suficientemente importante como para incluir el lenguaje de PHP.

Historical yearly trends in the usage statistics of content management systems

This report shows the historical trends in the usage of the top content management systems since January 2012.

	2012 1 Jan	2013 1 Jan	2014 1 Jan	2015 1 Jan	2016 1 Jan	2017 1 Jan	2018 1 Jan	2019 1 Jan	2020 1 Jan	2021 1 Jan	2022 1 Jan	2023 1 Jan	2023 23 Dec
None	71.0%	68.2%	64.8%	61.7%	56.6%	53.3%	51.3%	45.3%	43.1%	38.3%	33.8%	32.3%	31.5%
WordPress	15.8%	17.4%	21.0%	23.3%	25.6%	27.3%	29.2%	32.7%	35.4%	39.5%	43.2%	43.1%	43.1%
Shopify			0.1%	0.3%	0.4%	0.6%	0.9%	1.4%	1.9%	3.2%	4.4%	3.8%	4.1%
Wix		<0.1%	0.1%	0.1%	0.2%	0.3%	0.4%	1.0%	1.3%	1.5%	1.9%	2.4%	2.6%
Squarespace	0.1%	<0.1%	0.1%	0.2%	0.4%	0.5%	0.7%	1.4%	1.5%	1.4%	1.8%	2.0%	2.1%
Joomla	2.8%	2.8%	3.3%	3.3%	3.3%	3.4%	3.2%	3.0%	2.6%	2.2%	1.7%	1.8%	1.7%
Drupal	1.9%	2.3%	1.9%	2.0%	2.1%	2.2%	2.3%	1.9%	1.7%	1.5%	1.3%	1.2%	1.1%
Adobe Systems												1.1%	1.0%
PrestaShop		0.3%	0.4%	0.5%	0.6%	0.6%	0.6%	0.8%	0.7%	0.5%	0.5%	0.7%	0.8%
Google Systems												0.8%	0.7%
Webflow								0.1%	0.1%	0.2%	0.4%	0.6%	0.7%
Bitrix	0.3%	0.3%	0.3%	0.4%	0.6%	0.7%	0.7%	0.6%	0.9%	1.1%	0.9%	0.8%	0.7%
OpenCart				0.3%	0.4%	0.4%	0.4%	0.4%	0.5%	0.6%	0.6%	0.5%	0.5%

Ilustración 19 - Tabla de histórico de cuota de mercado, donde aparece WordPress.

El motivo de seleccionar estos lenguajes ha sido el amplio uso de estos en las organizaciones. En la Ilustración 21 vemos como ambos lenguajes siguen en los 12 primeros puestos (en los 10 primeros si no tenemos en cuenta lenguajes como HTML, SQL y bash/shell).

Para analizar el alcance de las herramientas es interesante hacer una evaluación de la cobertura de las reglas y los CWE que puede analizar respecto los CWE que contienen los Tests Suite de SARD. Para ello, se va a disponer un diagrama de Venn similar al de la Ilustración 20. En el caso de lenguajes como C/C++, hemos visto en la Tabla 2 como Semgrep no es una herramienta de SAST adecuada, además de contar SARD con un bajo volumen de vulnerabilidades en sus Tests Suites. En el diagrama de Venn el alcance de la herramienta Semgrep para estos lenguajes, es muy limitada. Ya que en el caso de máxima inteferencia del diagrama sería 7 coincidencias. Por lo tanto, C y C++ son excluidos del estudio de lenguajes a analizar.

[5] Este producto permite realizar firmas electrónicas de documentos con el DNI de España.

Ilustración 20 - Diagrama de Venn de cobertura de CWE de una herramienta respecto los CWE de SARD. Elaboración propia.

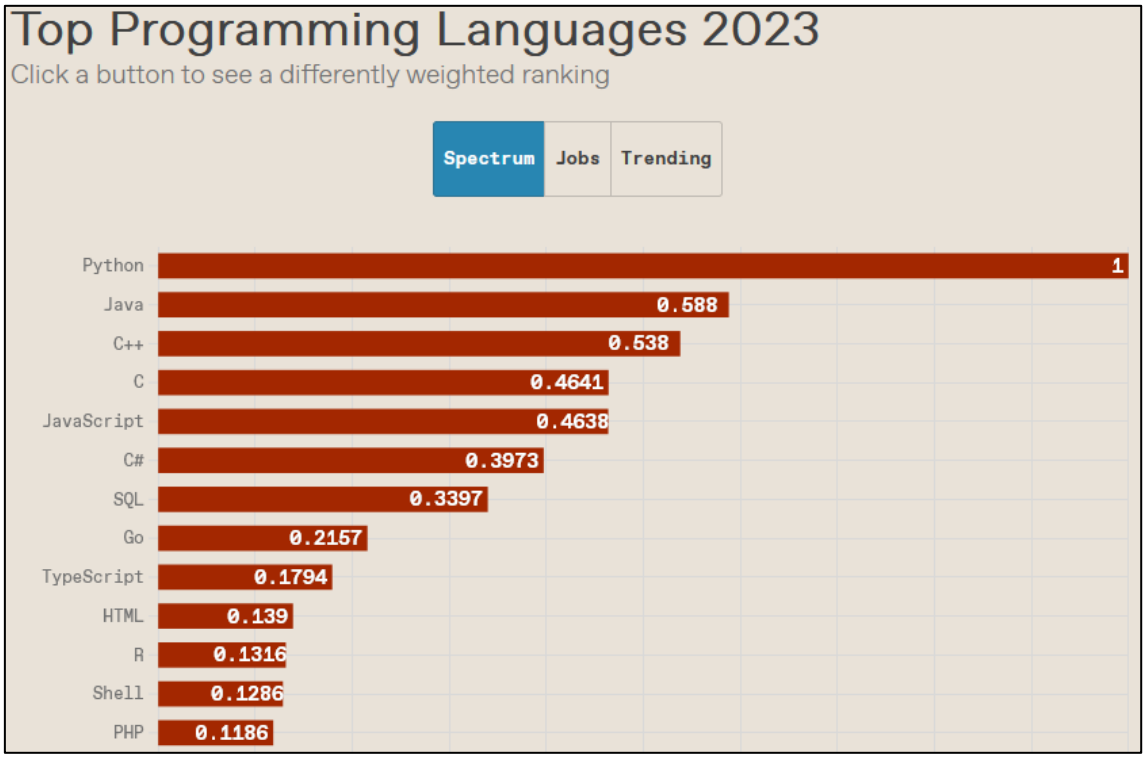

Ilustración 21 - Ranking de lenguajes programación en 2023. Fuente: https://spectrum.ieee.org/the-top-programming-languages-2023.

5.1 Caso Lenguaje Java

En la presente sección, se llevará a cabo un análisis exhaustivo de los tests suites SARD de SAMATE mediante la herramienta Semgrep, focalizándonos específicamente en los archivos correspondientes al lenguaje de programación Java (aunque que en los tests suite de Java también incluye ficheros “jsp” o similares, correspondientes a la capa vista de los proyectos). El listado completo de Test Suite de Java puede verse en la Ilustración 22. La composición de estos archivos se estructura en cinco categorías, cada una con su respectiva cantidad de archivos:

- Standalone Applications (Applications): Comprende un total de 8 archivos.
- Mobile Applications (Mobile): Integrado por 2 archivos.
- Stand Alone Suites (Stand_Alone): Constituido por 1 archivo.

- Web Applications (Web Aplic): Consta de 2 archivos.
- SARD Suites (Sard): Incluye 9 archivos.
- Archives: Compuesto por 8 archivos.

ID	Name	Author	Language	# Test cases	Created on	Type
21	Open Fire 3.6.0	SAMATE Team Staff	Java	12	18/09/2017	Applications
9	CoffeeMUD 5.8	Charles Oliveira	Java	478	27/10/2015	Applications
16	JTree	Charles Oliveira	Java	160	27/10/2015	Applications
11	Apache Jena 2.11.0	Charles Oliveira	Java	476	27/10/2015	Applications
10	Elasticsearch 1.0.0	Charles Oliveira	Java	478	27/10/2015	Applications
15	Apache POI 3.9	Charles Oliveira	Java	479	27/10/2015	Applications
12	Apache JMeter 2.8	Charles Oliveira	Java	160	27/10/2015	Applications
14	Apache Lucene 4.5.0	Charles Oliveira	Java	480	27/10/2015	Applications
23	VLC 2.1.3 deprecated	SAMATE Team Staff	Java	16	19/09/2017	Mobile
2	Card Board Sample 1.0	Charles Oliveira	Java	1	15/04/2015	Mobile
111	Juliet Java 1.3	NSA Center for Assured Software	Java	28881	01/10/2017	Stand_Alone
20	JSP Wiki 2.5.124-beta	SAMATE Team Staff	Java	3	18/09/2017	Web Aplic
13	Apache Lenya 2.0.4	Charles Oliveira	Java	477	27/10/2015	Web Aplic
96	Jetty 6.1.16	SAMATE Team Staff	Java	6	31/07/2014	Sard
97	Jspwiki 2.5.124	SAMATE Team Staff	Java	3	31/07/2014	Sard
95	Apache-tomcat 5.5.13	SAMATE Team Staff	Java	37	31/07/2014	Sard
98	Openfire 3.6.0	SAMATE Team Staff	Java	12	31/07/2014	Sard
64	Java Test Suite for Source Code Analyzer - false positive	Michael Koo	Java	27	03/02/2010	Sard
65	Java Test Suite for Source Code Analyzer - weakness suppresion	Michael Koo	Java	10	03/02/2010	Sard
63	Java Test Suite for Source Code Analyzer - weakness	Michael Koo	Java	27	03/02/2010	Sard
87	Juliet Java 1.2 with extra support deprecated	SAMATE Team Staff	Java	25477	14/05/2013	Archives
85	Juliet Java 1.2 deprecated	NSA Center for Assured Software	Java	25477	01/05/2013	Archives
32	IARPA STONESOUP Phase 1 - Injection for Java 1.0	IARPA	Java	36	01/11/2012	Archives
29	IARPA STONESOUP Phase 1 - Tainted Data for Java 1.0	IARPA	Java	35	01/11/2012	Archives
30	IARPA STONESOUP Phase 1 - Numeric Handling for Java 1.0	IARPA	Java	59	01/11/2012	Archives
28	Juliet Java 1.1.1	NSA Center for Assured Software	Java	23957	01/09/2012	Archives
69	Juliet Java 1.0 with extra support deprecated	SAMATE Team Staff	Java	14184	07/04/2011	Archives
24	Juliet Java 1.0	NSA Center for Assured Software	Java	14184	01/12/2010	Archives

