ANALÍTICAS DE SEGURIDAD DE CÓDIGO FUENTE

Trabajo de fin de grado

Joaquín López-Cortijo Guijarro
NIA: 100291500

Grado en Ingeniería Informática 2016/17

uc3m | Universidad Carlos III de Madrid

Agradecimientos

Al ser este mi último trabajo de la universidad, me gustaría agradecer el apoyo recibido durante esta etapa de mi vida, ya que sin él no habría llegado hasta donde estoy ahora.

En primer lugar, a mi padre Joaquín López-Cortijo García, por haber estado ahí siempre que le he necesitado para lo bueno y para lo malo, siempre con una sonrisa, este proyecto se lo dedico a él. Te quiero mucho y te echo de menos.

A mi madre y mis dos hermanas por todo el apoyo que me han dado en mi día a día, en momentos en los que ni siquiera yo no me veía capaz de conseguir mis metas.

A mi tío y tutor de este proyecto Román, por la ayuda e implicación que me ha prestado y por ser una de las mejores personas que he conocido en mi vida.

A los compañeros de clase con los que he trabajado, muchos de ellos ahora son amigos. Gracias a ellos la vida en la universidad ha sido mucho más que estudiar, sin ellos, la universidad habría sido una etapa vacía de mi vida.

A los profesores de la UC3M que me han demostrado que les gusta enseñar y sobre todo han influido en mi forma de ver la vida.

También quiero dar las gracias a la comunidad de SonarQube por brindarme ayuda inmediata con los problemas que me surgieron durante el desarrollo de este proyecto. Mediante el grupo de Google que ofrecen para los usuarios del analizador, me respondieron las cuestiones que me impedían continuar con la implementación de las reglas personalizadas. Aquí se puede encontrar todo el hilo de preguntas y respuestas por parte del equipo de SonarQube.

Índice de contenidos

Índice de imágenes

1. Introducción

A día de hoy, la información es el bien más preciado de cualquier entidad, sea una empresa o un gobierno, ya que, utilizada correctamente permite tener un conocimiento muy valioso que puede servir para muchos propósitos. Las compañías, sean del tamaño que sean, enfocan la explotación de la información a la obtención de beneficios económicos o ventajas frente a la competencia, esto se está volviendo cada vez más frecuente y lo vemos en todo tipo de sitios: Amazon averigua las tendencias de los clientes para enviarles anuncios personalizados, recomendaciones, etc. A su vez, las agencias que trabajan para el estado utilizan la información para el espionaje de los ciudadanos y así intentan controlar cualquier actividad sin ningún tipo de filtro en todo momento.

Es por eso que la seguridad de la información se ha convertido en el campo más importante junto con el análisis de datos en los tiempos que corren. Según estadísticas, el año pasado quedaron más de 300.000 puestos de ciberseguridad sin cubrir, esto nos da una idea de la gran y creciente demanda que hay en este sector. La información se debe proteger correctamente para evitar que sea sustraída por terceras manos, es por ello que cada día se invierte más en todos los sectores que se sustentan principalmente de la información en poner medidas de protección para evitar que estos sucesos puedan ocurrir.

Cualquier brecha de seguridad es importante, hay que adoptar medidas de seguridad desde el primer momento y es por ello que este proyecto se centra en la seguridad desde el inicio del ciclo de vida de cualquier tecnología: el desarrollo. Se pueden utilizar medidas de seguridad de todo tipo: firewalls, antivirus, sistemas de detección de intrusiones, sistemas de monitorización de eventos de seguridad (más conocidos como SIEM), etc. pero si el funcionamiento de la tecnología sea la que sea no es adecuado, un atacante puede aprovecharse de ello y las medidas de seguridad adicionales quedarían fuera de juego, permitiéndole acceder, inhabilitar o causar cualquier tipo de daño que finalmente se traduciría en pérdidas económicas o de legitimidad en el mejor de los casos.

1.1 Motivación y objetivos

El motivo principal que me ha llevado a realizar este proyecto es el enfoque de mi carrera hacia el ámbito de la seguridad en las tecnologías de la información o más conocida hoy en día como ciberseguridad. Creo que este es un campo muy interesante y que está en pleno auge, ya que cada vez los ataques contra las compañías y gobiernos son más y más sofisticados. También he aprovechado los conocimientos que he adquirido en mi actual empleo como analista de seguridad, para plasmar aquí los defectos que creo que tienen la mayoría de las empresas, como por ejemplo la programación con el objetivo del funcionamiento sin tener en cuenta los riesgos y la seguridad.

El objetivo principal del proyecto es la construcción de reglas que permitan detectar fallos o vulnerabilidades en el código fuente de las aplicaciones. Poniendo el foco en el

lenguaje más utilizado a día de hoy para el desarrollo de aplicaciones web: Java; aunque también se pueden encontrar en este documento algunos ejemplos de código en PHP que es bastante conocido y mundialmente utilizado. Creo que los productos que más se ven hoy en día son las aplicaciones web y es por ello que he centrado el proyecto en la detección de fallos para este tipo de aplicaciones, aunque las reglas generadas se podrían aplicar a otros ámbitos como aplicaciones de escritorio o Android (ya que estas últimas están implementadas sobre Java).

La idea consiste en comprobar que se cumplan ciertas reglas en el código referentes a malas prácticas de seguridad que pueden llevar a que la aplicación sea utilizada de una forma distinta al propósito con el que inicialmente se creó. Se han diseñado reglas intentando cubrir el mayor rango de vulnerabilidades conocidas posibles, como son SQL Injection, Cross Site Scripting (más conocido como XSS), Cross Site Request Forgery (más conocido como CSRF) y algunas otras que se comentarán a lo largo de este documento

Para el desarrollo de las reglas se ha utilizado SonarQube, una herramienta de código abierto cuyo propósito es el análisis estático del código, es decir analizar el código fuente sin ejecutarlo, simplemente tomando como referencia los símbolos léxicos, sintácticos y semánticos de los que se vale cualquier compilador. Para lograr producir código de calidad y que cumpla todos los requisitos de seguridad, también sería necesario analizar el código de forma dinámica, es decir, observar su comportamiento en tiempo de ejecución, pero este proyecto solo se centra en el análisis estático de código. Mediante el mismo tipo de reglas que se van a detallar a lo largo del documento se podrían detectar bugs, así como malas prácticas de programación (code smells). El flujo de análisis se podría resumir en el siguiente gráfico:

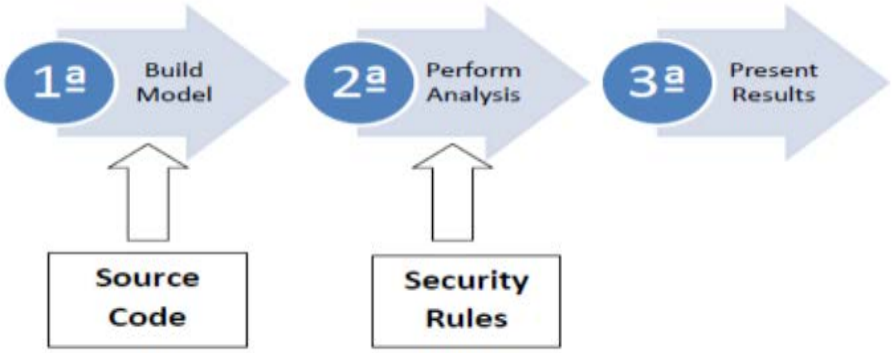

Imagen 1: Flujo de análisis SonarQube (Fuente SonarQube)

Como se puede observar, primero se construye el modelo que se obtiene del código fuente. En este proyecto, se ha creado código fuente vulnerable a propósito para que las reglas detecten los fallos deseados, se hará hincapié tanto en el fallo como en la solución o posibles soluciones. Una vez se tiene claro lo que se quiere detectar, se implementa la regla y posteriormente se analiza el código con las reglas creadas: SonarQube construirá un árbol a partir del código fuente, el cual contiene toda la información que se necesita para poder detectar los fallos. Sobre ese árbol se trabaja, analizando cada nodo (el cual puede ser un método, una variable, etc.) y poniendo las

condiciones que se decidan para que la regla salte en caso de detectar la vulnerabilidad a implementar. Después de realizar el análisis, se pueden hacer muchas más tareas mediante SonarQube como sacar informes de deuda técnica (término que hace referencia a las consecuencias de un desarrollo apresurado de software), gráficas de control de calidad, etc.

Resumiendo, ya que creo que es importante hacer hincapié en qué nos vamos a encontrar en esta documentación: Utilizando un analizador estático de código fuente llamado SonarQube, se han desarrollado una serie de reglas para detección de vulnerabilidades de seguridad en código fuente Java cubriendo los fallos más severos indicados en el OWASP Top 10 y dando una visión de cómo podrían afectar esos fallos a una empresa privada o a una institución pública, también ofreciendo una posible solución o soluciones a los mismos a aplicar en el código fuente.

2. Planteamiento del problema

2.1 Análisis del estado del arte

En esta sección, vamos a ver el entorno en el que está encuadrado el proyecto, así como ejemplos de su implementación que nos ayudarán a comprender mejor el enfoque del mismo.

2.1.1 Vulnerabilidades

En informática, una vulnerabilidad[1] es una debilidad en el software que permite a un atacante reducir la seguridad de la información de un sistema. Es la intersección entre el fallo o debilidad de un sistema, el acceso de un atacante a ese fallo y la capacidad de explotarlo.

La gestión de vulnerabilidades es el proceso cíclico de identificar, clasificar, remediar y mitigar vulnerabilidades. Esta práctica normalmente hace referencia a vulnerabilidades de seguridad en sistemas críticos, que son revisados periódicamente contrastando con las bases de datos mundiales como CVE (detalladas más abajo).

Una vulnerabilidad se puede identificar mediante cinco factores que la hacen única:

-**Producto**: es el software afectado por la vulnerabilidad. En un mismo software puede ocurrir que la vulnerabilidad solo exista en una determinada versión.

-**Dónde**: Dentro del software afectado, define en qué componente o módulo se encuentra la vulnerabilidad. Por ejemplo, una vulnerabilidad de manejo de certificados en un navegador no tiene por qué afectar al módulo que procesa el código HTML.

-**Causa y consecuencia**: define cuál es el origen del problema, cuál fue el fallo del programador que construyó ese código. Por ejemplo, puede que el programador no comprobase bien los valores de retorno de una función en concreto.

-**Impacto**: medida de los daños que podría causar un atacante si explotase la vulnerabilidad, normalmente se clasifican por severidad dependiendo de las características (más adelante entraremos en detalle de los tipos de vulnerabilidades según su impacto).

-**Vector**: Es la forma en la que un atacante puede explotar la vulnerabilidad, la forma más común es el envío de información manipulada a un determinado puerto del sistema esperando que el sistema devuelva una respuesta no esperada y que pueda ser aprovechada para fines distintos a los que realmente tiene.

[1] *"The Three Tenets of Cyber Security" U.S. Air Force Software Protection Initiative*

En los últimos años, el número de vulnerabilidades Zero-Day (vulnerabilidades desconocidas tanto para el fabricante como para el usuario del producto) que intentan ser explotadas crece enormemente, como se puede observar en la siguiente gráfica:

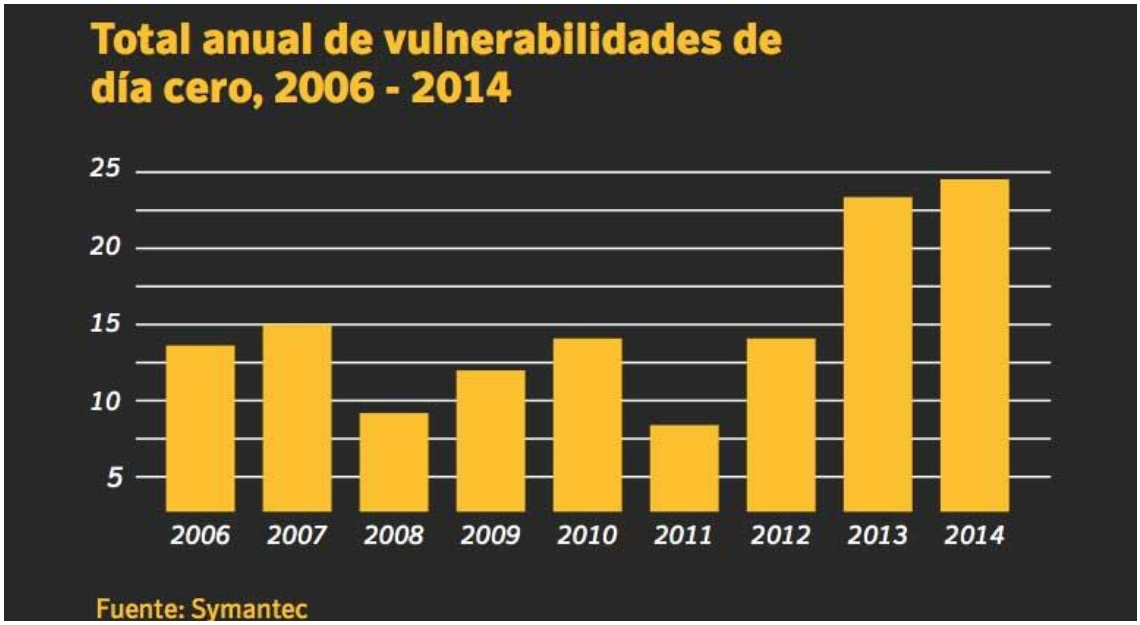

Imagen 2: Vulnerabilidades Zero-Day 2006-2014 (Fuente Symantec)

Es por ello que las compañías cada vez están dedicando más esfuerzos y recursos a mitigar esta situación desde el inicio de vida del software. Es una realidad que cuanto más tiempo se tarda en arreglar un fallo, más caro resulta.

Vamos a poner un ejemplo de vulnerabilidad, la conocida como “Directory traversal”. Esta vulnerabilidad se aprovecha de alguna debilidad que le permita acceder a archivos del sistema enviando información modificada al servidor. Estos archivos inicialmente no se pretenden mostrar al público, pero por una mala configuración del software, o por no poner las suficientes medidas de seguridad, son accesibles remotamente por un atacante. Esta vulnerabilidad puede resultar en filtraciones de información sensible como contraseñas, datos de clientes, etc. La pesadilla de cualquier empresa.

La siguiente imagen es un claro ejemplo de código vulnerable, en el cual se muestra un template (vista de una página web) por defecto si la Cookie enviada no tiene el parámetro “template” fijado. Como se puede observar, no hay ningún tipo de comprobación en la variable “template”, por tanto si en la Cookie enviada al servidor introducimos una secuencia de caracteres de tipo “../../../” el servidor intentará buscar en esa ruta y podremos navegar por las carpetas a las que el usuario que ejecuta el servidor web tenga acceso y de la misma forma acceder a los archivos a los que tenga acceso.

```
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
   $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/phpguru/templates/" . $template );
?>
```

Imagen 3: Ejemplo de código fuente vulnerable a Directory Traversal (Fuente "Conviction for using directory traversal")

Si enviamos la siguiente petición al servidor web, procesará la cadena de caracteres manipulada y devolverá el resultado de la imagen número seis, mostrando el fichero deseado:

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../etc/passwd
```

Imagen 4: Petición web manipulada (Fuente "Conviction for using directory traversal")

En el ejemplo se utiliza un sistema UNIX y por ello se va a acceder al fichero **/etc/passwd** el cual contiene información de los usuarios del sistema (antiguamente también contenía las contraseñas, en sistemas más modernos se almacenan en **/etc/shadow**). En este caso es una Cookie la que controla el acceso a archivos no permitidos, pero podría ser otro parámetro el manipulado, como por ejemplo una URL.

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OIf31.:182:100:Developer:/home/users/phpguru/:/bin/csh
```

Imagen 5: Respuesta del servidor web (Fuente "Conviction for using directory traversal")

Como se puede observar, el servidor devuelve el contenido del fichero /etc/passwd, mostrando los usuarios que existen en el sistema y otros datos como cuándo caduca la contraseña, qué tipo de Shell utilizan, etc… Con esta información un atacante podría pasar a la siguiente fase: fuerza bruta, ingeniería social…

Una solución posible a esta vulnerabilidad, sería comprobar que en la variable "template" no se puedan introducir secuencias de caracteres de ese tipo, y que la plantilla a la que se quiere acceder tenga la extensión "php". De esta manera, solo se podrá acceder a los archivos de tipo "php" contenidos en la carpeta "/home/users/phpguru/templates". Otro problema que tiene este código es de permisos, el servidor se está ejecutando con demasiados permisos y por tanto se puede acceder al fichero /etc/passwd. Si el usuario que corre el servidor tuviera los

permisos correctamente configurados, solo se debería tener acceso a los mínimos recursos necesarios, que en este caso sería la carpeta donde están alojadas las plantillas a mostrar.

2.1.2 Tipos de vulnerabilidades

Es imposible estudiar las vulnerabilidades durante mucho tiempo sin empezar a elegir patrones y relaciones entre los diferentes tipos de fallos que cometen los programadores. Desde un alto nivel, los defectos de los programas se dividen en dos grandes categorías: genéricas y específicas de un contexto.
Un defecto genérico es un fallo que puede ocurrir en casi cualquier programa escrito en un lenguaje dado. El buffer overflow es un excelente ejemplo de defecto genérico en programas escritos en C y C++. Representa un problema de seguridad en casi cualquier contexto, y muchas de las funciones y porciones de código que pueden llevar a un buffer overflow son las mismas, sin importar el propósito del programa.

Encontrar fallos específicos, requiere conocimientos específicos sobre la semántica del programa a analizar. Si imaginamos un programa que tiene que manejar números de tarjetas de crédito, para cumplir con PCI-DPS (Payment Card Industry Data Protection Standard) nunca debe mostrar un número de tarjeta completo a un usuario. Como no hay funciones estándar o estructuras de datos para manejar y presentar datos de tarjetas de crédito, encontrar un fallo en un programa de estas características requiere una comprensión del programa y las estructuras contenidas en el mismo.

Adicionalmente a la cantidad de contexto necesaria para identificar un fallo, muchos de ellos pueden ser encontrados únicamente en una representación particular del programa. Problemas de alto nivel son normalmente visibles solo en el diseño del programa, mientras que errores de implementación como la omisión de validación de entradas pueden ser encontrados normalmente examinando el código fuente del programa. Los lenguajes orientados a objetos como Java tienen librerías muy extensas y ampliamente documentadas, que hacen posible entender mejor el código simplemente leyéndolo. Las clases derivadas de una librería estándar llevan mucha semántica incluida en ellas, pero nunca es tarea fácil realizar ingeniería inversa para obtener el diseño a partir de la implementación.

Los defectos o fallos de seguridad comparten tantas características y patrones entre ellos que es razonable definir una nomenclatura para describirlos. Los investigadores han estado creando sistemas de clasificación de defectos de seguridad desde los años 70, pero los métodos antiguos normalmente fallan en representar las relaciones que vemos hoy en día. En los últimos años se ha visto renovado el interés en esta área. En el proyecto “Common Weakness Enumeration” (CWE) se está construyendo una lista formal y un esquema de clasificación para fallos de seguridad. El proyecto “OWASP Honeycomb” es una comunidad de investigadores creada para definir términos y relaciones entre principios de seguridad, amenazas, ataques, vulnerabilidades y contramedidas. Sin embargo, una organización que nos proporcione suficiente vocabulario para hablar a los programadores acerca de los fallos que normalmente llevan a problemas de seguridad, sería suficiente.

Los siete reinos perniciosos[2]

Es una taxonomía creada por Tsipenyuk, Chess y McGraw en 2005. El término reino es utilizado por los biólogos en su taxonomía de los organismos vivos para indicar una agrupación de miembros similares. Estos son los siguientes:

1. Validación y representación de entradas: problemas causados por meta caracteres, codificaciones alternativas y representaciones numéricas. Son causados por la confianza depositada en las entradas de los usuarios. Entre estos problemas encontramos los buffer overflows, XSS, SQL Injection y muchos otros.

2. Abuso de APIs: Una API es un contrato entre dos elementos, el llamante y el llamado. La forma más común de abuso de APIs es que el llamante no cumpla el final de este contrato. Este contrato también lo puede violar el llamado, por ejemplo, si una clase Java extiende java.util.Random, y devuelve valores que no son random, este contrato es violado.

3. Características de seguridad: Aquí entran problemas relacionados con la implementación de autenticación, control de acceso, confidencialidad, criptografía y gestión de privilegios. Escribir una contraseña de acceso a una base de datos en el código fuente es un ejemplo de este tipo de fallos.

4. Tiempo y estado: A los programadores piensan en el código como si fuera ejecutado ordenadamente, ininterrumpidamente y linealmente. Los sistemas operativos multitarea corriendo sobre procesadores multi-core no juegan con estas reglas. Este tipo de defectos son causa de interacciones inesperadas entre hilos, procesos, tiempo y datos. Estas interacciones ocurren a través de un estado compartido: semáforos, variables, sistemas de ficheros.

5. Manejo de errores: Incluye defectos de pobre manejo de excepciones y errores, o de ni siquiera manejarlos en absoluto. También se incluyen los defectos donde los errores producidos devuelven demasiada información que puede ser utilizada para comprometer la seguridad de la aplicación.

6. Calidad de código: Una mala calidad del código puede llevar a un comportamiento impredecible de la aplicación. Esto normalmente se manifiesta como una usabilidad muy pobre. Para un atacante es una oportunidad para estresar el sistema de maneras inesperadas.

7. Encapsulación: consiste en fijar barreras o fronteras entre distintos elementos. Por ejemplo, en un servidor, hay que diferenciar entre los datos validados y los no validados, datos que unos usuarios pueden ver y otros no, etc.

[2] Brian Chess, Jacob West - Secure Programming with Static Analysis

La mayoría de los fallos de seguridad de los sistemas se clasifican en un pequeño conjunto de categorías:

- **Buffer overflows**: ocurre cuando una aplicación intenta escribir datos después del final (en ocasiones antes del principio) de un buffer. Éstos pueden comprometer los datos, proveer un vector de ataque para escalar privilegios e incluso producir un "crash" de la aplicación. Los libros de seguridad del software los mencionan como la mayor fuente de vulnerabilidades. No se conocen cifras exactas, pero aproximadamente el veinte por ciento de los exploits publicados reportados por US-CERT (United States Computer Emergency Readiness Team) involucran buffer overflows. Excepto en casos especiales, los datos de entrada del usuario se almacenan en la pila o heap asociado al proceso, este tipo de fallos comprometen la pila, el heap o ambos.

- **Validación incorrecta de entradas**: Como regla general, todas las entradas deben ser validadas para asegurar que los datos son razonables. Cualquier entrada no validada de una fuente no confiable es un potencial ataque a una aplicación (en este contexto un usuario ordinario es una fuente no confiable). Ejemplos de entradas de una fuente no confiable serían: cadenas de texto, comandos a través de una URL, audio, video, gráficos, lecturas de datos de un servidor a través de la red, etc. Los atacantes buscan cualquier entrada no validada correctamente como puerta de entrada a los sistemas, este es uno de los métodos más comunes de explotación. Primero interactúan con las entradas y cuando encuentran una que cause un "crash" o un comportamiento diferente de la aplicación, pasan a la siguiente fase que sería encontrar una manera de explotar el fallo. Entradas invalidadas han sido utilizadas para tomar control de sistemas operativos, robar datos, corromper discos duros, etc. Incluso se utilizó esta técnica para llevar a cabo el primer "JailBreak" de un iPhone.

- **Condiciones de carrera**: Ocurren cuando cambios en el orden de eventos cambian el comportamiento de la aplicación. Si un orden concreto de ejecución de los eventos es requerido por el programa esto se considera un fallo. Si un atacante puede aprovecharse de esto para insertar código malicioso, cambiar el nombre de un fichero, o de cualquier manera interferir con el funcionamiento normal del programa, la condición de carrera se convierte en vulnerabilidad. Los atacantes a veces pueden aprovecharse de pequeños lapsos de tiempo en el procesamiento del programa para interferir en la secuencia de operaciones, que luego pueden explotar.

- **Problemas en el control de acceso**: el control de acceso es el proceso de supervisar a quién le está permitido hacer una determinada operación, o poder ver o manipular ciertos recursos. Desde el control del acceso físico a un sistema (manteniendo los servidores en una nave protegida, por ejemplo) a la especificación de quién tiene acceso a un recurso y qué operaciones puede realizar sobre ese recurso. Algunos mecanismos de control acceso son forzados por el sistema operativo como por ejemplo las operaciones sobre ficheros.

