Análisis de seguridad y calidad de aplicaciones (Sonarqube)

Juan Pablo Ospina Delgado

Universidad Obterta de Cataluyna

Ingeniería, Departamento de Informática

Manizales, Colombia

2015

Análisis de seguridad y calidad de aplicaciones (Sonarqube)

Juan Pablo Ospina Delgado

Memoria del proyecto presentado como requisito parcial para optar al título de:

Máster en Seguridad de las tecnologías de la información y de las comunicaciones .

Consultor:

Pau del Canto Rodrigo

Línea del proyecto:

Seguridad del software

Universidad Obterta de Cataluyna

Ingeniería, Departamento de informática

Manizales, Colombia

2015

Todo es teóricamente imposible, hasta que se hace.

Robert A. Heinlein

Agradecimientos

Primero quiero agradecer a mi familia por el cariño y apoyo constante que me han brindado todo este tiempo, a mi novia Alejandra por ser cada día una fuente de inspiración y amor. Además al tutor de este trabajo Pau del Canto Rodrigo por ofrecerme una guía y colaboración constante en la realización de esta tesis.

Resumen

Esta tesis presenta una revisión general del estado de las herramientas de análisis de código estático enfocadas en detectar vulnerabilidades en aplicaciones web y su implementación dentro del ciclo de desarrollo del software. Estas herramientas constituyen una forma rápida, eficiente y sencilla de asegurar los proyectos de software, incluso si no todos los fallos son detectados. La evaluación busca demostrar el estado actual y real de los programas de análisis estático, con el fin que ANCERT logre mejorar la seguridad de sus proyectos a partir de la implementación de la herramienta que mas se ajuste a sus necesidades y metodologías de desarrollo. Por último el trabajo pretende ser una fuente de consulta para futuras pruebas y avances en el campo del análisis estático de código en la detección de vulnerabilidades.

Palabras clave: Análisis de código estático, seguridad del software, seguridad aplicaciones web, seguridad del código.

Abstract

This thesis presents an overview state of the static code analysis tools that focused on detecting vulnerabilities in web applications and their implementation within the software development life-cycle. These tools provide a fast, efficient and easy way to ensure software projects, even if not all faults are detected. The evaluation aims to demonstrate the current and real state of static analysis programs, in order to achieve ANCERT improve the security of their projects from the implementation of the tool that best meets their needs and development methodologies. Finally the work is intended as a resource for future tests and developments in the field of static code analysis for detecting vulnerabilities.

Keywords: Static code analysis, software security, web application security, secured code.

Contenido

Pág.

Resumen **5**

Lista de figuras **8**

Lista de tablas **10**

1. Motivación **12**

2. Estado del Arte **14**

3. Objetivos **17**

4. Capítulo Fundamentos **18**

4.1 Análisis de código estático 19
4.1.1 Técnicas para el análisis estático 20
4.1.2 Usos del análisis de código estático 23
4.1.3 Ventajas y desventajas 26
4.2 SonarQube 28
4.2.1 Ejecutando el primer análisis 30
4.3 Taxonomía de vulnerabilidades en aplicaciones web 37
4.3.1.1 Clasificación Owasp top 10 39

5. Capítulo Evaluación **44**

5.1 Métricas 45
5.1.1 Reportes 46
5.1.2 Integración con la gestión de proyectos 46
5.1.3 Capacidad para la detección de errores 46
5.2 Juliet Test Suite 46
5.2.1 Estructura de las pruebas 48
5.2.2 Estructura de la suite 50

5.3 Resultados casos de prueba....52
5.3.1 SonarQube....52
5.3.2 VisualCode Grepper....56
5.3.3 Fortify HP on Demand....58

6.Conclusiones y trabajo futuro....62

6.1 Conclusiones....62
6.2 Trabajo futuro....64

Bibliografía....65

Lista de figuras

Pág.

Figura 4-1: Flujo de control [21]....21

Figura 4-2: Salida consola primer análisis....32

Figura 4-3: Dashboard Sonarqube....32

Figura 4-4: Widget Project....33

Figura 4-5: Métrica tamaño....33

Figura 4-6: Métrica incumplimiento de estándares y defectos....34

Figura 4-7: Métrica pruebas unitarias....35

Figura 4-8: Métrica complejidad....36

Figura 4-9: Métrica comentarios....36

Lista de tablas

Pág.

Tabla 4-1: Lenguajes soportados por Sonarqube [32]........29
Tabla 4-2: Taxonomía con relación al OWASP Top 10 [35]........42
Tabla 5-1: Resultados inyección sql (Sonarqube)........52
Tabla 5-2: Resultados XSS (Sonarqube)........53
Tabla 5-3: Resultados XSS-2 (Sonarqube)........54
Tabla 5-4: Resultados criptografía-1 (Sonarqube)........54
Tabla 5-5: Resultados criptografía-2 (Sonarqube)........55
Tabla 5-6: Resultados criptografía-3 (Sonarqube)........55
Tabla 5-7: Resultados inyección sql (VGC)........56
Tabla 5-8: Resultados XSS-1 (VCG)........57
Tabla 5-9: Resultados XSS-2 (VCG)........57
Tabla 5-10: Resultados criptografía-1 (VCG)........57
Tabla 5-11: Resultados criptografía-2 (VCG)........57
Tabla 5-12: Resultados criptografía-3 (VCG)........58
Tabla 5-13: Resultados inyección sql (Fortify HP)........59
Tabla 5-14: Resultados XSS-1 (Fortify HP)........59
Tabla 5-15: Resultados XSS-2 (Fortify HP)........60
Tabla 5-16: Resultados criptografía-1 (Fortify HP)........60

Tabla 5-17: Resultados criptografía-2 (Fortify HP)..60

Tabla 5-18: Resultados criptografía-3 (Fortify HP)..61

1. Motivación

El impacto de la revolución tecnológica afectó la economía, la sociedad y la vida cotidiana de las personas, en igual medida las empresas y en general las organizaciones que se sumaron a este cambio con la incorporación de soluciones informáticas en sus procesos y en la gestión de sus negocios, así que la tecnología se transformó en uno de los recursos de mayor impacto y valor para las industrias (energía, finanzas, transporte, etc). A medida que crece la propagación de los sistemas de información en la sociedad, los individuos toman mayor conciencia de la importancia de la seguridad de sus sistemas. Aún así el estado actual de la seguridad informática en la práctica es pobre, pues la gran mayoría de sistemas son guiados por modelos correctivos y no preventivos; los problemas de ejecutar procedimientos correctivos pos-desarrollo son sus altos costos y su falta de escalabilidad. Por lo tanto, gran parte de las investigaciones realizadas en la última década sobre temas de seguridad en el desarrollo de sistemas, van orientadas a la aplicación de métodos y procesos en la ingeniería de software que aporten una mejora considerable en la seguridad del código [1, 2], algunos resultados de estas investigaciones son el modelado de amenazas de Microsoft stride, el cual se aplica en el ciclo de vida de sus desarrollos; sin embargo, muchos de los resultados de estas investigaciones como stride o dread no son soluciones reales para empresas cuyos recursos económicos y humanos sean limitados, de manera que la gran mayoría de las aplicaciones web creadas en la actualidad son vulnerables [3].

Por otro lado, la incorporación de estas técnicas dentro de los marcos de las metodologías ágiles no son factibles dada su complejidad y requisito de tiempo. Por lo cual cada vez se

consideran más importantes las herramientas que automatizan procesos en pro de mejorar la seguridad como son los escáneres de vulnerabilidades o los analizadores de código estático, sobre todo estos últimos toman gran relevancia al ser programas mucho más precisos que un escáner, dado su acceso directo al código, además de ser alternativas que se integran perfectamente en el ciclo de desarrollo del software, independientemente de la metodología; por último no son herramientas factibles para utilizar en el pos-desarrollo, lo que obliga a los programadores a desarrollar aplicaciones de calidad con buenos estándares en seguridad.

2. Estado del Arte

Desde 1988 que se explotó una vulnerabilidad de tipo bufferover flow con el gusano Morris, desarrollado por Rober Morris, la seguridad en el software ha tomando gran notoriedad y atención. A esto se sumó el gran crecimiento de Internet desde los 90's, dando espacio a la creación de nuevas tecnología y con ellas nuevas vulnerabilidades. A partir de la deficiencia de la seguridad en el software, y teniendo en cuenta su gran impacto en la sociedad y el elevado número de ataques que se producían, se introdujo el concepto de seguridad del software en el año 2000 con los textos de Viega, McGraw y Howard, LeBlanc [4, 5]. En ellos se toma la posición de asegurar el software durante la fase de diseño y se critica como las compañías hacen frente a la seguridad de sus programas por medio de parches, culpando directamente la implementación de un sistema de desarrollo que se enfoca principalmente en añadir características y no en la calidad y la seguridad del código. Este debate se trasladó al avance y actualización del modelado de amenazas [6] que se puede definir como un procedimiento para mejorar la seguridad de los sistemas a través de la identificación de objetivos y vulnerabilidades, con el fin de definir contra medidas para mitigar los efectos de las amenazas.

Continuando con la evidente falta de incorporación de métodos en las fases iniciales del desarrollo, que aseguren el software, se desarrollaron en analogía con los patrones de diseño que tienen por propósito estructurar bien el software, los patrones en seguridad [7, 8], que se enfocan en imponer un nivel adicional de seguridad en la fase de diseño junto

con el modelado de amenazas. Estas metodologías en parte fueron incorporadas en el ciclo de desarrollo seguro (SDL) [9], creando así un conjunto de procesos que logran reducir en más de un 50 por ciento las vulnerabilidades del software [10].

Junto a estos avances en la seguridad de software existen alternativas antiguas como la inspección de fagan [11] que busca, por medio de un proceso estructurado de revisión de documentos de desarrollo, encontrar defectos en las aplicaciones, este concepto puede ser utilizado en la seguridad de software basado en revisiones manuales, no solo del código sino también en la documentación, los requerimientos y el diseño del desarrollo. El procedimiento tiene como principal virtud ser aplicable en cualquier fase del ciclo de vida del software y sobre cualquier tipo de metodología, sea ágil o no, aun así es un procedimiento que requiere de una capacitación previa hacia los involucrados y una enorme responsabilidad por parte de los desarrolladores.

La inspección de fagan dio paso al análisis estático de código como alternativa al aumento de la seguridad en el software. El análisis estático de código es efectivo, pero es un proceso poco óptimo, dado que cubre aproximadamente entre 100 y 200 líneas de código por hora [12], si se tiene en cuenta que cada proyecto puede contener cientos o miles de vulnerabilidades. Es difícil, entonces, para un desarrollador realizar un análisis sobre un fragmento que contenga diversos tipos de fallos, lo que ocasiona que una fracción de código deba ser revisada varias veces. Como alternativa se desarrollaron herramientas que automatizan el análisis estático de código, logrando una solución que alcanza a identificar un gran número de vulnerabilidades a la vez, a una proporción mucho mayor que cualquier desarrollador experimentado. De igual manera el análisis automático de código no es capaz aún de reemplazar completamente las revisiones manuales, dada la gran cantidad de falsos positivos y falsos negativos que este tipo de herramientas generan.

Desde el desarrollo de la primera herramienta para el análisis de código estático llamada Lint, la cual fue desarrollada para el lenguaje c a finales de los 70 [13] se han creado una

gran cantidad de programas de este tipo, enfocados a la calidad del código[1] y a partir de estas soluciones se han desarrollado alternativas enfocadas a la seguridad del software[2]. En la actualidad en el campo del análisis de código estático existen gran cantidad de investigaciones encaminadas a la verificación de la corrección del análisis [14] o a la optimización de su ejecución [15]. Esta tesis tiene como enfoque la evaluación e implementación de las herramientas de análisis de código estático dirigidas a la mejora en la seguridad de las aplicaciones web desarrolladas en java y php.