Ilustración 22 - Lista de casos por Test Suites completo de Java. Elaboración propia a partir de la información de SARD SAMATE.

Es preciso destacar que se procederá a suprimir aquellos archivos cuyo estado se encuentre marcado como "deprecated," indicando su obsolescencia. En este contexto, se excluye el archivo denominado "Juliet Java 1.3" debido a diversos inconvenientes, principalmente su formato especial en relación con la etiqueta "state", que se presenta de manera distinta al resto de los ficheros del test.

Este archivo en particular presenta una complejidad adicional, ya que su campo "state" exhibe tres estados posibles: 'good', 'bad', y 'mixed', siendo esta última etiqueta la mayoritaria. Sin embargo, dicha información no se encuentra explícitamente en la misma ubicación que el resto de los ficheros, sino que está almacenada en una sección distinta y adopta la forma de una lista. Esta particularidad ha generado inconvenientes considerables, entre ellos la imposibilidad de realizar un análisis convencional mediante un parseador estándar. Como solución, se ha optado por obtener del archivo SARIF todos los los reportes “bad” y “good", para el caso de los “mixed” se ha efectuado una implementación que busca en los nombres de los archivos las cadenas de “_bad” y “_good”, ya que en la documentación se explica que en este tipo se pueden especificar

de esta manera. No obstante, el número de archivos “mixed” nombrados con esta etiqueta era lo suficientemente bajo, que se excluyó de las pruebas.

Adicionalmente, se debe descartar los archivos IARPA STONESOUP, Card Board, JSP Wiki y Jspwiki y todos los de la clase Juliet distintos del mencionado en el párrafo anterior, debido a que sus formatos no son compatibles con SARIF. Además, se identifica Apache Lenya como corrupto y no puede abrirse. Los Tests Suite eliminados pueden verse en la Ilustración 23.

ID	Name	Author	Language	# Test cases	Created on	Type
23	VLC 2.1.3 deprecated	SAMATE Team Staff	Java	16	19/09/2017	Mobile
87	Juliet Java 1.2 with extra support deprecated	SAMATE Team Staff	Java	25477	14/05/2013	Archives
85	Juliet Java 1.2 deprecated	NSA Center for Assured Software	Java	25477	01/05/2013	Archives
69	Juliet Java 1.0 with extra support deprecated	SAMATE Team Staff	Java	14184	07/04/2011	Archives
28	Juliet Java 1.1.1	NSA Center for Assured Software	Java	23957	01/09/2012	Archives
24	Juliet Java 1.0	NSA Center for Assured Software	Java	14184	01/12/2010	Archives
2	Card Board Sample 1.0	Charles Oliveira	Java	1	15/04/2015	Mobile
111	Juliet Java 1.3	NSA Center for Assured Software	Java	28881	01/10/2017	Stand_Alone
32	IARPA STONESOUP Phase 1 - Injection for Java 1.0	IARPA	Java	36	01/11/2012	Archives
29	IARPA STONESOUP Phase 1 - Tainted Data for Java 1.0	IARPA	Java	35	01/11/2012	Archives
30	IARPA STONESOUP Phase 1 - Numeric Handling for Java 1.0	IARPA	Java	59	01/11/2012	Archives
20	JSP Wiki 2.5.124-beta	SAMATE Team Staff	Java	3	18/09/2017	Web Aplic
97	Jspwiki 2.5.124	SAMATE Team Staff	Java	3	31/07/2014	Sard
13	Apache Lenya 2.0.4	Charles Oliveira	Java	477	27/10/2015	Web Aplic

Ilustración 23 - Distribución de casos por Test Suites para Java que no van a formar parte del estudio. Elaboración propia a partir de la información de SARD SAMATE.

Los archivos que eliminar de la muestra suponen un cambio muy significativo en el test-case de este lenguaje. Únicamente se conservan las clases Applications y SARD, el resto desaparecen de la muestra.

Se observa que solo se identificaron vulnerabilidades en los capítulos:

- A01:2021 (Broken Access Control).
- A03:2021 (Injection).
- A05:2021 (Security Misconfiguration).

Efectuados estas correcciones el listado final de archivos a examinar puede observarse en la Ilustración 24.

	ID	Name	Author	Language	# Test cases	Created on	Type
0	21	Open Fire 3.6.0	SAMATE Team Staff	Java	12	18/09/2017	Applications
1	9	CoffeeMUD 5.8	Charles Oliveira	Java	478	27/10/2015	Applications
2	16	JTree	Charles Oliveira	Java	160	27/10/2015	Applications
3	11	Apache Jena 2.11.0	Charles Oliveira	Java	476	27/10/2015	Applications
4	10	Elasticsearch 1.0.0	Charles Oliveira	Java	478	27/10/2015	Applications
5	15	Apache POI 3.9	Charles Oliveira	Java	479	27/10/2015	Applications
6	12	Apache JMeter 2.8	Charles Oliveira	Java	160	27/10/2015	Applications
7	14	Apache Lucene 4.5.0	Charles Oliveira	Java	480	27/10/2015	Applications
8	96	Jetty 6.1.16	SAMATE Team Staff	Java	6	31/07/2014	Sard
9	95	Apache-tomcat 5.5.13	SAMATE Team Staff	Java	37	31/07/2014	Sard
10	98	Openfire 3.6.0	SAMATE Team Staff	Java	12	31/07/2014	Sard
11	64	Java Test Suite for Source Code Analyzer - false positive	Michael Koo	Java	27	03/02/2010	Sard
12	65	Java Test Suite for Source Code Analyzer - weakness suppresion	Michael Koo	Java	10	03/02/2010	Sard
13	63	Java Test Suite for Source Code Analyzer - weakness	Michael Koo	Java	27	03/02/2010	Sard

Ilustración 24 - Listado de los tests suites que aplican en la PoC. Elaboración propia a partir de SARD SAMATE.

Una vez analizado los Test Suites que pueden utilizarse para la PoC, se han efectuado los procesos de extracción de información y se ha procedido a obtener la información del análisis siguiendo los pasos de la metodología fijada.

Los resultados obtenidos por paquete son los expuestos en la Ilustración 25:

	language	package	state	Res Tool	Res Sarif	TP	FN	FP	TN
0	java	2010-02-03-java-test-suite-for-source-code-analyzer-false-positive	good	7	18	0	0	7	11
1	java	2010-02-03-java-test-suite-for-source-code-analyzer-weakness	bad	6	7	6	1	0	0
2	java	2010-02-03-java-test-suite-for-source-code-analyzer-weakness-suppresion	bad	2	2	2	0	0	0
3	java	2014-07-31-apache-tomcat-v5-5-13	bad	4	2	2	2	0	0
4	java	2014-07-31-jetty-v6-1-16	bad	2	2	2	0	0	0
5	java	2014-07-31-openfire-v3-6-0	bad	4	2	2	2	0	0
6	java	2015-10-27-apache-jena-v2-11-0	bad	36	36	36	0	0	0
7	java	2015-10-27-apache-jmeter-v2-8	bad	12	12	12	0	0	0
8	java	2015-10-27-apache-lucene-v4-5-0	bad	39	39	39	0	0	0
9	java	2015-10-27-apache-poi-v3-9	bad	40	40	40	0	0	0
10	java	2015-10-27-coffeemud-v5-8	bad	41	41	41	0	0	0
11	java	2015-10-27-elasticsearch-v1-0-0	bad	39	39	39	0	0	0
12	java	2015-10-27-jtree	bad	12	12	12	0	0	0
13	java	2017-09-18-open-fire-v3-6-0	bad	4	2	2	2	0	0

Ilustración 25 - Listado de paquetes de tests suites que aplican en la PoC, con los resultados. Elaboración propia a partir de SARD SAMATE.