Muchas de las vulnerabilidades de seguridad aparecen a causa del descuido o mal uso del control de acceso, o por ni siquiera utilizarlo en absoluto. Muchas de las discusiones en la literatura de la seguridad informática hacen referencia a los privilegios, es de particular interés para los atacantes obtener privilegios elevados en los sistemas (en sistemas UNIX/Linux lo ideal para ellos sería obtener una sesión con el usuario "root") que consiste en tener permisos ilimitados para realizar cualquier operación sobre el sistema sin restricción de ningún tipo.

- **Debilidades en la implementación de autenticación, autorización o de algoritmos criptográficos**: Los métodos de autenticación, autorización y cifrado proveen una capa de seguridad adicional a los sistemas y aplicaciones si son utilizados correctamente, pero una mala implementación de estos mecanismos puede llevar a agujeros de seguridad enormes. Imaginemos un esquema empresarial en el que se utiliza LDAP (directorio activo) para la autenticación de los usuarios en la red WiFi empresarial. Si un atacante consigue suplantar un punto de acceso WiFi e interceptar las credenciales de un usuario, tendrá acceso no sólo a la red empresarial, sino a todos los recursos a los que ese usuario tenga acceso dentro de la misma, por ello es muy importante tener en cuenta la granularidad y no utilizar las mismas credenciales para elementos completamente distintos de una red.

Según la criticidad, se pueden distinguir cuatro categorías de vulnerabilidades:

- **Crítica**: Afecta a la disponibilidad, integridad o confidencialidad de los sistemas de información. Requiere atención inmediata por parte de los administradores IT para su resolución. La no resolución de la misma podría causar graves pérdidas económicas o daños en la imagen comercial de la marca.

- **Importante**: Puede afectar a la disponibilidad, integridad o confidencialidad de los sistemas o datos de usuario. Si se llegan a explotar, afectaría de forma drástica al igual que las críticas.

- **Moderada**: El impacto podría reducirse aplicando ciertas configuraciones o mediante auditorías, etc. Es difícil de explotar y requiere conocimientos elevados para poder aprovecharse de ella.

- **Baja**: El impacto si se explotara sería mínimo, requiere de conocimientos elevados ya que son las más difíciles de explotar. Esto no quiere decir que no puedan suponer peligro para los sistemas.

Esta clasificación en función del impacto es muy importante para realizar el denominado "triage", que consiste en hacer una priorización de las vulnerabilidades a resolver. Mediante esta priorización se ahorra tiempo de resolución, y se enfoca esta tarea hacia las vulnerabilidades que realmente son importantes, eliminando falsos positivos previamente.

2.1.3 Análisis estático de código fuente

El análisis estático de código fuente es más usado de lo que la mayoría de la gente piensa, también es debido a que hay muchos tipos de herramientas de análisis estático, cada una con diferentes objetivos y ámbitos. En esta sección, se van a ver las diferentes categorías de herramientas de análisis estático.

- **Comprobación de tipos:** Es la forma más conocida de análisis estático, todos los programadores están familiarizados con ella, aunque no le den mucha importancia, ya que los lenguajes de hoy en día están fuertemente tipados. La comprobación de tipos elimina categorías enteras de errores de programación. Simplemente se comprueba que los tipos son compatibles en la asignación de valores, inicialización de variables, definición de funciones, etc.

- **Comprobación de estilos:** Poseen reglas relacionadas con espacios en blanco, nombres, funciones obsoletas, comentarios y estructura del programa. Como muchos programadores están anclados a su propio estilo de programación la mayoría de estas reglas son muy flexibles y permiten definir un estilo propio en el código.

- **Entendimiento del programa:** Estas herramientas ayudan a los usuarios a entender código fuente muy largo. Un ejemplo simple de funcionalidad de estas herramientas sería encontrar el número de veces y el lugar donde una función es utilizada a lo largo del código. Los IDEs siempre incluyen funciones de este tipo, además pueden contener algunas más avanzadas como la representación del flujo del programa en diagramas, etc.

- **Verificación del programa y comprobación de propiedades:** Estas herramientas toman como entrada unas especificaciones y código fuente. Se encargan simplemente de comprobar que el código se ajusta completamente a las especificaciones. Este tipo de comprobación no es usual en los proyectos de desarrollo de software debido a la complicación de su implementación continua.

- **Análisis de bugs:** el propósito de este tipo de herramientas es indicar porciones del código donde el programa podría comportarse de forma inesperada para el programador. Findbugs es un buen ejemplo de este tipo de herramientas.

- **Revisión de seguridad:** Este tipo de análisis utiliza (entre otras) las herramientas mencionadas en los puntos anteriores, aunque más enfocadas a identificar problemas de seguridad, estas técnicas anteriores son aplicadas de forma diferente. Las primeras herramientas de este tipo (RATS, Flawfinder, ITS4, etc.) simplemente identificaban cosas muy simples como llamadas a funciones, en este sentido eran casi parecidas a los comprobadores de estilo. Con el paso del tiempo estas herramientas han ido evolucionando, y son conocidas por generar una gran cantidad de falsos positivos, ya que normalmente la gente piensa que devuelven una lista de fallos a partir del

código, y esto es un error. Las herramientas modernas son una mezcla entre analizadores de bugs y analizadores de propiedades. Como se ha comentado, aunque los analizadores de bugs son muy útiles, no se puede hacer caso omiso de la cantidad de falsos positivos que pueden llegar a generar. Esto significa que se requiere de una revisión humana para verificar que los fallos indicados son realmente fallos. Este proyecto está centrado en la revisión de seguridad continua de código fuente.

Normalmente este tipo de análisis se realiza como parte de la revisión de código, también conocido como “test de caja blanca”. Un test de caja blanca es una prueba de seguridad donde se conoce la infraestructura donde se trabaja y todos los activos que la componen y se lleva a cabo en la fase de implementación de un SDL (Security Development Lifecycle).

Usualmente este análisis estático se refiere a la ejecución de herramientas automatizadas con el fin de resaltar posibles vulnerabilidades en código estático (código que no se está ejecutando) usando las técnicas descritas a continuación. Estas técnicas se combinan para ofrecer un único analizador de vulnerabilidades y son derivadas de las técnicas que se utilizan en la compilación de código.

Análisis del flujo de datos

Se usa para recolectar información utilizada en tiempo de ejecución mientras el código permanece en un estado estático. Hay tres términos comunes usados en el análisis de flujo de datos: bloque básico (el código), análisis control de flujo y análisis control del camino que toman los datos. Un bloque básico es una secuencia de instrucciones que no puede ser alterada si se empieza a ejecutar, es decir, si se empieza por la primera instrucción, la última instrucción ejecutada será la del fin de ese bloque.

Grafo de control de flujo

Representación abstracta en forma de grafo donde cada nodo representa un bloque básico. Los arcos entre los nodos representan el salto de un bloque básico a otro. En la siguiente imagen se puede ver un ejemplo de este tipo de grafos de control de flujo:

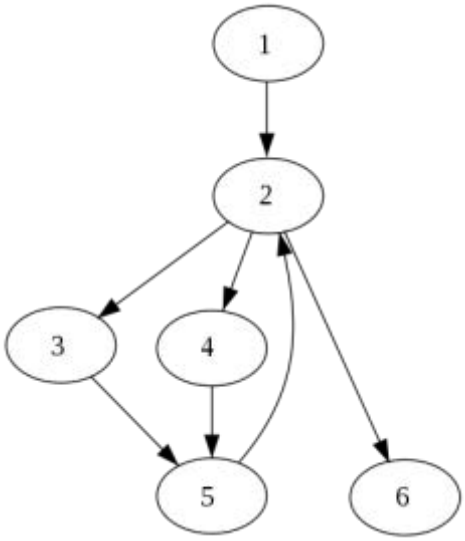

Imagen 6: Grafo de control de flujo (Fuente WikiVisually)

En este grafo tenemos el punto de entrada que sería el nodo número uno, y el nodo de salida que sería el número seis. Los saltos entre nodos representan bucles y condiciones.

Taint

Intenta identificar variables que contienen entradas del usuario y las rastrea hasta encontrar posibles funciones vulnerables que las usen. Si la variable se pasa como parámetro a la función sin ser previamente comprobada esto se marcará como una vulnerabilidad. Algunos lenguajes de programación como Ruby tienen en análisis Taint activado por defecto en determinadas situaciones como datos de entrada pasados vía CGI (Common Gateway Interface).

Análisis léxico

El análisis léxico es una de las principales técnicas que utilizan los compiladores para poder generar instrucciones máquina. Esta técnica convierte el código fuente en "tokens" con el fin de abstraer los conceptos y hacerlos más sencillos de manipular. Pongamos un ejemplo de análisis léxico, imaginemos que tenemos un código con dos variables y una función. Un analizador léxico muy sencillo devolvería tres tokens: VAR, VAR y FUNC. Simplemente hemos identificado que hay ciertos elementos en el programa, pero no se sabe qué hacen. Ésta es precisamente la función del analizador léxico, identificar los elementos mínimos que componen el código para posteriormente poder tratarlos y convertirlos en instrucciones que el procesador puede entender.

2.1.4 Falsos positivos VS Falsos negativos

Un **falso positivo** es vulnerabilidad que ha sido detectada por cualquier mecanismo pero que realmente no existe mientras que un **falso negativo** es vulnerabilidad no detectada que realmente sí existe. ¿Qué es más relevante?

Esta pregunta parece muy clara, pero el caso es que los dos son relevantes. Es cierto que un falso negativo podría parecer mucho más importante pero no es realmente así. Un falso positivo no detectado puede llevar a trabajo inservible, es decir horas de empleados tiradas a la basura, ya que intentar resolver una vulnerabilidad que no existe, es inútil. Por ello es muy importante que los administradores de activos de información tengan clara la tecnología que están utilizando, para poder detectar a la mayor brevedad posible este tipo de incongruencias. Por ejemplo, si se ha detectado una vulnerabilidad que afecta a una determinada versión de Apache en un servidor que no corre tecnología Apache, lo más probable es que sea un falso positivo y hay que descartarlo cuanto antes para evitar trabajo adicional que se traduce en pérdidas económicas para una compañía.

Un falso negativo podría tener un impacto mucho mayor, ya que permanece inadvertido a los administradores de seguridad encargados de revisar las vulnerabilidades o malas configuraciones de los activos, y un atacante podría aprovechar ese desconocimiento para robar información o causar cualquier tipo de daño. No por ello es más relevante que un falso positivo.

La clave es un balance lo más nivelado posible entre falsos positivos y falsos negativos porque en la vida real es imposible reducir estos porcentajes al cero por ciento. Como se ha comentado en la introducción este es uno de los problemas más importantes a tener en cuenta a la hora de analizar vulnerabilidades y es una de las mayores dificultades de este proyecto. Hay dos posturas a tomar, la primera consiste en restringir al máximo, es decir tener en cuenta cualquier vulnerabilidad y generar una alerta para cada una, y entonces se generarán muchos falsos positivos. La otra postura es más laxa y consiste en restringir menos, es decir generar menos alertas, pero en este caso tendremos muchos más falsos negativos y por tanto muchísimo riesgo de que nuestros activos sean atacados satisfactoriamente. En este proyecto se han propuesto reglas para una postura intermedia, con el objetivo de obtener un balance adecuado entre falsos positivos y falsos negativos.

2.1.5 Herramientas de análisis de seguridad de código Fuente

Las herramientas de análisis de seguridad código fuente, conocidas como SAST (Static Application Security Testing) se encargan de chequear código fuente o código compilado para ayudar a encontrar fallos de seguridad. Lo ideal sería que estas herramientas encontraran problemas con tal nivel de confianza que efectivamente cada fallo que encuentren sea realmente un fallo. Esto no ocurre así en la mayoría de los casos debido a la complejidad de los fallos de seguridad y del entendimiento del diseñador del software, por lo que estas herramientas sirven como apoyo a los analistas para encontrar porciones de código relevantes y para que puedan identificar los fallos de una manera mucho más eficiente que observando el código manualmente cada vez que se modifica.

Algunas de estas herramientas están empezando a acoplarse directamente en los entornos integrados de desarrollo o más conocidos como IDEs (Eclipse, Netbeans, Microsoft Visual Studio, etc.) debido a la eficiencia a la hora de analizar proyectos. Cada vez que se inserta código en el proyecto, éste es analizado y se resaltarán las alertas activadas en los fallos identificados por el analizador. Para el tipo de problemas que pueden ser detectados durante la fase de desarrollo en sí misma, es importante hacer uso de estas herramientas para obtener resultados inmediatos acerca de errores o fallos que el programador ni siquiera ha tenido en cuenta. Normalmente los programadores están tan centrados en obtener la funcionalidad requerida que muchas veces se olvidan de pequeños problemas que parecen no tener importancia, pero realmente se pueden convertir en asuntos muy graves a largo plazo. Ésta es una de las razones por las que se utilizan este tipo de herramientas en la fase de desarrollo, para no cargar a los programadores con tantos parámetros de configuración. Los programadores se encargarán de la funcionalidad y de los asuntos de seguridad que se les pida en los requisitos, mientras que los fallos que puedan cometer durante el desarrollo serán identificados en su mayoría por los analizadores estáticos si están correctamente configurados y tienen las reglas de activación adaptadas al proyecto en concreto.

El resultado inmediato que proporcionan estas herramientas cuando están correctamente integradas en todo el ciclo desarrollo es muy útil, especialmente cuando lo comparamos a encontrar vulnerabilidades graves a posteriori en el software que está en producción. Es una realidad que los expertos de seguridad y los desarrolladores de software hablan lenguajes completamente diferentes, estas herramientas proveen un punto de encuentro entre ambos sectores en el cual ambas partes se sienten cómodas.

Al igual que cualquier tecnología, este tipo de herramientas tienen sus ventajas e inconvenientes cuando son utilizadas en el desarrollo de un proyecto. Las principales ventajas son:

- **Feedback para desarrolladores:** ofrecen información precisa acerca de los errores en el código como el fichero concreto en el que se encuentra el error, número de línea, nombre de método, etc. También se puede encontrar otro

tipo de información en las alertas como posibles soluciones, importancia del error, etc.

- **Escalabilidad:** pueden ejecutarse sobre mucho software repetidamente, es perfecto para proyectos de integración continua donde en el proyecto se inserta código reiteradamente en periodos de tiempo cortos.

- **Utilidad para descubrimientos con alto nivel de confianza:** hay cierto tipo de errores cuya detección tiene una probabilidad muy alta de no ser un falso positivo. Entre estos errores se encuentran los buffer overflows y los fallos de SQL Injection.

Los inconvenientes que puede tener el uso de estas herramientas se pueden resumir en los siguientes:

- **Dificultad de identificar vulnerabilidades:** es muy complicado encontrar cierto tipo de vulnerabilidades como problemas de autenticación, uso incorrecto de algoritmos criptográficos, etc. Por ello el actual estado del arte solo permite a las herramientas encontrar un pequeño porcentaje de fallos de seguridad, aunque se está mejorando cada vez más en la detección de este tipo de fallos en el código.

- **Cantidad elevada de falsos positivos:** Como se ha comentado antes este es un punto muy a tener en cuenta ya que hay que conseguir un balance entre falsos positivos y negativos.

- **Dificultad de encontrar errores de configuración:** Las configuraciones normalmente se aplican de forma externa al código, ya sea en ficheros XML o de cualquier otro tipo. Por este motivo es difícil para los analizadores encontrar estos errores, ya que solo analizan el código.

- **Dificultad para analizar código que no puede ser compilado:** los analistas de código frecuentemente son incapaces de compilarlo porque no tienen las librerías adecuadas, instrucciones de compilación, etc.

Es muy importante aplicar un criterio de selección a la hora de elegir utilizar o prescindir de estas herramientas, ya que el uso inadecuado de las mismas puede incluso empeorar el rendimiento del proyecto. Hay una serie de factores a tener en cuenta a la hora de elegir:

- **Tiene que soportar los lenguajes que se requieran en el proyecto**
- **Tipo de vulnerabilidades que la herramienta puede detectar**
- **Cómo es de preciso según las tasas de falsos positivos/negativos**
- **Si requiere un despliegue de software o no**
- **Cuál es la curva de aprendizaje**
- **Si puede ejecutarse de forma automática continuamente**
- **Coste de la licencia**

- **Si es compatible o no con las librerías y frameworks utilizados**

Hay una gran cantidad de herramientas SAST en el mercado, muchas de ellas distribuidas bajo licencia comercial ya que las empresas son muy reacias a utilizar herramientas de código libre, básicamente porque estas herramientas no suelen ofrecer un soporte aceptable. En este proyecto se ha decidido utilizar una herramienta de código libre debido a la documentación tan completa que ofrece y a que la comunidad de usuarios está realmente sensibilizada con el proyecto. Algunas de las soluciones que hay disponibles en el mercado son las siguientes:

Veracode[3]

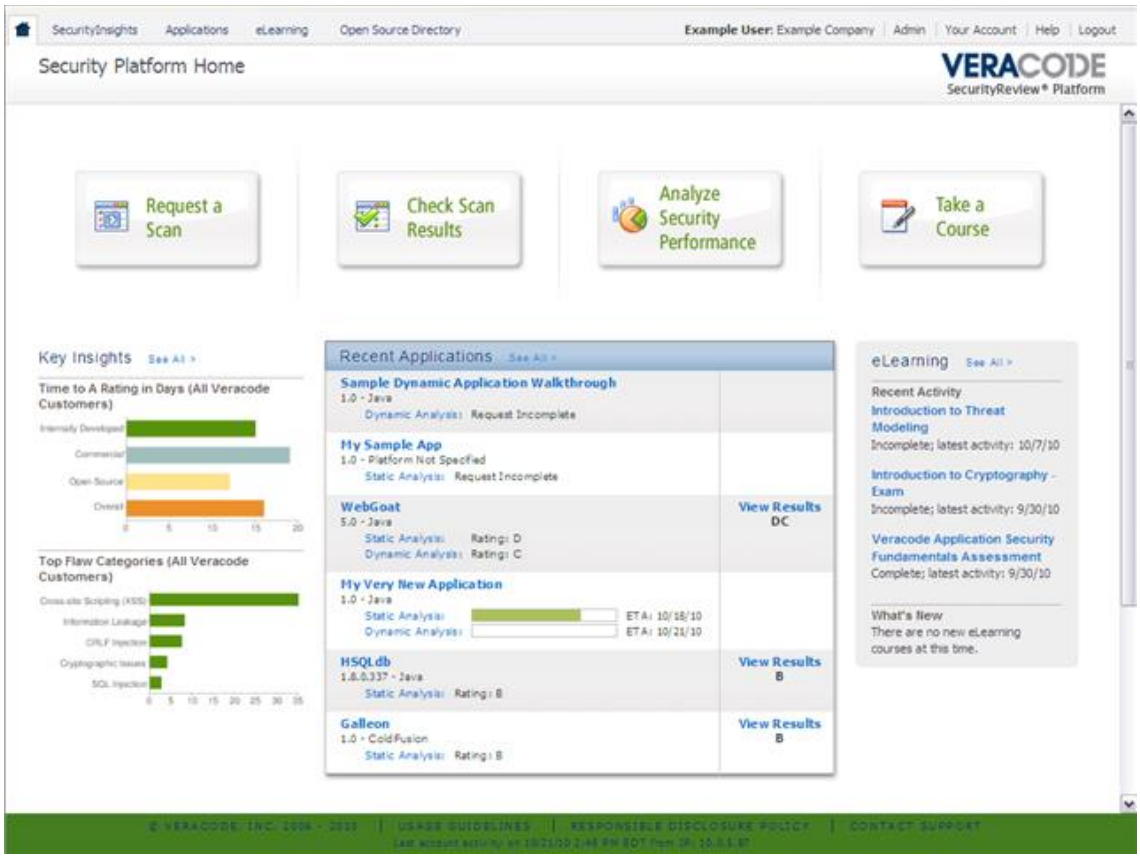

Imagen 7: Interfaz de Veracode (Fuente Veracode)

Veracode es una compañía de seguridad de Massachusetts, fundada en 2006. Esta compañía proporciona servicios de seguridad basados en la nube. Es una de las compañías pioneras en este tipo de tecnología y su solución SAST es una de las más utilizadas a nivel mundial junto con HP Fortify. Es muy intuitiva, aunque ciertas funcionalidades complejas pueden requerir una mayor curva de aprendizaje para ser utilizadas correctamente. Permite la elaboración de reportes de calidad de código en función de la seguridad, aunque también permite hacerlos basándose en otro tipo de métricas de auditoría de código.

[3] https://www.veracode.com/products/binary-static-analysis-sast

BugScout

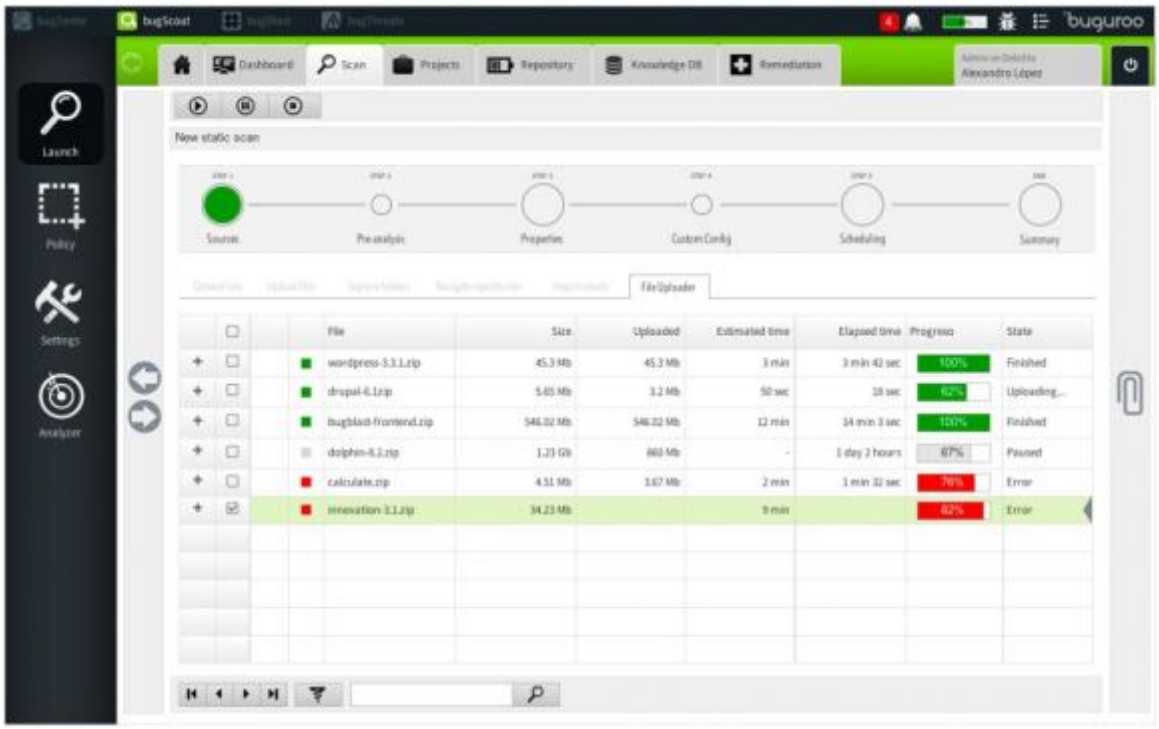

Imagen 8: Interfaz de BugScout (Fuente Buguroo)

Esta herramienta proporciona un motor propio de análisis de código, implementado por un equipo experto de prestigio internacionalmente reconocido. Tiene una gran eficiencia, llega a alcanzar tasas de varios millones de líneas de código en pocos minutos. Es compatible con múltiples lenguajes e incorpora Sonar dentro de la misma, la herramienta clave de este proyecto. Está enfocada en los estándares de seguridad más importantes: CWE, CVSS y OWASP. También permite la posibilidad de elaborar informes a medida.