1 http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

2 National Institute of Standards, http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

3. Objetivos

3.1 Objetivo general

Implantar dentro del ciclo de vida del desarrollo de aplicaciones el cómputo automático de una serie de métricas sobre la calidad del código y la búsqueda de vulnerabilidades. Lo anterior se hará realizando la integración de estos módulos de análisis dentro de la herramienta de software libre SonarQube.

3.2 Objetivos específicos

- Analizar las amenazas más comunes y frecuentes para las aplicaciones Web. Especialmente se tendrán en cuenta las aplicaciones Web desarrolladas en el lenguaje de programación JAVA. Se estudiarán los aspectos de seguridad de cada uno de estos lenguajes y las vulnerabilidades más extendidas

- Valorar SonarQube como herramienta de análisis de código estático con base en diferentes métricas que se irán estableciendo a lo largo del trabajo, priorizando su evaluación sobre el aporte que ofrece en el incremento de la seguridad en las aplicaciones web.

- Integrar y definir el análisis de código estático dentro del ciclo de desarrollo seguro del software como apoyo en el aumento de los estándares de seguridad en las aplicaciones web.

4. Capítulo Fundamentos

En este capítulo se definen los conceptos necesarios para desarrollar e integrar una herramienta de análisis de código estático como SonarQube, en la búsqueda de vulnerabilidades de las aplicaciones web desarrolladas en JAVA.

Primero se estudiará el análisis de código estático, cuáles son las técnicas mas utilizadas para automatizar este proceso, cómo las herramientas de análisis de código pueden aportar en la mejora de la calidad del código y, en general, a todo el ciclo de desarrollo del software. Además se precisa cuáles son las aplicaciones que tienen este tipo de análisis.

A continuación se realiza un estudio sobre las funcionalidades y características de SonarQube a partir de ejecutar un primer análisis, usando el runner y un archivo básico de propiedades. Una vez se complete el proceso se va a detallar todo el cuadro de resultados centrado en los siete pecados capitales del desarrollo de software. Luego de comprender cada uno de los siete principios propuestos por SonarQube, se abordan las convenciones de la interfaz y la jerarquía de paquetes y clases. Por último se realiza una breve comparación de SonarQube con las otras herramientas que existen en el mercado de este tipo, siempre que puedan detectar vulnerabilidades en cualquier lenguaje.

Al final del capitulo se explica cómo se clasifica un fallo en términos de vulnerabilidad, esto hace más fácil determinar cuál es el efecto de la vulnerabilidad que se detecta y a partir de una taxonomía se puede describir la causa de la misma, seguido de una explicación en detalle para cada caso.

4.1 Análisis de código estático

El análisis estático es un proceso que se realiza sobre el código de una aplicación sin necesidad de ejecutarse. La definición general puede incluir el análisis estático manual realizado por un humano. Pero, para términos de esta tesis, el análisis estático es un análisis realizado de forma automatizada por una herramienta. La idea principal de estas herramientas es encontrar, tan pronto como sea posible, defectos de codificación en el ciclo de vida del desarrollo, generalmente después de escribir el código fuente de la aplicación.

Las primeras herramientas de análisis de código estático eran sencillos programas de chequeo que no eran mas sofisticados que un simple comando grep o find de unix. Estos software fueron rápidamente precedidos por herramientas comerciales mas sofisticadas que buscaban automatizar la revisión de código para una variedad de lenguajes de programación. Estos programas analizan los fallos mas comunes para cada lenguaje en particular y ayudan a los desarrolladores a crear códigos mas estables y de mayor calidad.

La complejidad de su análisis varía según si se tiene en cuenta el comportamiento de instrucciones y declaraciones individuales o si incluyen el código fuente completo de una aplicación. La información recopilada puede ser utilizada para indicar posibles problemas de codificación o incluso demostrar por medio de métodos formales ciertas propiedades del software.

El ideal de estas herramientas sería que encontraran automáticamente fallos de seguridad con un alto grado de confianza, pues gran cantidad de los resultados mostrados son falsos positivos o falsos negativos. Sin embargo, actualmente existen varias investigaciones encaminadas a mejorar la precisión del análisis [16,17]. Por lo tanto las herramientas actuales sirven como ayuda para un analista para volver más eficiente su trabajo y ofrecer un apoyo que se convierte en guía, mas no son herramientas que logran detectar los defectos de forma completamente automática [18].

4.1.1 Técnicas para el análisis estático

4.1.1.1 Análisis de flujo de datos

El análisis de flujo de datos es un proceso de recogida de información en el software de forma dinámica sin necesidad de ejecutar el programa. Sin embargo, no utiliza la semántica de los operadores. La semántica de la sintaxis del lenguaje es capturada de forma implícita en el algoritmo [19].

Existen tres términos comunes cuando se habla del análisis de flujo de datos que van a ser definidos a continuación.

Bloque básico: Es una porción de código de un programa que solo contiene un punto de entrada y un punto de salida. Esto hace que un bloque básico sea óptimo para el análisis. Los compiladores usualmente descomponen el programa en sus bloques básicos como primer paso para el proceso de análisis [20].

El código de un bloque básico tiene

1. Un punto de entrada, quiere decir que ningún código en el bloque tiene como destino un salto de instrucción.

2. Un punto de salida, significa que solo la última instrucción del programa puede causar la ejecución del código de un bloque básico diferente.

Bajo esta circunstancia, cada vez que se ejecuta la primera instrucción de un bloque básico, el resto de las instrucciones deben ser ejecutadas exactamente una vez en orden.

Grafo de flujo de control: Un gráfico de flujo de control es una representación abstracta de un procedimiento o programa. Cada nodo en la gráfica representa un bloque básico. Las aristas dirigidas son para representar los saltos en el flujo de control. Si un bloque no tiene ninguna línea de entrada entonces se nombra como un bloque de entrada, y si no tiene ninguna línea saliente entonces es un bloque de salida. A continuación hay un gráfico que representa un flujo de control.

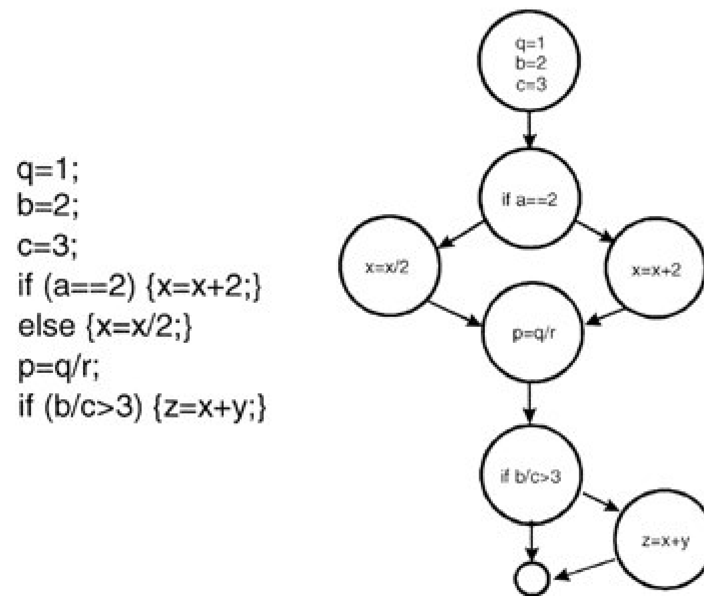

Figura 4-1: Flujo de control [21].

Orientación del flujo de control: La orientación en el flujo de control es un camino que inicia en el grafo con un bloque de entrada y termina con un bloque de salida. Normalmente existen muchos posibles caminos, a menudo se pueden encontrar una cantidad infinita de posibilidades dado el impredecible funcionamiento de algunos bucles. Uno de los principales usos que se les da es identificar la vía más larga en el flujo de control para establecer cuál es el tiempo de ejecución en el peor de los casos.

El análisis de flujos de datos se puede completar mediante la definición de cada bloque que modifica datos y luego realizar iteraciones desde el punto fijo (nodo) a través de todo el grafo. Este enfoque es particularmente útil para la optimización del software. Y más relevante para el alcance de este trabajo, el análisis de flujo de datos puede facilitar la identificación de las dependencias entre datos y realizar un seguimiento en los efectos que provoca la entrada de datos en cada bloque.

El seguimiento de las entradas es un concepto esencial en la seguridad del software, pues todos los datos que provienen del exterior del programa deben ser considerados

potencialmente maliciosos, incluyendo los archivos de configuración y las comunicaciones entre sistemas locales. Si un valor es procesado sin haber sido verificado puede conducir a la explotación de un fallo. Por lo tanto, el análisis de flujo de datos sufre de una complejidad computacional alta, incluso aplicando heurísticas avanzadas en el flujo de datos se puede consumir mucho tiempo en el análisis de programas grandes.

4.1.1.2 Modelo de verificación

El modelo de control es una técnica algorítmica que revisa la descripción de un sistema a partir de especificaciones. A partir de la descripción, junto con una especificación lógica, el algoritmo del modelo de verificación prueba que esa descripción satisface las especificaciones programadas, o si es necesario reporta un contra ejemplo donde se violan estas especificaciones. Los software que aplican modelos de verificación sobre el código implementado han sido una área de amplia investigación. [22,23,24] La entrada para este tipo de programas es el código fuente del programa (descripción del sistema) y una propiedad temporal (especificación). La especificación usualmente es dada por la instrumentación de un programa definido por un autómata, el cual observa si la ejecución del código viola una propiedad deseada, como una característica especifica de seguridad. La salida de un modelo de verificación sería idealmente una prueba de corrección del programa que pueda ser validada de forma individual, o un contra ejemplo que detalle el camino en la ejecución del programa.

El modelo de control encaja perfectamente con las necesidades de los desarrolladores de software que requieren fiabilidad en sus construcciones o incluso garantías sobre programa críticos como sistemas bancarios o sistemas que automaticen procesos industriales delicados. Aún así esta técnica tiene un nivel de complejidad alto y por lo tanto es costosa, hablando en términos computacionales, de suerte que no resulta adecuada su integración en proyectos que requieran bajos estándares de seguridad.

Un ejemplo de este tipo de software es el Blast desarrollado en el año 2007 por el profesor T.A Henzinger [25]

4.1.1.3 Basado en patrones de texto

La forma más sencilla y rápida de encontrar vulnerabilidades triviales en lenguajes de programación como C, es buscar a lo largo de los archivos de código por llamados a funciones débiles y potenciales que puedan provocar vulnerabilidades como los son el get o los métodos de la librería strcpy, por nombrar algunos.

Si bien esta técnica no es de mucha utilidad para personas que no son especialistas en seguridad, porque se debe conocer de antemano que patrón en particular debe ser buscado, las últimas herramientas que se desarrollan piensan en mejorar este aspecto. Estas herramientas vienen con una lista de patrones predeterminados, los cuales contienen el diseño de diferentes vulnerabilidades.

4.1.2 Usos del análisis de código estático

El análisis estático es usado de forma más amplia que lo que simplemente conocen los desarrolladores. Particularmente porque existen diferentes enfoques en las herramientas de este tipo. Por esta razón se van a explicar todos los usos que tiene este tipo de análisis, identificados en su libro por Brian Chess y Jacob West [26].