Resultados por OWASP (donde se concentran por CWE) se pueden consultar en la Ilustración 26:

	OWASP DESC	CWE_CODE	package	Res Sarif	Res Tool	TP	FN	FP	TN
0	A01:2021 – Broken Access Control	CWE-200	2014-07-31-jetty-v6-1-16	2	2	2	0	0	0
1	A03:2021 – Injection	CWE-78	2010-02-03-java-test-suite-for-source-code-analyzer-false-positive	5	3	0	0	3	2
2	A03:2021 – Injection	CWE-79	2010-02-03-java-test-suite-for-source-code-analyzer-false-positive	9	3	0	0	3	6
3	A03:2021 – Injection	CWE-79	2010-02-03-java-test-suite-for-source-code-analyzer-weakness	3	3	3	0	0	0
4	A03:2021 – Injection	CWE-79	2010-02-03-java-test-suite-for-source-code-analyzer-weakness-suppresion	1	1	1	0	0	0
5	A03:2021 – Injection	CWE-89	2010-02-03-java-test-suite-for-source-code-analyzer-false-positive	4	1	0	0	1	3
6	A03:2021 – Injection	CWE-89	2010-02-03-java-test-suite-for-source-code-analyzer-weakness	4	3	3	1	0	0
7	A03:2021 – Injection	CWE-89	2010-02-03-java-test-suite-for-source-code-analyzer-weakness-suppresion	1	1	1	0	0	0
8	A03:2021 – Injection	CWE-89	2014-07-31-openfire-v3-6-0	2	4	2	2	0	0
9	A03:2021 – Injection	CWE-89	2015-10-27-apache-jena-v2-11-0	36	36	36	0	0	0
10	A03:2021 – Injection	CWE-89	2015-10-27-apache-jmeter-v2-8	12	12	12	0	0	0
11	A03:2021 – Injection	CWE-89	2015-10-27-apache-lucene-v4-5-0	39	39	39	0	0	0
12	A03:2021 – Injection	CWE-89	2015-10-27-apache-poi-v3-9	40	40	40	0	0	0
13	A03:2021 – Injection	CWE-89	2015-10-27-coffeemud-v5-8	41	41	41	0	0	0
14	A03:2021 – Injection	CWE-89	2015-10-27-elasticsearch-v1-0-0	39	39	39	0	0	0
15	A03:2021 – Injection	CWE-89	2015-10-27-jtree	12	12	12	0	0	0
16	A03:2021 – Injection	CWE-89	2017-09-18-open-fire-v3-6-0	2	4	2	2	0	0
17	A05:2021 – Security Misconfiguration	CWE-614	2014-07-31-apache-tomcat-v5-5-13	2	4	2	2	0	0

Ilustración 26 - Distribución de los resultados en función de OWASP TOP 10. Elaboración propia a partir de SARD SAMATE.

Como puede verse en la lustración la herramienta identifica correctamente las vulnerabilidades existentes. Sin embargo, hay que añadir que los test cases contienen un número de casos muy bajo e incluye un numero de vulnerabilidades muy bajo, esto es, es muy poco representativo y exhaustivo del Top10 de OWASP, ya que no se abarcan todos los capítulos de OWASP.

- CWE-78 (Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')): Esta categoría se refiere a la falta de neutralización adecuada de elementos especiales utilizados en un comando del sistema operativo. La identificación de esta vulnerabilidad sugiere la posibilidad de que las aplicaciones puedan ser susceptibles a ataques de inyección de comandos, donde un atacante podría insertar comandos maliciosos en entradas de usuario para ejecutar operaciones no autorizadas en el sistema.

- CWE-79 (Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')): Aquí se aborda la falta de neutralización adecuada de la entrada durante la generación de páginas web, lo que puede dar lugar a ataques de tipo Cross-Site Scripting (XSS). Este tipo de vulnerabilidad permite a un atacante inyectar scripts maliciosos que se ejecutarán en el navegador de otros usuarios, comprometiendo la seguridad de la aplicación y la privacidad del usuario.

- CWE-89 (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')): La presencia de esta categoría indica la falta de neutralización adecuada de elementos especiales utilizados en comandos SQL. Un ataque de inyección SQL puede ocurrir cuando las entradas de usuario no se validan correctamente, permitiendo que un atacante manipule consultas SQL para acceder, modificar o eliminar datos en la base de datos.

- CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor): Esta categoría aborda la exposición de información sensible a actores no autorizados.

La detección de esta vulnerabilidad sugiere que la aplicación puede estar compartiendo inadvertidamente datos confidenciales, lo que podría ser explotado por un atacante para obtener información privilegiada.

- CWE-614 (Sensitive Cookie in HTTPS Session Without 'Secure' Attribute): Aquí se señala la presencia de cookies sensibles en sesiones HTTPS sin la presencia del atributo 'Secure'. Esto puede representar un riesgo de seguridad al permitir que las cookies se transmitan sin cifrado en conexiones no seguras, lo que facilita ataques como el secuestro de sesión.

En la imagen Ilustración 27 se puede ver el alcance de las reglas de Semgrep en cuanto a CWE respecto a los CWE que contienen los Test Suites utilizados. En este caso, se puede ver como la herramienta y los Test Suites tienen CWE mayoritariamente difentes. Los resultados indican que existen 63 tipos de CWE de SARD que no tienen una regla de Semgrep que pueda evaluarlos, 9 tipos de CWE que sí son detectables desde Semgrep y 44 que puede detectar Semgrep, pero no hay un test que pueda evaluarlo . Esto puede darse por una identificación de las vulnerabilidades poco precisa en el respositorio de Test Case o que existe un número bajo de reglas de Semgrep. Para determinar este punto, habría que analizar caso por caso, y queda fuera del alcance de la investigación.

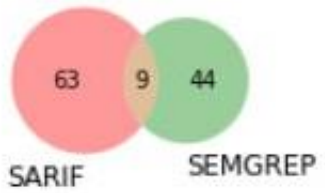

Ilustración 27 - Diagrama de Venn de los CWE del archivo SARIF de los SARD respecto los CWE de las reglas de Semgrep de Java. Elaboración propia.

5.2 Caso Lenguaje PHP

En la presente sección, se llevará a cabo un análisis exhaustivo de los tests suites SARD de SAMATE mediante la herramienta Semgrep, focalizándonos específicamente en los archivos correspondientes al lenguaje de programación PHP. El caso de PHP es más conciso que el visto en el caso Java únicamente hay 4 Test Suite (4 archivos), para dos categorías:

- Web Applications (Web Aplic), 1 archivo.
- SARD suites (Sard), 3 archivos.

El número de archivos es bajo, si bien hay que mencionar que todos se leen bien, la información esta correctamente consignada con lo que la extracción no presenta problemas. En la Ilustración 28 se puede ver la composición de los archivos.

ID	Name	Author	Language	# Test cases	Created on	Type
1	WordPress 2.0	Charles Oliveira	PHP	23	31/03/2015	Web Aplic
103	PHP Vulnerability Test Suite	Bertrand C. Stivalet	PHP	42212	27/10/2015	Sard
99	Wordpress 2.0	SAMATE Team Staff	PHP	22	31/07/2014	Sard
31	Web Applications in PHP	Romain Gaucher	PHP	15	23/10/2006	Sard

Ilustración 28 - Listado completo de tests suites de PHP. Elaboración propia a partir de SARD SAMATE.

A diferencia del caso de Java en el que cada paquete es good o bad, en este case tenemos el archivo PHP Vulnerability que contiene ambos casos.

	language	package	state	Res Tool	Res Sarif	TP	FN	FP	TN
0	php	2014-07-31-wordpress-v2-0	bad	6	1	1	5	0	0
1	php	2015-03-31-wordpress-v2-0	bad	6	1	1	5	0	0
2	php	2015-10-27-php-vulnerability-test-suite	bad	653	504	504	149	0	0
3	php	2022-05-12-php-test-suite-sqli-v1-0-0	bad	7584	5228	5228	2356	0	0
4	php	2022-05-12-php-test-suite-sqli-v1-0-0	good	17012	11820	0	0	11820	5192

Ilustración 29 - Listado de test suites de PHP por paquete y sus resultados. Elaboración propia a partir de SARD SAMATE.

Puede observarse en la Ilustración 29, todas las vulnerabilidades son detectadas, en el caso de los archivos Wordpress el ratio de falsos resultados es notablemente más alto que en el caso de los archivos PHP pese a la desigual distribución de casos entre ambas clases.

	OWASP DESC	CWE_CODE	package	Res Sarif	Res Tool	TP	FN	FP	TN
0	A03:2021 – Injection	CWE-78	2015-10-27-php-vulnerability-test-suite	62	124	62	62	0	0
1	A03:2021 – Injection	CWE-79	2014-07-31-wordpress-v2-0	1	6	1	5	0	0
2	A03:2021 – Injection	CWE-79	2015-03-31-wordpress-v2-0	1	6	1	5	0	0
3	A03:2021 – Injection	CWE-79	2015-10-27-php-vulnerability-test-suite	342	342	342	0	0	0
4	A03:2021 – Injection	CWE-89	2015-10-27-php-vulnerability-test-suite	100	187	100	87	0	0
5	A03:2021 – Injection	CWE-89	2022-05-12-php-test-suite-sqli-v1-0-0	17048	24596	5228	2356	11820	5192

Ilustración 30 - Distribución de los resultados en función de OWASP TOP 10. Elaboración propia a partir de SARD SAMATE.

Los resultados para este lenguaje son muy específicos. Cómo se puede apreciar en la Ilustración 30, únicamente se detectan vulnerabilidades para el capitulo A03 de OWASP Top Ten, centrado en las vulnerabilidades de inyección, revela una serie de debilidades críticas que pueden permitir a un atacante insertar y ejecutar código malicioso en la aplicación. A continuación, se presenta un análisis exhaustivo de las vulnerabilidades específicas identificadas en este capítulo para el lenguaje PHP, considerando cada código CWE asociado.