Kiuwan

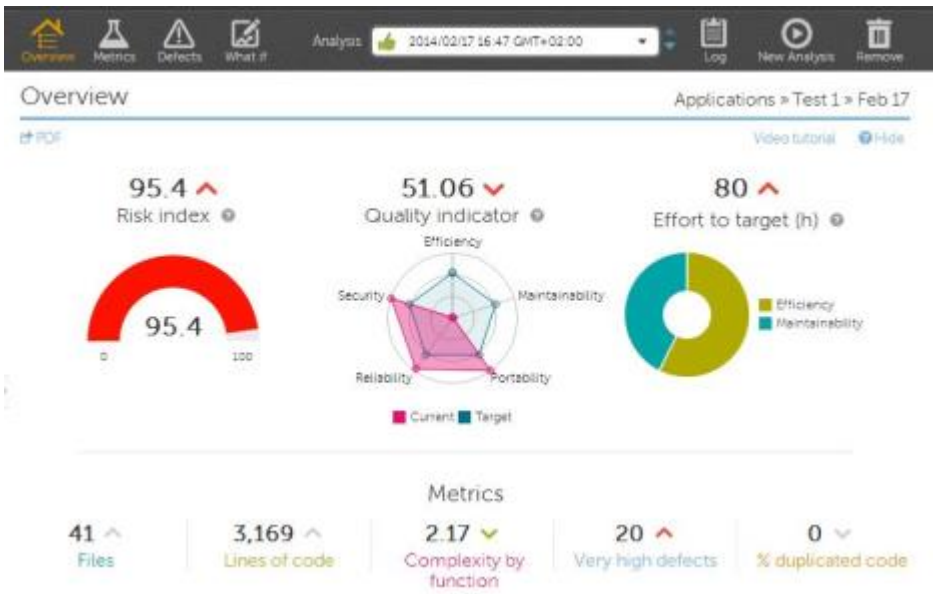

Imagen 9: Interfaz gráfica de Kiuwan (Fuente Kiuwan)

Es una solución de análisis estático de código basada en la nube, implementada en forma de SaaS (Software as a Service) y especialmente enfocada en la seguridad. Está orientada a medir, analizar y comprobar la calidad y seguridad del código fuente. Al ser SaaS no requiere la instalación de ningún tipo de componente en la parte del cliente. Es multilenguaje y también tiene la capacidad de elaboración de informes bajo demanda.

HP Fortify

Imagen 10: Interfaz gráfica de HP Fortify (Fuente HP)

Al igual que las dos anteriores, esta herramienta se distribuye bajo licencia comercial, en este caso la herramienta ha sido diseñada por la compañía Hewlett Packard. Es utilizada por los desarrolladores y expertos en seguridad para el análisis de código fuente en busca de errores de seguridad. Identifica el origen de la vulnerabilidad y proporciona resultados precisos con indicadores de severidad como CVSS. Está orientada a la corrección de líneas de código. Es muy fácilmente escalable e intuitivo, esto es una ventaja grande ya que no requiere una curva de aprendizaje excesivamente larga.

FindBugs

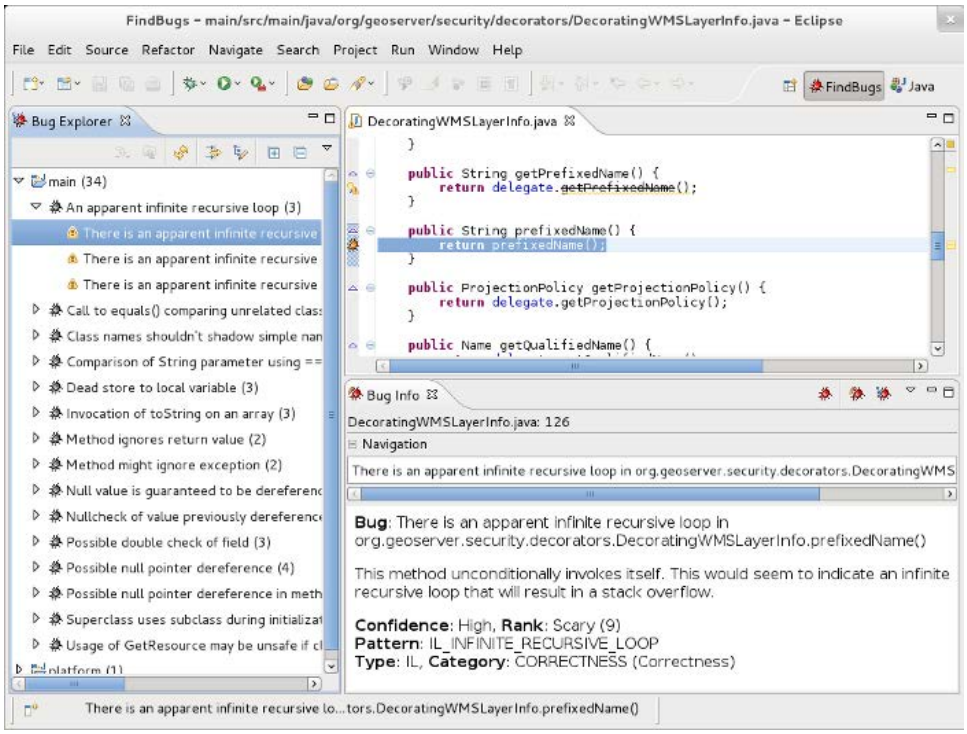

Imagen 11: Interfaz gráfica FindBugs (Fuente FindBugs)

FindBugs es una herramienta de software libre creada por William Pugh para encontrar fallos en el lenguaje de programación Java. Opera sobre bytecode, el código intermedio que utiliza la máquina virtual de Java para ejecutarse. Es distribuida como una interfaz gráfica independiente, aunque también es posible integrarla en los entornos de programación, como se puede ver en la imagen la integración con Eclipse, uno de los IDEs más utilizados. Proporciona un complemento llamado FindSecBugs que se encarga únicamente de identificar errores de seguridad. Es integrada en Sonar por defecto para la obtención de métricas.

2.1.6 Añadiendo revisiones de seguridad a un proyecto existente

Es fácil hablar sobre integrar la seguridad en el proceso de desarrollo de software, pero puede llegar a ser una gran transición si los programadores y diseñadores la ignoran por completo, centrándose únicamente en obtener las funcionalidades requeridas por la aplicación.

Evaluar y seleccionar una herramienta de análisis es la tarea más fácil de la integración de seguridad en un proyecto. Estas herramientas pueden ayudar a los programadores y analistas de seguridad a ser más eficientes encontrando y solucionando fallos, pero las herramientas por sí mismas no solucionan el problema. En otras palabras, el análisis estático tiene que ser utilizado como parte del SDL (Secure Development Lyfecycle) no como reemplazo del mismo.

Cualquier iniciativa de seguridad exitosa requiere que los programadores tengan en la cabeza la idea de que la seguridad es importante. De forma incremental, la llegada de las herramientas de análisis estático, es parte de la necesidad de añadir seguridad al código. Por esta razón, el análisis estático es tan nuevo que no ha habido estudios todavía para medir el impacto en el software construido por grandes organizaciones.

Todas las compañías de desarrollo conocidas, son al menos un poco caóticas, y cambiar el comportamiento de un sistema caótico no es una tarea fácil. En primera instancia, la integración de herramientas de análisis estático puede no parecer un problema para el ciclo de desarrollo: coger la herramienta, ejecutarla, obtener los problemas y solucionarlos. Esto no es realmente así, no es realista pensar que por integrar una nueva herramienta en el proyecto las actitudes de las personas involucradas en él vayan a cambiar, esto llevará a objeciones por parte de estas personas, como las siguientes:

- "*Tardan mucho en ejecutarse*"
- "*Arrojan muchos falsos positivos*"
- "*No es compatible con mi forma de trabajar*"

Todas estas objeciones nacen de la idea de que la seguridad es algo opcional, y como requiere esfuerzo, no se quiere hacer. Las siguientes tres cuestiones tienen que estar claras para adoptar una herramienta de forma exitosa:

- **¿Quién se encarga de ejecutar la herramienta?** Idealmente no importaría quien ejecuta la herramienta, pero algunas cuestiones prácticas hacen que esta pregunta sea importante. Muchas organizaciones consideran las dos opciones más obvias: el equipo de seguridad o los programadores. Para que el equipo de seguridad se encargue de ejecutar la herramienta, estos tienen que tener amplios conocimientos en desarrollo de código, y conocer la estructura del mismo. De esta manera pueden tener una visión amplia de los riesgos que podrían causar determinados fallos en la aplicación y considerar los más importantes en primera instancia. Los programadores tienen el conocimiento de cómo funciona su código, combinar esto con los resultados que arrojan las

herramientas es una ventaja para elegirlos como encargados de ejecutar el análisis. Por otro lado, también es probable que incluso con entrenamiento, nunca lleguen a tener el mismo nivel de conocimiento en seguridad que un equipo experto. Otra opción, es agrupar personal de seguridad y programadores, de esta manera los resultados son más precisos y se lleva a cabo un análisis más exhaustivo, esto supone menos carga tanto para los programadores como para el equipo experto en seguridad.

- **¿Cuándo se ejecuta la herramienta dentro del ciclo de desarrollo?** Decidir cuándo se ejecuta la herramienta durante el ciclo desarrollo determina la manera en la que una compañía adopta la seguridad. Hay muchas respuestas posibles, pero las más comunes son: mientras el código es implementado, en tiempo de compilación o cuando el proyecto es finalizado. La respuesta correcta, depende de cuánto tiempo tarda en ejecutarse la herramienta y la forma en la que los resultados van a ser interpretados y utilizados. Los estudios dicen que el coste de arreglar un fallo de programación incrementa con el tiempo, así que sería inteligente revisar el código nuevo de forma inmediata.

- **¿Qué ocurre con los resultados?** El proceso más importante y el trabajo más complejo vienen después de la ejecución de la herramienta, y esto es algo que muchas veces no se tiene en cuenta. En algunas organizaciones el equipo de seguridad prioriza la salida del programa como una fuente de alimentación para nuevas versiones del software. Esto lo vemos bastante en sistemas operativos, cuando actualizamos por ejemplo IOS, si nos fijamos en las mejoras de la nueva versión respecto de la anterior, siempre hay un apartado donde pone “Bug fixes”.

Las herramientas de análisis tienden a venir configuradas para detectar todos los tipos de fallos posibles. Esto está muy bien si se pretende probar la capacidad de detección de la herramienta, pero puede ser una sobrecarga para el que lo ejecuta en primera instancia y es encargado de interpretar y solucionar cada posible fallo que la herramienta muestra. Siempre hay que empezar poco a poco. Lo ideal sería empezar por los problemas bien conocidos y centrarse en un pequeño rango de fallos para después ir ampliándolo de forma incremental. No importa lo que se haga, un código muy largo no va a ser perfecto de un día para otro, además, el personal de la organización agradecerá la priorización de los fallos más prioritarios.

2.2 Requisitos

En esta sección se detallan todos los elementos que componen este proyecto: las herramientas utilizadas, las vulnerabilidades que se van a tratar mediante las reglas, etc. Primero se explicará en qué consiste la herramienta SonarQube, y posteriormente se verán las vulnerabilidades que se van a cubrir con esta herramienta mediante ejemplos.

2.2.1 SonarQube

Imagen 12: Logo SonarQube (Fuente Sonar)

SonarQube (conocido en como Sonar entre los profesionales del sector) es una herramienta de código libre multiplataforma distribuida bajo la licencia GPL (GNU General Public License) y soportada por la comunidad de usuarios y desarrolladores. Tiene como objetivo la inspección continua y evaluación de código fuente. Utiliza distintas herramientas de análisis estático para la obtención de métricas de evaluación de código, algunas de ellas son CheckStyle, PMD o FindBugs. Sus principales características son las siguientes:

- **Código fuente limpio**: informa de la salud total del proyecto y lo más importante, señala problemas encontrados en el código nuevo. Esto permite un desarrollo conciso y evita muchos problemas típicos de los proyectos de software como pueden ser bugs, malas configuraciones, etc.

- **Detección de bugs**: el analizador está equipado con reglas que permiten detectar muchos de los fallos que se cometen. Aparte de bugs el analizador también detecta vulnerabilidades y “code smells” (código duplicado, por ejemplo). Se permite añadir nuevas reglas al analizador para satisfacer las necesidades de cada proyecto.

- **Multilenguaje:** Más de veinte módulos analizadores de código cubren todos los lenguajes importantes a día de hoy como son C++, C#, JavaScript, Java, COBOL, PL/SQL, PHP, Python, etc.

- **Integración DevOps:** Para lenguajes dinámicos como JavaScript o PHP ejecutar un análisis es tan sencillo como pasarle a SonarQube los ficheros fuente del proyecto. Sin embargo, en otro tipo de proyectos como por ejemplo Java con Maven, no tiene sentido analizar el código sin incluir el analizador como parte de la construcción del proyecto. Es por ello que se proporcionan soluciones para MSBuild, Maven, Gradle, Ant y Makefiles.

- **Centralización de la calidad del software:** sitio único donde poder compartir en un proyecto la visión de la calidad del código para desarrolladores, managers y ejecutivos envueltos en uno o en muchos proyectos simultáneamente.

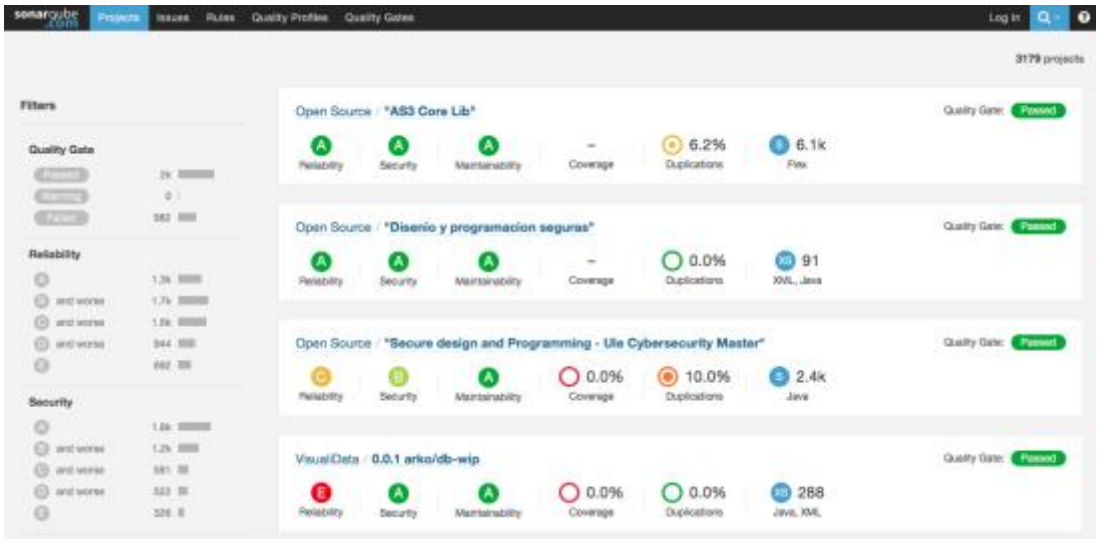

Imagen 13: Interfaz gráfica SonarQube (Fuente SonarQube)

SonarQube se puede integrar en un proyecto de desarrollo de formas distintas: se puede descargar para ejecutar localmente en una red siguiendo el modelo cliente/servidor, utilizarlo online (en la imagen se puede ver la interfaz que ofrece en su web para análisis de proyectos), integrarlo en IDEs como Eclipse, etc. Posiblemente la forma más adecuada para un proyecto sería la integración en los IDEs de los desarrolladores para posteriormente realizar una revisión por parte de los analistas en un punto central de la red.

Aunque no está enfocado específicamente en reglas de seguridad, ofrece una forma muy sencilla de implementación de reglas personalizadas, y es por ello que se ha elegido SonarQube como herramienta SAST. Una vez se tienen las reglas, se puede enfocar perfectamente a la evaluación de seguridad de forma recurrente, perfecto para entornos de integración continua.

El modelo de calidad de SonarQube se enfoca en tres tipos diferentes de reglas:

- Reglas de fiabilidad, para detección de bugs.
- Reglas de mantenibilidad, para detección de los llamados “code smells”.
- Reglas de seguridad, para detección de vulnerabilidades.

Si lo dividimos de otra manera solo hay dos tipos de reglas: reglas de seguridad y todas las demás. Esta distinción no se refiere al tipo de fallos detectados si no al origen de las reglas y los estándares impuestos sobre las mismas.

El estándar para muchas de las reglas implementadas en los plugins de SonarQube es muy estricto y tiene como objetivo cero falsos positivos, aunque esto en la práctica sea muy complicado (o imposible) de llevar a cabo en las reglas de seguridad. En otro tipo

de reglas que no son de seguridad, la tasa de falsos positivos es muy cercana a cero o incluso puede llegar a ser nula. Para las reglas de seguridad la historia es un poco distinta, por ejemplo, muchas investigaciones de seguridad informática hablan de cómo se debe tratar la información sensible (no guardarla sin encriptar, no loguear información innecesaria, etc.) pero en una regla es imposible saber qué información es sensible y cuál no, entonces las posibilidades son las siguientes: mantener el estándar de cero falsos positivos y no implementar reglas de seguridad o implementar reglas de seguridad con un estándar distinto. Ésta es la razón por la que las reglas de seguridad son tan difíciles de implementar. La idea objetivo es que las reglas generen una alerta por cada elemento sospechoso que encuentren en el código, y dejar a los humanos la decisión de si ese elemento es realmente una amenaza o no.

La mayoría de las reglas de seguridad implementadas en esta herramienta tienen como origen los estándares establecidos y reconocidos internacionalmente: CWE, OWASP Top 10, etc.

2.2.2 Reglas a implementar

Para proponer una solución compatible con las reglas ya existentes de SonarQube, que pueda integrarse y realmente ser un complemento útil, se ha realizado una comparativa de lo que debería tener, basada en el listado OWASP Top 10 y contrastada con las reglas que existen ya en la herramienta.

Para realizar la comparativa primero se han filtrado en Sonar las reglas correspondientes al lenguaje Java, en este caso se han seleccionado únicamente las reglas de detección de vulnerabilidades:

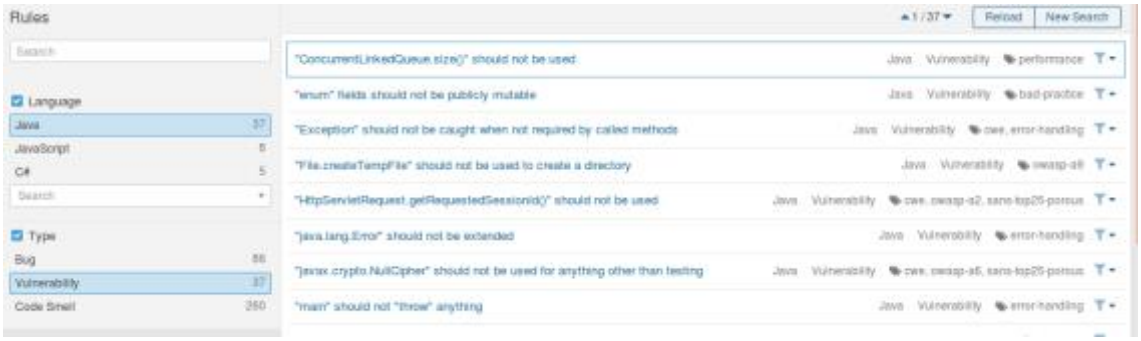

Imagen 14: Filtrado de reglas en Sonar (Fuente propia)

Como se puede observar en la imagen, por defecto, únicamente vienen incluidas treinta y siete reglas de seguridad, esto a simple vista parece un número muy bajo teniendo en cuenta la cantidad de vulnerabilidades que existen, sin embargo, Sonar permite la posibilidad de integrar nuevos paquetes de reglas en forma de archivo JAR. En este caso vamos a integrar nuestro propio plugin, con las reglas que consideramos importantes para un proyecto Java, y en concreto nos vamos a centrar en proyectos Web, ya que son los más utilizados hoy en día.

A continuación, se muestra una tabla donde se puede observar a qué categoría de OWASP pertenece cada regla contenida en el proyecto base de SonarQube (Hay algunas reglas que caen en distintas categorías de forma simultánea).

Categoría OWASP	Reglas Sonar
A1 – Inyección	- Las clases no deben ser cargadas de forma dinámica. - Los valores pasados a LDAP deben ser validados previamente. - Los valores pasados a comandos del sistema operativo deben ser validados previamente. - Los valores pasados a consultas SQL deben ser validados previamente. - Las aplicaciones web deben utilizar filtros de validación.
A2 – Rotura de autenticación y gestión de sesiones	- "ConcurrentLinkedQueue.size()" no debe ser utilizado. - "HttpServletRequest.getRequestedSessionId()" no debe ser utilizado. - Las cookies deben tener el flag “secure” activado. - Las credenciales nunca deben estar escritas en el código. - Las direcciones IP nunca deben estar escritas en el código. - No se debe confiar en los “HTTP referers”.
A3 – Cross Site Scripting (XSS)	
A4 – Rotura del control de acceso	
A5 – Malas configuraciones de seguridad	- Los algoritmos criptográficos RSA deben incorporar siempre OAEP (Optimal Asymmetric Encryption Padding).
A6 – Exposición de datos sensibles	- "javax.crypto.NullCipher" debe utilizarse únicamente para pruebas. - Las cookies deben tener el flag “secure” activado. - No se deben lanzar excepciones desde métodos contenidos en un servlet. - Los algoritmos criptográficos RSA deben incorporar siempre OAEP (Optimal Asymmetric Encryption Padding). - No se deben lanzar excepciones genéricas. - No se debe utilizar DES (Data Encryption Standard) ni 3DES como algoritmos de cifrado. - Solo deben utilizarse algoritmos criptográficos estándar. - Generadores pseudoaleatorios (PRNGs) no deben utilizarse en contextos seguros. - SHA-1 y Message-Digest no deben ser utilizados como funciones resumen.
A7 – Protección ante ataques insuficiente	- Los campos "enum" no deben ser públicamente mutables.

	- Las restricciones de seguridad deben ser definidas. - Una excepción no debe ser lanzada si el método llamante no lo requiere. - “java.lang.Error” no debe ser extendido. - El método main no debe lanzar excepciones. - Los campos “public static” deben ser constantes. - Los arrays “static final” deben ser privados. - Debe utilizarse la función “wait” en lugar de “Thread.sleep” cuando un hilo se apropia de un bloqueo de memoria. - Se debe definir la visibilidad de las variables. - Los miembros mutables no deben ser almacenados o devueltos directamente. - Los recursos deben cerrarse una vez utilizados. - “Object.finalize()” no debe utilizarse. - “Throwable” y “Error" no deben ser manejadas. - “Throwable.printStackTrace” no debe ser utilizado. - No se deben almacenar datos no validados en las sesiones. - Las aplicaciones web no deben tener un método “main”.
A8 – Cross Site Request Forgery (CSRF)	
A9 – Utilización de componentes con vulnerabilidades conocidas	- "File.createTempFile" no debe ser utilizado para crear un directorio.
A10 – APIs protegidas	

Si se observa la tabla, la mayoría de las reglas contenidas son comprobaciones simples, como puede ser el uso de determinadas funciones con vulnerabilidades conocidas, uso de algoritmos criptográficos obsoletos, etc.

Viendo la relación de las reglas con las categorías de OWASP, se comprueba que no existen reglas específicas para Cross Site Scripting (A3) ni para Cross Site Request Forgery (A8). Estas son dos de las vulnerabilidades más importantes en aplicaciones web, por lo que se ha decidido implementar una regla para cada una de ellas.

Aunque parezca una vulnerabilidad muy antigua, por experiencia propia he observado que todavía quedan muchos sitios vulnerables a SQL Injection, la validación incorrecta de entradas sigue siendo un punto muy débil de las aplicaciones, y por tanto también se ha decidido integrar en la solución una regla específica de SQL Injection, que estaría encuadrada en “A1 – Inyección”. Como se puede observar la única regla que hay para SQL Injection es muy genérica, simplemente dice que los valores pasados a una consulta SQL deben ser validados previamente.

Tampoco existe ninguna regla asociada a la rotura del control de acceso, un tema muy importante a tener en cuenta en una aplicación, define qué recursos están disponibles

a determinados usuarios, sin un control de acceso bien configurado, cualquier usuario podría acceder a determinadas partes restringidas de la aplicación. Por tanto, también se ha decidido implementar una regla que actúe sobre el control de acceso, para detectar posibles fallos de este tipo.

Por tanto, la lista final de reglas a implementar quedaría así:

- A1 – Regla para detección de SQL Injection
- A3 – Regla para detección de XSS
- A4 – Regla para detección de rotura del control de acceso
- A8 – Regla para detección de CSRF

Una vez concretadas las vulnerabilidades que se van a cubrir y la herramienta que se va a utilizar para la implementación de las reglas, es necesario aclarar lo que se va a obtener finalmente, es decir el producto final derivado de la implementación de esta solución.