4.1.2.1 Type checking

El type checking es la forma más común en la que se usa análisis estático, y con la que mayor número de programadores están familiarizados. Las reglas de este tipo de revisiones generalmente están predefinidas en los lenguajes de programación y se obliga su uso por medio de un compilador, por lo tanto los desarrolladores no tienen claro que esto sea un tipo de análisis estático. El type checking elimina varios tipos de errores, por ejemplo, previene a los programadores de asignar accidentalmente un valor integral en una variable de tipo objeto. La captura de los fallos se da en tiempo de compilación y de esta forma evita la ejecución del programa defectuoso.

4.1.2.2 Revisión de estilo

Las revisiones de estilo generalmente son más superficiales que las revisiones de tipo "type checking", las revisiones de estilo puras obligan a utilizar estándares tales como nombres de espacio, nombramiento de funciones, re-definir funciones obsoletas, comentarios, estructuras del programa, y similares. Como muchos programadores tienen su propia percepción de cual es un buen estilo de codificación, entonces la mayoría de herramientas de revisión de estilo son flexibles sobre su número de reglas. Las alertas lanzadas por los revisores de estilo generalmente afectan la legibilidad y el mantenimiento del código, pero no indican errores particulares que puedan ocurrir cuando el programa se ejecuta.

4.1.2.3 Entender el programa

Las herramientas que apuntan al entendimiento de programas ayudan a dar una visión general de su código base. Los ambientes de desarrollo integrados (IDE) generalmente incorporan alguna funcionalidad o análisis que permita comprender los proyectos. Algunos ejemplos incluyen "encontrar todas las implantaciones de una función" o "encontrar las declaraciones globales de una variable", otros análisis más avanzados pueden soportar la refactorización de un programa, como renombrar un conjunto de variables o dividir un simple método en múltiples métodos. Todo esto es útil, especialmente si un desarrollador quiere entender un fragmento de código nuevo o si quiere refactorizar ese fragmento.

4.1.2.4 Encontrar errores

El propósito de los analizadores que encuentran errores no es identificar problemas de formato, como una revisión de estilo, ni tampoco errores que pueda identificar un compilador, sino comportamientos inesperados que se puedan presentar en la ejecución del programa. Errores comunes de este tipo son consecuencias de unas malas prácticas de programación, comportamientos incorrectos, problemas de rendimiento o código poco confiable. Los software de análisis con este enfoque implementan patrones como reglas que posteriormente son comparadas con el código fuente de la aplicación. Un ejemplo de estas

herramientas es Findbug[3] la cual es integrada en las herramientas de análisis de SonarQube.

4.1.2.5 Seguridad

Las herramientas de análisis estático que tienen como principio la seguridad implementan técnicas similares a los otros programas, pero más enfocadas en su objetivo, lo que significa que aplican esas técnicas de forma diferente.

Los primeros programas que se desarrollaron con este fin eran muy cercanos a los de revisión de estilo, pues realizaban un escaneo del código en busca de funciones que podían provocar fallos de seguridad, pero no necesariamente eran causa de las mismas, solo indicaban líneas de código que representaban un alto grado de preocupación para los desarrolladores. En la última década estas herramientas han sido criticaras por arrojar en sus resultados un gran número de falso positivo pues los programadores buscan en estos software una lista fiable de fallos, más no un soporte para una revisión manual de código.

En la actualidad estas herramientas de análisis de código estático enfocadas en mejorar la seguridad del software, se han optimizado y la mayoría utilizan patrones para encontrar vulnerabilidades, combinando técnicas basadas en modelos de verificación y patrones de texto, acoplando así métodos utilizados en los programas de detección de errores y de revisión de estilo.

3 FindBugs. FindBugsTM - Find Bugs in Java Programs. http://findbugs.sourceforge.net/

4.1.3 Ventajas y desventajas

Basado en el documento de OWASP donde se especifican las ventajas y desventajas del análisis de código estático se puede enumerar los siguiente puntos[4].

Ventajas:

- Escala bien, puede ser implementado en múltiples proyectos y es repetible.
- Pueden detectar con alta confianza vulnerabilidades bien documentadas como los desbordamientos de pila, la inyecciones SQL etc.
- La ejecución del análisis generalmente es rápida.
- Detecta la causa del problema pues se realiza una búsqueda directa en el código, mientras los test de penetración solo establecen el problema, mas no el motivo.

Desventajas:

- Muchas clases de vulnerabilidades son difíciles de encontrar automáticamente, como los problemas de autenticación, los accesos de control, la lógica del negocio, etc. El estado del arte actual solo permite encontrar un pequeño porcentaje de las vulnerabilidades.
- Los resultados contienen un alto número de falsos positivos.
- Generalmente no pueden encontrar problemas de configuración, pues estos archivos no están representados en el código.
- Muchas de estas herramientas tienen dificultades para analizar código que no se encuentra compilado.

4 https://www.owasp.org/index.php/Static_Code_Analysis

4.1.3.1 Lista de herramientas de análisis de código estático

A continuación se comparte una lista de herramientas enfocadas en el análisis de código estático para la detección de vulnerabilidades[5].

Libres/Código abierto

- Google CodeSearchDiggity (Multiple)
- PMD (Java)
- FlawFinder (C/C++)
- Microsoft FxCop (.NET)
- Splint (C)
- FindBugs (Java)
- RIPS (PHP)
- Agnitio (Objective-C, C#, Java & Android)
- Microsoft PreFast (C/C++)
- Fortify RATS (C, C++, Perl, PHP & Python)
- DevBug (PHP)
- Brakeman (Rails)
- VisualCodeGrepper (C/C++, C#, VB, PHP, Java & PL/SQL)

Comerciales

- Fortify
- Veracode
- GrammaTech
- ParaSoft
- Armorize CodeSecure

5 https://www.owasp.org/index.php/Static_Code_Analysis

- Checkmarx Static Code Analysis
- Rational AppScan Source Edition
- Coverity
- BugScout
- Insight

4.2 SonarQube

Sonarqube es una plataforma de código abierto usada por los equipos de desarrollo para controlar la calidad del código. Sonar fue desarrollado con el principal objetivo de hacer accesible la administración de la calidad del código con un mínimo esfuerzo. Como tal, Sonar contiene en su núcleo de funcionalidades un analizador de código, una herramienta de reportes, un módulo que detecta defectos y una función para regresar los cambios realizados en el código.

Prácticamente en todas las industrias, los grandes líderes utilizan métricas. Ya se trate de defectos y perdidas en la manufactura, ventas y ganancias, o la tracción en una página web, existen esas métricas que les dice a los líderes como están haciendo su trabajo y que tan efectivo es, en general si se están obteniendo mejores o peores resultados. Ahora existen este tipo de métricas para el desarrollo de software, empaquetadas y presentadas a través de una plataforma estandarizada, basada en servidor (sin necesidad que el cliente instale algo) que utiliza herramientas respetadas en la industria, como Findbug, PMD y JaCoCo. Esos resultados son presentados de forma intuitiva por medio de una interfaz web que además ofrece RSS de las alertas generadas cuando los umbrales de calidad establecidos se vulneran.

En términos de lenguajes Sonarqube soporta Java en su núcleo principal, pero es flexible a través de plugins comerciales o libres, por lo tanto se puede extender a lenguajes como Actionscript, PHP, PL/SQL, Python (etc.) ya que el motor de reportes es independiente

del lenguaje de análisis. A continuación se deja una lista con los lenguajes que soporta Sonarqube.

Tabla 4-1: Lenguajes soportados por Sonarqube [27].

Lenguaje	Pago/Gratis	Métricas
Abap	Pago	Tamaño, comentarios, complejidad, duplicados y errores
C	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
C++	Gratis/Pago	Tamaño, comentarios, complejidad, duplicados y errores (También existen plugins de pago).
C#	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas, errores. El análisis de C# se ejecuta por una seria de herramientas externas que deben ser instaladas de forma individual.
Cobol	Pago	Tamaño, comentarios, complejidad, errores. Métricas especificas del lenguaje como salida y entrada de declaraciones.
Delphi	Gratis	Tamaño, comentarios, complejidad, diseños, pruebas duplicados y errores
Drools	Gratis	Tamaño, comentarios, y errores
Flex/Actionscript	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores
Groovy	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores
Javascript	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores

Natural	Pago	Tamaño, comentarios, complejidad, duplicados y errores
Php	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
PL/1	Pago	Tamaño, comentarios, complejidad, duplicados y errores
PL/Sql	Pago	Tamaño, comentarios, complejidad, duplicados y errores
Pytho	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
Visual Basic 6	Pago	Tamaño, comentarios, complejidad, duplicados y errores
JSP	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
XML	Gratis	Tamaño y errores

4.2.1 Ejecutando el primer análisis

Después de realizar la instalación de sonarqube como se muestra en el apéndice A es necesario correr el programa Sonarqube runner que se encarga de ejecutar realmente el análisis. Este análisis primero requiere establecer algunas propiedades a través del archivo “sonar.propieties”.

Los valores específicos del proyecto se pueden definir en el runtime, por la línea de comandos o como se nombró antes en un archivo de propiedades. El archivo de propiedades más simples que un proyecto pueda contener, sería similar al siguiente:

 # required metadata

sonar.projectKey=my:project

sonar.projectName=MyProject

sonar.projectVersion=1.0

 # required path to source directories

sources=srcDir1,srcDir2

Para ejecutar el análisis con sonar runner es necesario acceder a la url del proyecto que contiene el código a analizar, y se ejecuta de la siguiente manera:

cd [directorio del proyecto]

/directorio-de-sonar-runner/sonar-runner

El análisis comenzará de forma inmediata al ejecutar estos comandos, al final de la línea de comandos se muestra el tiempo que tomó el análisis en procesarse, este tiempo varía según el tamaño del proyecto y, si es el caso, la velocidad de la red. En la siguiente figura se muestran los resultados arrojados en consola al realizar el análisis sobre uno de los proyectos de prueba que se facilitan en la página de Sonarqube[6].

```
16:00:55.026 INFO  - Execute decorators...
16:00:55.506 INFO  - Store results in database
16:00:55.845 INFO  - Analysis reports generated in 23ms, dir size=1 KB
16:00:55.853 INFO  - Analysis reports compressed in 8ms, zip size=2 KB
16:00:55.970 INFO  - Analysis reports sent to server in 117ms
16:00:55.970 INFO  - ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashbo
16:00:55.970 INFO  - Note that you will be able to access the updated dashboard once
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
Total time: 14.501s
Final Memory: 20M/700M
INFO: ------------------------------------------------------------------------
[gnupablo@chakra-pc java-sonar-runner-simple]$
```

Figura 4-2: Salida consola primer análisis.

6 https://github.com/SonarSource/sonar-examples

4.2.1.1 Dashboard sonarqube

Una vez completado el análisis se pueden observar las primeras métricas en el dashboard de Sonarqube. Para eso se debe ingresar al enlace http://localhost:9000 en el caso de haber instalado Sonarqube en la máquina local, de otra forma se debe reemplazar en la url el localhost por el dominio correspondiente.