- CWE-78 - Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). La vulnerabilidad detectada en la suite de pruebas de vulnerabilidad de PHP (2015-10-27) revela que existe una inadecuada neutralización de elementos especiales utilizados en un comando del sistema operativo (OS Command Injection).
- CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). Se identificaron tres instancias de CWE-79 en diferentes contextos. Dos de ellas están asociadas a la suite de pruebas de vulnerabilidad de PHP (2014-07-31 y 2015-03-31) y la tercera a la suite de pruebas de SQL Injection (2015-10-27).
- CWE-89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). Dos instancias de CWE-89 fueron identificadas en la suite de pruebas de vulnerabilidad de PHP (2015-10-27 y 2022-05-12).

En la imagen Ilustración 31 se puede ver el alcance de las reglas de Semgrep en cuanto a CWE respecto a los CWE que contienen los Test Suites utilizados. En este caso, es similar al detectado en Java. Los resultados indican que existen 18 tipos de CWE de SARD, de los cuales 9 que no tienen una regla de Semgrep que pueda evaluarlos, 9 tipos de CWE que sí son detectables desde Semgrep y un total de 28 tipos que puede detectar Semgrep, pero de los cuales 19 no hay un test que pueda evaluarlo. Para determinar este punto, habría que analizar caso por caso, y queda fuera del alcance de la investigación.

Ilustración 31 - Diagrama de Venn de los CWE del archivo SARIF de los SARD respecto los CWE de las reglas de Semgrep para PHP. Elaboración propia.

6 Conclusiones y trabajos futuros

Esta investigación ha partido de una idea inicial y se ha conformado según se ha ido encontrando información interesante durante la fase de estado del arte. En ese punto, se determinó que existía una base de set de pruebas (SARD) facilitada por una organización (NIST) relevante en el mundo de la ciberseguridad.

6.1 Conclusiones

Sobre los sets de pruebas, se ha podido averiguar que los tests suites de SARD de SAMATE abarcan varios lenguajes de programación y diferentes tipos de aplicaciones. Además, existen tests suites creados desde diferentes perspectivas:

- Proyectos open source: de este tipo de paquete se disponen de versiones de software libre en los que se detectaron vulnerabilidades y que sirven ahora para que las herramientas de revisión de código busquen estos problemas conocidos.
- Proyectos Juliet: este tipo de proyectos están orientados a contener un número de vulnerabilidades muy superiores a los que se detectaría en proyectos reales, ya que son proyectos abiertamente vulnerables pensados para ser una batería de pruebas completa. Estos se componen de un número de archivos y de vulnerabilidades, en principio, más variado que los paquetes open source.

Este conjunto de pruebas también adjunta la información necesaria para su compilación e incluso correr el proyecto. Sin embargo, durante la fase de análisis se trató de automatizar el proceso de compilación de los proyectos y se ha encontrado problemas que dificultan la compilación, como la necesidad de versiones específicas del compilador de Java y la herramienta de Ant.

En este caso, se ha analizado el conjunto de Tests Suites donde se han detectados que los tests suites de tipo open source suelen contener pocos casos de pruebas, aunque están clasificados correctamente como vulnerables o falsos positivos. Sin embargo, los del tipo Juliet son mucho las voluminosos en casos, pero la clasificación general de los casos es demasiado ambigua (ya que no se especifica si se trata de TP, FP, TN o FN… Se define como “mixto”), siendo el número de casos útiles para un benchmark demasiado bajo.

Por otro lado, se ha efectuado una implementación en un formato de Prueba de Concepto de la metodología, donde se ha ejecutado una evaluación de una herramienta de SAST. Con el sistema propuesto se pueden efectuar benchmark de una manera más sencilla una comparación de herramientas utilizando el formato estandarizado de SARIF y con una batería de pruebas del NIST se pueden hacer comparaciones para 5 lenguajes de programación. En el caso de que la herramienta de SAST requiera que el código este compilado, se requiere crear al menos un paso adicional de compilación que puede requerir un esfuerzo de personalización de la construcción (aunque esta de partida muy simplificado).

En cuanto a los resultados obtenidos para la herramienta de Semgrep, para los lenguajes de programación de Java y PHP, se ha podido comprobar que el conjunto de reglas de Semgrep que analizan un subconjunto de las vulnerabilidades de los sets de pruebas, tienen tasas muy buenas en cuanto a TP y FP (puede verse en detalle en el capítulo “Análisis de los resultados”). Esta herramienta demuestra que es de gran utilidad para un análisis de tipo SAST.

Por otro lado, la inclusión de Spotbugs no ha sido viable en el plazo fijado de la investigación implementar esta herramienta debido a que sólo analiza código compilado. Debido a la compilación de los Tests Suites es viable pero no es genérica, hay que automatizar una subtarea adicional a la tarea de compilar. Por tanto, no se ha podido llevar a cabo y se propone para futuros trabajos.

Como conclusión final, esta investigación ha demostrado que es factible realizar la automatización propuesta, aunque han aparecido puntos a tener en cuenta para llevarlo a cabo. Por tanto, se ha conseguido los dos principales objetivos de esta investigación, donde se define una metodología de comparación de herramientas mediante formatos estandarizados (en este caso se ha descrito los factores más objetivos) y la creación de una PoC para poner a prueba la viabilidad de la metodología. En el caso de los objetivos secundarios, en el caso del uso de esta metodología implementado en un ciclo de S-SDLC puede ser mediante la implantación de una propuesta (como la que se hará en el apartado de “trabajos futuros” mediante Jenkins (u otras herramientas y ciclos de DevOps). Otro objetivo secundario, es la obtención de una comparativa automatizada, lo cual se demuestra que es viable, ya que desde los datos obtenidos y guardados en la base de datos se podría presentar fácilmente los datos obtenidos.

6.2 Futuros trabajos

Tras la investigación se han identificados posibles continuaciones de esta línea de investigación es mejorar la propia automatización de pruebas a partir de la experiencia de la presente investigación. Para llevar a cabo una automatización más sencilla, modular y eficiente que “esquive” algunos de los problemas detectados, es mediante el uso de soluciones existentes orientadas a realizar tareas programadas.

El diagrama del benchmark para cubrir soluciones en las que se requiera compilar el código fuente para efectuar el análisis:

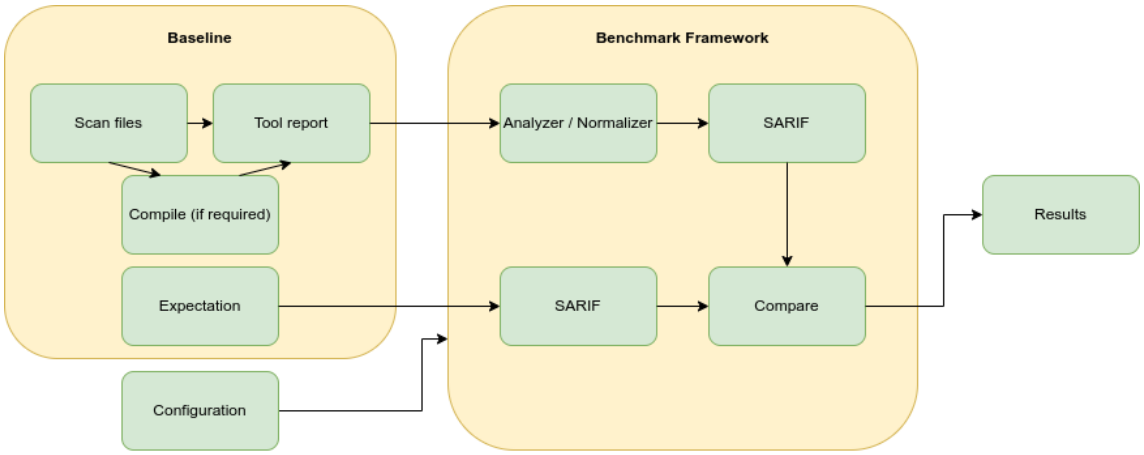

Ilustración 32 - Nuevo planteamiento del sistema de benchmark. Imagen de elaboración propia.

El diseño de como implementar esta investigación a otro nivel sería:

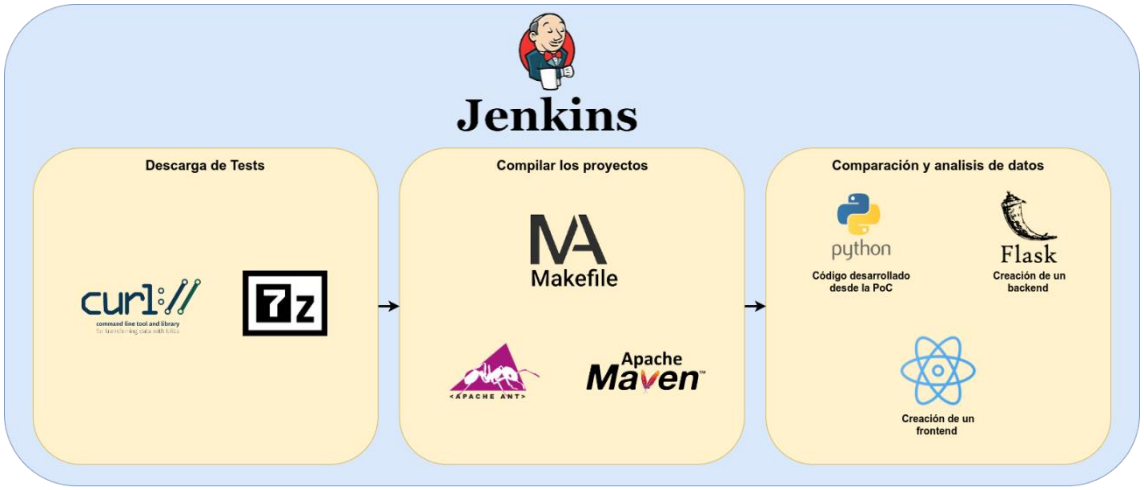

Ilustración 33 - Propuesta para desarrollar una nueva versión del benchmark. Imagen de elaboración propia.