SonarQube integra las reglas en forma de plugins como se verá en el detalle de la solución técnica. Estos plugins no son más que paquetes Java, con extensión ".jar", que se integran en la herramienta de forma muy sencilla, simplemente hay que introducirlos en una carpeta y la instancia de Sonar los cargará una vez arrancado el servicio. Por tanto, el producto final de este proyecto será un paquete Java que se puede integrar en cualquier instancia de SonarQube, ya que es multiplataforma, este paquete contiene reglas para detección de las vulnerabilidades mencionadas en la lista anterior.

2.2.3 SQL Injection

Una inyección SQL consiste en la ejecución de código arbitrario en un intérprete de comandos DBMS por una mala o inexistente validación de las entradas. La vulnerabilidad se produce por una incorrecta comprobación o filtrado de las variables usadas para llevar a cabo la consulta a la base de datos.

Ésta es una de las vulnerabilidades más conocidas y más peligrosas (número uno en la lista OWASP Top 10), debido a que una explotación exitosa de la misma podría llevar a la filtración de una base de datos completa incluyendo usuarios, contraseñas, etc.

Veamos un código de ejemplo:

```
private static void printUserData(String id, String pwd) throws ClassNotFoundException, SQLException {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    con = DBConnection.getConnection();
    stmt = con.createStatement();
    String query = "SELECT name, password FROM users WHERE id ='"+id+"' AND password='"+pwd+"'";
    System.out.println(query);
    rs = stmt.executeQuery(query);
    while(rs.next())    System.out.println("Name = "+rs.getString("name"));
}
```

Imagen 15: Código vulnerable a SQL Injection (Fuente propia)

Se puede ver que el código es muy sencillo, simplemente es una función que recibe como parámetro el "id" y "password" de un usuario, a través de esos parámetros realiza una consulta SQL para obtener el nombre asociado a ese usuario y una vez obtenido se imprime por pantalla. Vamos a ver qué pasa si interactuamos con la función de forma normal:

```
wifi-87-113:~ joaquin$ ./testSQLi.sh
Introduce tu usuario
joaquin
Introduce tu contraseña
password
Tu nombre es:
Joaquin
wifi-87-113:~ joaquin$
```

Imagen 16: Ejemplo SQLi (Fuente propia)

La función devuelve el nombre del usuario asociado, tal y como se esperaba, pero ¿qué pasaría si introducimos una cadena de texto especialmente manipulada? No hay ningún tipo de comprobación en las variables por tanto podemos introducir la cadena de texto que nosotros deseemos. Vamos a introducir una cadena de texto que convierta la consulta en otra diferente, inicialmente tenemos esta consulta:

SELECT name,password FROM users WHERE id='<id>' AND password='<password>'

Si alteramos la condición que se ejecuta en el WHERE de la consulta podremos modificar el comportamiento de la misma. Introduciendo en el campo "id" la cadena de caracteres:*' OR '1'='1'#*, será modificada a la siguiente:

SELECT name,password FROM users WHERE id='' OR '1'='1'

Como no hay ningún tipo de comprobación de la variable, hemos modificado el String que contiene la consulta a nuestro gusto. Al introducir la almohadilla (utilizada para identificar un comentario en muchos DBMS) el intérprete ignorará el código posterior, en este caso el intérprete de los comandos es MySQL, si fuera otro intérprete DBMS por ejemplo un Microsoft SQL Server habría que introducir el comentario correspondiente (en SQL Server se utiliza --). Como se puede observar hemos modificado el valor de la consulta y la condición siempre se cumple, por tanto,

podremos obtener los nombres de todos los usuarios de la base de datos. Si utilizamos una herramienta automatizada de SQL Injection como **sqlmap** podríamos llegar a obtener incluso una **shell** para manejar el DBMS remotamente. Vamos a ver cómo reacciona nuestra función vulnerable a la consulta modificada:

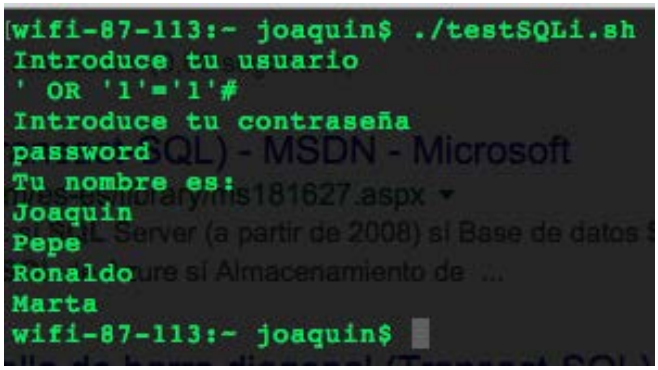

Imagen 17: Ejemplo SQLi (Fuente propia)

Al haber introducido la almohadilla, el campo "password" queda inmediatamente omitido, ya que el DBMS lo interpretará como un comentario (no habría hecho falta introducir la contraseña ya que es ignorada). Así conseguimos ejecutar nuestra consulta modificada en el servidor y obtener información que a priori no deberíamos poder obtener.

El proceso que se ha llevado a cabo se puede ver de forma muy simple en el siguiente gráfico:

Imagen 18: Gráfico funcionamiento SQLi (Fuente propia)

1. El atacante envía al servidor web una entrada modificada, en este caso la entrada es: *'' OR '1'='1'#*
2. El servidor web no valida la entrada y envía la cadena al gestor de base de datos.
3. El servidor de base de datos devuelve todos los usuarios de la base datos al servidor web.
4. El servidor web envía la información de todos los usuarios al atacante.

Una posible solución a este fallo sería la verificación de las variables de entrada, es decir comprobar que no contengan caracteres que puedan modificar la consulta. Esta sería una solución muy pobre para este tipo de consultas que se ejecutan repetidamente en los servidores (por ejemplo, un login, un formulario de registro…),

por ello Java nos ofrece la clase PreparedStatement, una extensión de la clase Statement modificada especialmente para realizar consultas repetitivas y parametrizadas como es este caso. La consulta es compilada para ejecutarse las veces que sea necesario pasando únicamente los parámetros de la misma. Ésta sería una posible solución correcta a largo plazo y por tanto es la que se ha decidido implementar en las reglas personalizadas en el analizador SonarQube (detallado en el diseño de la solución técnica).

2.2.4 Cross Site Scripting (XSS)

Cross Site Scripting, conocido como XSS en el lenguaje técnico de la seguridad informática, es un tipo de fallo de seguridad que permite la inyección de código JavaScript o similar en sitios web evitando las medidas de control existentes. Mediante esta vulnerabilidad se explota la confianza que un usuario posee en un determinado sitio. Permite a un atacante ejecutar scripts maliciosos en el navegador de la víctima, podría causar problemas como: robo de sesiones, defacing (modificación del código de la aplicación en el backend), redirección a otros sitios dominados por el atacante y algunos otros ataques. Puede ser llevado a cabo de dos maneras diferentes:

- **Reflejado**: Consistente en hacer una modificación de los valores que la web utiliza para pasar variables entre páginas distintas. El atacante podría robar las cookies de sesión para luego robar la identidad de un usuario, pero para ello debe conseguir que la víctima ejecute un determinado comando en su navegador. Esto se podría conseguir enviando un link malicioso en un correo electrónico, por ejemplo.

- **Almacenado**: Modificación del código HTML en sitios donde no hay una validación correcta de las entradas, de esta forma se podría hacer un deface de una página, y todos los demás usuarios lo verían ya que el código es modificado de forma “almacenada”, es decir es persistente a las sesiones al contrario que el XSS reflejado.

Para entender su funcionamiento, vamos a trasladarlo a una situación real: Imaginemos que una aplicación web como Facebook es vulnerable a XSS. Un atacante consigue inyectar código que redirige a los usuarios a otra página idéntica, pero controlada por el atacante. Los usuarios introducen sus credenciales pensando que están entrando en la página verdadera, proporcionándoselos directamente al atacante.

JSP (Java Server Pages) es una tecnología que permite generar páginas HTML dinámicas mediante la introducción de código Java. Observemos el siguiente formulario:

```
<html>
<head>
  <title>XSS Form</title>
</head>
<body>
    <form action="formulario.jsp" method="post">
        <input type="text" name="campo" value="">
        <input type="submit" name="enviar" value="enviar">
    </form>
</body>
</html>
```

Imagen 19: Ejemplo XSS (Fuente propia)

El código es muy simple, tenemos un campo para introducir texto y un botón para enviar los datos al JSP "formulario". Supongamos que el código JSP que trata los datos es el siguiente:

```
<html>
<head>
  <title>XSS</title>
</head>
<body>
  <% request.getParameter("campo") %>
</body>
</html>
```

Imagen 20: Código vulnerable a XSS (Fuente propia)

No hay ningún tipo de comprobación en el input "campo", por lo que una vez más, podremos introducir código en él, en este caso en vez de ser código SQL es código HTML y JavaScript. Por ejemplo, si introducimos la siguiente cadena de caracteres:

<script>alert(document.cookie)</script>

El navegador mostrará una alerta con la Cookie de sesión. En este gráfico se puede ver de forma muy clara el proceso explicado:

Imagen 21: Gráfico funcionamiento XSS (Fuente propia)

1. El atacante envía un correo específicamente diseñado para que el usuario haga click en el link

http://vulnerablexss/formulario.jsp?campo=<script>alert(document.cookie)</script>

2. El usuario hace click en el link y envía la petición del atacante al servidor
3. El servidor devuelve al cliente una alerta con la cookie de sesión.

Esto es simplemente un sencillo ejemplo de XSS, un atacante podría introducir código mucho más sofisticado para los propósitos antes mencionados. Este es uno de los fallos más comunes en los desarrollos, no validar correctamente las entradas tanto en el lado del cliente como en el lado del servidor. Un código equivalente y no vulnerable para el formulario HTML sería aplicando una expresión regular en el campo de texto, para admitir solo los valores que se deseen, en HTML5 es muy sencillo:

```
1  <html>
2  <head>
3    <title>XSS Form</title>
4  </head>
5  <body>
6      <form action="formulario.jsp" method="post">
7          <input type="text" name="campo" value="" pattern="[A-Za-z0-9]+">
8          <input type="submit" name="enviar" value="enviar">
9      </form>
10 </body>
11 </html>
```

Imagen 22: Validación de entradas en la parte del cliente (Fuente propia)

Ahora en el formulario solo se pueden introducir letras mayúsculas, minúsculas y números y tendremos que introducir al menos un carácter para que la entrada sea validada correctamente. Esta es la parte del cliente, que ya hemos solucionado, ahora vamos con la parte del servidor. Antes de imprimir, hay que chequear el texto a imprimir, hay una clase llamada JSoup que puede ser muy útil para este tipo de comprobaciones, su uso es muy sencillo y por ello se ha decidido implementar esta regla en SonarQube:

```
1  <html>
2  <head>
3    <title>XSS</title>
4  </head>
5  <body>
6    <% JSoup.clean(request.getParameter("campo"),Whitelist.basic())%>
7  </body>
8  </html>
```

Imagen 23: Validación de entradas en la parte del servidor (Fuente propia)

JSoup comprueba en una lista blanca de caracteres y después devuelve el valor modificado al cliente, evitando ataques de Cross Site Scripting. Como se puede ver la comprobación de entradas es uno de los temas más graves y la vía de ataque más frecuente en aplicaciones de tipo web. Esta regla se ha implementado en este proyecto y está detallada en el punto 3.

2.2.5 Rotura del control de acceso[4]

El control de acceso define qué recursos son accesibles para determinados usuarios o perfiles de usuarios (más conocidos como roles). Según Microsoft, un rol es una entidad de base de datos que permite que una aplicación se ejecute con sus propios permisos de usuario. Los roles de aplicación son utilizados para permitir el acceso a datos específicos únicamente a aquellos usuarios que se conecten a través de una aplicación concreta.

Un sistema de roles y usuarios se puede definir como la clasificación de privilegios de operación en una determinada aplicación:

- **Usuarios:** son los elementos que interactúan con el sistema, pueden ser personas u otras aplicaciones, y hay que determinar qué operaciones les están permitidas sobre los distintos elementos de la aplicación.

- **Roles:** son los perfiles que se asignarán a los usuarios para concederles privilegios sobre los elementos de la aplicación.

- **Recursos:** son las distintas zonas en las que se divide la aplicación, por ejemplo: panel de control, mi perfil, etc.

Una vez definidos estos términos se puede comprender mejor en qué consiste la rotura del control de acceso. Imaginemos que un usuario tiene restringido el acceso a un determinado recurso como por ejemplo el panel de control, y cambiando un parámetro en la petición a la aplicación web, esta le permite el acceso. Bien, esto sería la forma más común de explotar este control.

Un ejemplo de escenario de ataque sería el siguiente:

1. Imaginemos un recurso que accede a información de cuentas bancarias en una aplicación: **http://ejemplo.es/app/datosCuenta?cuenta=XXXX**

2. Un atacante modifica el número de cuenta en el parámetro enviado a la aplicación para acceder a la cuenta de otro usuario: **http://ejemplo.es/app/datosCuenta?cuenta=CuentaModificadaAtacante**

3. Si la aplicación no tiene bien configuradas las medidas de control de acceso, el atacante puede acceder a los datos de la cuenta de otro usuario.

Para saber si una aplicación es vulnerable a rotura de control de acceso, hay que considerar lo siguiente:

- Para accesos a datos, la aplicación asegura que el usuario está autorizado mediante un mapa de referencias o verificación de control.

[4] https://www.owasp.org/index.php/Top_10_2017-A4-Broken_Access_Control

- Para peticiones a funciones no públicas, la aplicación asegura que el usuario está autenticado y tiene los roles necesarios para usar esa función.

En la revisión del código de la aplicación se puede verificar que estos controles están implementados correctamente y están presentes en todos los sitios donde se requieran. El análisis manual es más eficiente en la detección de fallos de este tipo, las herramientas automatizadas normalmente no comprueban este tipo de fallos (de ahí que en Sonar no exista ninguna regla asociada) porque, de nuevo, no se pueden identificar en un análisis sintáctico/léxico los elementos que requieren protección.

Para evitar la rotura del control de acceso:

- **Comprobar acceso**: cada uso de una referencia directa desde una fuente no confiable debe incluir una comprobación de control de acceso para asegurar que el usuario está autorizado para utilizar ese recurso.

- **Usar referencias a objetos**: este patrón de programación previene a atacantes de actuar directamente contra los recursos no autorizados.

- **Verificación automática**: implementar mecanismos de verificación automática de control de acceso para un correcto despliegue de la autorización es muy común.

OWASP ESAPI[5] (Enterprise Security API) es una librería de código abierto enfocada a la implementación de medidas de control en aplicaciones web, que permite a los programadores escribir aplicaciones con mayores medidas de seguridad y menor riesgo de una forma sencilla.

El diseño básico es el siguiente:

- Hay un conjunto de interfaces de control de seguridad que definen, por ejemplo, los tipos de parámetro que son pasados a los controles de seguridad.

- Hay una referencia por cada control de seguridad, la lógica no es específica de la aplicación ni del contexto. Por ejemplo, validación de entradas mediante cadenas de caracteres.

- Opcionalmente, cabe la posibilidad de implementar controles de seguridad personalizados. Esto puede ser utilizado por ejemplo para autenticación en empresas (LDAP).

[5] https://static.javadoc.io/org.owasp.esapi/esapi/2.1.0.1/org/owasp/esapi/AccessController.html

La arquitectura se puede observar en la siguiente imagen:

Imagen 24: Arquitectura ESAPI (Fuente OWASP)

Esta API es muy sencilla de utilizar, proporciona funciones como "isAuthorizedForData()", "isAuthorizedForFile()" o "isAuthorizedForFunction()", muy útiles a la hora de hacer comprobaciones de control de acceso. Ésta es la razón por la que se ha elegido esta API en el listado de reglas para implementar en Sonar, la sencillez y consistencia que proporciona.

2.2.6 Cross Site Request Forgery

CSRF es un ataque que fuerza a un usuario a ejecutar acciones involuntariamente en una aplicación web en la que está autenticado. Este tipo de ataques no se usan para el robo de información ya que el atacante no tiene forma de ver la respuesta del servidor en la mayoría de los casos, en su lugar, se utilizan para ataques muy específicos como podría ser la transferencia de puntos o dinero virtual en aplicaciones vulnerables. Se suele combinar con ataques de ingeniería social para inducir a los usuarios a hacer clic en links maliciosos y ejecutar las acciones correspondientes. Si la víctima es un usuario corriente, la vulnerabilidad podría conducir a transferencias, cambio de dirección mail, etc. Si la víctima es un administrador, CSRF podría comprometer la aplicación entera.

Al contrario que XSS, CSRF explota la confianza que un sitio tiene en un determinado usuario. Esta vulnerabilidad presenta muchos riesgos, ya que en sistemas críticos como firewalls o sistemas de gestión de transacciones de cualquier tipo su explotación conllevaría pérdidas desastrosas. Permite forzar al navegador de la víctima a generar peticiones que la aplicación vulnerable no bloquea ya que las encuentra legítimas.

Imaginemos que una aplicación de transferencia de divisas como PayPal permite a los usuarios hacer el siguiente tipo de peticiones de cambio de estado:

http://aplicación.es/enviarDinero?cantidad=1000&cuentaDestino=3782458749

Un atacante puede hacer uso de la ingeniería social y enviar un link introduciendo la cantidad deseada y su cuenta para transferir dinero, por ejemplo, introduciendo este link en una imagen e induciendo al usuario a que haga clic en él. Al igual que XSS, es frecuente que este tipo de ataques se produzcan conjuntamente con técnicas ingeniería social.

En la siguiente imagen se puede ver el funcionamiento del ataque:

Imagen 25: Gráfico ejemplo CSRF (Fuente propia)

1. El atacante envía el link antes mencionado a la víctima, mediante un correo electrónico.
2. La víctima hace click en el link, y si tiene sesión abierta en el sitio en concreto, el atacante ejecutará la acción que quiere llevar a cabo mediante la sesión del usuario víctima.

Soluciones para evitar esta vulnerabilidad son las siguientes:

- Incorporar un identificador único en los campos ocultos o URLs
- Verificar el origen y el destino en la cabecera de las peticiones
- Se puede incorporar un captcha para verificar las peticiones de cambio de estado

En el proyecto también se incorpora una regla para detectar este tipo de vulnerabilidades, se ha decidido implementar la solución mediante el captcha de Google, aunque se podría haber utilizado un identificador.

2.3 Marco regulador

Existen normas y estándares que indican cómo utilizar las medidas adecuadas para la seguridad y protección de datos de carácter confidencial y en ellas están muy presentes los términos disponibilidad, integridad y confidencialidad. ISO 27001 es una norma internacional emitida por la Organización Internacional de Normalización que describe cómo se debe gestionar la seguridad de los activos de información en una compañía y puede ser implementada en cualquier tipo de organización. Está definida por los mayores expertos del mundo y proporciona una metodología a seguir, a su vez, permite la certificación de las empresas, es decir confirma que la seguridad ha sido correctamente implementada cumpliendo con la normativa ISO 27001. Es una de las principales normas a nivel mundial, en el siguiente gráfico se muestra el crecimiento del número de certificados emitidos hasta 2012:

Imagen 26: Certificados ISO 27001 hasta 2012 (Fuente encuesta ISO)

Usualmente las compañías y gobiernos realizan auditorías de seguridad para comprobar el cumplimiento de las normas y políticas estándar de seguridad. Para estas auditorías se contratan individuos expertos en seguridad informática, con amplios conocimientos en tests de intrusión. Un test de intrusión es un proceso por el cual se evalúa el estado de los activos frente a ataques, no hay mejor forma de comprobar cómo de seguro es un sistema que atacarlo y ver cómo reacciona. Se compone de las siguientes fases:

- Planificación: identificación de sistemas.
- Auditoría: realización de ataques contra los sistemas.
- Documentación: redacción de resultados y de posibles soluciones a los problemas encontrados.

2.3.1 OWASP (Open Web Application Security Project)[6]

OWASP es un proyecto de código abierto que se centra en los estándares y configuraciones de seguridad que se deberían cumplir por parte de los administradores de sitios Web para mantenerlos seguros. Provee un listado de los diez riesgos más críticos que hay en el mercado, y en este proyecto se tomará como guía el listado para el desarrollo de las reglas, ya que es reconocido internacionalmente por

[6] https://www.owasp.org/index.php/Main_Page

organizaciones tanto gubernamentales como privadas. En 2017 actualizan su Top 10 al siguiente:

- **A1 – Inyección**
- **A2 – Rotura de autenticación y gestión de sesiones**
- **A3 – Cross Site Scripting (XSS)**
- **A4 – Rotura del control de acceso**
- **A5 – Malas configuraciones de seguridad**
- **A6 – Exposición de datos sensibles**
- **A7 – Protección ante ataques insuficiente**
- **A8 – Cross Site Request Forgery (CSRF)**
- **A9 – Utilización de componentes con vulnerabilidades conocidas**
- **A10 – APIs protegidas**

En este proyecto se verán algunas de estas vulnerabilidades, y se comentarán otras, aunque no se especifique ninguna regla para solucionarlas debido a que su implementación sería muy abstracta y daría lugar a muchísimos falsos positivos. Es difícil elaborar reglas de seguridad por este mismo motivo, no se sabe qué información puede llegar a ser sensible mediante un análisis sintáctico/léxico, es el diseñador quién conoce esa información de primera mano, y es su responsabilidad poner las medidas de seguridad necesarias para que su código no sea susceptible a cualquiera de estas vulnerabilidades.

2.3.2 CVE (Common Vulnerabilities and Exposures)

Es la base de datos de información almacenada más utilizada sobre vulnerabilidades de seguridad conocidas, cada una tiene un identificador que la hace única. De esta manera se da carácter público a este tipo de problemas y se facilita la compartición de estos datos para su uso y mitigación de fallos de seguridad. El formato que siguen los identificadores en esta base de datos es el siguiente:

CVE-YYYY-NNNN

Donde YYYY identifica el año en que fue registrada y NNNN el número de la vulnerabilidad. Cuando una vulnerabilidad es candidata a ser registrada en CVE la nomenclatura sigue otro formato:

CAN-YYYY-NNNN

El proceso de incorporación de una vulnerabilidad a la lista CVE es el siguiente:

- Etapa de presentación inicial: se presenta un informe del fallo y los riesgos que pueden derivar de la explotación del mismo.

- Etapa de candidatura: una vez aceptada la presentación inicial se mantiene como candidata con el formato de identificación arriba indicado.

- Etapa de ingreso en la lista: al aprobarse su candidatura se incluye en la base de datos relacional con un identificador único como el indicado arriba.

La razón de que en CVE no aparezcan vulnerabilidades Zero-Day es precisamente el proceso de incorporación de las mismas a la base de datos.

2.3.3 OSVDB (Open Source Vulnerability Database)

Es una base de datos abierta e independiente. El propósito del proyecto fue proveer información precisa, detallada, actual e imparcial sobre vulnerabilidades de seguridad. Este proyecto ayudó a una colaboración más abierta entre empresas e individuos del sector de la ciberseguridad. El núcleo del proyecto es una base de datos relacional que enlazaba información de vulnerabilidades en una fuente de datos común y referencias cruzadas. En noviembre de 2013 la base de datos contenía información catalogada de alrededor de 100.000 vulnerabilidades.

Es soportada por la Open Security Foundation, creada para asegurar la continuidad del soporte del proyecto. Brian Martin y Jake Kouns son líderes del proyecto OSVDB. Es una implementación cliente/servidor que consiste en un demonio “mysqld” y muchos programas/librerías de cliente diferentes. Proporciona una API para poder hacer uso de esta base de datos para recopilar información.

Originalmente contenía informes de seguridad, advertencias y exploits recopilados en fuentes de información externas, la nueva base de datos contiene únicamente un título y enlaces a otras fuentes externas de información acerca de la misma vulnerabilidad, previa comprobación de veracidad y estabilidad de la información.

2.3.4 NVDB (National Vulnerability Database)

Es el repositorio estándar del gobierno de Estados Unidos para la gestión de vulnerabilidades usando el Security Content Automation Protocol (SCAP). Estos datos permiten la automatización de medidas de seguridad y cumplimiento de políticas. Incluye bases de datos de listas de verificación de seguridad, fallos de seguridad en software, malas configuraciones, nombres de productos y métricas de impacto parecidas a CVSS.

2.3.5 CWE (Common Weakness Enumeration)

Es una lista comprensiva de fallos de seguridad comunes desarrollada por la comunidad de usuarios y soportada por MITRE. Sirve como un lenguaje común, una unidad de medida para herramientas de seguridad de software y como línea base para prevención, identificación y mitigación de debilidades. El propósito del proyecto es entender mejor los fallos de seguridad y la creación de herramientas para identificarlos automáticamente. SANS Top 25 es una lista actualizada de los fallos de CWE con más riesgo.