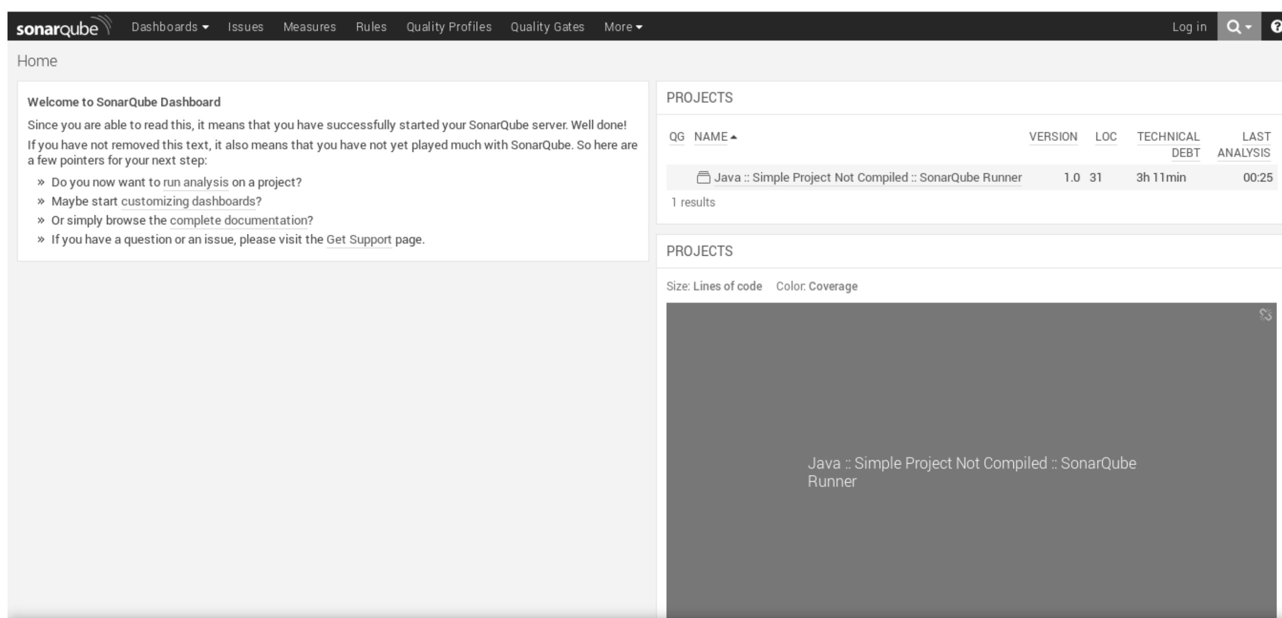

Figura 4-3: Dashboard Sonarqube.

En la figura 4-3 se muestra la página inicial de Sonarqube, que está compuesta por 3 cajas principales, la del lado izquierdo superior contiene un mensaje de bienvenida y algunos enlaces de utilidad, el widgets derecho superior muestra la lista de proyectos activos y por último la de la parte inferior en el sector izquierdo expone un gráfico en árbol de los proyectos. Para visualizar el verdadero potencial de Sonarqube, las métricas, se debe presionar clic sobre el nombre del proyecto en el widget “Projects”.

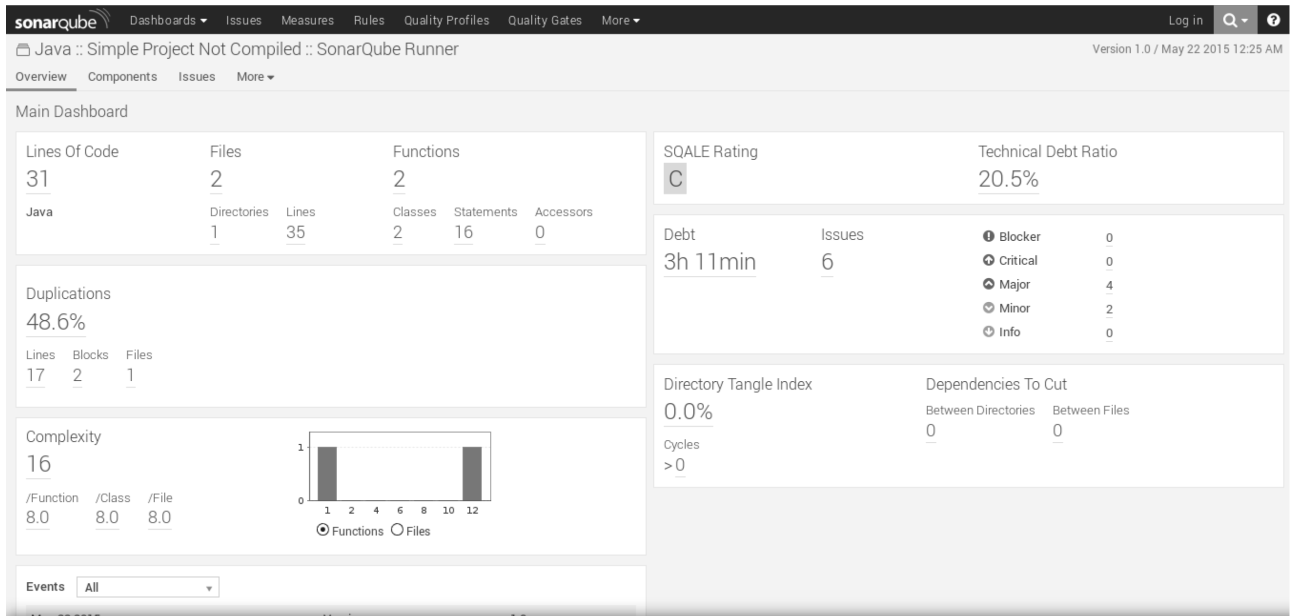

Figura 4-4: Widget Project.

Tamaño

La métrica de tamaño está ubicada en el widget superior izquierdo. En ella se percibe cuántas líneas de código, métodos, clases y paquetes fueron encontrados durante el análisis, como muestra la figura 1.6.

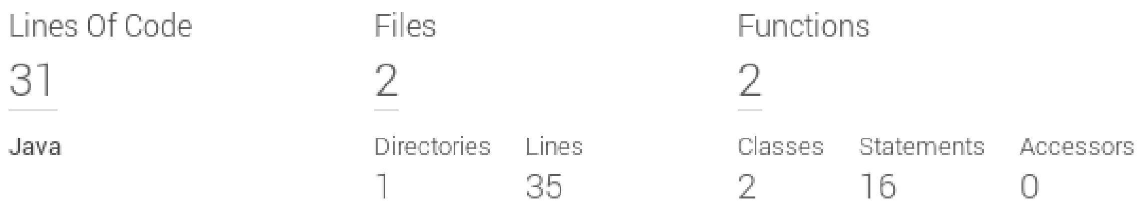

Figura 4-5: Métrica tamaño.

4.2.1.2 Los siete pecados capitales para Sonarqube

El resto de de métricas establecidas en los widgets del panel de control se refieren particularmente a los siete pecados capitales definidos por Sonarqube[7]. A continuación se listan.

7 http://docs.sonarqube.org/display/HOME/Developers%27+Seven+Deadly+Sins

- Defectos y defectos potenciales
- Incumplimiento de estándares
- Duplicados.
- Falta de pruebas unitarias.
- Mala distribución de la complejidad.
- Código espagueti.
- Insuficientes o demasiados comentarios

Incumplimiento de estándares y defectos

Los defectos o defectos potenciales es el pecado más prioritario, pues representa las líneas de código que se están ejecutando mal o que podrían ejecutarse mal en un futuro. Por ejemplo, una referencia nula en una condición o un mal cierre de corchetes se puede considerar un claro ejemplo de un defecto.

Teniendo en cuenta los ejemplos, es claro que algunos problemas son peores que otros. Por eso Sonarqube clasifica los errores por severidad: bloqueante, crítica, mayor, menor, info. El número que está abajo de “Debt” se refiere a la deuda técnica, este valor está basado en el número de días y horas requeridos para corregir los errores encontrados, su cálculo está basado en la metodología SQALE[8]. Mientras sea menor su valor mejor.

Figura 4-6: Métrica incumplimiento de estándares y defectos.

8 http://en.wikipedia.org/wiki/SQALE

Falta de pruebas unitarias

Las pruebas unitarias son una forma de comprobar el correcto funcionamiento de un fragmento de código. Esto permite determinar si un módulo de la aplicación está completo y correctamente terminado. La falta de pruebas unitarias ofrece una baja confiabilidad en el código, y por lo tanto se tiene un proyecto de baja calidad. La siguiente figura muestra la métrica de pruebas unitarias en Sonarqube.

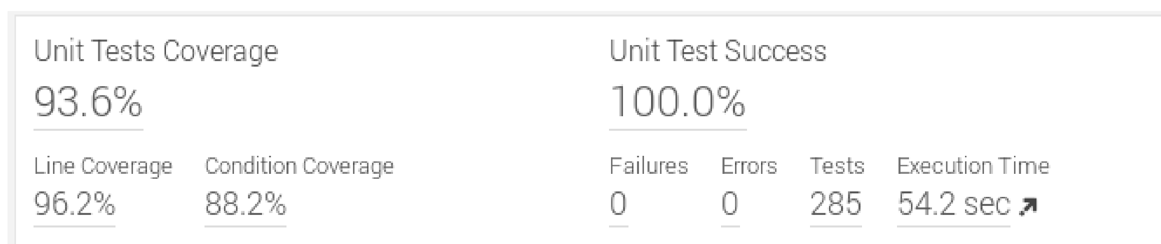

Figura 4-7: Métrica pruebas unitarias.

Mala distribución de la complejidad/Código espagueti

El desarrollo de aplicaciones requiere en sus métodos y clases un nivel de complejidad considerable, pero si se intenta llenar de lógica las soluciones codificadas, entonces se incurre en el problema de generar un código espagueti el cual es difícil de comprender, poco óptimo y en el caso que un nuevo programador necesite trabajar sobre él, entonces, deberá perder mucho tiempo entendiéndolo. La premisa que tiene sonarqube de este indicador es que entre más pares de llaves existan, mayor complejidad tiene el código. Por lo tanto, es mejor si en el widget se muestra un número de complejidad menor. La siguiente figura muestra un ejemplo.

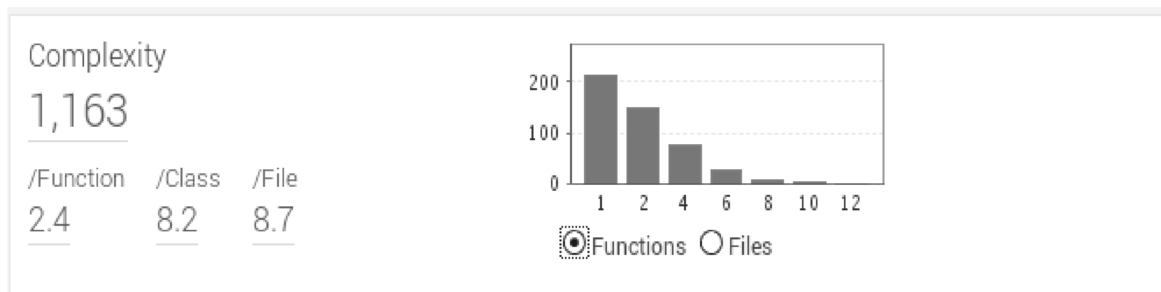

Figura 4-8: Métrica complejidad.

Insuficientes o demasiados comentarios

Existen dos tipos de comentarios: los de una sola línea que se encuentran en cualquier método (público o privado) que quieren mostrar algún detalle de la lógica del código, y los que se encuentran afuera de un método público que intentan comunicar como y porque se debe usar. Los dos tipos de comentarios son necesarios pero en una medida justa, por eso Sonarqube los mide sobre la idea que los desarrolladores deben gastar un mayor porcentaje de su tiempo escribiendo el código de la aplicación y no incurrir en exceder el número de comentarios, ya que así requieren una mayor dedicación en su mantenimiento y se convierten en barrera para los objetivos reales del sistema.