Cómo orquestador de gestión de tareas, se podría utilizar Jenkins, conocida solución muy utilizada en los entornos de integración continua de los ciclos de DevOps. Precisamente por su capacidad de efectuar tareas programadas, se puede fácilmente programar:

- Descarga de cada Test Suite
- Descompresión de cada uno de los tests.
- Verificación de la existencia de archivos SARIF.
- Compilación de cada proyecto mediante los archivos facilitados en el SARD del NIST. Este apartado requiere especial revisión, ya que es probable que haya que efectuar algún tipo de adaptación para que la compilación se efectúe con las versiones de los gestores de paquetes específicos de cada test. Este paso puede requerir una automatización personalizada.
- Comparación siguiendo los pasos explicados en esta investigación, de donde se pueden extraer las experiencias previas.

- Finalmente, se pude construir un desarrollo para presentar los datos de manera amigable en un formato Web. Esta parte, sería completamente opcional y personalizable.

Con este planteamiento se minimizan los problemas encontrados y se simplifica el código para efectuar las pruebas. Esta propuesta permite independizar las tareas y aplicar un “divide y vencerás”. No obstante, este diseño requiere el conocimiento de las herramientas descritas para efectuar la automatización. En el caso de Jenkins, puede requerir el uso de Groovy para automatizar pipelines.

Otras posibles vías de investigación se pueden destacar:

- Implementar nuevos o ampliar los tests suites de SARD publicando diferentes propuestas que mejoren el set de pruebas. Se podría explorar la vía de generar código vulnerable y código no vulnerable (pero que sirva para la detección de falsos positivos) de manera automatizada mediante inteligencia artificial.
- Identificar unívocamente los reportes en el SARIF de los paquetes existentes cuya clasificación de las vulnerabilidades es “mixed”.
- Revisar las herramientas existentes open source que no tienen soporte para SARIF e implementar y realizar un “push request” con la implementación para que la comunidad la incorpore.
- Analizar los reportes de todas las herramientas de SAST para verificar su cumplimiento del formato de SARIF, es decir, que su reporte ubica la información en los nodos adecuados y que se emite toda la información necesaria.
- Crear una variante de esta investigación para otro tipo de pruebas como el DAST y SCA, ya que en el formato de SARIF ha evolucionado hacía una estructura que permite una cierta flexibilidad y existe la posibilidad de usarlo para estas pruebas. Por otro lado, hay herramientas de este tipo de pruebas que ya reportan en este formato cómo, por ejemplo, OWASP ZAP para el DAST y trivy para el SCA.
- En caso de evolucionar en la dirección correcta la automatización de los benchmarks, se podría realizar pruebas de detección mediante analizadores de código fuente que implementen Inteligencia Artificial.

7 Anexos

7.1 Análisis de los Tests Suite de SARD para C

Durante la fase de análisis de los Tests Suite se verificó que el mayor conjunto de tests era el de C/C++. En este caso, se analiza los datos volcados a la base de datos y se detecta que existen diferentes maneras de clasificar los reportes dentro de los archivos de SARIF de los proyectos.

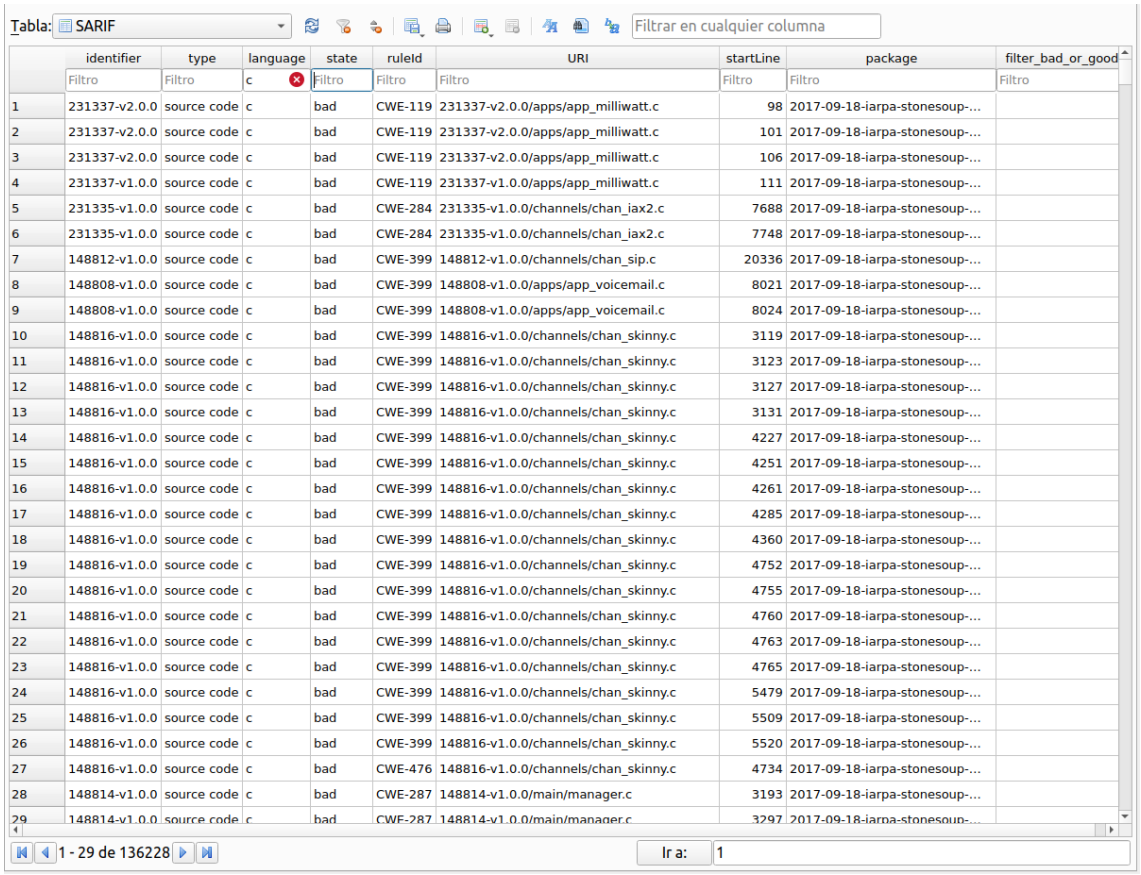

Tabla: SARIF — Filtrar en cualquier columna

	identifier	type	language	state	ruleId	URI	startLine	package	filter_bad_or_good
	Filtro	Filtro	c	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	231337-v2.0.0	source code	c	bad	CWE-119	231337-v2.0.0/apps/app_milliwatt.c	98	2017-09-18-iarpa-stonesoup-...	
2	231337-v2.0.0	source code	c	bad	CWE-119	231337-v2.0.0/apps/app_milliwatt.c	101	2017-09-18-iarpa-stonesoup-...	
3	231337-v2.0.0	source code	c	bad	CWE-119	231337-v2.0.0/apps/app_milliwatt.c	106	2017-09-18-iarpa-stonesoup-...	
4	231337-v1.0.0	source code	c	bad	CWE-119	231337-v1.0.0/apps/app_milliwatt.c	111	2017-09-18-iarpa-stonesoup-...	
5	231335-v1.0.0	source code	c	bad	CWE-284	231335-v1.0.0/channels/chan_iax2.c	7688	2017-09-18-iarpa-stonesoup-...	
6	231335-v1.0.0	source code	c	bad	CWE-284	231335-v1.0.0/channels/chan_iax2.c	7748	2017-09-18-iarpa-stonesoup-...	
7	148812-v1.0.0	source code	c	bad	CWE-399	148812-v1.0.0/channels/chan_sip.c	20336	2017-09-18-iarpa-stonesoup-...	
8	148808-v1.0.0	source code	c	bad	CWE-399	148808-v1.0.0/apps/app_voicemail.c	8021	2017-09-18-iarpa-stonesoup-...	
9	148808-v1.0.0	source code	c	bad	CWE-399	148808-v1.0.0/apps/app_voicemail.c	8024	2017-09-18-iarpa-stonesoup-...	
10	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	3119	2017-09-18-iarpa-stonesoup-...	
11	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	3123	2017-09-18-iarpa-stonesoup-...	
12	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	3127	2017-09-18-iarpa-stonesoup-...	
13	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	3131	2017-09-18-iarpa-stonesoup-...	
14	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4227	2017-09-18-iarpa-stonesoup-...	
15	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4251	2017-09-18-iarpa-stonesoup-...	
16	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4261	2017-09-18-iarpa-stonesoup-...	
17	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4285	2017-09-18-iarpa-stonesoup-...	
18	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4360	2017-09-18-iarpa-stonesoup-...	
19	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4752	2017-09-18-iarpa-stonesoup-...	
20	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4755	2017-09-18-iarpa-stonesoup-...	
21	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4760	2017-09-18-iarpa-stonesoup-...	
22	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4763	2017-09-18-iarpa-stonesoup-...	
23	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	4765	2017-09-18-iarpa-stonesoup-...	
24	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	5479	2017-09-18-iarpa-stonesoup-...	
25	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	5509	2017-09-18-iarpa-stonesoup-...	
26	148816-v1.0.0	source code	c	bad	CWE-399	148816-v1.0.0/channels/chan_skinny.c	5520	2017-09-18-iarpa-stonesoup-...	
27	148816-v1.0.0	source code	c	bad	CWE-476	148816-v1.0.0/channels/chan_skinny.c	4734	2017-09-18-iarpa-stonesoup-...	
28	148814-v1.0.0	source code	c	bad	CWE-287	148814-v1.0.0/main/manager.c	3193	2017-09-18-iarpa-stonesoup-...	
29	148814-v1.0.0	source code	c	bad	CWE-287	148814-v1.0.0/main/manager.c	3297	2017-09-18-iarpa-stonesoup-...	