2.3.6 CVSS (Common Vulnerability Scoring System)

Es un sistema de puntuación abierto y estándar que permite estimar el impacto que una vulnerabilidad puede causar, es decir, ayuda a calificar la severidad que puede tener una vulnerabilidad en concreto si es explotada de forma exitosa. Es amparado por Forum of Incident Response and Security Teams (FIRST) pero su uso es completamente libre para cualquier individuo o empresa.

Normalmente las vulnerabilidades se encuentran almacenadas con su puntuación (severidad) en las bases de datos de vulnerabilidades mundialmente reconocidas (CVE, OSVDB, NVDB).

Para determinar el impacto de una vulnerabilidad con CVSS se utiliza una escala del 0 al 10: la severidad de la misma será baja si está entre 0 y 3.9, impacto medio si está entre 4 y 6.9, alto cuando es mayor que 7. CVSS se basa en métricas para puntuar, la siguiente imagen muestra los parámetros utilizados:

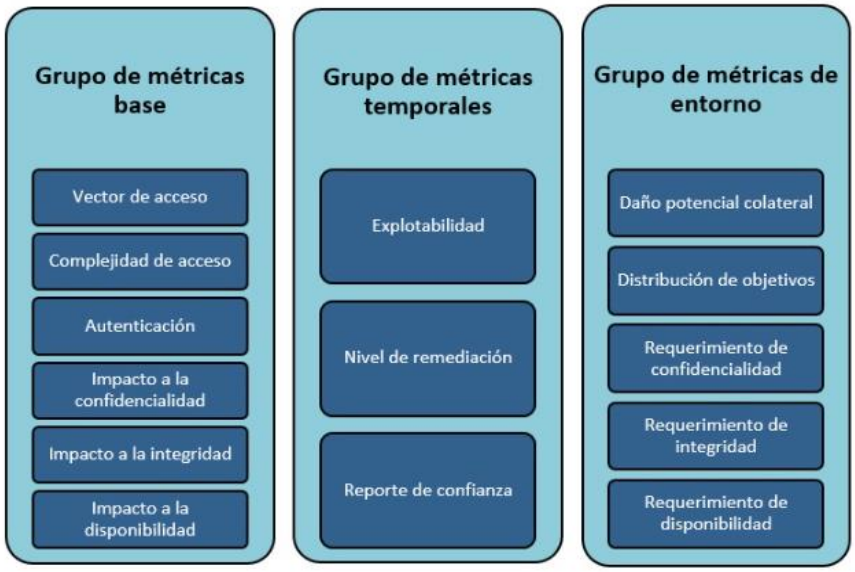

Imagen 27: Métricas de evaluación CVSS (Fuente WeLiveSecurity)

-**Métricas base**: son propiedades intrínsecas de la vulnerabilidad, siendo constantes en el tiempo y entorno utilizado. Permiten la definición del acceso a una vulnerabilidad y si se cumplen las condiciones necesarias para que sea explotada. Determina el grado de pérdida de confidencialidad, disponibilidad o integridad.

-**Métricas temporales**: son las características de la vulnerabilidad que podrían ser dinámicas con el paso del tiempo, pero son estáticas para la visión del usuario. Como los riesgos pueden cambiar con el tiempo, son considerados tres factores: explotabilidad, nivel de remediación y reporte de confianza (disponibilidad del código o técnicas de explotación). Hay que tener en cuenta que estas métricas no afectan a la evaluación ya que son totalmente opcionales y se pueden omitir.

-**Métricas de entorno**: características que son únicas y tienen relevancia para un entorno en particular. De forma similar a las temporales, son opcionales e incluyen un valor que no afecta a la evaluación, utilizado para poder omitirla en caso de que un usuario lo considere necesario.

3. Diseño de la solución técnica

3.1 Entorno de desarrollo

En esta sección se detallan los recursos utilizados en la parte técnica de este proyecto, se incluyen las máquinas, versiones de las herramientas utilizadas, etc. En el entorno de desarrollo se han utilizado las siguientes máquinas:

	Físico	Virtual
Nombre Fabricante	Apple MacBook Pro 13”	VMWare Fusion
Procesador	2,5 GHz Intel Core i5	2,5 GHz Intel Core i5
Núcleos del procesador	2 (4 threads)	1 (2 threads)
RAM	16 GB	8 GB (virtualizado)
SSD	128 GB	20 GB (virtualizado)
Sistema operativo	macOS Sierra 10.12.2	Ubuntu Linux 16.04 LTS

La plataforma virtual está construida sobre la plataforma física, utilizando el hipervisor VMWare Fusion, para tener la posibilidad de tomar snapshots (fotografía de la máquina en un instante de tiempo determinado) y volver a un punto concreto en caso de que fuese necesario. En cuanto a las versiones de los componentes utilizados:

Herramienta	Versión
Linux	4.8.0-52-generic (64 bits)
Java	JDK 1.8.0_121 (64 bits)
Apache Maven	3.3.9
Eclipse	Neon 2 Release (4.6.2)
SonarQube	5.6.6 LTS
SonarQube Scanner	2.8

Para construir el entorno, se ha creado una máquina virtual mediante el hipervisor VMWare Fusion. Sobre esa instancia virtual se ha instalado una versión de Java (la indicada en la tabla) desde los repositorios oficiales de Oracle. Posteriormente se ha configurado el entorno integrado de desarrollo Eclipse Neon, que se ejecutará sobre esta misma versión de Java. Una vez configurado el entorno del IDE, se ha incluido en el sistema una instancia de SonarQube, siguiendo la documentación[7] oficial de la herramienta donde se detalla cómo instalar SonarQube en una red local. Todos los elementos se han configurado para utilizar la misma versión de Java, para asegurar la compatibilidad en todo momento y evitar posibles errores.

[7] https://docs.sonarqube.org/display/SONAR/Documentation

3.2 Cómo se construyen reglas personalizadas en SonarQube[8]

SonarQube ofrece la posibilidad de implementar reglas mediante plugins. A partir de una plantilla de proyecto proporcionada desde Github, se implementan ciertos ficheros que permiten la detección de fallos. A continuación, se detalla el proceso de creación de reglas personalizadas en SonarQube poniendo ejemplos del funcionamiento del nuevo plugin implementado:

1. **Custom plugin**

Maven es un software que facilita la puesta en común de dependencias en proyectos de desarrollo de software. Mediante un POM (Project Object Model) que es un fichero XML, se indican las librerías necesarias y la versión concreta que se desea de las mismas, Maven a través de sus repositorios (o de los que se le indiquen) descargará las dependencias necesarias facilitando al programador la labor de despliegue.

SonarQube proporciona un proyecto Maven que sirve como punto de partida para empezar a implementar reglas. Una vez descargado se importa directamente al IDE que se esté utilizando, en este caso Eclipse. La estructura de este proyecto es la siguiente:

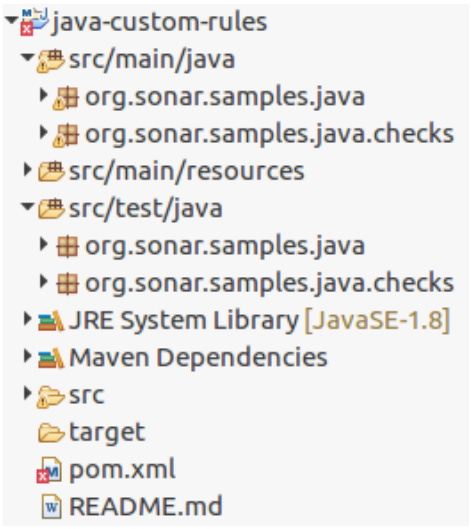

Imagen 28: Estructura del proyecto inicial (Fuente propia)

Se puede observar que el proyecto se divide básicamente en dos bloques: el bloque principal y el bloque de pruebas (main y test respectivamente). En el primero se introducen los ficheros encargados de hacer las comprobaciones, que serán clases Java implementadas específicamente para detectar ciertos patrones utilizando las librerías de SonarQube.

En el bloque de pruebas se introducirán los ficheros JUnit necesarios para comprobar el correcto funcionamiento de nuestras reglas, pasando como parámetro a las pruebas un fichero susceptible a la regla, para que la detecte.

[8] https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101

2. Ficheros necesarios para crear una regla

Vamos a ver los elementos necesarios para crear una regla en SonarQube, éstos son tres:

- Fichero que contiene código de ejemplo, tanto susceptible como no susceptible a la regla a implementar
- Clase JUnit que contiene la prueba unitaria del correcto funcionamiento de la regla
- Clase que contiene la implementación de la regla

A modo de ejemplo, se va a implementar una regla muy sencilla que satisfaga la siguiente restricción de un famoso Guru[9]:

"Si una función tiene un único parámetro, el tipo de retorno y el del parámetro no debe ser el mismo"

Para la implementación de este tipo de reglas es bueno seguir una metodología TDD (Test Driven Development), que básicamente consiste en empezar con las pruebas y continuar con el desarrollo de la funcionalidad en sí. Por tanto, primero se crea un fichero de ejemplo en la carpeta correspondiente a ficheros de prueba, que contiene código que cumple la regla y también código que no la cumple, veamos un ejemplo:

```
1 class Ejemplo {
2
3   int     funcion1() {return 0;}
4   int     funcion2(int valor) {return 0;} // Noncompliant
5   Clase   funcion3(Clase valor) {return null;} // Noncompliant
6   int     funcion4(int value, String nombre) {return 0;}
7   int     funcion5(int ... valores) {return 0}
8
9 }
```

Imagen 29: Ejemplo creación de reglas SonarQube (Fuente propia)

Se puede ver que se marcan las líneas que no cumplen como "Noncompliant", esto hará que el motor interno de SonarQube pueda comprobar si de verdad la regla está funcionando bien. Ahora se va a ver cómo sería la clase de pruebas, que comprueba el correcto funcionamiento de la regla:

[9] **Gandalf - Why Program When Magic Rulez (WPWMR, p.42)**

```java
package org.sonar.template.java.checks;

import org.junit.Test;
import org.sonar.java.checks.verifier.JavaCheckVerifier;

public class PruebaTest {

    @Test
    public void test() {
      JavaCheckVerifier.verify("src/test/files/Ejemplo.java", new Prueba());
    }
}
```

Imagen 30: Ejemplo creación de reglas SonarQube (Fuente propia)

Ésta no es más que una clase JUnit, que nos permitirá comprobar de forma muy sencilla nuestras reglas. En este caso, queremos utilizar la regla contenida en la clase "Prueba" para probar contra el archivo de ejemplo "Ejemplo.java".

Ya tenemos los ficheros de prueba, ahora hay que crear la regla. Para ello, se crea una clase, dentro del paquete "main", e introducimos dos métodos, el primero será el encargado de decirle a SonarQube qué tipo de nodos queremos visitar para hacer las comprobaciones. En este caso, vamos a visitar métodos ya que esta regla solo aplica a métodos, por tanto, el método quedaría así:

```java
@Override
public List<Kind> nodesToVisit() {
  return ImmutableList.of(Kind.METHOD);
}
```

Imagen 31: Ejemplo creación de reglas SonarQube (Fuente propia)

Con esto le estamos diciendo a SonarQube que en esta regla solo queremos visitar métodos. Ahora tenemos que implementar la comprobación sobre el método, que irá contenida en la función Java "visitNode":

```java
@Override
public void visitNode(Tree tree) {
  MethodTree method = (MethodTree) tree;
  if (method.parameters().size() == 1) {
    MethodSymbol symbol = method.symbol();
    Type firstParameterType = symbol.parameterTypes().get(0);
    Type returnType = symbol.returnType().type();
    if (returnType.is(firstParameterType.fullyQualifiedName())) {
      reportIssue(method.simpleName(), "No hagas eso");
    }
  }
}
```

Imagen 32: Ejemplo creación de reglas SonarQube (Fuente propia)

Las librerías de SonarQube proporcionan todos los elementos necesarios para poder analizar el código. Primero, comprobamos el número de parámetros de la función, ya que solo queremos actuar sobre métodos con un único parámetro, después, verificamos si los tipos de retorno y del parámetro son iguales, en ese caso mostramos una alerta al usuario. Este es un ejemplo muy sencillo, SonarQube tiene capacidades mucho mayores, se puede actuar sobre todo tipo de elementos ya sean variables,

funciones, condicionales, bucles, etc. Ya tenemos nuestra regla, ahora hay que probarla mediante el test JUnit. Lo ejecutamos y obtenemos la salida:

Imagen 33: Ejemplo creación de reglas SonarQube (Fuente propia)

La prueba es correcta, esto quiere decir que cada línea ha sido identificada por SonarQube, las que no están marcadas no cumplen la regla, y las que están marcadas la cumplen, y por tanto la regla funciona correctamente.

3. Integración de la regla en SonarQube

Ya tenemos nuestra regla implementada, ahora tenemos que integrarla en una instancia de SonarQube para que actúe sobre código real. Para ello, en la clase de la implementación de la regla agregamos los metadatos correspondientes mediante la etiqueta *@Rule*:

```
@Rule(
        key = "Prueba",
        name = "Tipo de retorno",
        description = "Para un metodo con un solo parametro, el tipo de retorno y del parametro no pueden ser el mismo",
        priority = Priority.CRITICAL,
        tags = {"bug"})
```

Imagen 34: Ejemplo creación de reglas SonarQube (Fuente propia)

Una vez hecho, tenemos que construir el proyecto mediante maven, para ello simplemente hay que situarse en la carpeta raíz del proyecto y ejecutar el comando "mvn clean install" que generará el archivo JAR correspondiente para poder integrar en SonarQube. Si todo va bien y la ejecución de Maven es correcta, obtendremos el siguiente mensaje:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 8.780 s
[INFO] Finished at: 2017-06-04T07:04:45-07:00
[INFO] Final Memory: 25M/249M
[INFO] ------------------------------------------------------------------------
joaquin@ubuntu:~/Project/workspace/CustomRules$
```

Imagen 35: Ejemplo creación de reglas SonarQube (Fuente propia)

El archivo JAR que genera este comando se encuentra en el directorio target del proyecto:

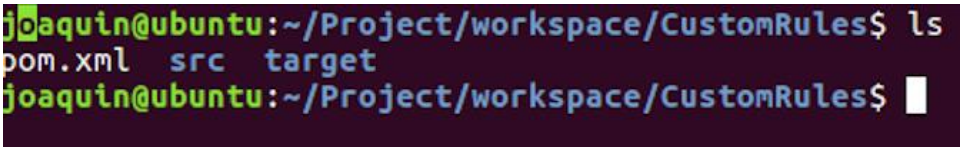

Imagen 36: Ejemplo creación de reglas SonarQube (Fuente propia)

Para integrarlo en SonarQube, simplemente hay que mover este archivo a la carpeta <Ruta del directorio SonarQube>/extensions/plugins:

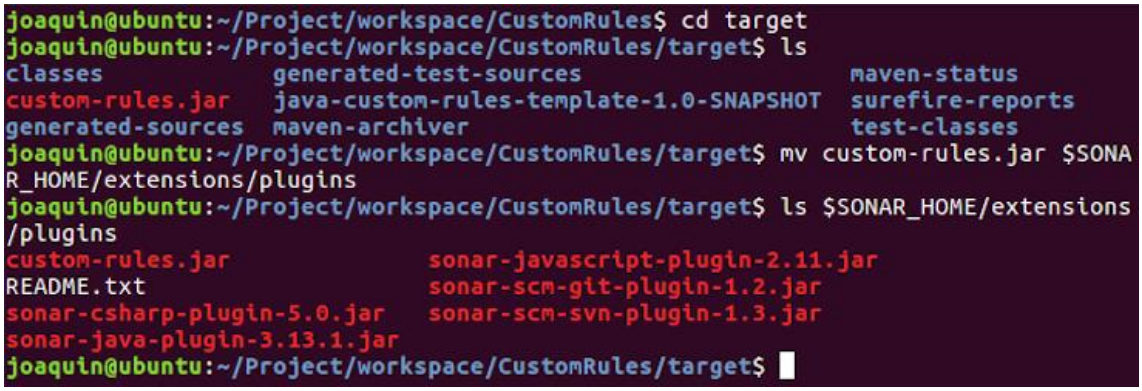

Imagen 37: Ejemplo creación de reglas SonarQube (Fuente propia)

En el listado del directorio plugins de SonarQube podemos ver que tenemos el archivo generado con Maven. Ahora simplemente lanzamos SonarQube y ya tendremos nuestra regla incluida en los repositorios. Lanzar SonarQube es muy sencillo, simplemente hay que ejecutar el script correspondiente a la arquitectura de nuestro procesador y sistema operativo indicando si queremos arrancar (start) o parar el servicio (stop):

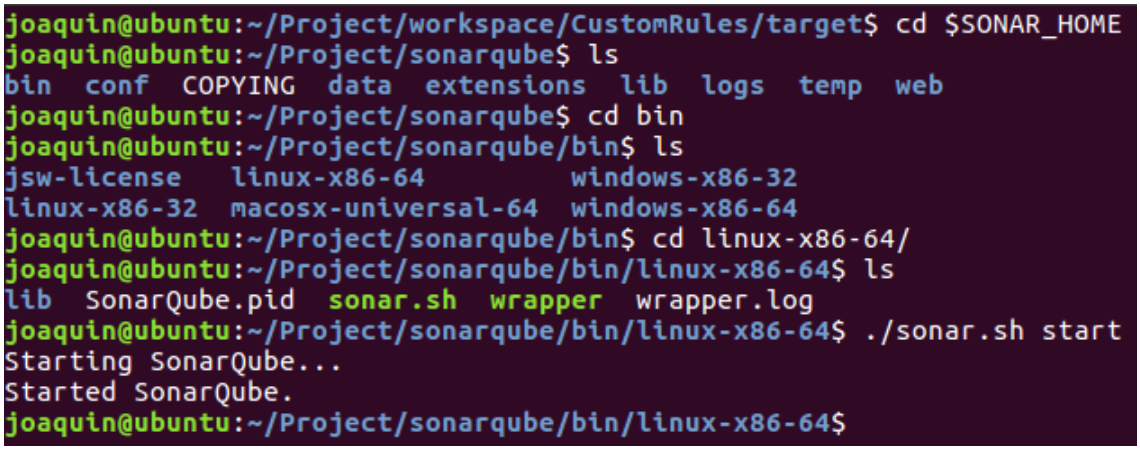

Imagen 38: Ejemplo creación de reglas SonarQube (Fuente propia)

En este caso se ha seleccionado Linux de 64 bits, ya que es la versión que corresponde con el equipo utilizado, detallado en la sección del entorno de desarrollo. Accedemos a la interfaz de SonarQube, localizada en la máquina local, en el puerto nueve mil. Se corresponde con la URL https://localhost:9000. En la sección “Rules” del menú principal podremos ver nuestra regla, para ello, hay que filtrar la búsqueda de la siguiente manera:

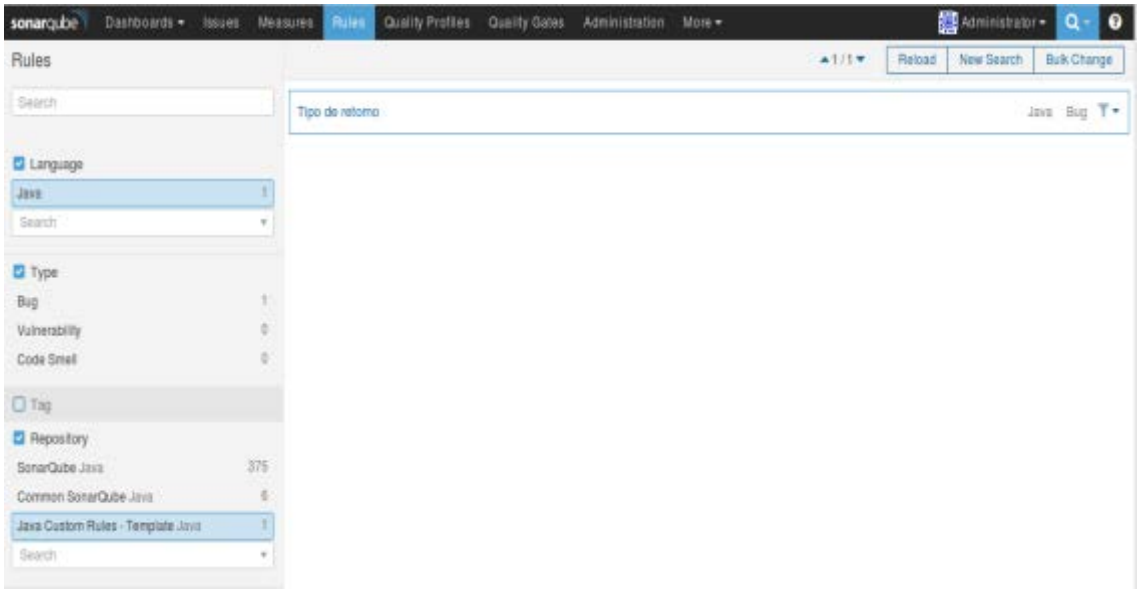

Imagen 39: Ejemplo creación de reglas SonarQube (Fuente propia)

Se ha filtrado por lenguaje (Java) y solo se seleccionan las reglas del repositorio introducido, que en este caso tiene el nombre "Java Custom Rules - Template".

Si entramos en la descripción de la regla podremos comprobar que se han integrado todos los metadatos introducidos anteriormente:

Imagen 40: Ejemplo creación de reglas SonarQube (Fuente propia)

Ya tenemos nuestra regla personalizada integrada en SonarQube y lista para actuar sobre proyectos reales. Ahora solo faltaría integrar proyectos en esta instancia de SonarQube y activar la regla sobre los proyectos deseados, o también podremos activarla para que actúe sobre el espacio de trabajo al completo. Ahora que ya hemos visto cómo implementar e integrar reglas, vamos con nuestras reglas específicas para detección de vulnerabilidades.

3.3 Regla para detección de SQL Injection

En el lenguaje Java, las dos clases más utilizadas para lanzar consultas a un gestor de base de datos o DBMS, son la clase Statement y su extensión PreparedStatement. Cada una tiene unas características diferentes y su correcto uso está muy ligado a las mismas. Dependiendo del tipo de consulta que se vaya a realizar, sería adecuado utilizar una clase u otra. Veamos las características de cada una de estas clases:

Statement

- Se usa para ejecutar consultas SQL estándar.
- Es utilizada cuando una consulta se va a realizar una única vez.
- No se pueden pasar parámetros a la consulta.
- Es principalmente utilizada para consultas de definición de datos (DDL): CREATE, ALTER, DROP, etc.
- La eficiencia de estas consultas es realmente baja.

PreparedStatement

- Es utilizada para ejecutar consultas SQL dinámicas o parametrizadas.
- Es preferible cuando una consulta se va a ejecutar repetidamente.
- Se pueden pasar parámetros a la consulta en tiempo de ejecución.
- Es utilizada para cualquier tipo de consulta SQL que se vaya a ejecutar muchas veces.
- El rendimiento es mejor que el de la clase Statement cuando es usada para la misma consulta repetidamente.

La clase PreparedStatement parece mucho más recomendable para consultas parametrizadas y repetitivas, el problema es que no se puede detectar desde el analizador cuándo una consulta es repetitiva, y aquí nos encontramos con la primera decisión. Como la mayoría de consultas de una aplicación web son repetitivas por naturaleza, se ha decidido dejar a decisión del programador si es necesario o no usar la clase PreparedStatement aunque la mayoría de las veces no dará falsos positivos ya que las tareas de DDL (Data Definition Language, se refiere a los comandos CREATE, ALTER y DROP utilizados para definir elementos en SQL) se suelen realizar mediante procesos externos al código de la aplicación web.

La regla implementada detectará cuando la clase Statement está siendo utilizada y mostrará una alerta recomendando el uso de la clase PreparedStatement para mejor rendimiento y para evitar ataques de SQL Injection. Este fallo se considerará como una vulnerabilidad en los metadatos asociados a la regla. Siguiendo el mismo proceso explicado anteriormente, primero se implementan los casos de prueba y posteriormente el código que visita los nodos.

Para los casos de prueba vamos a crear dos objetos, uno objeto Statement y un objeto PreparedStatement:

```
Statement st = new Statement(); // Noncompliant
PreparedStatement ps = new PreparedStatement();
```

Imagen 41: Fichero de prueba para regla SQLi (Fuente propia)

El nuevo objeto Statement lo marcaremos como erróneo y el objeto perteneciente a la clase PreparedStatement como correcto. De esta manera, si el test es correcto, comprobará que efectivamente la regla detecta bien los nuevos objetos Statement.

Para detectar cuando se instancia un nuevo objeto, se ha utilizado el tipo NEW_CLASS en el método que selecciona los nodos a visitar, esto garantiza que la regla solo actuará cuando un nuevo objeto es creado. El método selector de nodos según su tipo, queda así:

```
@Override
public List<Tree.Kind> nodesToVisit() {
  return ImmutableList.of(Tree.Kind.NEW_CLASS);
}
```

Imagen 42: Método que selecciona los nodos SQLi (Fuente propia)

Una vez seleccionamos los nuevos objetos creados, vamos a comprobar si el nuevo objeto pertenece a la clase Statement. En ese caso, mostraremos una alerta con un mensaje personalizado:

```
@Override
public void visitNode(Tree tree) {
  //Comprobacion para creación de nuevos objetos Statement
  if (((NewClassTree) tree).symbolType().isSubtypeOf("java.sql.Statement")) {
    reportIssue(tree, "Evita el uso de Statement para consultas parametrizadas y repetitivas,"
        + " en su lugar, utiliza PreparedStatement");
  }
}
```

Imagen 43: Método que visita los nodos SQLi(Fuente propia)

El método que visita los nodos es muy sencillo de comprender. Con un condicional se comprueba si el nuevo objeto pertenece a la clase que no queremos utilizar para consultas SQL. Hay que tener en cuenta que esta regla actuará sobre todos los objetos Statement y puede que alguna de las veces que salte la alerta finalmente resulte ser un falso positivo, por ello es tarea del analista de seguridad detectar si realmente es un riesgo o no.

En los metadatos de la regla, se ha categorizado como una vulnerabilidad crítica, ya que un fallo de este tipo podría poner en compromiso toda la información de una compañía y por tanto sería un problema a tener muy en cuenta para actuar lo antes posible.