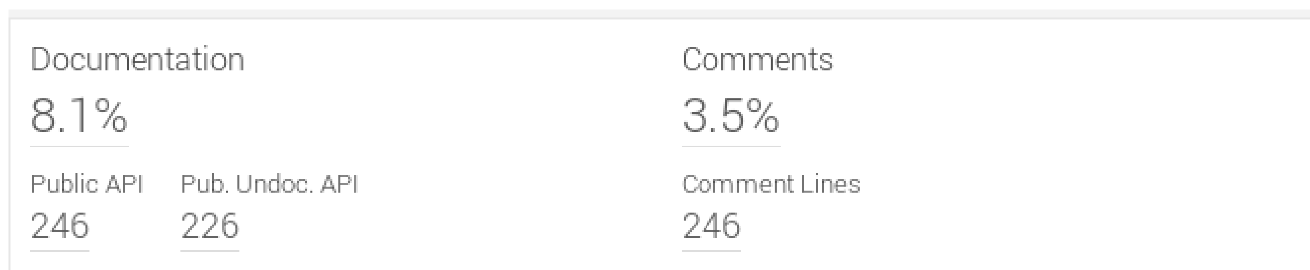

Figura 4-9: Métrica comentarios.

4.3 Taxonomía de vulnerabilidades en aplicaciones web

La palabra taxonomía es definida de manera formal como "Ciencia que trata de los principios, métodos y fines de la clasificación"[9]. Esta clasificación es dada con base en las característica de los objetos, una buena taxonomía es un requisito necesario para un estudio sistemático. Por lo tanto definir una taxonomía para los errores en el software puede ayudar a entender a los desarrolladores y consultores los errores más comunes que afectan la seguridad de las aplicaciones. Para este trabajo se utilizo una de las clasificaciones más difundidas en la actualidad creada por Tsipenyuk, Chess, y McGraw [28]. Esta taxonomía se enfoca en la implementación de los fallos y es especialmente útil para las herramientas de análisis de código estático. Esta clasificación explica la causa de la vulnerabilidad, pero no necesariamente que la afecta. A continuación se enumera y se da una breve descripción de cada uno:

1. Validación de entradas y representación.
2. Abuso de las interfaces de programación (API).
3. Características de seguridad.
4. Tiempo y estado.
5. Manejos de errores.
6. Calidad del código.
7. Encapsulamiento.

Validación de entradas y representación

Los problemas de validación de entradas y representación son causados principalmente por la falta de validación de los datos, los meta-caracteres y las representaciones numéricas. Quiere decir que el fallo en seguridad resulta de confiar en los valores de entrada de la

9 Diccionario en línea de la RAE, http://lema.rae.es/drae/

aplicación. Estos errores incluyen inyecciones sql, desbordamiento de buffer, cross site scripting, por nombrar los mas importantes. Este tipo de vulnerabilidades son en la actualidad los más peligrosos y frecuentes en el software.

Abuso de las interfaces de programación

Una API representa la capacidad de comunicación entre dos entidades, la entidad que solicita por medio del API un resultado de una función, rutina o procedimiento y la otra entidad que aloja el servicio. Normalmente una violación de una interfaz de programación ocurre cuando el solicitante incumple con la condiciones del destinatario o el servicio.

Características de seguridad

Las características de seguridad se refieren a la implementación incorrecta de estas, por ejemplo se encuentran temas como la autenticación, los accesos de control, la confidencialidad, la criptografía, o la administración de privilegios. Existen casos donde las contraseñas de la base de datos se encuentran codificadas en archivos sin protección, esta es una característica de seguridad (autenticación) que esta mal implementada.

Tiempo y estado

La computación distribuida trata principalmente sobre tiempo y estados. Generalmente los desarrollos tiene una ejecución lineal, sin interrupción, pero para los sistemas distribuidos no aplica esta regla dado que dos eventos pueden ocurrir justo en un mismo instante. Los fallos en este tipo de sistemas son causados por interacciones inesperadas entre hilos, procesos, tiempo, estado, y datos. Principalmente en los lapsos de tiempo entre un evento y su respuesta un atacante puede beneficiarse de ese momento y obtener alguna ventaja en las negociaciones de las entidades.

Manejos de errores

Los errores de un programa pueden ser una gran fuente de información para un atacante o inclusive pueden resultar en formas de identificar la existencia o no de una vulnerabilidad.

Este tipo de fallos es muy utilizado en la identificación de vulnerabilidades del tipo "Validación de entrada" como los son las inyecciones SQL o los XSS.

Calidad del código

Una calidad pobre del código conduce en comportamientos impredecibles en una aplicación lo que se manifiesta en una pobre usabilidad. Un software con una baja calidad en el código provee a los atacantes la oportunidad de forzar el sistema de diferentes maneras. Por ejemplo provocar un loop infinito podría ocasionar una denegación de servicio.

4.3.1.1 Clasificación Owasp top 10

Después de definir un nivel de abstracción mas general sobre el tipo de fallos en el software respecto a la seguridad es pertinente continuar con la revisión de una de las listas más reconocidas la cual recopila los riesgos más importantes que afecta a las aplicaciones web, la OWASP top 10 [29]. Este documento pretende concienciar a la industria sobre los riesgos de seguridad, en el ámbito de las aplicaciones web, identificando y remarcando las diez amenazas más serias a las que se exponen.

- Inyección.
- Pérdida de autenicación y gestión de sesiones.
- Secuencia de comandos en sitios cruzados (xss).
- Referencia directa insegura a objetos.
- Configuración de seguridad incorrecta.
- Exposición de datos sensibles.
- Ausencia de control de acceso a funciones.
- Falsificación de peticiones en sitios cruzados.

- Utilización de componentes con vulnerabilidades conocidas.
- Redirecciones y reenvios no validos.

Inyección

Las vulnerabilidades de tipo inyección, tales como: SQL, OS, LDAP se presentan cuando se ingresan a las funciones de este tipo consultas maliciosas o mal formadas que puedan burlar el sistema y ejecutar comandos no deseados que permiten consultar o acceder a datos sin autorización.

Pérdida de autenticación y gestión de sesiones

La autenticación y gestión de sesiones son funciones que comúnmente son implementadas de forma errónea en las aplicaciones web, dando paso para que los atacantes logren obtener acceso no autorizado en los sistemas con la suplantación de la identidad de otros usuarios.

Secuencia de comandos en sitios cruzados (xss)

Los comandos en sitios cruzados se presentan en los lenguajes interpretados por el navegador como son javascript o html, se produce cuando las aplicaciones reciben datos no confiables y los envían sin ningún tipo de verificación al sistema de renderizado. Esta vulnerabilidad permite a un usuario malicioso capturar sesiones de usuario, modificar páginas del sitio web, o re-direccionar a las victimas hacia sitios maliciosos.

Referencia directa insegura a objetos

"Una referencia directa a objetos ocurre cuando un desarrollador expone una referencia a un objeto de implementación interno, tal como un fichero, directorio, o base de datos. Sin un chequeo de control de acceso u otra protección, los atacantes pueden manipular estas referencias para acceder a datos no autorizados".

Configuración de seguridad incorrecta

Cada aplicación que compone un sistema web debe tener definida y configurada unas métricas de seguridad optimas. Todas estas implementaciones deben ser correctamente mantenidas y definidas pues por lo general no son seguras por defecto. Algunas acciones como actualizar el software de forma constante o utilizar capas adicionales de verificación son algunos ejemplo de configuración segura.

Exposición de datos sensibles

Este punto va directamente relacionado con la implementación de sistemas criptográficos seguros pues si no se protege de forma adecuada cada dato sensible de la aplicación los atacantes externos o internos pueden capturar o modificar los datos con propósitos delictivos, como el fraudes o el robo de identidad.

Ausencia de control de acceso a funciones

"La mayoría de aplicaciones web verifican los derechos de acceso a nivel de función antes de hacer visible en la misma interfaz de usuario. A pesar de esto, las aplicaciones necesitan verificar el control de acceso en el servidor cuando se accede a cada función. Si las solicitudes de acceso no se verifican, los atacantes podrán realizar peticiones sin la autorización apropiada".

Falsificación de peticiones en sitios cruzados

Un ataque CSRF (Cross site request forgery) conocido también como ataque de un-clic es un tipo de fallo malicioso que se presentan en las aplicaciones web cuando ejecutan comandos no autorizados enviados por un atacante a través de las credenciales de un usuario confiable. Este fallo se explota en el navegador de la victima donde se generan pedidos que la aplicación vulnerable piensa son peticiones legítimas provenientes de un usuario autorizado.

Utilización de componentes con vulnerabilidades conocidas

"Algunos componentes tales como las librerías, los frameworks y otros módulos de software casi siempre funcionan con todos los privilegios. Si se ataca un componente vulnerable esto podría facilitar la intrusión en el servidor o una perdida seria de datos. Las aplicaciones que utilicen componentes con vulnerabilidades conocidas debilitan las defensas de la aplicación y permiten ampliar el rango de posibles ataques e impactos".

Redirecciones y reenvios no validos

"Las aplicaciones web frecuentemente redirigen y reenvían a los usuarios hacia otras páginas o sitios web, y utilizan datos no confiables para determinar la página de destino. Sin una validación apropiada, los atacantes pueden redirigir a las víctimas hacia sitios de phishing o malware, o utilizar reenvíos para acceder páginas no autorizadas".

4.3.1.2 Taxonomía con relación al Owasp Top 10

Por último se hace un paralelo entre la capa de abstracción descrita por Tsipenyuk, Chess, y McGraw con respecto a la clasificación del OWASP top 10, de esta forma se tiene una taxonomía completa que puede ser utilizada como guía en los análisis y resultados de las herramientas de análisis de código estático.

Tabla 4-2: Taxonomía con relación al OWASP Top 10 [30].

Taxonomía	OWASP Top 10 2013
1. Validación de entradas y representación.	1. Inyecciones. 3. Secuencia de comandos en sitios cruzados (xss). 8. Falsificación de peticiones en sitios

	cruzados. 9. Utilización de componentes con vulnerabilidades conocidas.
2. Abuso de las interfaces de programación (API). 3. Características de seguridad.	7. Ausencia de control de acceso a funciones. 2. Pérdida de autenicación y gestión de sesiones. 4. Referencia directa insegura a objetos.
4. Tiempo y estado.	
5. Manejos de errores.	6. Exposición de datos sensibles.
6. Calidad del código. 7. Encapsulamiento.	10. Redirecciones y reenvíos no validos. 5. Configuración de seguridad incorrecta.

5. Capítulo Evaluación

En la actualidad existen una serie de herramientas de análisis de código estático enfocadas en la seguridad de las aplicaciones. En el capitulo anterior se listan algunos programas de este tipo. Aún así el desarrollo de estas herramientas esta en crecimiento, sobre todo en las alternativas de software libre pues el nivel de madurez de los programas propietarios es relativamente alto pero sus costos son bastante elevados.

En la primera sección del capitulo se definen algunas métricas básicas con las cuales los software de análisis de código estático son seleccionados y valorados, luego de definir cuales son las métricas para evaluar las herramientas se procede a detallar la estructura y el funcionamiento general de los casos de prueba, para este trabajo se opto por el Juliet test suite, esta suite es mantenida y desarrollada por el NIST específicamente para realizar esta clase de experimentos. Por último se ejecutan las herramientas seleccionadas sobre los casos de prueba más representativos para la seguridad de las aplicaciones web en Java y se procede a exponer y describir cada uno de los resultados obtenidos en los test.

5.1 Métricas

Antes de realizar las respectivas pruebas es necesario definir cuales son las característica que se van a considerar en el momento de escoger las herramientas adecuadas para los casos de prueba. Además se debe precisar algunas métricas con las cuales se va a evaluar

cada uno de los software en su función de análisis de código estático para mejorar la seguridad de las aplicaciones web.