1 - 29 de 136228 — Ir a: 1

Ilustración 34 - Vista de los reportes de la tabla "SARIF" donde se vuelcan los datos de los proyectos del SARD del NIST. En este caso, filtrando por C/C++. Elaboración propia.

En la Ilustración 34 observamos el contenido de la tabla filtrando por la tecnología de C/C++. En la Ilustración 35 filtramos por los tipos de vulnerabilidad “mixed”

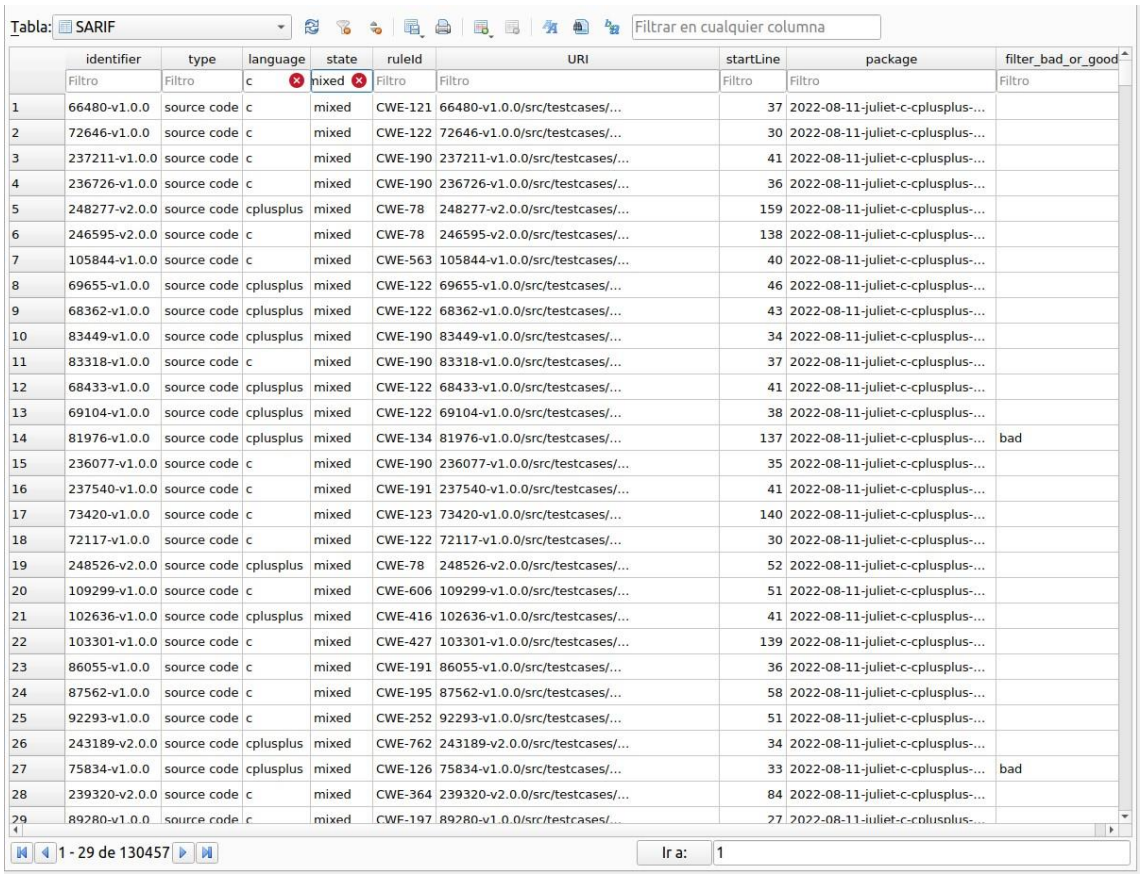

Tabla: SARIF — Filtrar en cualquier columna

	identifier	type	language	state	ruleId	URI	startLine	package	filter_bad_or_good
	Filtro	Filtro	c	mixed	Filtro	Filtro	Filtro	Filtro	Filtro
1	66480-v1.0.0	source code	c	mixed	CWE-121	66480-v1.0.0/src/testcases/...	37	2022-08-11-juliet-c-cplusplus-...	
2	72646-v1.0.0	source code	c	mixed	CWE-122	72646-v1.0.0/src/testcases/...	30	2022-08-11-juliet-c-cplusplus-...	
3	237211-v1.0.0	source code	c	mixed	CWE-190	237211-v1.0.0/src/testcases/...	41	2022-08-11-juliet-c-cplusplus-...	
4	236726-v1.0.0	source code	c	mixed	CWE-190	236726-v1.0.0/src/testcases/...	36	2022-08-11-juliet-c-cplusplus-...	
5	248277-v2.0.0	source code	cplusplus	mixed	CWE-78	248277-v2.0.0/src/testcases/...	159	2022-08-11-juliet-c-cplusplus-...	
6	246595-v2.0.0	source code	c	mixed	CWE-78	246595-v2.0.0/src/testcases/...	138	2022-08-11-juliet-c-cplusplus-...	
7	105844-v1.0.0	source code	c	mixed	CWE-563	105844-v1.0.0/src/testcases/...	40	2022-08-11-juliet-c-cplusplus-...	
8	69655-v1.0.0	source code	cplusplus	mixed	CWE-122	69655-v1.0.0/src/testcases/...	46	2022-08-11-juliet-c-cplusplus-...	
9	68362-v1.0.0	source code	cplusplus	mixed	CWE-122	68362-v1.0.0/src/testcases/...	43	2022-08-11-juliet-c-cplusplus-...	
10	83449-v1.0.0	source code	cplusplus	mixed	CWE-190	83449-v1.0.0/src/testcases/...	34	2022-08-11-juliet-c-cplusplus-...	
11	83318-v1.0.0	source code	c	mixed	CWE-190	83318-v1.0.0/src/testcases/...	37	2022-08-11-juliet-c-cplusplus-...	
12	68433-v1.0.0	source code	cplusplus	mixed	CWE-122	68433-v1.0.0/src/testcases/...	41	2022-08-11-juliet-c-cplusplus-...	
13	69104-v1.0.0	source code	cplusplus	mixed	CWE-122	69104-v1.0.0/src/testcases/...	38	2022-08-11-juliet-c-cplusplus-...	
14	81976-v1.0.0	source code	cplusplus	mixed	CWE-134	81976-v1.0.0/src/testcases/...	137	2022-08-11-juliet-c-cplusplus-...	bad
15	236077-v1.0.0	source code	c	mixed	CWE-190	236077-v1.0.0/src/testcases/...	35	2022-08-11-juliet-c-cplusplus-...	
16	237540-v1.0.0	source code	c	mixed	CWE-191	237540-v1.0.0/src/testcases/...	41	2022-08-11-juliet-c-cplusplus-...	
17	73420-v1.0.0	source code	c	mixed	CWE-123	73420-v1.0.0/src/testcases/...	140	2022-08-11-juliet-c-cplusplus-...	
18	72117-v1.0.0	source code	c	mixed	CWE-122	72117-v1.0.0/src/testcases/...	30	2022-08-11-juliet-c-cplusplus-...	
19	248526-v2.0.0	source code	cplusplus	mixed	CWE-78	248526-v2.0.0/src/testcases/...	52	2022-08-11-juliet-c-cplusplus-...	
20	109299-v1.0.0	source code	c	mixed	CWE-606	109299-v1.0.0/src/testcases/...	51	2022-08-11-juliet-c-cplusplus-...	
21	102636-v1.0.0	source code	cplusplus	mixed	CWE-416	102636-v1.0.0/src/testcases/...	41	2022-08-11-juliet-c-cplusplus-...	
22	103301-v1.0.0	source code	c	mixed	CWE-427	103301-v1.0.0/src/testcases/...	139	2022-08-11-juliet-c-cplusplus-...	
23	86055-v1.0.0	source code	c	mixed	CWE-191	86055-v1.0.0/src/testcases/...	36	2022-08-11-juliet-c-cplusplus-...	
24	87562-v1.0.0	source code	c	mixed	CWE-195	87562-v1.0.0/src/testcases/...	58	2022-08-11-juliet-c-cplusplus-...	
25	92293-v1.0.0	source code	c	mixed	CWE-252	92293-v1.0.0/src/testcases/...	51	2022-08-11-juliet-c-cplusplus-...	
26	243189-v2.0.0	source code	cplusplus	mixed	CWE-762	243189-v2.0.0/src/testcases/...	34	2022-08-11-juliet-c-cplusplus-...	
27	75834-v1.0.0	source code	cplusplus	mixed	CWE-126	75834-v1.0.0/src/testcases/...	33	2022-08-11-juliet-c-cplusplus-...	bad
28	239320-v2.0.0	source code	c	mixed	CWE-364	239320-v2.0.0/src/testcases/...	84	2022-08-11-juliet-c-cplusplus-...	
29	89280-v1.0.0	source code	c	mixed	CWE-197	89280-v1.0.0/src/testcases/...	27	2022-08-11-juliet-c-cplusplus-...	