```
@Rule(key = "SQLiCheck",
      name = "SQL Injection",
      description = "Evita el uso de Statement para consultas parametrizadas y repetitivas,"
          + " en su lugar, utiliza PreparedStatement",
      priority = Priority.CRITICAL,
      tags = {"vulnerability"})
```

Imagen 44: Metadatos regla SQLi (Fuente propia)

En la sección de resultados y evaluación, se podrá observar el comportamiento de esta regla sobre proyectos reales de desarrollo. Ésta regla es la que mejor resultado tiene de este proyecto, ya que los fallos de tipo SQL Injection (así como los buffer overflows) son mucho más sencillos de detectar desde un analizador estático de código, como se ha comentado anteriormente.

3.4 Regla para detección de XSS

Para la detección de Cross Site Scripting en código Java, se ha implementado una solución mediante la librería JSoup[10]. JSoup es una librería escrita en Java que permite trabajar con código HTML real. Proporciona una API para extracción y manipulación de datos, tomando como base lo mejor de DOM (Document Object Model), CSS (Cascading Style Sheet) y métodos JQuery (librería JavaScript).

JSoup convierte el HTML en un DOM al igual que lo haría un navegador, por eso es la solución perfecta para XSS ya que es un ataque que finalmente tendría impacto en los navegadores de los usuarios finales. Sus principales características se pueden resumir en las siguientes:

- Filtrar y analizar código HTML desde una URL, fichero o String
- Encontrar y extraer datos usando DOM o CSS
- Manipular elementos HTML, atributos y texto
- Limpiar contenido introducido por el usuario para evitar XSS
- Salida ordenada de código HTML

Para la implementación de esta regla, nos hemos centrado en la cuarta característica de la lista. JSoup permite la limpieza de cadenas de caracteres mediante una lista blanca. Esta lista blanca no es más que una colección de caracteres que se deberían permitir en una entrada de texto introducida por el usuario. Por supuesto esta regla actuará sobre código que se ejecuta en el servidor, por lo tanto, la utilización de la misma no excluye otros métodos de comprobación de entradas como por ejemplo validación mediante JavaScript o validación nativa en HTML, en el lado del cliente (navegador).

Al igual que en las reglas anteriores, se seguirá utilizando una metodología TDD. Por tanto, primero se empieza con los ficheros de prueba. En este caso, como se pretende actuar sobre código Java, la regla está diseñada para actuar sobre los servlets, y no sobre los JSP mencionados anteriormente. Esos JSP posteriormente se traducen a código Java, y ahí es donde nuestra regla actuará, sobre los servlets Java.

La implementación ideal detectaría cada parámetro sensible en el código, pero puesto que esto no es posible debido a los problemas de detección de datos sensibles que plantean las herramientas SAST, la regla detectará cualquier parámetro introducido mediante la variable “request” y en caso de encontrar alguno, verificará si la clase JSoup está siendo utilizada para validar ese parámetro. En concreto se verificará la utilización del método clean de JSoup, visto anteriormente en el detalle de la vulnerabilidad XSS.

El fichero de prueba, una vez más será muy sencillo, en una clase introducimos un parámetro pasado por petición del usuario sin utilizar la clase JSoup. En este caso la regla

[10] https://jsoup.org

genera la alerta que pretendemos que salte cuando se encuentre este tipo de fallo. En otra clase, introducimos un parámetro mediante petición, pero esta vez validado mediante JSoup, aquí la regla no salta ya que la implementación es validada como correcta.

```
void prueba(){
    System.out.println(request.getParameter("name")); // Noncompliant
}

void prueba2(){
    System.out.println(JSoup.clean(request.getParameter("name"),Whitelist.basic()));
}
```

Imagen 45: Fichero de prueba regla XSS (Fuente propia)

Una vez implementados los ficheros de pruebas, pasamos a la implementación de la lógica de la regla. Nuestra regla escaneará las clases Java en busca de la variable request. En caso de encontrar alguna referencia al método getParameter, buscará si la librería JSoup está siendo utilizada, en concreto el método clean, para limpiar los parámetros pasados al servlet. Por ello la regla tendrá que actuar sobre invocaciones a métodos. Y el método que indica los nodos a visitar queda así:

```
@Override
public List<Tree.Kind> nodesToVisit() {
  return ImmutableList.of(Tree.Kind.METHOD_INVOCATION);
}
```

Imagen 46: Método que selecciona los nodos XSS (Fuente propia)

En este caso, como se puede observar, se ha utilizado el tipo de nodo METHOD_INVOCATION, que hace referencia a llamadas a funciones. En la comprobación, primero nos aseguramos de que la clase HttpServletRequest es utilizada:

```
@Override
public void visitNode(Tree tree) {

   if(tree.getClass().getSimpleName().equals("HttpServletRequest"))   parameter = true;
   if(parameter && (tree.getClass().getSimpleName().equals("JSoup"))) jsoup = true;
   if (parameter && !jsoup){
       reportIssue(tree, "Utiliza JSoup para limpiar las variables pasadas al servlet,"
                 + "evitando posibles ataques de XSS");
   }
}
```

Imagen 47: Método que visita los nodos XSS (Fuente propia)

En este método que visita los nodos, después de comprobar que existe una referencia a un método perteneciente a la clase HttpServletRequest, utilizamos dos variables lógicas para controlar la salida de la regla. Si encontramos una variable request ponemos la variable parámetro a verdadera. Si hemos encontrado un parámetro comprobamos si existen referencias a métodos de la clase JSoup, en este caso la regla no debe saltar. La regla sólo salta cuando hay parámetros y ninguna referencia a JSoup, es decir, cuando la variable parámetro es verdadera pero la variable jsoup es falsa.

De esta manera, nos aseguramos de que cada vez que un parámetro es pasado a un servlet, se comprobará la verificación del mismo, en este caso mediante la solución concreta de JSoup, pero se podría adaptar para utilizar cualquier otro método de verificación como por ejemplo expresiones regulares, parseo mediante otras librerías de Java e incluso OWASP ESAPI.

También incluimos los metadatos de esta regla en el plugin:

```
@Rule(key = "XSSCheck",
      name = "Cross Site Scripting",
      description = "Utiliza JSoup para limpiar las variables pasadas al servlet,"
            + "evitando posibles ataques de XSS",
      priority = Priority.CRITICAL,
      tags = {"vulnerability"})
```

Imagen 48: Metadatos regla XSS (Fuente propia)

Una vez más, esta regla está configurada para proyectos específicos, y su uso correcto sería en aplicaciones web. No tiene sentido ejecutar esta regla en aplicaciones que no sean web ya que una vulnerabilidad XSS se basa en código HTML y JavaScript. Por supuesto está sujeta a falsos positivos y también será tarea del diseñador y analista de seguridad poner en común los puntos a modificar.

3.5 Regla para detección de rotura del control de acceso

Como se ha comentado en el punto dos, para la implementación de esta regla, se ha escogido una librería de OWASP llamada ESAPI, que proporciona mecanismos sencillos para el despliegue de autorización en una aplicación.

Al igual que cualquier otra librería Java, para trabajar con ella en una aplicación se debe importar al proyecto, es decir incluir el código de la librería en el proyecto Java. Para ello hay que situar al inicio de la clase la línea “import” en este caso, la librería que se va a importar es OWASP.

Por tanto, la regla se ha diseñado para detectar cuando la librería es importada al proyecto, ya que la única forma existente de trabajar con esta librería es importándola.

Las capacidades de esta librería son muy interesantes y están detalladas en el análisis del estado del arte. Provee de todo lo necesario para implementar la seguridad en un proyecto de desarrollo. Proporciona una manera sencilla de implementar autenticación, control de usuarios y acceso, validadores (para SQL Injection y XSS), cifradores, registros, etc. Parece muy recomendable incluir esta librería ya que resulta mucho más fácil para el programador utilizar funciones existentes que implementar las suyas propias, además ahorra tiempo de desarrollo y por tanto dinero.

Por tanto, esta regla se encargará de detectar si la librería está siendo utilizada o no, va a ser muy sencilla. En este caso lo único que tenemos que comprobar es si la librería está anexada al proyecto, y posteriormente se podrían hacer comprobaciones de uso de determinadas funciones en accesos a recursos, etc.

En este caso, se ha decidido implementar la regla utilizando una plantilla contenida ya en Sonarqube a partir de la cual se pueden crear nuevas reglas. Esta plantilla busca dependencias en las inclusiones de librerías al proyecto:

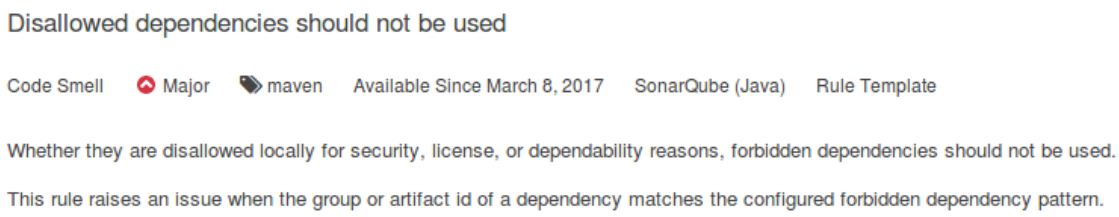

Imagen 49: Plantilla regla dependencias (Fuente propia)

Detecta cuando una dependencia prohibida es incluida al proyecto, en este caso le daremos la vuelta a la lógica de la regla, ya que es precisamente la librería que queremos incluir la que buscaremos entre las dependencias, y no una librería prohibida para el proyecto.

La regla de detección de rotura de control de acceso queda así:

Imagen 50: Regla rotura de control de acceso (Fuente propia)

En este caso la regla se integra directamente desde la consola de Sonar, y no hace falta anexarla mediante un plugin como en el resto de reglas. Desde la misma consola podemos seleccionar el perfil en el que queremos que actúe la regla, la severidad, etc. En este caso se ha decidido simplemente informar de si la librería está siendo utilizada o no, no se le ha asignado severidad como en las otras reglas.

3.6 Regla para detección de CSRF

El caso de CSRF es más complicado de abordar que los anteriores. El motivo es que el primer objetivo de la regla es detectar qué peticiones URL son de cambio de estado, y en el analizador la máxima información que tenemos es el mapeo de los controladores que manejan las peticiones a los recursos, que están escritos en Java en un proyecto JSP.

Un controlador no es más que una función que actúa cuando un cierto recurso del servidor web es solicitado. Por ejemplo, si enviamos la siguiente URL como petición:

http://aplicacion/recurso

El servidor web tendrá implementado un controlador que decida qué servlet se ejecutará cuando el usuario solicite acceso al elemento "recurso". El analizador tendría que detectar qué recursos son susceptibles de tener riesgo en caso de que un usuario con una sesión establecida acceda a esa URL.

Según OWASP, una de las formas de mitigar o solucionar este ataque es utilizando captchas. La solución que se ha decidido implementar es la introducción de un captcha en las vistas asociadas a recursos. El captcha que se integra es el de Google ReCaptcha por ser uno de los más conocidos y utilizados mundialmente, además su integración en un proyecto web es muy sencilla. Como se ha comentado, es imposible detectar qué recursos son de cambio de estado desde el analizador, por tanto, se buscarán controladores en base a nombres concretos, una empresa debería modificar esta lista de nombres para que la regla se adapte a sus controladores de cambio de estado en concreto. Aquí se utilizarán nombres como "transfer", "perform", "action", etc. Muy comunes en transacciones de este tipo. Se utiliza una lista negra de nombres que podría ser modificada a gusto de un diseñador para introducir esta regla en su entorno de desarrollo.

Una vez identificado el controlador en base al nombre, la regla deberá identificar si en la vista asociada a ese controlador hay embebido un captcha de Google o no. Esto se puede conseguir mediante el uso de una expresión regular que detecte una cadena de texto específica. En este caso, la cadena que se va a detectar es la URL que se utiliza para introducir un captcha[11] de Google en una aplicación Web. Esta URL la proporciona Google directamente para facilitar a los desarrolladores la tarea de embeber una verificación de este tipo en sus aplicaciones. La URL es la siguiente:

https://www.google.com/recaptcha/api.js

Este es el punto de entrada del captcha. Para incluir este elemento en una aplicación web también sería necesario vincularlo con una clave única que Google proporciona a cada sitio o usuario que quiere implementarlo. Finalmente, la vista que contiene el captcha mostraría un widget para que el usuario verifique que es humano, que sería el elemento mostrado en la imagen, debajo del formulario. Esto es importante, un captcha no verifica la identidad de una persona, simplemente verifica que realmente quien accede a un recurso es humano.

[11] https://webdesign.tutsplus.com/es/tutorials/how-to-integrate-no-captcha-recaptcha-in-your-website--cms-23024

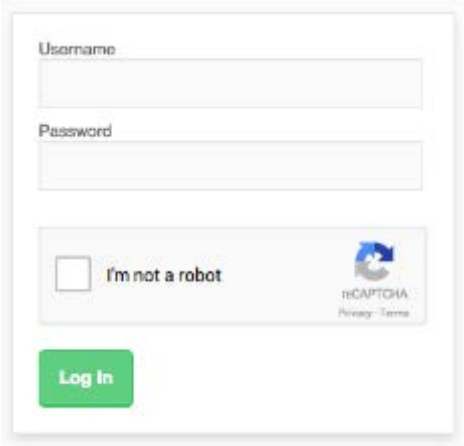

Imagen 51: Ejemplo de formulario web que utiliza captcha (Fuente Google)

Como el framework más conocido y utilizado en los últimos años para el desarrollo de aplicaciones web es Spring, se ha decidido llevar a cabo una aproximación para este tipo de proyectos. Esto se hace porque en Spring las anotaciones son fundamentales, en este caso nos vamos a centrar en las anotaciones tipo @Controller para identificar los controladores de la aplicación, y de una forma muy sencilla detectaremos la vista asociada y la analizaremos. Dentro de un controlador, se identificarán las anotaciones @RequestMapping y se comprobará si su nombre corresponde con alguno de la lista negra. En caso de que coincidan se analizará la vista asociada a ese mapeo para comprobar si tiene un captcha embebido. Una vez explicado, vamos con la implementación.

Como anteriormente, primero empezamos por los ficheros de prueba. En este caso el fichero de ejemplo que se pasará a JUnit para verificar el comportamiento de la regla es un poco más complejo, ya que tendremos que hacer uso de anotaciones y nombres concretos para que la regla salte. Por tanto, se crea un controlador con dos anotaciones @RequestMapping de acceso a recursos:

```
@Controller
class EjemploCSRF {

    @RequestMapping
    String vulnerable("/perform"){ // Noncompliant
        return "perform";
    }

    @RequestMapping
    String novulnerable("/view"){
        return "view";
    }
}
```

Imagen 52: Fichero de prueba regla CSRF (Fuente propia)

Un controlador es usualmente el encargado de preparar los datos que maneja la aplicación (el modelo) y seleccionar la vista deseada para mostrar esos datos. Como se puede observar, el controlador habla por sí mismo. En este ejemplo tenemos dos

vistas: perform y view. Como se ha comentado anteriormente, los nombres de las vistas son relevantes en esta regla, ya que en base a ellos se decidirá si se analiza la vista asociada a ese modelo o no. Por tanto, cuando se detecte el nombre perform o cualquiera de los nombres contenidos en la lista negra, ésta se activará.

La clase que implementa la lógica de la activación de la regla es más compleja que en los casos anteriores. Esto se debe a la necesidad de explorar ficheros externos (vistas) en busca de la URL mencionada anteriormente. La función que visita los nodos en esta clase Java es la siguiente:

```
@Override
public void visitMethod(MethodTree tree) {

  String listaNegra = "(action|perform|transfer|pay)+";
  String url = "https://www.google.com/recaptcha/api.js";
  Symbol.MethodSymbol simbolo = tree.symbol();

  SymbolMetadata propietario = simbolo.owner().metadata();

  boolean esControlador = propietario.isAnnotatedWith("org.springframework.stereotype.Controller");

  if (esControlador) {
    if (simbolo.metadata().isAnnotatedWith("org.springframework.web.bind.annotation.RequestMapping")) {
      for (VariableTree param : tree.parameters()) {

        TypeTree valorRetorno = param.type();

        if(valorRetorno.getClass().getSimpleName().matches(listaNegra)){
          File vista = new File(valorRetorno.getClass().getSimpleName());
          FileReader lectorVista;
          boolean captcha = false;
          try {
              lectorVista = new FileReader(vista);
              String linea;
              while((linea = new BufferedReader(lectorVista).readLine()) != null){
                  if(linea.matches(url)){
                      captcha = true;
                  }
              }
          }
          catch (IOException e) {
              System.out.println("No se encuentra la vista");
          }
          if(!captcha)    reportIssue(tree, "Utiliza un captcha en los recursos de cambio de estado"
                  + "para prevenir posibles ataques CSRF");
        }
      }

    }
  }
  super.visitMethod(tree);
}
```

Imagen 53: Regla para detección de CSRF (Fuente propia)

Se puede ver que la complejidad de esta regla es más alta que las anteriores por el tamaño del código y los motivos anteriormente comentados. Su funcionamiento es sencillo, aunque el código parezca muy enrevesado:

1. Primero se inicializa la lista negra (aquí una empresa podría introducir los nombres de sus controladores de cambio de estado). Esta lista negra se ha decidido implementar como una expresión regular, ya que es una forma muy eficiente de comprobar si el texto cumple algún patrón.

2. Se introduce la URL a detectar en la vista (URL de inclusión de reCaptcha mediante JavaScript), también se introduce como una expresión regular, en este caso tiene que coincidir exactamente con la cadena de texto.

3. Si se encuentra un controlador, se sigue buscando en los mapeos, si no se encuentra ninguno la regla no se activará.

4. Si se ha encontrado un controlador, se busca en los mapeos (anotados con @RequestMapping) para ver si alguno coincide con los especificados en la lista negra.

5. En caso de que algún mapeo coincida con un nombre de la lista negra, se abre la vista asociada (si no se encuentra la vista devuelve un error) y se comprueba si contiene la URL de reCaptcha.

6. En caso de que la vista no contenga la URL de reCaptcha la regla saltará indicando en el @RequestMapping que hay un posible fallo de Cross Site Request Forgery.

En este caso también incluimos los metadatos de la regla para mejor visualización en la consola de administración:

```
@Rule(key = "CSRFCheck",
      name = "Cross Site Request Forgery",
      description = "Utiliza un captcha en los recursos de cambio de estado"
        + "para prevenir posibles ataques CSRF",
      priority = Priority.CRITICAL,
      tags = {"vulnerability"})
```

Imagen 54: Metadatos regla CSRF (Fuente propia)

4. Resultados y evaluación

En esta sección del documento se puede observar el comportamiento del analizador tanto sobre código real de proyectos web como en código realizado a propósito para comprobar el correcto funcionamiento de las mismas.

Los proyectos que se han decidido analizar son proyectos de Github, ya que es la mayor fuente de código libre del mercado y qué mejor sitio para obtener código fuente que una de las mayores comunidades de compartición de software y buenas prácticas de programación. También se ha analizado un proyecto de fin de grado de la Universidad Politécnica de Madrid, proporcionado por su autor David Márquez Delgado para su análisis y medición de la seguridad en el código del mismo. Se indicarán los enlaces utilizados para obtener el código para que se puedan observar los proyectos y su estructura sin ningún tipo de problema.

El listado de proyectos Java que se han analizado mediante la solución de seguridad descrita a lo largo de este documento es el siguiente:

- Aplicación Java Spring llamada Diseño de Aplicaciones Web (DAW), proyecto de fin de grado cedido por un alumno de la Universidad Politécnica de Madrid

- Ejemplo de aplicación web Java[12]

- Shopizer 2.0.5[13] para Java 1.8, proyecto de código libre de sitio e-commerce

- Sample Jersey Webapp[14], proyecto de muestra basado en Java en la parte del backend

Los siguientes apartados incluyen los test de seguridad realizados, así como sus resultados y evaluación de las tasas de fallo de la solución implementada.

[12] https://github.com/tomcz/example-webapp
[13] https://github.com/shopizer-ecommerce/shopizer
[14] https://github.com/rkazarin/sample-jersey-webapp

4.1 Funcionamiento de las reglas en la consola de Sonar

Para comprobar que las nuevas reglas introducidas en el analizador detectan los fallos correctamente desde la consola de visualización, se han realizado unas pequeñas pruebas con código diseñado a propósito para que las reglas salten. A continuación, se muestran los resultados para cada regla, poniendo un ejemplo de falso positivo para cada una de ellas.

SQL Injection

Para el primer caso, el código que se ha utilizado para probar el analizador es el siguiente:

```
1  package test1;
2
3  import java.sql.Connection;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7
8  public class SQLTest {
9
10     void verTabla(Connection con, String nombre) throws SQLException {
11
12           Statement stmt = null;
13           String query = "SELECT * FROM personas WHERE nombre=" + nombre;
14           try {
15               stmt = con.createStatement();
16               ResultSet rs = stmt.executeQuery(query);
17               while (rs.next()) {
18                   String aux = rs.getString("nombre");
19                   System.out.println(aux);
20               }
21           }
22           catch (Exception e ) {
23               System.out.println("Error en la consulta SQL");;
24           }
25
26           finally {
27               if (stmt != null)   stmt.close();
28           }
29       }
30 }
```

Imagen 55: Ejemplo SQL Injection (Fuente propia)

Este código refleja el error más típico de SQL Injection, pero que aún en 2017 es común encontrar en las aplicaciones, bien porque están desactualizadas o por falta de buenas prácticas de programación. En el recuadro rojo se muestra el código vulnerable, el motivo de que sea vulnerable es que el parámetro pasado a la función (nombre) no tiene ningún tipo de validación, por lo que podríamos introducir una secuencia de caracteres para alterar el funcionamiento de la consulta y obtener información de la base de datos siguiendo el mismo proceso que el explicado anteriormente en el punto 2.2.3.

En la siguiente imagen se puede observar cómo la regla de SQL Injection informa del fallo cuando la clase Statement es utilizada por alguno de los componentes de la aplicación:

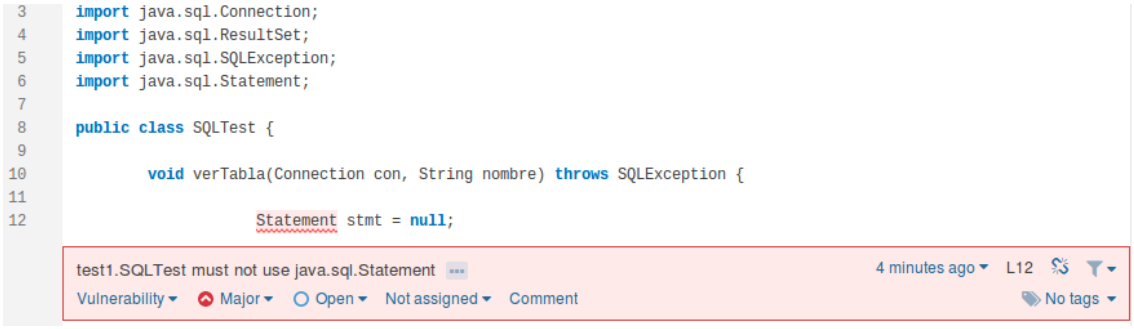

Imagen 56: Detección de SQL Injection Sonar (Fuente propia)

Al ejecutar el analizador sobre este fichero de prueba, se etiqueta la vulnerabilidad como crítica al haber introducido esos metadatos en la regla. Estos parámetros pueden ser modificados a medida, cada proyecto tiene sus propias prioridades en cuanto a la severidad de las reglas.

Esto en un principio es lo que queremos, pero hay un problema a tener en cuenta que ya se ha mencionado anteriormente, los falsos positivos.

En este caso un falso positivo se produce cuando las entradas se están validando, pero, aun así se está utilizando la clase Statement. La regla que hemos creado no sabe detectar cuándo se hace la validación (ni siquiera detecta si existe alguna validación), por lo que, si se valida la variable nombre, por ejemplo, utilizando una expresión regular o JSoup, el analizador lo tomará como un fallo de igual manera.

En este caso se ha decidido implementar una expresión regular, ya que JSoup se ha utilizado para la regla de Cross Site Scripting. Si el nombre hace “match” con la expresión regular entonces será permitida esa entrada, de lo contrario se producirá un error. La modificación es muy sencilla y el código resultante es el siguiente:

```java
package test2;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SQLTest {

    void verTabla(Connection con, String nombre) throws SQLException {
            String validacion = "[A-Za-z]+";
            if (nombre.matches(validacion)){
                Statement stmt = null;
                String query = "SELECT * FROM personas WHERE nombre=" + nombre;
                try {
                    stmt = con.createStatement();
                    ResultSet rs = stmt.executeQuery(query);
                    while (rs.next()) {
                        String aux = rs.getString("nombre");
                        System.out.println(aux);
                    }
                }
                catch (Exception e ) {
                    System.out.println("Error en la consulta SQL");;
                }

                finally {
                    if (stmt != null)   stmt.close();
                }
            }
            else    System.out.println("Error en la entrada");
        }
}
```

Imagen 57: Falso positivo SQL Injection (Fuente propia)

Al introducir la expresión regular la vulnerabilidad de SQL Injection desaparece, pero cuando ejecutamos el analizador sobre este código el resultado es el siguiente:

```java

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SQLTest {

        void verTabla(Connection con, String nombre) throws SQLException {
                    String validacion = "[A-Za-z]+";
                    if (nombre.matches(validacion)){
                        Statement stmt = null;
```

test2.SQLTest must not use java.sql.Statement ... 20 hours ago L13

Vulnerability Major Open Not assigned Comment No tags

Imagen 58: Detección falso positivo SQL Injection en Sonar (Fuente propia)

El analizador sigue detectando el código como vulnerable, ya que la única lógica que tiene la regla es identificar cuando la clase Statement se está utilizando en el código, el analizador no sabe si las variables pasadas a la consulta han sido validadas previamente o no.

Cross Site Scripting

El código de prueba para la regla de detección de XSS se trata de una simulación del paso de un parámetro a un servlet. Estos parámetros se recogen en peticiones (HttpServletRequest) que son pasadas a los servlets (clases Java utilizadas para generar código HTML dinámico), en el código de ejemplo se intenta extraer el parámetro sin ningún tipo de validación.

El código vulnerable a XSS es el siguiente:

```
1  package test1;
2
3  import javax.servlet.http.HttpServletRequest;
4
5  public class XSSTest {
6
7          HttpServletRequest request = new HttpServletRequest();
8          String prueba = request.getParameter("name");
9
10 }
11
```

Imagen 59: Ejemplo XSS (Fuente propia)

Como se puede ver, el parámetro se introduce en la variable prueba sin ningún tipo de comprobación previa. En este caso el código es muy simple, ya que simular la interacción de una aplicación Web ensuciaría el código y no se vería bien lo que se quiere mostrar aquí, la no validación de entradas del usuario (o de otra aplicación).

```
6
7          HttpServletRequest request = new HttpServletRequest();
8          String prueba = request.getParameter("name");
9
```

Imagen 60: Detección XSS en Sonar (Fuente propia)

De nuevo, si validamos las entradas de otra forma que no sea utilizando JSoup, el analizador seguirá detectando la vulnerabilidad, ya que no contempla este tipo de situaciones, en estos casos se producirán falsos positivos y será tarea del analista de seguridad determinar qué es un falso positivo y qué es realmente un riesgo para la aplicación.

Un ejemplo de falso positivo sería el siguiente, una vez más se utilizan expresiones regulares para validar la entrada:

```java
package test1;

import javax.servlet.http.HttpServletRequest;

public class XSSTest {

    HttpServletRequest request = new HttpServletRequest();
    String exp = "[A-Za-z]+";
    String prueba;
    if (request.getParameter("name").matches(exp))  prueba = request.getParameter("name");
    else    prueba = null;
}
```

Imagen 61: Falso positivo XSS (Fuente propia)

En este caso, igual que en el caso anterior de SQL Injection, el analizador detecta la vulnerabilidad, aunque esta no se podría explotar ya que los parámetros sí que son validados, tratándose de un falso positivo.

Control de acceso

En la regla de rotura de control de acceso se comprueba que la librería OWASP ESAPI está integrada en el Proyecto, por ello no se va a generar un ejemplo de falso positivo ya que no se da el caso. La lógica es que la librería esté incluida o no, no hay puntos intermedios.

Cuando la librería está integrada en el proyecto el código Maven (Project Object Model) queda así:

```xml
<dependency>
    <groupId>org.owasp.esapi</groupId>
    <artifactId>enterprise-security-api</artifactId>
</dependency>
```

Imagen 62: Ejemplo control de acceso (Fuente propia)

El analizador nos avisa cuando esta librería no está incluida, por tanto, para probar la regla se ha creado una clase que no incluye esta librería:

```java
package test1;


public class AccessControlTest {

    public void funcion1(){
        System.out.println("prueba1");
    }

    public void funcion2(){
        System.out.println("prueba1");
    }

    public void funcion3(){
        System.out.println("prueba1");
    }
}
```

Imagen 63: Ejemplo control de acceso (Fuente propia)

Cuando analizamos esta clase:

Imagen 64: Detección control de acceso Sonar(Fuente propia)

El analizador nos advierte de que no se está utilizando la librería ESAPI. Como no se le ha asignado severidad, aparece como una nota informativa y por ello el testigo se muestra en verde como "Info". Se ha registrado así debido a que se pueden utilizar otras muchas librerías e incluso implementar los controles a mano.

Cross Site Request Forgery

Para simular un código vulnerable a CSRF, se ha creado una petición a un recurso, utilizando la anotación RequestMapping presente en Spring. Esta anotación facilita la tarea al programador, añadiéndola al principio de la función que tratará el recurso, el framework sabrá qué función ejecutar cuando un usuario intente acceder al mismo.

En este caso como la detección se basa en una lista negra de nombres hemos escogido uno de los nombres que contiene la lista para que la regla salte, como el analizador no ha encontrado la URL de reCaptcha en la vista asociada a la petición, arroja la alerta para que se incluya este elemento:

Imagen 65: CSRF Sonar (Fuente propia)

Como hay otras maneras de evitar este tipo de ataques, esta regla puede arrojar falsos positivos al analizar la vista asociada a la petición del recurso. Por ejemplo, si se incorpora un identificador único en un campo oculto (una de las soluciones propuestas por OWASP para este tipo de vulnerabilidades) el captcha quedaría fuera de lugar, aunque el analizador seguiría detectando la vulnerabilidad.

Vamos a hacer la prueba, si añadimos a la vista un identificador único en un campo oculto, el analizador seguirá arrojando la misma alerta que antes ya que lo que busca en la vista es la URL de reCaptcha:

```
<!DOCTYPE html>
<head>
    <title>VISTA FORMULARIO</title>
</head>
<body>
<div class="container">
    <div>
        <form action="/ad" method="post">
            <input type="hidden" id="fa34534isdaufhjsofijsao34211"/>
            <div>
                <label>Nombre</label>
                <div>
                    <input type="text"/>
                </div>
            </div>
            <div class="form-group">
                <label>Descripcion:</label>
                <div>
                    <input type="text"/>
                </div>
            </div>
            <div class="row">
                <button type="submit">Aceptar</button>
            </div>
        </form>
    </div>
</div>
</body>
</html>

```

Imagen 66: Vista con identificador único (Fuente propia)

Este identificador es ignorado, no está presente en la lógica de la regla, y por tanto una vez más será tarea del encargado de la seguridad del proyecto determinar si se puede producir una explotación o no, y en caso de que la respuesta sea sí, tomar las medidas necesarias para prevenir los ataques.

Por tanto, al volver a ejecutar el analizador una vez modificada la vista arroja el mismo resultado que antes:

Imagen 67: Falso positivo CSRF (Fuente propia)

4.2 Análisis de proyectos

Al analizar los proyectos mencionados anteriormente en el principio de esta sección, se observa que todos ellos pasan los test de calidad, se nota que se han utilizado herramientas de medición de calidad durante el desarrollo, o que se han adoptado muy buenas prácticas de programación de desarrolladores expertos.

El análisis se ha realizado mediante la herramienta sonar-scanner, simplemente hay que crear un fichero de propiedades y situarlo en la carpeta raíz de cada uno de los proyectos. El fichero es el siguiente:

```
joaquin@ubuntu:~/Project/workspace/TestRules$ cat sonar-project.properties
sonar.projectKey=casosprueba
sonar.projectName=Casos de prueba
sonar.projectVersion=1.0
sonar.sources=.

joaquin@ubuntu:~/Project/workspace/TestRules$
```

Imagen 68: Fichero "properties" (Fuente propia)

En este caso el fichero corresponde al proyecto utilizado para probar las reglas. Este fichero contiene los siguientes elementos:

- **projectKey:** Es el identificador único del proyecto para una instancia de Sonar, no puede haber dos proyectos con el mismo identificador.
- **projectName:** Es el nombre del proyecto que se mostrará en la consola de Sonar.
- **projectVersion:** Versión del proyecto.
- **sources:** Indica la ruta donde Sonar buscará el código fuente de la aplicación para parsearlo y analizarlo, lo normal es situar este fichero en la carpeta raíz, entonces el directorio donde Sonar tiene que empezar a buscar es el directorio actual.

Una vez creado el fichero de propiedades, lo único que hay que hacer para analizar el proyecto es ejecutar el escáner desde el directorio donde está localizado este fichero, en este caso desde la carpeta raíz del proyecto:

```
joaquin@ubuntu:~/Project/workspace/TestRules$ sonar-scanner
INFO: Scanner configuration file: /home/joaquin/Project/sonar-scanner-2.8/conf/s
onar-scanner.properties
INFO: Project root configuration file: /home/joaquin/Project/workspace/TestRules
/sonar-project.properties
INFO: SonarQube Scanner 2.8
INFO: Java 1.8.0_121 Oracle Corporation (64-bit)
INFO: Linux 4.10.0-33-generic amd64
INFO: User cache: /home/joaquin/.sonar/cache
INFO: Load global repositories
INFO: Load global repositories (done) | time=249ms
INFO: User cache: /home/joaquin/.sonar/cache
INFO: Load plugins index
INFO: Load plugins index (done) | time=9ms
INFO: SonarQube server 5.6.6
INFO: Default locale: "en_US", source code encoding: "UTF-8" (analysis is platfo
rm dependent)
INFO: Process project properties
INFO: Load project repositories
INFO: Load project repositories (done) | time=221ms
```

Imagen 69: Sonar scanner (Fuente propia)

Si el escáner se ejecuta correctamente, el programa lanzará una traza indicando que el análisis ha sido exitoso, indicando también el total de memoria consumido y el tiempo total de ejecución:

```
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 9.509s
INFO: Final Memory: 44M/157M
INFO: ------------------------------------------------------------------------
joaquin@ubuntu:~/Project/workspace/TestRules$
```

Imagen 70: Sonar Scanner (Fuente propia)

Cabe destacar que para que el escáner pueda trabajar necesitamos que la instancia de Sonar en la que queremos observar el proyecto esté levantada, por defecto el escáner buscará la instancia en la máquina local en el puerto TCP 9000.

Una vez realizado este proceso para cada uno de los proyectos a analizar, simplemente navegamos hacia el cuadro de mando de Sonar, donde podremos observar los indicadores de calidad para cada uno de los proyectos:

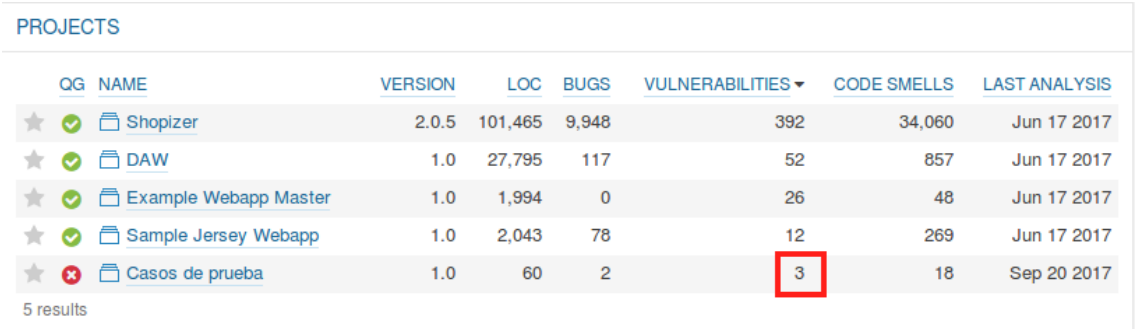

Imagen 71: Dashboard proyectos Sonar (Fuente propia)

En el caso del proyecto “Casos de prueba” solo aparecen tres vulnerabilidades ya que a la regla de control de acceso no se le ha asignado severidad, simplemente es informativa.

A simple vista se puede ver que todos los proyectos han pasado las pruebas de calidad excepto en el proyecto que se ha creado a propósito para que las reglas salten.

En el dashboard principal de Sonar se pueden observar la versión (indicada en el fichero de propiedades), las líneas totales de código y el número de fallos que se han encontrado categorizados según su tipo: Bugs, Vulnerabilities y Code Smells. Por supuesto aquí no solo actúan nuestras reglas si no todas las reglas contenidas en el perfil “SonarQube Way”. Se ha decidido incluir las reglas creadas en este perfil porque ya contiene muchas reglas interesantes y así se puede ofrecer una visión un poco más avanzada de cómo funciona la herramienta. Si se quiere observar la deuda técnica de un proyecto, hay que pinchar en el mismo y nos dará un resultado más detallado.

A continuación, se muestran los resultados para cada proyecto:

DAW

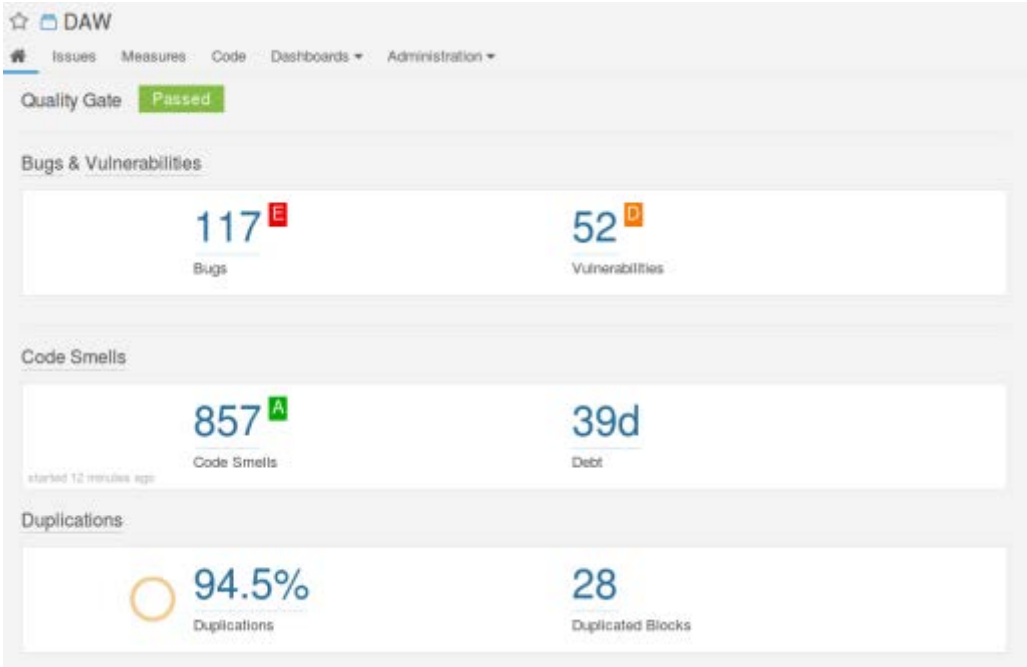

Imagen 72: Resultado del análisis de Diseño de Aplicaciones Web (Fuente propia)

Example Webapp Master

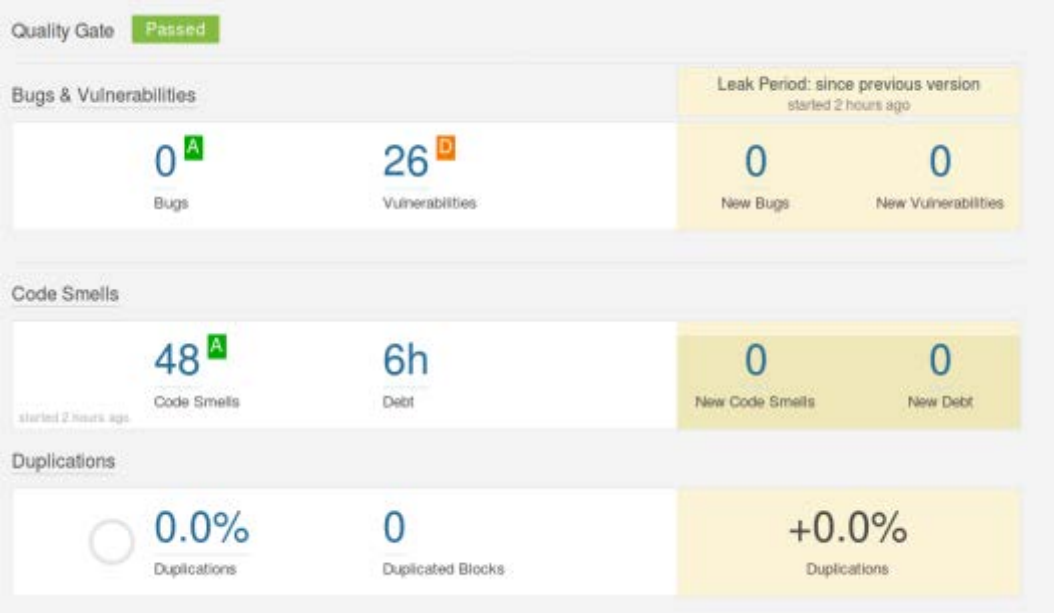

Imagen 73: Resultado del análisis de Example Webapp Master (Fuente propia)

Sample Jersey Webapp

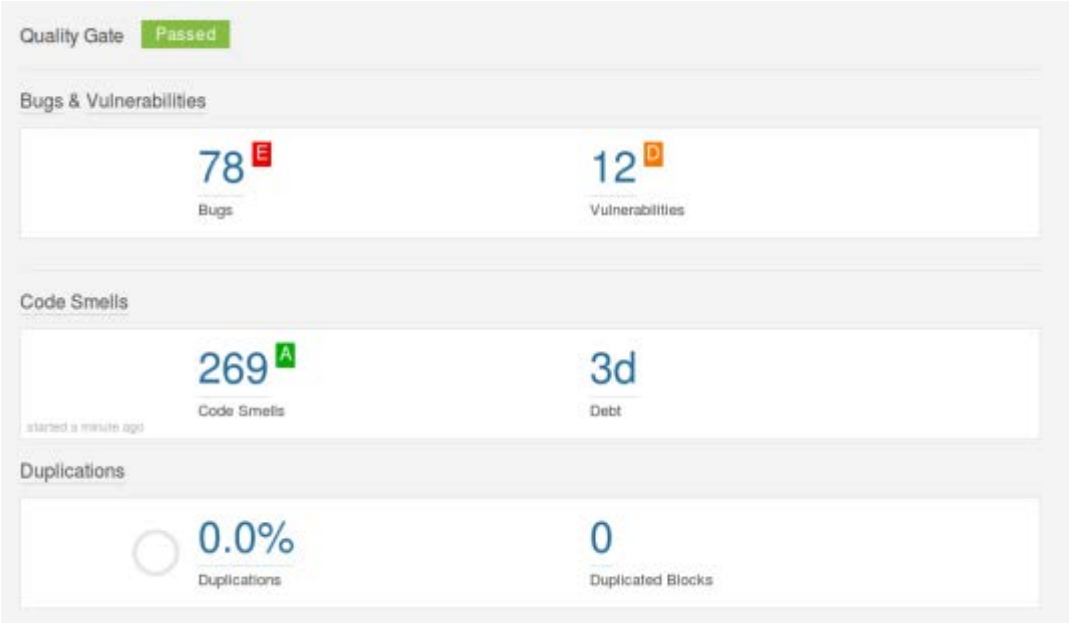

Imagen 74: Resultado del análisis de Sample Jersey Webapp (Fuente propia)

Shopizer

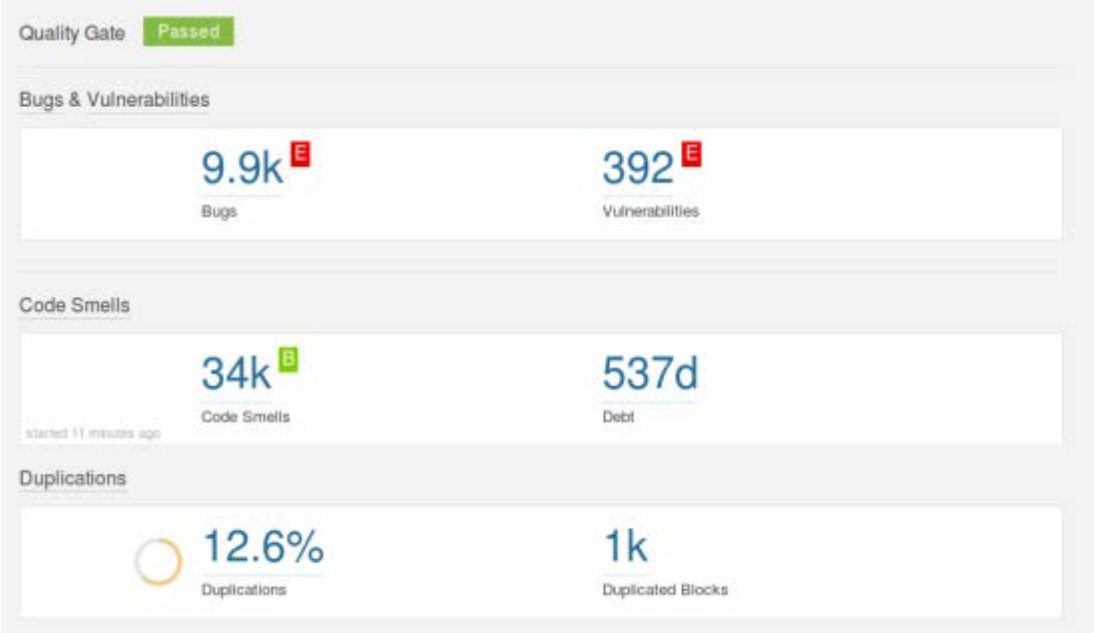

Imagen 75: Resultado del análisis de Shopizer 2.0.5 (Fuente propia)

Estos indicadores ofrecen una visión de la deuda técnica que tiene un proyecto. Sonar realiza una aproximación de las horas que llevaría solucionar toda la deuda técnica del proyecto, se puede ver en la sección “Debt”. Por supuesto este número puede diferir mucho de la realidad ya que hay muchos fallos que los analizadores considerarían falsos positivos o que no tendrían mucha importancia.

Lo ideal sería centrarse en fallos concretos, por ejemplo, empezar por los bugs de programación, seguido de las reglas de mantenibilidad, y una vez solucionados o ignorados esos fallos pasar a las reglas de seguridad. Es impensable solucionar todos los fallos de golpe, sería un dolor de cabeza para el personal dedicado al proyecto.

4.3 Evaluación de la solución

Los resultados de la solución implementada han sido bastante satisfactorios para el caso de SQL Injection y XSS, pero no para CSRF. Esto puede ser debido a que los proyectos han utilizado otro método de protección que no sea un captcha, como un identificador. Esta regla habría que mejorarla en futuras implementaciones de la solución para enfocar la detección de otra manera ya no ha detectado correctamente ningún fallo en ninguno de los proyectos.

A continuación, se muestra una tabla con el número total de fallos detectados por las reglas sobre todo el conjunto de proyectos y la tasa global de falsos positivos:

Regla	Ocurrencias	Falsos positivos	% FP
SQL Injection	43	9	20.93%
XSS	26	7	26,92%
Control de acceso	4	2	50%
CSRF	9	9	100%
Paquete conjunto	80	25	31,25%

La tasa de aciertos del paquete de reglas propuesto es la cual puede no parecer muy buena, pero teniendo en cuenta lo difícil que es la detección de estos fallos desde un analizador, y siendo la primera versión de las mismas, es un resultado bastante interesante.

En la siguiente gráfica se muestra el porcentaje de aciertos de cada una de las reglas, siendo SQL Injection la más exitosa en detección:

Esto puede deberse a que muchos desarrolladores no conocen la clase PreparedStatement, y no la utilizan. Aparte de poder acabar en fallos de SQL Injection si las entradas no son bien validadas, el rendimiento es mucho peor como hemos podido ver en las características de esta clase.

En cuanto a la regla de XSS, es común la mala validación de entradas por parte de los desarrolladores, ya sea por querer entregar rápido el producto o por no utilizar herramientas externas que prevean este tipo de fallos. Algunos de los fallos de XSS se producen en los campos ocultos de los formularios, también es importante validar estos campos ya que también son una entrada y pueden ser modificados desde la parte del cliente.

5. Conclusiones

Como conclusión, creo que se han cumplido las metas que tenía en mente para este trabajo. Se ha intentado dar una visión clara del funcionamiento de las herramientas de análisis estático de seguridad y se han mostrado los puntos débiles de estas herramientas, como son la dificultad para detectar cierto tipo de fallos o la curva de aprendizaje que muchas de ellas presentan para los desarrolladores y auditores de seguridad de código fuente.

Creo que la mayoría de las empresas esperan demasiado tiempo para ejecutar los análisis de seguridad en sus aplicaciones. Es necesario incluir análisis de seguridad desde el inicio de la vida del software, el desarrollo. Incluir la seguridad en el SDL (Secure Development Lifecycle) es la forma más adecuada de solucionar los problemas, si se espera al final del desarrollo los fallos se podrían convertir en daños económicos para las compañías y el precio de solucionarlos es siempre más caro si los fallos son detectados a posteriori.

Ha faltado mejorar detalles y me gustaría haber tenido más tiempo para perfilar la regla de CSRF, así como otros pequeños puntos como incluir algún apartado de elaboración de informes mediante SonarQube para ofrecer una visión más amplia de la herramienta que es bastante extensa y compleja, aquí solo se ha mostrado una pequeña parte de las capacidades de Sonar. También me hubiera gustado incluir una regla para buffer overflows, pero al trabajar con un lenguaje de tan alto nivel como Java, es imposible tratar estos asuntos porque están a un nivel al que no se puede llegar debido a la gran abstracción de la memoria que nos presenta el hipervisor de Java.

En futuras implementaciones y mejoras de esta solución sería interesante sobre todo mejorar la regla de CSRF para que se adapte de forma más genérica a los proyectos y no dependa tanto del uso de ese captcha en concreto, ya que es muy probable que se haya incluido otro tipo de protección y entonces la regla queda inservible, como se ha podido observar en el análisis de estos cuatro proyectos. El desarrollo de este proyecto me ha sido de gran utilidad para aplicar los conocimientos que he adquirido tanto durante la carrera como en mi corta vida laboral. Creo que es muy interesante observar los avances que se van haciendo y cómo cada vez los conceptos se comprenden y relacionan mejor. Me parecería interesante que en la carrera de ingeniería informática se estudiaran más temas de seguridad, ya que la única asignatura relacionada que yo he cursado ha sido Criptografía. Este proyecto creo que se podría relacionar con las asignaturas de desarrollo de software ya que es la etapa de desarrollo donde se analizaría el código mediante las herramientas y soluciones detalladas en este documento.

El sector de la tecnología es muy extenso, hay muchos entornos distintos y cada uno de ellos puede ser muy distinto del resto. Posiblemente en un futuro la ingeniería informática como la conocemos hoy en día se dividirá en muchas ramas diferentes y surgirán nuevas carreras, de hecho, muchas de ellas están apareciendo en los últimos años como Big Data, etc. Creo que en algún momento alguna carrera relacionada con

la seguridad de las tecnologías de la información también tendrá su hueco entre las universidades.

6. English summary

Source code security analytics

6.1 Introduction

Nowadays, information is the most appreciated asset for any company. That's because correctly used, information leads to a big knowledge which is useful for a lot of purposes. Most companies, regardless their size, are focusing their efforts in data mining for economic benefits or for any kind of advantage. This is becoming popular and we can see it everywhere: Amazon is predicting its customer's purchases… besides, governments are using data mining for massive spying on citizens.

This environment is perfect for the IT security growth, it has become the most popular professional field in the last decade along with data mining. According to a research, last year 300.000 cybersecurity employments were unfilled, this gives us a big idea on the growing rate in the IT security field. Information must be protected correctly to avoid data leaks. Because of this, companies are expending lots of money finding and hiring the best security experts from all over the world.

Any security breach is important, security measures must be applied since the very first day, and that's what this is about, implementing security measures from the beginning: the software development. Companies may expend lots of money in extra-security measures like firewalls, IDS and IPS systems, Antivirus software, etc. but the truth is, if the software is vulnerable, any of those systems will be immediately useless and an attacker will be able to access, crash or make any kind of damage which could lead to company losses in the best scenario.

The most important reason for me to make this project is that I am focusing my career in cybersecurity. I think this is an interesting field, and is becoming popular in these days because attacks are more sophisticated each time. I have also applied knowledge from my current employment as a security operator in this project.

The main goal in this project is to construct rules which can detect security bugs in applications source code, focusing in the main languages for application development: Java and JavaScript. The idea is to make some checks in source code, to make sure that it meets some custom security rules. These rules are implemented based on the most important security bugs globally, trying to cover well-known vulnerabilities such as SQL Injection and Cross Site Scripting (XSS).

Rule development has been implemented in SonarQube, an open source tool that statically analyses code. This means it analyses the code without executing it, based only on lexical symbols and compiler techniques. To be successful in producing quality code which meets all the security standards dynamic analysis is also needed, but this project is only focused in static code analysis.

So, the information we are going to find in this document is related to the following points:

- Security standards
- Implementation of security rules

- Vulnerable code examples
- Solution to vulnerabilities in code
- Security reports

6.2 State of the art

Before diving into vulnerabilities and code issues, we are going to review a few important concepts in the cybersecurity field. First, the key elements on any IT security project:

- **Availability**: any authorized user must have access to a resource when needed, required assets should be available at any time the user wants to access them. To ensure availability, a wide variety of tools are used, including Anti-DDoS tools.

- **Integrity**: information as any process associated to it must be complete and accurate. Integrity guarantees information hasn't been modified by an untrusted third party using, for example, digest algorithms such as SHA-256.

- **Confidentiality**: information assets only must be accessed by authorized users. Here come in play cipher algorithms, used to protect data in transit and statically stored.

Most common consequences caused by successful exploitation on any of the previous elements are:

- **Interrupt**: service stops because of a successful vulnerability exploitation. Resources become unavailable until a solution or mitigation is applied by security administrators. An example would be a DDOS attack.

- **Interception**: an attacker has gained access to non-authorized data. The attack to SONY would be an example of interception.

- **Modification**: Stored information has been altered by an untrusted third party and isn't valid anymore. A recent example is Donald Trump's web deface.

- **Fabrication**: creation of a product with the purpose of steal confidential information. Usually occurs when an attacker takes control of the DNS system, in consequence, users are redirected to an attacker-controlled domain.

6.2.1 SAST (Static Application Security Testing)

Source code static analysis is usually accomplished as part of code review, also known as white box testing. A white box test is a security test where the infrastructure to test as any asset included in it is well known by the auditors, and it is performed in the implementation phase of a SDL (Security Development Lifecycle).

This static analysis usually refers to execution of automated tools to highlight possible issues in static code (code that is not being executed) using the below techniques. These techniques are combined to provide a single static security analysis solution.

- **Data flow analysis**: It's used to recollect execution time information as the code remains in a static state. A basic block is a sequence of instructions which cannot be altered once executed.

- **Flow control graph**: Abstract representation of basic blocks using graphs. Each node connection represents a jump from a basic block to another.

- **Taint analysis**: This analysis tries to identify user input variables and track them until a vulnerable function that uses them is found. If the variable is passed as a function parameter without being check first this will be considered as a vulnerability.

- **Lexical analysis**: This is one of the main compiler techniques used to convert source code to machine code. It transforms source code to tokens in order to abstract concepts and make them easier to put together.

6.2.2 Vulnerabilities

A vulnerability is an issue or weakness in software which allows an unintended use and can cause further damages in an entire system or network. It can be identified by five unique factors:

- **Product**: Software affected by the vulnerability. There can be some versions of the same software affected while others not.

- **Where**: This refers to the concrete vulnerable module which resides inside the affected software.

- **Causes and consequences**: Defines which is the origin of the problem, the error behind the vulnerability and the impact if successfully exploited.

- **Impact**: measure of the damages that a potential attacker may cause if the security error is triggered**.**

- **Attack vector**: Defines the way an attacker may use to exploit the vulnerability. The most common vector is the delivery of specially crafted requests to a concrete port of a remote server.

Depending on the way vulnerabilities affect to systems in a determinate company, they can be classified into three categories:

- **Known vulnerabilities on installed systems**: the company creator of the affected product is aware of the vulnerability and a patch or update is released to address that issue.

- **Known vulnerabilities on non-installed systems**: these are also known by the software creators, but if the concrete company doesn't use that technology it won't be affected.

- **Not yet known vulnerabilities or zero-day vulnerabilities**: these are the most dangerous security vulnerabilities because they are not known yet by anyone. If an attacker discovers a new vulnerability will likely exploit it in order to obtain benefits.

Most security bugs in systems fall into the next five categories:

- **Buffer overflows**: occurs when an application try to write data past the end (or sometimes before the beginning) of a buffer. This kind of error may compromise data, provide an attack vector to escalate privileges or directly crash a system.

- **Input validation**: as a general mandatory rule, all input and output must be validated before using it in an application environment to be sure that the data is legit. Any non-validated input or output is a potential attack vector into the application.

- **Race conditions**: it happens when the order of events is relevant. If an attacker can change the behaviour of the application by modifying the event sequence, this is considered a vulnerability.

- **Access control**: lots of vulnerabilities have the same origin: bad use or misconception of access control. This refers to bad configuration of privileges in resources.

- **Weaknesses in implementation of authentication, authorization or cryptographic algorithms**: authentication, authorization and cryptographic functions provide an extra security layer to systems and applications if correctly used. Bad configuration of these mechanisms could lead to big damages and risks.

We may classify vulnerabilities by its severity into four categories:

- **Critical**: It affects system's availability, integrity or confidentiality and requires immediate attention to be solved. No solution would end in economic losses or damages in the brand's image.

- **Important**: May affect system's availability, integrity or confidentiality. If successfully exploited it would affect companies the same way as critical vulnerabilities.

- **Moderate**: Impact could be assessed by applying certain configurations or through security audits. It is hard to exploit and requires advanced skills to take advantage of the flaw.

- **Low**: If exploited, impact would be minimum, and they require high skills as moderate vulnerabilities. This doesn't mean they cannot harm the systems, but they are difficult to exploit.

6.3 Technical solution design

This section describes the vulnerabilities included in the final solution and the tool used to achieve our goals: detect and alerting security flaws in code.

6.3.1 SonarQube

SonarQube (also known as Sonar) is a multiplatform open source tool distributed under GPL license. Its main objective is continuous code inspection and source code evaluation. It uses a wide variety of static analysis tools to obtain code metrics, such as are CheckStyle, PMD or FindBugs.

Its main features are the following:

- Clean source code
- Bug detection
- Multilanguage
- DevOps Integration
- Quality code centralization

Sonar can be integrated in a development project in various ways: it can be downloaded to be executed locally in a private network, integrated in IDEs like Eclipse or Netbeans and some others. Although Sonar isn't focused only in security rules, it offers a very simple way to include custom rules, that's why it has been chosen as the SAST tool for this project. Once custom security rules are implemented it can be used for continuous security code inspection, perfect for continuous integration software projects.

Quality model is focused on three main different types of rules:

- Reliability
- Maintainability
- Security

Standard for rules is very strict and its main objective is to obtain zero false positives in an analysis. This is not possible in practice because of code complexity and many other factors which are discussed in this document. The idea is that rules help developers and security auditors to find security issues. It's the task of the human to verify if the triggered rule is a false positive or not.

Most security rules included in Sonar follow the world's most important standards like OWASP, CWE, etc.

The main reasons for choosing Sonar as a SAST tool is that it is free and open source and it offers a very simple of writing custom rules, in this case security rules. You can make your own rule profile to include a set of rules to meet certain needs. It allows to integrate an unlimited number of rules and it's very straightforward.

Custom rule implementation has been done in Java. Once the rules have been implemented, project is built and a JAR file is generated, then this package of rules is integrated in Sonar by only moving it to a certain folder inside analyser's root directory.

6.3.2 Vulnerabilities covered by the solution

It has been difficult to find vulnerabilities to cover with custom rules. It's difficult for SAST tools to find this kind of issues only looking at lexical symbols. Code is not executed, so behavioural analysis must be a mandatory complement to this analysis to produce legacy code.

The designed solution covers the main vulnerabilities in Web applications, recognised by the OWASP Top 10 security project which is powered by security experts from all over the world. These are:

SQL Injection

A SQL Injection flaw consists in arbitrary code execution on a DBMS through non-validated or incorrectly validated input fields. Vulnerability is caused by an incorrect check or filtering of input variables from the end user. This vulnerability has been included in the solution because is one of the most important vulnerabilities from the last years.

In Java, there are two main classes for handling interaction with a DBMS. Its main features are detailed below.

Statement:

- Used for standard SQL queries
- Used when a certain query is to be executed only once
- Query parameters aren't allowed
- Mainly used for DDL queries

- Low efficiency

PreparedStatement:

- Used for dynamic or parametrized SQL queries
- Preferred when a query is to be executed multiple times
- Query parameters are allowed
- Used for any query that will be executed repeatedly
- Performance is better than in Statement because query is precompiled and only parameters are dynamic

It seems that PreparedStatement is the correct class to use when implementing SQL queries against a DBMS in a web application environment. It is common in web application to make repeated queries, for example in a login, search form, etc. So that's why

The objective of the implemented rule is to find where the Statement class is used, and recommend the developer not to use this class, as using it could lead to SQL Injection flaws in most cases.

XSS

Cross Site Scripting is a security flaw which allows injection of JavaScript code in web applications bypassing existing control measures. By exploiting this vulnerability, user's trust to a certain site is exploited. It allows an attacker to execute malicious scripts in the victim browser and could cause damages like: stolen sessions through stolen cookies, defacing, attacker's site redirections…

This attack can be performed in two different ways:

- Reflected: Modifying values that the application uses to communicate between pages. Attacker could steal user cookies and consequently the identity of that user.

- Stored: Modifying HTML code in an application where incorrect on inexistent validation is in place. Attacker could deface the page and all users will see it because attacker's code is stored on the server.

To implement this rule JSoup has been used. JSoup is a Java library which allows to work with real HTML. Its main features are the following:

- Analyse HTML code from a URL, file or String
- Find and extract data using DOM or CSS
- Manipulate HTML objects, attributes and text
- Clean user input
- Ordered HTML output

The feature we have used for implementing this rule is fourth. JSoups is able to parse user input through the use of the clean function and a Whitelist of allowed characters. Implemented rule detects if a parameter is passed to a servlet, and then if JSoup is not used, an alert will be triggered and the developer will be aware of a possible XSS flaw in his application.

CSRF

Cross Site Request Forgery is an attack which forces an end user to execute unwanted actions in a web application. Victim must have a session open in that web application for this attack to work. This type of attacks isn't used to steal information because attacker is not able to see the response from the server. Instead, it is used for very specific tasks like virtual money transfers. If the victim is an administrator of the vulnerable application this attack could compromise the entire site.

OWASP recommended solutions to this flaw are:

- Unique identifier in hidden fields or URLs
- Verify origin and destiny in headers
- Embed a captcha to verify state change requests

The third option has been the desired approach to implement the CSRF detection rule. reCaptcha (Google's captcha) has been the desired technology to correct this flaw. The rule itself detects application's state change requests through a blacklist of common names which could be modified to suit certain company's needs. Once a state change request is identified, rule checks if the associated view contains a captcha, and if it doesn't, rule is activated and the developer will see an alert for this vulnerability.

6.4 Results and evaluation

Selected projects for analysis have been obtained from GitHub, one of the most important source code communities around the world. An external project from an UPM scholar has been also included in the analysis, it has been obtained directly from his author David Marquez Delgado.

The following lists includes the four projects analysed with the designed solution:

- DAW, final university project from an UPM scholar
- Example Java Webapp from GitHub
- Shopizer 2.0.5 e-commerce site from GitHub
- Sample Jersey Webapp, from GitHub

Note that all analysed projects are web application projects. That's because the rules implemented are related to well-known web application security flaws and it would be weird to execute web vulnerability analysis against non-web code.

Results have been successful for the SQL Injection rule and XSS rule, but not for CSRF. This could be because another protection mechanism is in place for CSRF for example an identifier in the URL. This rule must be improved in next implementations to identify this kind of flaws in another way, because it hasn't been successful in any of the projects analysed.

Below is a table showing rule's activation count and false positive rate. This numbers are obtained from the analysis of the four projects:

Rule	Total	False positives	% False Positives
SQL Injection	43	9	20.93%
XSS	26	7	26,92%
Access control	4	2	50%
CSRF	9	9	100%
Rule package	80	25	31,25%

SQL Injection rule has been successful in a 79.07 %. This could be because programmers aren't aware of the dangers regarding the Statement Java class. I think a lot of developers are still using this class as usual, and it's his responsibility not to use it, as it could lead to critical security flaws like the whole database leaked.

XSS rule has been also useful in a 73,08 %. Most XSS flaws are caused by non-validated or bad validated hidden input fields. This kind of input must be also validated in order to produce secure code.

The overall tool success is 68,75%. This may seem a low success rate, but it is very good taking into account the difficulty of finding security flaws statically in source code.

6.5 Conclusions

As a final thought, I believe all the main objectives on this project have been met. I tried to offer a clear vision on the performance of SAST tools and I shown the main weaknesses of this tools, like the difficulty to find bugs and the learning curve that some of them present to developers and source code security auditors.

I think most companies wait too long to introduce security in their environments. Security analysis is needed from the beginning: the software development. Including security in the Software Development Lifecycle is the most appropriate way to solve the problems without expending too much money. If companies wait until the software is in its production stage, it will be a headache to find the root cause of problems.

I would have liked to improve some details like profiling the CSRF rule to be more accurate and precise. In future implementations and improvements, it would be interesting to improve CSRF rule, and some other things like including a point for Sonar

reports, and a rule for buffer overflows. This last point is more complicated in Java because in this language you work at a very high level, abstracting you from the memory allocation and that kind of tasks.

Development of this project has been very useful for me to apply knowledge I have acquired through my career and professional life. I think it's important to see the advances you make and how concepts become easier to understand each time. It would be interesting to include more security subjects in the computer engineering degree, because the only subject I have had related to security was cryptography. This project may be more related to software engineering as it is in the software development phase where it should be implemented.

7. Bibliografía

Libros

- *Chess, Brian; West, Jacob. "**Secure programming with static analysis**", First Edition McGraw, 2007*

- *Mueller, John Paul. "**Security for Web developers**", First Edition O'Reilly Media Inc, 2015*

- *Charalampos S. Arapidis. "**Sonar Code Quality Testing Essentials**", First Edition Packt publishing, 2012*

- *Y.E, Liang. "**JavaScript Security**", First Edition Packt publishing, 2014*

Enlaces

-*Neelam, Jain. "**Security vulnerabilities in Java-based web applications**"* https://www.3pillarglobal.com <https://www.3pillarglobal.com/insights/security-vulnerabilities-java-based-web-applications>. *2014*

-*Pankaj, "**JDBC Statement vs PreparedStatement – SQL Injection Example**"* http://www.journaldev.com <http://www.journaldev.com/2489/jdbc-statement-vs-preparedstatement-sql-injection-example>. 2014

-*INTECO, "**¿Qué son las vulnerabilidades del software?**"* http://www.jesusamieiro.com <http://www.jesusamieiro.com/wp-content/uploads/2011/08/Que_son_las_vulnerabilidades_del_-software.pdf>.

-*UNED, "**Introducción al análisis automático de la seguridad de aplicaciones web**"* http://www.scc.uned.es <http://www.scc.uned.es/jornadasmaster/pdf/Charla2.pdf>. 2012

-*UNAM, **"Capítulo 2. Amenazas y vulnerabilidades de la seguridad informática"*** http://www.ptolomeo.unam.mx:8080 <http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/217/A5.pdf?sequence=5>. 2012

-*Miguel Ángel Mendoza, **"Vulnerabilidades: ¿qué es CVSS y cómo utilizarlo?"*** https://www.welivesecurity.com <https://www.welivesecurity.com/la-es/2014/08/04/vulnerabilidades-que-es-cvss-como-utilizarlo/>. 2014

-*Apple Inc, **"Types of security vulnerabilities"*** https://developer.apple.com<https://developer.apple.com/library/content/documentation/Security/Conceptual/SecureCodingGuide/Articles/TypesSecVuln.html>. 2016

-*Cassius Puodzius,* ***"4 familias de ransomware con errores de cifrado"***
https://www.welivesecurity.com <https://www.welivesecurity.com/la-es/2016/09/14/ransomware-errores-de-cifrado/>. 2016

-*Ignacio Bruna,* ***"Confidencialidad, disponibilidad e integridad de la información"***
http://www.belt.es <http://www.belt.es/expertos/experto.asp?id=2245>

-*Advisera,* ***"¿Qué es la norma ISO 27001?"***
https://advisera.com <https://advisera.com/27001academy/es/que-es-iso-27001/>. 2014

-*OWASP,* ***"Source Code Analysis Tools"***
https://www.owasp.org
<https://www.owasp.org/index.php/Source_Code_Analysis_Tools>. 2017

-*Buguroo,* ***"Bugscout: El analizador estático de Buguroo"***
https://buguroo.com <https://buguroo.com/es/bugscout-el-analizador-estatico-de-codigo-de-buguroo>. 2016

-*CWE,* ***"Common Weakness Enumeration"***
https://cwe.mitre.org <https://cwe.mitre.org>. 2017

-*Testeando Software,* ***"Kiuwan, ¿qué es?"***
https://testeandosoftware.com <https://testeandosoftware.com/kiuwan-que-es/>. 2015

-*MicroFocus,* ***"Fortify Static Code Analyzer"***
http://www8.hp.com <http://www8.hp.com/es/es/software-solutions/static-code-analysis-sast/>. 2017

-*FindBugs, "****Find Bugs in Java Programs****"*
http://findbugs.sourceforge.net <http://findbugs.sourceforge.net>. 2015

-*Ignacio Pérez,* ***"Comprendiendo la vulnerabilidad XSS (Cross-site Scripting) en sitios web"***
https://www.welivesecurity.com <https://www.welivesecurity.com/la-es/2015/04/29/vulnerabilidad-xss-cross-site-scripting-sitios-web/>. 2015

-*Sonar,* ***"Writing custom Java rules 101"***
https://docs.sonarqube.org
<https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101>. 2017

-*OWASP,* ***"Static code analysis"***
https://www.owasp.org <https://www.owasp.org/index.php/Static_Code_Analysis>. 2017

-Sonar, ***"Security-related rules"***
https://docs.sonarqube.org <https://docs.sonarqube.org/display/SONAR/Security-related+rules>. 2017