La primera propiedad que deben cumplir estos programas es que su estado de actividad del desarrollo debe ser dinámico, esto se mide con base en la fecha del último cambio realizado en la herramienta, pues en el campo de la seguridad del software es necesario actualizar constantemente el estado de las vulnerabilidades, de igual forma es importante que las técnicas de análisis utilizadas en estas herramientas se mejoren cada vez mas. Como segunda característica se tiene en cuenta la posibilidad de descarga e instalación, esto quiere decir que en el caso de un licenciamiento propietario el software debe permitir utilizar alguna versión de prueba u ofrecer un modelo freemium, pues en otro caso es difícil tener acceso a este tipo de herramientas propietarias por su costo; con los licenciamiento libre no existe este tipo de inconvenientes. Por último la herramienta debe ser compatible con el lenguaje de programación Java, ya que es uno de los lenguajes más difundidos en el mundo del desarrollo de aplicaciones web, a la vez es importante limitar el número de pruebas y lenguajes utilizado, esto con el fin de obtener resultados homogéneos en los test.

Además de definir las características en común de cada uno de los programas seleccionados para las pruebas es importante establecer con que tipo de métricas van a ser evaluados para obtener resultados concisos y de valor en cada prueba. Se van a utilizar alguna de las métricas expuestas en la tesis realizada por Thomas Hofer [31] en conjunto con algunas establecidas para este trabajo.

5.1.1 Reportes

Esta métrica abarca la calidad de los reportes arrojados por las herramientas. Así se mide el nivel de simpleza en la presentación de la información arrojada en cada análisis y el grado de relevancia de estos datos, todo debe ser presentado de una formar sencilla para

la comprensión rápida de los programadores. Se califica con tres valores, “sencillos”, “complejos” y “no tiene”.

5.1.2 Integración con la gestión de proyectos

La métrica determina si la herramienta es compatible con proyectos completos o no. Esta característica es relevante sobre todo para reconocer si existe interacción entre los diferentes ficheros de código, y también para valorar la usabilidad del software pues de lo contrario seria necesario analizar archivo por archivo, esto es significante si se tiene en cuenta que la mayoría de proyectos contienen cientos de archivos.

5.1.3 Capacidad para la detección de errores

Esta métrica en particular es la más importante de todas pues determina el nivel de fiabilidad de las herramientas a partir de los análisis realizados en los archivos de prueba. Para medir esta métrica se debe identificar en cada análisis los resultados de tipo falso positivo, falso negativo y positivo valido para cada vulnerabilidad localizada en el análisis.

Hay que tener presente que el enfoque del trabajo va orientado a la seguridad de las aplicaciones web por lo tanto no se tienen en cuenta los tipos de fallos fuera de este contexto, como evaluar el estilo del código o recomendar mejores prácticas en la programación, por lo que errores de esta naturaleza son valorados como nulos.

5.2 Juliet Test Suite

Como requisito para comparar cada una de las herramientas de análisis estático en función de su fiabilidad y precisión al momento de identificar vulnerabilidades en el código es necesario utilizar un objeto de prueba que contenga de forma predeterminada fallos de seguridad en su programación. Para este caso en particular algunas aplicaciones útiles pueden ser el Webgoat desarrollado por OWASP el cual tiene como lenguaje de programación JAVA Y JSP y una serie de vulnerabilidades basadas en el OWASP top 10,

otra posibilidad con las mismas características pero utilizando PHP es el Damn Vulnerable. Estas aplicaciones web se desarrollaron con el principal objetivo de enseñar la forma como se presentan las vulnerabilidades en una página web y como pueden ser explotadas, entonces se puede decir que están mas enfocadas en el aprendizaje de técnicas de pentesting que en su uso para evaluar herramientas de análisis de código, sea dinámico o estático. Lo ideal para esta serie de test es la implementación de casos de prueba que cumplan con los requisitos en documentación, corrección, codificación y calidad para probar cada herramienta de análisis de código estático frente a una vulnerabilidad en particular, por lo cual se va a utilizar un suite que cumpla con estas características conocida como "Juliet test suite".

Juliet test suite es una colección de programas en C/C++ y java con vulnerabilidades conocidas y reportadas por CWE[10] (Common Weakness Enumeration) en total 57,099 casos de prueba en C/C++ y 23,957 casos de prueba en JAVA [32]. Cada programa o caso de prueba contiene una o dos páginas de código, y la mayoría incluye pruebas similares pero discriminatorias.

Juliet cubre 181 tipos de fallos documentados por el Common Weakness Enumeration, e incluyen problemas de autenticación y accesos de control, control de buffer, inyecciones, uso de algoritmos criptográficos, control de errores, fugas de información (etc). Esta suite ofrece una serie de ejemplos con programas sencillos para cada caso, así como variaciones de la vulnerabilidad en al menos una docena de patrones diferentes. Fue desarrollado por el NIST[11] (National Institute of Standars and Technology), es de dominio publico y no esta sujeto a la protección de los derechos de autor, lo que lo hace ideal para el enfoque de este trabajo. Cada caso de prueba es sometido a revisiones con el afán de verificar que cada archivo contiene la vulnerabilidad y la variación que pretende ofrecer.

10 https://cwe.mitre.org/

11 http://samate.nist.gov/SARD/testsuite.php

Cada fragmento de código contiene unicamente un fallo; como resultado en cada caso se tiene una estructura mucho más simple que un software en producción, así es sencillo realizar un análisis veraz de cada prueba sin necesidad de establecer el número de ocurrencias o el verdadero alcance de la vulnerabilidad.

5.2.1 Estructura de las pruebas

Cada archivo de prueba contiene un formato especifico y su codificación se basa en una estructura predefinida.

Nombre

Cada conjunto de ficheros tiene una estructura en su nombre la cual se basa en los siguientes componente: El número de CWE, una pequeña descripción del fallo, una variante funcional del fallo, un número de dos dígitos continuo, un indicador opcional hacia otro archivos y la respectiva extensión, por ejemplo .c o .java. En el indicador opcional para otro archivo se refiere a que el caso de prueba esta compuesto por diferentes fragmentos de código repartidos en varios ficheros, por ejemplo code_ejemplo_54 puede contener 2 archivos más code_ejemplo_54a, code_ejemplo_54b.

Codificación

Al inicio de cada clase se identifica los comentarios que describen el nombre de la prueba, el funcionamiento básico, y las variantes del código. Luego hay otros métodos con su respectivo comentario de “bad” que es la porción de código que tiene el error, uno o más métodos con una correcta implementación y por último esta la función “main” que es utilizada para compilar de forma correcta el archivo.

Las porciones de código libres de fallo (métodos “good”) son implementados de tal forma que tengan un comportamiento similar a las porciones mal codificadas, con el fin de probar la habilidad de las herramientas para distinguir entre errores y no, de este modo se logra obtener los resultados de tipo falso positivo, falso negativo y positivo valido.

La documentación de cada caso de prueba contiene:

- Una descripción con el propósito del caso de prueba.
- El nombre del autor.
- Un método “good” (falsa alarma), “malo” (positivo), o “mixed” (algunas falsas alarmas y otras alertas positivas validas).
- Se especifica el tipo de prueba. Posiblemente los valores incluyen “Código fuente”, “Binario” o “Pseudo código”.
- Si el caso de la prueba es de tipo código fuente, se determina el lenguaje (C, C++, Java).
- Si es necesario compilar, analizar o ejecutar la prueba, las instrucciones son incluidas.
- Cualquiera que sea la clasificación de la prueba sea “good”, “bad”, o “mixed”, debe tener el ID CWE y el nombre.
- Si es un método tipo “bad” o “mixed” se especifica la linea de código vulnerable.
- Si la complejidad del código (como loops, procedimientos internos sobre el flujo de datos, suavizado del buffer) es parte del caso de prueba, el fragmento de código complejo debe ser documentado.

El código fuente debe:

- Compilarse o correr sin ningún error fatal.
- Correr sin ningún mensaje de error fatal o mensajes esperado para un programa incompleto.
- No generar ningún mensaje de tipo “warning” (a menos que la alerta sea parte de la prueba).
- Contener la documentación referente al fallo en los métodos “bad” y “mixed”.
- No contener ningún fallo si la función es del tipo “good”.

5.2.2 Estructura de la suite

Juliet suite se centra en las funciones disponibles directamente en cada lenguajes, por lo tanto no utilizar bibliotecas de terceras partes o de frameworks. Los casos de prueba hacen énfasis en las funciones neutrales de cada plataforma pero pueden contener algunas funciones especiales de Windows. Los casos de prueba en Java cubren aplicaciones de escritorio y servlets, pero no utiliza applets o el formato JSP, de igual forma no cubre aplicaciones móviles o código embebido, estos puntos son críticos para el objetivo de la tesis pero ofrece algunas pruebas de tipo inyección, algoritmos criptográficos y XSS (cross site scripting) por lo que sigue siendo útil para el experimento ya que estos tres tipos de vulnerabilidades son las más criticas en las aplicaciones web JAVA de acuerdo al OWASP [33]. Sin embargo estos casos tienen como defecto la falta de una estructura de datos compleja para evaluar por completo la calidad de las herramientas, pues muchas de las vulnerabilidades que se presentan en un código en producción son de este tipo.

A continuación se muestra una visión general de como se realiza la exploración del conjunto de pruebas de la suite Juliet, y como a partir de la estructura de sus directorios se va automatizar el análisis de cada herramienta. Esta figura fue tomada del trabajo de tesis de Christian Gotz [34] en la cual se basa el proceso de pruebas realizadas para este trabajo.

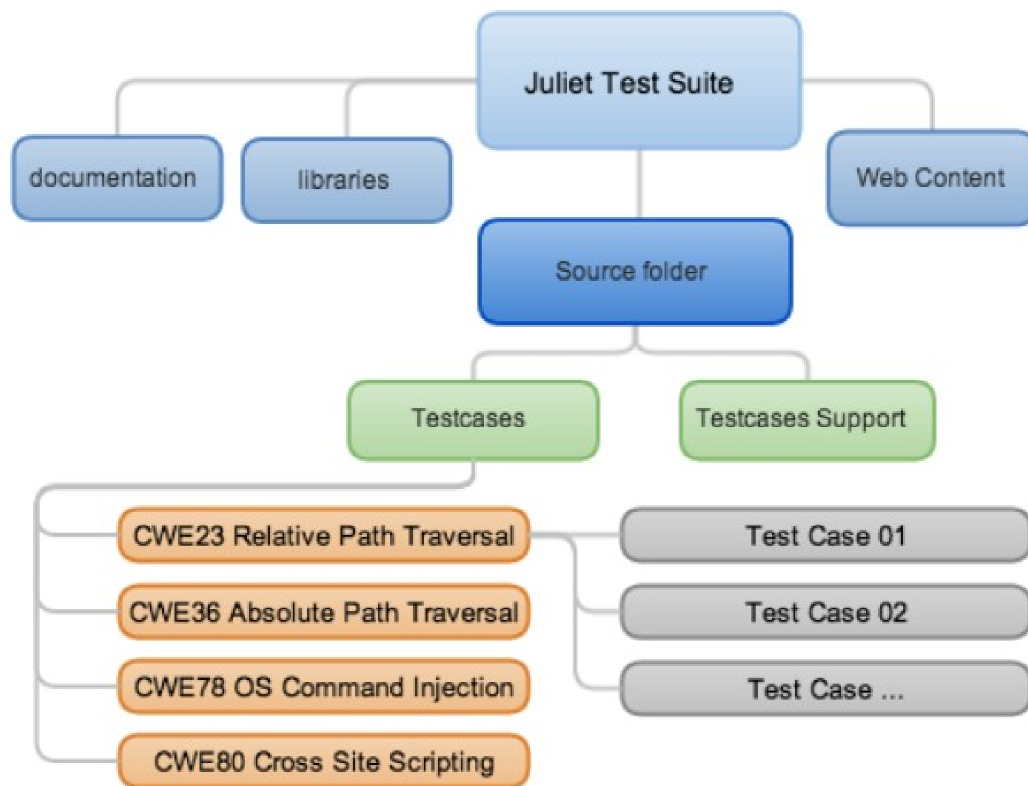

Figura 4-10: Estructura Juliet test suite [34]

Como se puede observar en la figura anterior el Juliet test suite contiene en el primer nivel de carpetas la documentación, las librerías, los códigos y el contenido web; para el alcance de este trabajo solo se va a trabajar dentro de la carpeta "testcaste" del padre "source" que esta compuesta a su vez por una serie de carpetas identificadas con el id del CWE y una pequeña descripción de la vulnerabilidad que contiene cada caso de prueba. Esta carpeta comprende un total de 41136 archivos, por lo tanto con la finalidad de sintetizar las pruebas y resultados de cada análisis se opta por 3 de las vulnerabilidad más importantes en las aplicaciones web actuales y son las inyecciones SQL, los Cross site scripting y los tipos de criptografía débil. En la siguiente sección se encuentra la descripción de los test y los resultados arrojados por cada herramienta.

5.3 Resultados casos de prueba

5.3.1 SonarQube

Sonarqube es una plataforma para evaluar el código fuente de las aplicaciones. Es software libre e implementa diversas herramientas de análisis de código estático como Findbug o PMD para JAVA, a partir de los resultados de estos componentes el software genera una serie de reportes con base en las métricas definidas. Tiene varias características para resaltar, principalmente la capacidad para no solo reportar errores en el estilo o en las prácticas de programación si no también vulnerabilidades de diversos tipos. A continuación se detalla con mayor precisión algunas de las pruebas realizadas en Sonarqube sobre el Juliet test suite.

5.3.1.1 Inyecciones SQL

La primera prueba se realiza con las vulnerabilidades de tipo inyección SQL, se puede señalar los resultados como relativamente exitosos pues Sonarqube reportó gran cantidad de estos fallos. Aún así algunos casos de prueba lograron enmascarar la vulnerabilidad y gran parte del reporte son falsos positivos, como se especifica mas atrás cada archivo del Juliet test contiene generalmente dos clases de métodos, uno con código vulnerable y otro con una mala codificación pero sin riesgo de ser explotado, muchas del segundo tipo de funciones fueron reportadas como vulnerables. A continuación están los resultados.

Tabla 5-1: Resultados inyección sql (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE89	2112	753	1550	1359

5.3.1.2 Cross Site Scripting

En este punto Sonarqube reporto un total de 63 errores de tipo “security” respecto a la vulnerabilidad “CWE81 XSS Error Message” y un total de 4896 errores sobre aspectos en el estilo y buenas prácticas de la codificación .

La gran mayoría de las amenazas se encontraban en lineas de código similares a:

```
response.sendError(404, "<br>bad() - Parameter name has value " + data);
```

El código del script es enviado directamente al cliente por medio de un error tipo 404, esto teniendo en cuenta que el valor data no es correctamente filtrado ni tampoco fueron depuradas las etiquetas de la variable.

Como se observa en la siguiente tabla se especifica que ninguna de las lineas de código vulnerables de XSS fueron reportadas por Sonarqube, en total son 542 falsos negativos, igual se especifica que el número de reportes son reales pero no del tipo cross site scripting, por lo tanto los falsos positivos son iguales a 0 y los positivos validos igual 0.

Tabla 5-2: Resultados XSS (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE81	63	0	542	0

Como segunda prueba para el tipo de vulnerabilidades XSS se uso la carpeta “CWE80_XSS” la cual contiene un total de 1088 pruebas. Igual que las anteriores se obtuvieron resultados similares, no se detecto ningún fallo de esta clase como se especifica en la siguiente tabla.

Tabla 5-3: Resultados XSS -2 (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE80	126	0	1084	0

5.3.1.3 Criptografía

El Juliet test suite ofrece diferentes tipos de vulnerabilidades relativo a la seguridad criptográfica de las aplicaciones JAVA, para validar la eficacia de la herramienta se seleccionaron principalmente tres:

- CWE256_Plaintext_Storage_of_Password: Sonarqube reporto gran cantidad de fallos validos pero no fue preciso en identificar la línea de código que contenía la amenaza real la cual es el almacenamiento de la variable password a partir de un texto sin necesidad de descifrar su información fue pasada directamente a la conexión de la base de datos.

Tabla 5-4: Resultados criptografía-1 (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE256	299	0	61	183

- CWE259_Hard_Coded_Password: En este caso se nota una gran falta para identificar y comprender la lógica del programa con fines de seguridad pues la variable data es asignada con un texto plano y posteriormente utilizada en la conexión de la base de datos, por lo tanto la herramienta en estos casos debe identificar un problema de Hard coded password, pero solo reporto errores menores.

Tabla 5-5: Resultados criptografía-2 (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE259	21	0	182	0

- CWE327_Use_Broken_Crypto: Por último se realizó un análisis sobre un caso de prueba que tiene como fallo utilizar un sistema criptográfico débil, aunque aparentemente es sencillo identificar sobre una clase Cipher si se utiliza un algoritmo débil como DES, Sonarqube parece que no lo hace y no reporta ningún tipo de error referente a la seguridad de la aplicación.

Tabla 5-6: Resultados criptografía-3 (Sonarqube).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE327	0	0	36	0

5.3.1.4 **Evaluación**

Nombre	Reportes	Integra proyectos?	Errores	Costo licencia
SonarQube	Sencillos + Completos	Si	No detecta ningún caso de XSS y es débil para identificar sistemas criptográficos vulnerables	Software Libre

5.3.2 VisualCode Grepper

VCG es una herramienta libre para análisis de código estático que identifica rápidamente código inseguro. También tiene la posibilidad de procesar archivos de configuración para todos los lenguajes lo que permite a los usuarios personalizar su escaneo con sus requerimientos y necesidades. Este analizador descompone las vulnerabilidades en 6 niveles predefinidos de seguridad y sus resultados pueden ser exportados en formato XML. Además es compatible con los lenguajes C++, C#, VB, PHP, Java y PL/SQL.

5.3.2.1 Inyecciones SQL

Los resultados del reporte en las vulnerabilidades de inyección SQL están plagados de falsos positivos y falsos negativos, superando en este aspecto a Sonarqube en mas de 3000 registro. Aún así los dos son similares en el números de positivos validos, pero la fiabilidad de Sonarqube es mayor.

Tabla 5-7: Resultados inyección sql (VGC).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE89	3808	2540	2540	1268

5.3.2.2 Cross Site Scripting

VCG en la detección de XSS superó ampliamente a Sonarqube reconociendo diferentes tipos de amenazas cross site scripting en la codificación, principalmente con la función FileInputStream de Java la cual no era correctamente sanitizada en las pruebas.

Tabla 5-8: Resultados XSS-1 (VCG).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE81	768	256	30	512

Tabla 5-9: Resultados XSS-2 (VCG).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE80	2438	1354	0	1084

5.3.2.3 Criptografía

- CWE256_Plaintext_Storage_of_Password: Las contraseñas almacenadas en texto plano no fueron reconocidas por VCG, se obtuvo un pésimo resultado con un total de 647 fallos reportados y de estos 0 fueron vulnerabilidades reales.

Tabla 5-10: Resultados criptografía-1 (VCG).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE256	647	647	61	0

- CWE259_Hard_Coded_Password: Igual que Sonarqube VCG no logro reportar ningún error valido, solo problemas menores y similares a los identificados en las pruebas realizadas sobre los casos de prueba cross site scripting.

Tabla 5-11: Resultados criptografía-2 (VCG).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE259	324	324	62	0

- CWE327_Use_Broken_Crypto: En esta última prueba se obtuvo un resultados mucho más preciso que Sonarqube, incluso logro reportar cada una de las fallas de este tipo, que constaban en reconocer el uso de un algoritmo criptográfico débil como DES.

Tabla 5-12: Resultados criptografía-3 (VCG).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE327	63	27	0	36

5.3.2.4 Evaluación

Nombre	Reportes	Integra proyectos?	Errores	Costo licencia
VCG	Complejos + Básico	No, sólo carpetas.	El método o herramientas de análisis son muy buenas pero el sistema de reportes es pésimo.	Software Libre

5.3.3 Fortify HP on Demand

Según el cuadro mágico de Gartner para las aplicaciones de análisis de código estático enfocada en la seguridad [35] existen dos lideres principales en el mercado y son el programa IBM Security AppScan y el HP Fortify evaluados por su inversión en el desarrollo e investigación de la herramienta, además del tiempo que llevan establecidos en el mercado, por lo tanto son las propuestas más maduras que existen actualmente en el mercado.

Para este trabajo se probó Fortify por el simple hecho de ofrecer una versión de prueba donde permiten realizar hasta 5 análisis por cuenta, por otra parte el software de IBM

tiene un costo promedio por licenciamiento de 20.000 USD sin la opción de una versión demo, por lo tanto se descarta su evaluación en esta tesis.

5.3.3.1 Inyección SQL

Los resultados obtenidos en las pruebas con Fortify HP on Demand se pueden establecer como precisos pero no completos, quiere decir que el gran porcentaje de errores reportados son validos pero quedan muchas vulnerabilidades por detectar, un total de 2145 fallos no fueron localizados por esta herramienta.

Tabla 5-13: Resultados inyección sql (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE89	1664	450	2154	764

5.3.3.2 Cross Site Scripting

Similar a la anterior prueba Fortify reportó un total de 182 vulnerabilidades de las cuales el 100% fueron validas, pero más del 50% de los fallos restantes no fueron detectados correctamente. Con relación a Sonarqube, Fortify es más preciso en el momento de identificar XSS, pero mucho menos efectivo que VCG.

Tabla 5-14: Resultados XSS (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE81	1664	450	2154	764

Tabla 5-15: Resultados XSS (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE80	650	0	434	650

5.3.3.3 Criptografía

- CWE256_Plaintext_Storage_of_Password: En la detección de problemas relacionados con almacenamiento de contraseñas en texto plano Fortify fue mucho más útil que las alternativas libre ya que detecto todos los errores. Si bien el nivel de falsos positivos es alto, es relativamente sencillo descartarlos pues su sistema de reportes es muy práctico.

Tabla 5-16: Resultados criptografía-1 (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE256	168	125	0	63

- CWE259_Hard_Coded_Password: Aunque mejora en este aspecto frente a Sonarqube y VCG sólo detecto el 25% de las vulnerabilidades de esta clase, pero sigue demostrando que el nivel de fiabilidad de sus reportes es muy alto.

Tabla 5-17: Resultados criptografía-2 (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE259	16	0	46	16

- CWE327_Use_Broken_Crypto: Al igual que VCG, Fortify detectó cada uno de los errores codificados en las pruebas, por lo tanto Sonarqube es el más flojo a la hora de detectar sistemas criptográficos débiles.

Tabla 5-18: Resultados criptografía-3 (Fortify HP).

Nombre	Errores reportados	Falsos positivo	Falsos negativos	Positivos validos
CWE259	16	0	46	16

5.3.3.4 Evaluación

Nombre	Reportes	Integra proyectos?	Errores	Costo licencia
Fortify HP on Demand	Sencillos + Completos	Si	Es una herramienta que genera pocos falsos positivos, por lo cual sus reportes son muy fiables, pero sus métodos de análisis de código no son mucho más completos que las alternativas libres.	3000-7000 USD

6. Conclusiones y trabajo futuro

6.1 Conclusiones

En el presente trabajo se realizó un estudio sistemático de la actualidad del análisis de código estático y, para ser más precisos, del análisis de código estático enfocado en detectar vulnerabilidades en aplicaciones web JAVA. En la primera parte se hizo una breve presentación sobre los avances y el adelanto actual de la revisión automatizada de código y su implementación en el desarrollo de proyectos. Luego se describió cada una de las técnicas utilizadas para realizar análisis estático y cuáles son los usos que se les pueden dar a estas herramientas, además se especifican las ventajas y desventajas que ofrecen al implementarse dentro del proceso de desarrollo de sistemas. Con esa base se procede a explicar las características y funcionalidades de SonarQube, por ser el software propuesto por el ANCERT como una alternativa viable para ser implementada en sus desarrollos. A partir de este punto se expone una taxonomía con las vulnerabilidades que tienen mayor importancia en las aplicaciones web, esto con el fin de evaluar esta caracterización frente algunas herramientas de código estático, que en la actualidad se enfocan en detectar vulnerabilidades web. Desde este apartado se comienza con el Capitulo 4 (Evaluación) donde se establecen algunas métricas para valorar tres programas de análisis de código estático, concretamente SonarQube, VCG y Fortify HP, los cuales cumplían con los requisitos de permitir su instalación o evaluación (si su licenciamiento es propietario), tener un estado de actividad reciente y soportar el lenguaje de programación JAVA. Por último, de la suite de pruebas Juliet se seleccionan tres fallos de seguridad importantes en

las aplicaciones web y se procede a ejecutar cada herramienta sobre los casos de prueba elegidos para describir, de esta forma, los resultados obtenidos junto con una corta evaluación de las generalidades de cada software.

Los resultados obtenidos en esta tesis demuestran que las herramientas de análisis de código estático actuales representan una poderosa alternativa para mejorar el nivel de seguridad del software. Y sugieren que todavía existe un futuro prometedor para este tipo de procedimientos automatizados que buscan una forma de generar sistemas o código seguro de un modo más simple y accesible para cualquier persona o entidad. Sin embargo, revelan lo lejos que se encuentran de ser alternativas completamente independientes; quiere decir, todavía son herramientas que ofrecen un importante soporte al revisado manual de código, más no son una opción con un nivel de automatización fiable, ya que en sus reportes no aseguran un gran porcentaje de positivos válidos.

Por otro lado, considerando la extensión y el tiempo para desarrollar este trabajo de tesis se puede decir que el número de pruebas realizado sobre el Juliet test suite es pequeño y por lo tanto algunas evaluaciones pueden variar con otros tipos de vulnerabilidades, lenguajes y casos. Aun así, fueron evaluados un total de 4809 casos de prueba diferentes sobre cada una de las herramientas, pero esta suite ofrece cerca de 40000 test adicionales que si son utilizados de forma estructurada se podría obtener resultados generales mucho mas precisos. Adicional a este problema se suma la falta de software comerciales que presenten una versión de prueba para evaluar su funcionamiento, por lo tanto en el capitulo de evaluación se vio reducido el número de herramientas que fueron sometidos a prueba.

6.2 Trabajo futuro

Con los resultados obtenidos y los inconvenientes que se identificaron a lo largo del proceso, se podría trabajar más adelante sobre los siguientes temas y desarrollos:

- Estructurar una propuesta sobre la integración de procedimientos automatizados como el análisis de código estático dentro del ciclo de vida del desarrollo de software, sobre todo en las metodologías ágiles, pues son herramientas que se pueden alinear muy bien con los principios generales de esta clase de técnicas.

- Incorporar el análisis de código estático dentro de los modelos de integración continua, puede ser sobre herramientas como Jenkins o TeamCity. Pues estos buscan, por medio de integraciones automáticas, detectar fallos lo antes posible; por lo tanto su función se acopla perfectamente con el análisis estático de código en la detección de vulnerabilidades.

- Desarrollar un complemento para SonarQube que ayude a mejorar la fiabilidad y en general la detección de vulnerabilidades en aplicaciones JAVA. Un ejemplo, aún no implementado, es la detección de algoritmos criptográficos inseguros en algunas funciones JAVA.

- Proponer algoritmos nuevos para la detección de vulnerabilidades que no solo apliquen el tipo de técnicas tradicionales y que fueron mencionadas en esta tesis, sino también utilizar técnicas de inteligencia artificial sobre los métodos ya investigados.

A. Instalación de SonarQube[12]

1. Primero se debe descargar la última versión de SonarQube (SonarQube v5.1). desde la página oficial.
2. Se descomprime el archivo .tar.gz en /home/usuario/ para tenerlo fácilmente accesible.
3. Luego se edita el archivo sonar.properties para configurar el acceso a la base de datos. El archivo se encuentra en /home/usuario/sonarqub-5.1/conf/ sonar.propiet ies/. Dentro del archivo de configuración hay que comentar la siguiente línea para no utilizar el SGBD de H2.

   #sonar.embeddedDatabase.port=9092

   Se descomenta las siguientes líneas y posteriormente se asigna los siguientes valores para indicar al servidor web de SonarQube qué base de datos se va a utilizar, los datos del usuario y contraseña de la base de datos y la información del servidor web donde se ejecutará una vez esté instalado. Por defecto se ejecuta sobre localhost con el puerto 9000.

   # DATABASE

   sonar.jdbc.username=sonarqube

   sonar.jdbc.password=sonarqube

   #----- MySQL 5.x

12 http://enrikusblog.com/sonarqube-instalacion-y-configuracion/

*sonar.jdbc.url=jdbc:mysql://localhost:3306/sonarqube?
useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true*

 # WEB SERVER

sonar.web.host=localhost

sonar.web.port=9000

4. El último paso consiste en ejecutar el servidor de SonarQube. En este caso se va a ejecutar la versión de 64 bits de Linux en la siguiente ruta:

   /home/usuario/sonarqube-5.1/bin/linux-x86-64/sonar.sh.

Bibliografía

[1] Devanbu, P. T., & Stubblebine, S. (2000, May). Software engineering for security: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 227-239). ACM.

[2] Busch, M., Koch, N., & Wirsing, M. (2014). Evaluation of Engineering Approaches in the Secure Software Development Life Cycle. En *Engineering Secure Future Internet Services and Systems* (pp. 234-265). Springer International Publishing.

[3] Cenzic, Application Vulnerability Trends Report, 2014.

[4] Viega, J., & McGraw, G. (2002). Building secure software: How to avoid security problems the right way. Boston, MA:Addison Wesley.

[5] Howard, M., & LeBlanc, D. 2002. Writing secure code. Redmond, WA: Microsoft Press.

[6] Shawn Hernan, Scott Lambert,Tomasz Ostwald y Adam Shostack, "Uncover Security Design Flaws Using The STRIDE Approach", MSDN magazine, Noviembre 2006.

[7] Programming (PLoP), 1998. P.J. Brooke and R.F. Paige, “FaultSource code analysis tools can search a program for hundreds of different security flaws at once, at a rate far greater than any human can review code. Trees for Security System Design and Analysis,” Computers and Security, vol. 22, no. 3, p. 256-264, Abril 2003.

[8] B. Blakley, C. Heath, and Members of the Open Group Security Forum, Security Design Patterns: Open Group Technical Guide, 2004.

[9] Adam Shostack, “Experiences Threat Modeling at Microsoft”, presentado en “Modeling Security Workshop”, Toulouse, 2008.

[10] Microsoft, “Security Development Lifecycle for Agile Development”, Presentación en la BlackHat, 2009.

[11] Fagan, M. E.: Design and code inspections to reduce errors in program development. IBM (1976).

[12] Steve McConnell, Code Complete: A Practical Handbook of Software Construction, 2004.

[13] D. Evans, J. Guttag, J. Horning, Y Y.M. Tan. LCLint: a tool for using specifications to chek code. En SIGSOFT Sympom Foundations of Software Engineering, Diciembre 1994.

[14] Almeida, J. B., Barbosa, M., Sousa Pinto, J., y Vieira, B. Verifying cryptographic software correctness with respect to reference implementations. In FMICS ’09: Debates en el 14th International Workshop on Formal Methods for Industrial Critical Systems (Berlin,Heidelberg, 2009), Springer-Verlag, p. 37–52.

[15] Alt, M., and Martin, F. Generation of Efficient Interprocedural Analyzers with PAG. En SAS'95, Static Analysis Symposium (Septiembre 1995), p. 33–50.

[16] S. Hallem, D. Park, and D. Engler, "Uprooting Software Defects at the Source," Queue, vol. 1, 2003, pp. 64-71.

[17] Huang, Yao-Wen, et al. "Securing web application code by static analysis and runtime protection." *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004.

[18] Kupsch, J. A., and Miller, B. P. Manual vs. automated vulnerability assessment: A case study. In The First International Workshop on Managing Insider Security Threats (2009), West Lafayette.

[19] Khedker, U. Data flow analysis. In The Compiler Design Handbook. CRC Press, 2002, p 1–59.

[20] Frances E. Allen. "Control flow analysis". *SIGPLAN Not.* 5, 7, Julio 1970, p 1-19.

[21] Lee Copeland, "A Practitioner's Guide to Software Test Design", Enero 2004

[22] Huang, Yao-Wen, et al. "Verifying web applications using bounded model checking" *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004.

[23] Artzi, Shay, et al. "Finding bugs in web applications using dynamic test generation and explicit-state model checking." *Software Engineering, IEEE Transactions on* 36.4 (2010): 474-494.

[24] Cristel Baier, Joost-Pieter Kaoten, Principles of model checking. The MIT Press Cambridge, Massachusetts London, England, 2008.

[25] B[eyer, D., Henzinger, T., and Théoduloz, G. Configurable software verification: Concretizing the convergence of model checking and program analysis. In CAV: Computer-Aided Verification, Lecture Notes in Computer Science 4590. Springer, 2007, pp. 509–523.

[26] Brain Chess and Jacob West. Secure Programming with Static Analysis. Addison-Wesley, 2007.

[27] Patroklos P. Papapetrou, G. Ann Campbell, "Sonar In Action", Published by Manning, Noviembre 2013.

[28] Tsipenyui, K., B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," presentado en Automated Software Engineering, Long Beach, CA, 2005.

[29] OWASP Top 10, Los diez riesgos más críticos en Aplicaciones web, 2013.

[30] Brain Chess and Jacob West. Secure Programming with Static Analysis. Addison-Wesley, 2007. (pp. 19-20)

[31] Thomas Hofer. "Evaluating Static Source Code Analysis Tools". MA thesis. , Ecole Polytechnique Federale de Lausanne , 2010.

[32] Center for Assured Software. Juliet Test Suite v1.1 for Java User Guide. National Security Agency. 9800 Savage Road, Fort George G. Meade, MD 20755-6577, 2011.

[33] OWASP Foundation. “OWASP TOP 10 for JavaEE: The Ten Most Critical Web Application Security Vulnerabilities For Java Enterprise Applications”. In: (2007).

[34] Christian Gotz. “Vulnerability identification in web applications through static analysis”. MA thesis. Technischen Universitat Munchen, 2013.

[35] Joseph Feiman and Neil MacDonald. Magic Quadrant for Static Application Security Testing. Tech. rep. Gartner, 2010.