1 - 29 de 130457 — Ir a: 1

Ilustración 35 - Vista de los reportes de la tabla "SARIF" donde se vuelcan los datos de los proyectos del SARD del NIST. En este caso, filtrando por C/C++ y por los reportes de tipo "mixed". Elaboración propia.

Tal como se observa en las dos imágenes anteriores (y se ha explicado en el apartado "Análisis de los Tests Suite de SARD"), el subconjunto de vulnerabilidades clasificadas como "mixed" es demasiado amplio, disminuyendo la potencia de los Tests Suite de SARD, ya que la volumetría de datos útiles para efectuar las pruebas se ha mermado en exceso.

7.2 Análisis de resultados de C con Semgrep

En este apartado, se lleva a cabo un análisis exhaustivo de los Tests Suites de SARD de SAMATE mediante la herramienta Semgrep, focalizándonos específicamente en los archivos correspondientes al lenguaje de programación C. El caso de C es más extenso en test como se pudo observar en el "Análisis de los Tests Suite de SARD".

Para este caso, como se ha tratado en el apartado "Herramientas evaluadas", sólo existe un total de 7 reglas para efectuar el SAST con esta herramienta (para esta tecnología Semgrep considera experimental su análisis). En la Ilustración 36, se puede apreciar en el diagrama de Venn la cobertura de vulnerabilidades (CWE) de los tests y de la herramienta.

Los resultados para este lenguaje son muy específicos. A continuación, se presenta un análisis exhaustivo de las vulnerabilidades identificadas en este capítulo para el lenguaje

C, considerando cada código CWE asociado (al tratarse de una tecnología que no se utiliza demasiado en la construcción de Webs, no se agrupará mediante OWASP Top 10).

- CWE-134 – (Use of Externally-Controlled Format String). Ocurre cuando una aplicación utiliza una cadena de formato que puede ser controlada por un atacante externo, lo que puede llevar a vulnerabilidades de seguridad como la ejecución de código no deseado.
- CWE-415 - (Double Free). Este CWE se refiere a situaciones en las que se libera la misma memoria dos veces, lo que puede llevar a errores de acceso a la memoria y posiblemente a ataques de seguridad.
- CWE-416 – (Use After Free"). Es la vulnerabilidad asociada con este CWE. Ocurre cuando se hace referencia a una ubicación de memoria que ya ha sido liberada, lo que puede conducir a problemas de seguridad y errores de ejecución.
- CWE-676 - (Use of Potentially Dangerous Function). Ocurre cuando se utilizan funciones que pueden ser peligrosas si no se manejan adecuadamente, como funciones de copia de memoria sin límites de tamaño.
- CWE-774 - "Allocation of Resources Without Limits or Throttling). Sucede cuando una aplicación permite la asignación de recursos sin límites adecuados, lo que puede agotar los recursos del sistema y llevar a denegaciones de servicio u otros problemas.

En la Ilustración 36 se puede ver el alcance de las reglas de Semgrep en cuanto a CWE respecto a los CWE que contienen los test suites utilizados. En este caso, es mucho mayor (180 CWE) al detectado en Java (63 CWE) y PHP (18 CWE). Los resultados indican que existen un total de 180 tipos de CWE de SARD, de los cuales 175 que no tienen una regla de Semgrep que pueda evaluarlos, 5 tipos de CWE que sí son detectables desde Semgrep y un total de 7 tipos de CWE que detecta Semgrep de los cuales 4 no hay un test que pueda evaluarlo Para determinar este punto, habría que analizar caso por caso, y queda fuera del alcance de la investigación.

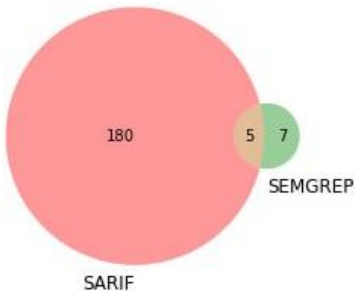

Ilustración 36 - Diagrama de Venn de cobertura de CWE de una herramienta respecto los CWE de SARD. Elaboración propia.

7.3 Detalles de interés sobre las reglas de Semgrep

Tal como se ha explicado en el apartado “Herramientas evaluadas”, en concreto en el subapartado “Semgrep”, una herramienta puede implementar un número de reglas

determinado que sólo comprueban un mismo CWE. Debido a esto, en la Tabla 2 se expone el número de reglas que existen en Semgrep por lenguaje de programación, pero este número no es directamente proporcional al número de CWE que cubre Semgrep. En la Ilustración 37, se comprueba que existen 23 reglas de Semgrep para Java que solamente cubren el CWE-89.

Tabla: SEMGREP_FULL_RULES

	RULE_ID	CWE	CWE_CODE
	java.	CWE-89	Filtro
1	java.lang.security.audit.formatted-sql-strin...	CWE-89: Improper Neutralization of Specia...	CWE-89
2	java.lang.security.audit.sqli.jdo-sqli.jdo-sqli	CWE-89: Improper Neutralization of Specia...	CWE-89
3	java.spring.security.jpa-sqli.jpa-sqli	CWE-89: Improper Neutralization of Specia...	CWE-89
4	java.servlets.security.nosql-injection-...	CWE-89: Improper Neutralization of Specia...	CWE-89
5	java.lang.security.audit.sqli.jdbc-sqli.jdbc-...	CWE-89: Improper Neutralization of Specia...	CWE-89
6	java.aws-lambda.security.tainted-...	CWE-89: Improper Neutralization of Specia...	CWE-89
7	java.servlets.security.tainted-sql-from-http...	CWE-89: Improper Neutralization of Specia...	CWE-89
8	java.lang.security.audit.sqli.turbine-...	CWE-89: Improper Neutralization of Specia...	CWE-89
9	java.lang.security.audit.sqli.vertx-sqli.vertx...	CWE-89: Improper Neutralization of Specia...	CWE-89
10	java.aws-lambda.security.tainted-sql-...	CWE-89: Improper Neutralization of Specia...	CWE-89
11	java.lang.security.audit.sqli.tainted-sql-fro...	CWE-89: Improper Neutralization of Specia...	CWE-89
12	java.servlets.security.tainted-sql-from-http...	CWE-89: Improper Neutralization of Specia...	CWE-89
13	java.lang.security.audit.jdbc-sql-formatted-...	CWE-89: Improper Neutralization of Specia...	CWE-89
14	java.spring.security.jdo-sqli.jdo-sqli	CWE-89: Improper Neutralization of Specia...	CWE-89
15	java.spring.security.injection.tainted-sql-...	CWE-89: Improper Neutralization of Specia...	CWE-89
16	java.lang.security.audit.sqli.jpa-sqli.jpa-sqli	CWE-89: Improper Neutralization of Specia...	CWE-89
17	java.spring.security.jdbctemplate-...	CWE-89: Improper Neutralization of Specia...	CWE-89
18	java.spring.security.hibernate-sqli.hibernat...	CWE-89: Improper Neutralization of Specia...	CWE-89
19	java.spring.security.audit.spring-sqli.spring...	CWE-89: Improper Neutralization of Specia...	CWE-89
20	java.lang.security.audit.sqli.hibernate-...	CWE-89: Improper Neutralization of Specia...	CWE-89
21	java.spring.security.spring-sqli-...	CWE-89: Improper Neutralization of Specia...	CWE-89
22	java.jboss.security.session_sqli.find-sql-...	CWE-89: Improper Neutralization of Specia...	CWE-89
23	java.lang.security.audit.formatted-sql-...	CWE-89: Improper Neutralization of Specia...	CWE-89

Ilustración 37 - Reglas de Semgrep para Java para cubrir el CWE-89 guardadas en la BBDD. Elaboriación propia.

Este análisis es relevante, ya que determina el alcance real de las reglas de Semgrep y ayuda a comprender el nivel de cobertura que aportan las herramientas de SAST realmente.

8 Glosario

Definición de los términos y acrónimos más relevantes utilizados en la Memoria.

- BBDD (Base de Datos): es un conjunto organizado de datos relacionados entre sí, que se almacenan y gestionan de manera estructurada. Las bases de datos se utilizan para almacenar información de manera eficiente, permitiendo el acceso, la búsqueda y la recuperación de datos de manera rápida y segura.
- Benchmark: es un conjunto de pruebas que evalúa y compara la efectividad y rendimiento de herramientas de seguridad en aplicaciones para detectar vulnerabilidades en el software, ayudando a elegir la más adecuada para cada necesidad específica.
- DAST (Dynamic Application Security Testing): Son las pruebas dinámicas de seguridad en aplicaciones. Es un proceso que identifica vulnerabilidades en aplicaciones de software en tiempo de ejecución. A diferencia del SAST, el DAST evalúa la aplicación en un entorno en funcionamiento, generalmente en etapas de prueba o preproducción (desaconsejándose en producción), buscando vulnerabilidades que pueden ser explotadas durante su operación normal.
- IAST (Interactive Application Security Testing): Prueba interactiva de seguridad en aplicaciones. Es una solución que combina elementos de SAST y DAST para identificar y mitigar vulnerabilidades de seguridad. IAST monitorea las aplicaciones en tiempo de ejecución para identificar problemas de seguridad, proporcionando una visión detallada del comportamiento de la aplicación y su interacción con los datos y el entorno.
- Java: Lenguaje de programación popular que es seguro, robusto y ampliamente utilizado en el desarrollo de aplicaciones empresariales y móviles.
- NIST (Instituto Nacional de Estándares y Tecnología): Agencia gubernamental de EE. UU. que desarrolla el Marco de Ciberseguridad, una guía ampliamente usada en seguridad cibernética.
- OWASP (Open Web Application Security Project): Organización sin fines de lucro que promueve la seguridad en aplicaciones web y mantiene la lista "OWASP Top Ten" de amenazas comunes.
- OWASP ZAP (OWASP Zed Attack Proxy): es una herramienta de código abierto ampliamente utilizada para probar la seguridad de aplicaciones web. Se utiliza principalmente para realizar pruebas de penetración y descubrir vulnerabilidades en aplicaciones web.
- PHP: Lenguaje de scripting ampliamente utilizado para el desarrollo web, especialmente en la creación de sitios web dinámicos y aplicaciones web.
- PoC (Proof of Concept): este concepto significa "Prueba de Concepto", es un prototipo que valida aspectos de un proyecto o solución. Sirve para establecer si un desarrollo es viable y escalable de cara a crear un producto más consolidado.
- SAMATE (Software Assurance Metrics And Tool Evaluation) es un proyecto del (NIST) que tiene como objetivo proporcionar métricas y evaluaciones para las herramientas de análisis de seguridad del software (SAST y DAST). SAMATE busca establecer una base sólida para comparar estas herramientas, brindando

información necesaria para tomar decisiones informadas y mejorar la seguridad del software.

- SARD (Software Assurance Reference Dataset): es un conjunto de fragmentos de código con ejemplos de vulnerabilidades de seguridad y sus versiones corregidas, provisto por el NIST dentro de su proyecto SAMATE. Sirve para evaluar y mejorar herramientas y técnicas de aseguramiento de software, proporcionando recursos para analizar y comprender las vulnerabilidades en el código, y cómo pueden ser detectadas y mitigadas.
- SARIF (Static Analysis Results Interchange Format): Formato para representar resultados de análisis estático de código fuente y seguridad del software, facilitando la compartición y la corrección de problemas.
- SATE (Static Analysis Tool Exposition): es una iniciativa del NIST que busca mejorar las herramientas de análisis estático, evaluándolas en un conjunto común de programas para exponer sus fortalezas y debilidades, proporcionar retroalimentación a los desarrolladores y ayudar a los usuarios a elegir la herramienta adecuada.
- SAST (Static Application Security Testing): Son las pruebas estáticas de seguridad en aplicaciones. Es un proceso que detecta vulnerabilidades en aplicaciones de software al evaluar el código fuente, bytecode o binarios de una aplicación sin ejecutarla. El SAST se realiza en una etapa temprana del ciclo de vida del desarrollo de software (SDLC) y puede identificar problemas de seguridad antes de que el código se ejecute en un entorno en vivo.
- SCA (Software Composition Analysis): Análisis de los componentes del software. Es un proceso que identifica y evalúa las bibliotecas y componentes externos (generalmente open source pero también pueden ser privativos) utilizados en una aplicación para detectar vulnerabilidades de seguridad, licencias y otros riesgos. El SCA ayuda a las organizaciones a gestionar y asegurar los componentes de terceros que se incorporan en sus aplicaciones.
- SDLC (Software Development Life Cycle): Es el Ciclo de Vida del Desarrollo de Software. Es un proceso estructurado utilizado por las organizaciones para diseñar, desarrollar, probar y desplegar software de alta calidad. El SDLC define las fases y tareas asociadas con el desarrollo de software, desde la concepción inicial hasta el mantenimiento y obsolescencia del desarrollo. Las fases típicas del SDLC incluyen: planificación, análisis, diseño, implementación, pruebas y mantenimiento.
- S-SDLC (Secure Software Development Life Cycle): Ciclo de Vida del Desarrollo de Software Seguro. Es una adaptación del SDLC tradicional que integra prácticas y procedimientos de seguridad en cada fase del desarrollo de software. El objetivo del S-SDLC es mejorar la seguridad cuenta desde el inicio y a lo largo de todo el proceso de desarrollo, reduciendo así el riesgo de vulnerabilidades y amenazas en el software producido. Las fases del S-SDLC incluyen actividades específicas de seguridad, como análisis de riesgos, revisión de código y pruebas de seguridad.
- Semgrep: es una herramienta de análisis estático de código abierto utilizada para identificar problemas de seguridad y calidad en el código fuente de software mediante reglas personalizables.

- SpotBugs: es una herramienta de análisis estático de código abierto utilizada para encontrar posibles errores y defectos en el código fuente de aplicaciones Java. Ofrece una revisión automatizada del código para identificar problemas de calidad y posibles vulnerabilidades de seguridad, ayudando a los desarrolladores a mejorar la calidad y la seguridad de su software.
- OASIS (Organization for the Advancement of Structured Information Standards): es un consorcio internacional sin ánimo de lucro cuyo objetivo es impulsar el desarrollo, la convergencia y la adopción de estándares abiertos para el sector de IT.
- Test Case: Un escenario específico de prueba que describe una condición o evento a probar en una aplicación, incluyendo los pasos a seguir y los resultados esperados.
- Test Suite: Un conjunto de casos de prueba relacionados o agrupados que se ejecutan de manera conjunta para probar una característica o componente de software específico.

9 Bibliografía

1. Higuera JR, Bermejo J, Montalvo JA, Villalba J, P閞ez J. Benchmarking Approach to Compare Web Applications Static Analysis Tools Detecting OWASP Top Ten Security Vulnerabilities. Comput Mater Contin. 30 de junio de 2020;64:1555-77.

2. Snyk [Internet]. 2021 [citado 10 de octubre de 2023]. You can't compare SAST tools using only lists, test suites, and benchmarks. Disponible en: https://snyk.io/blog/cant-compare-sast-tools-lists-test-suites-benchmarks/

3. Cartey L, Keaton D, Hagen S. OASIS Static Analysis Results Interchange Format (SARIF) TC | OASIS [Internet]. [citado 10 de octubre de 2023]. Disponible en: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif

4. OWASP. OWASP Benchmark | OWASP Foundation [Internet]. [citado 10 de octubre de 2023]. Disponible en: https://owasp.org/www-project-benchmark/

5. Krishnamurthy R. Benchmarking Static Analysis Tools for C [Internet]. CodeX. 2021 [citado 10 de octubre de 2023]. Disponible en: https://medium.com/codex/11-static-analysis-tools-for-c-4fe5f63c18a5

6. SAMATE. NIST [Internet]. 3 de febrero de 2021 [citado 25 de octubre de 2023]; Disponible en: https://www.nist.gov/itl/ssd/software-quality-group/samate

7. Software Assurance Reference Dataset (SARD). NIST [Internet]. 3 de febrero de 2021 [citado 27 de octubre de 2023]; Disponible en: https://www.nist.gov/itl/ssd/software-quality-group/samate/software-assurance-reference-dataset-sard

8. Static Analysis Tool Exposition (SATE). NIST [Internet]. 15 de abril de 2021 [citado 27 de octubre de 2023]; Disponible en: https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate

9. ZAP – SARIF JSON Report [Internet]. [citado 27 de octubre de 2023]. Disponible en: https://www.zaproxy.org/docs/desktop/addons/report-generation/report-sarif-json/

10. Haddad I. An Open Guide To Evaluating Software Composition Analysis Tools. noviembre de 2020; Disponible en: https://project.linuxfoundation.org/hubfs/Reports/An-Open-Guide-To-Evaluating-Software-Composition-Analysis-Tools_V2.pdf?hsLang=en

11. Haddad I. Open Source Compliance in the Enterprise [Internet]. 2018 [citado 29 de octubre de 2023]. Disponible en: https://www.linuxfoundation.org/resources/publications/open-source-compliance-in-the-enterprise

12. Hdiv Scores a Perfect 100% at OWASP Benchmark | Hdiv Security [Internet]. 2022 [citado 5 de noviembre de 2023]. Disponible en: https://web.archive.org/web/20220625103146/https://hdivsecurity.com/owasp-benchmark

13. Delaitre et al. - 2018 - SATE V report ten years of static analysis tool e.pdf [Internet]. [citado 5 de noviembre de 2023]. Disponible en: https://samate.nist.gov/SATE5/SATE5%20Report.pdf

14. Supported languages | Semgrep [Internet]. [citado 11 de diciembre de 2023]. Disponible en: https://semgrep.dev/docs/supported-languages/

15. Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf [Internet]. [citado 12 de diciembre de 2023]. Disponible en: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf