UNED

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

Desarrollo de software seguro - COD. 31105151

Procedimientos de actuación para evitar vulnerabilidades de software

Autor: José Antonio Amores Durán.

Director: José Antonio Cerrada Somolinos.

Curso Académico 2018/2019.

Convocatoria: febrero.

UNED

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

Desarrollo de software seguro - COD. 31105151

Tipo de trabajo B - Procedimientos de actuación para evitar vulnerabilidades de software

Autor: José Antonio Amores Durán.

Director: José Antonio Cerrada Somolinos.

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MÁSTER

Fecha: 23/01/2019

Quién suscribe:

Autor: José Antonio Amores Durán D.N.I/N.I.E/Pasaporte.: 47469831Q

Hace constar que es el autor del trabajo:

Título completo del trabajo. Procedimientos de actuación para evitar vulnerabilidades de software.

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

Impreso TFdM05_AutorPbl. Autorización de publicación
y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Resumen

Hoy en día el software ha tomado mucha importancia en nuestras vidas cotidianas, podemos encontrarlo prácticamente en todas partes, en vehículos, cajeros automáticos, en las aplicaciones web a través de las cuales hacemos la compra o buscamos información en la red y en muchos sitios más.

Dada esta importancia es necesario tener en cuenta que el software puede tener multitud de vulnerabilidades que pueden provocar un mal funcionamiento de los dispositivos o de las aplicaciones web donde esté implementado y comprometer tanto nuestros datos personales como nuestra seguridad.

Este proyecto pretende realizar un estudio de las principales vulnerabilidades de software existentes, con el objetivo de crear unas pautas que se deberán tener en cuenta durante todas las fases de desarrollo del código para evitar que se cometan dichos errores.

Nuestro objetivo principal será crear una serie de procedimientos o actuaciones que nos permitirán desarrollar nuestro código de forma segura para evitar en la medida de lo posible las vulnerabilidades de software.

Palabras clave: aplicación web, vulnerabilidad de software, desarrollo de software seguro.

Índice

Contenido

Listado de ilustraciones y tablas

Ilustraciones

Tablas

Introducción

Hoy en día el software ha tomado una gran importancia en nuestras vidas, ya que podemos encontrarlo prácticamente en todas partes. Está presente en nuestro día a día y lo utilizamos constantemente, incluso para operaciones que tienen una gran trascendencia, como pueden ser las transacciones bancarias, operaciones quirúrgicas, control de aeronaves, vehículos autónomos, etc. Dada la importancia que el software ha tomado en nuestra sociedad es necesario tener en cuenta que también puede presentar vulnerabilidades y fallos que pueden provocar un mal funcionamiento y que pueden dar lugar a errores importantes, desde comprometer nuestros datos personales hasta poner en riesgo nuestra seguridad.

También deberemos tener en cuenta que las técnicas de piratería han ido evolucionando con los años, y con estas las vulnerabilidades también han ido aumentando gradualmente. Cada año se registran nuevas vulnerabilidades de software, y la tendencia sigue aumentando. Sin embargo, la respuesta a estas vulnerabilidades hoy en día se basa en el análisis manual, provocando que la detección de esas vulnerabilidades no se realice a tiempo. Por este motivo es necesario desarrollar técnicas que puedan, por ejemplo, detectar y parchear vulnerabilidades automáticamente.

Muchas de las tecnologías y metodologías que presentemos en este documento pueden ser extrapoladas a cualquier tipo de desarrollo de software.

Nuestro objetivo es establecer unas metodologías que nos ayuden a detectar y evitar las vulnerabilidades de software que surjan durante todas sus fases.

Para este estudio nos hemos basado en nuestra experiencia profesional desarrollando software de diferentes tipos, aplicaciones web, aplicaciones de escritorio, servicios RESTFul, etc. De esta forma podremos aplicar a nuestra propia experiencia personal los análisis y estudios que vamos a realizar en este documento con el objetivo de mejorar nuestro código con nuevos métodos y técnicas que aún no conocemos, así como aportar las metodologías que ya conocemos y que consideramos útiles e indispensables para realizar un software de calidad.

Nuestro objetivo es aplicar metodologías que eviten vulnerabilidades y errores de software de la forma más genérica posible, es decir, que sean aplicables a cualquier lenguaje de programación y/o arquitectura de software. En nuestros desarrollos hemos implementado, aplicaciones con una arquitectura SOA, basada en Spring MVC y que utilizaban API RESTFul como protocolo de comunicación entre los diferentes servicios, aplicaciones de escritorio que tenían un propósito de análisis de muestras biométricas y que utilizaban el lenguaje de programación C#, aplicaciones móviles desarrolladas en Android, así como muchas más. A pesar de ser desarrollos software muy diferentes entre ellos tenían una cosa en común, las vulnerabilidades de software que podíamos implementar, y que podían llegar al producto final provocando errores de funcionamiento.

Si diseñásemos una metodología que nos ayudase a reducir el número de vulnerabilidades de software independientemente de la plataforma, lenguaje de programación y arquitectura y la aplicásemos a todos nuestros desarrollos, conseguiríamos reducir el número de vulnerabilidades que llegan a la fase final del desarrollo.

Estado del arte

En este apartado se introducirán las diferentes tecnologías y metodologías existentes para la detección de vulnerabilidades, del mismo modo nos parece interesante presentar las vulnerabilidades de software más comunes que podemos encontrarnos.

Principales vulnerabilidades de software y reglas de seguridad

Este apartado tiene como objetivo mostrar las vulnerabilidades más comunes que podemos encontrarnos durante el desarrollo de nuestro software, ya que uno de los mejores métodos para evitarlas es conocerlas, para identificar dichas vulnerabilidades nos basaremos en el estudio realizado por OWASP.

OWASP es un proyecto de código abierto cuyo objetivo principal se centra en la seguridad de las aplicaciones web. Periódicamente la comunidad OWASP desarrolla un documento en el que se abordan los fallos de seguridad más importantes que afectan a las aplicaciones y a los que se enfrentan las organizaciones que las utilizan actualmente. Este documento se basa en los datos obtenidos de empresas especializadas en seguridad de aplicaciones. A través de este informe se estudian las vulnerabilidades recopiladas por multitud de organizaciones, aplicaciones y APIs que se encuentran en uso actualmente.

OWASP Top 10 nace con el objetivo de instruir a la comunidad del software (desarrolladores, arquitectos de software, personal de pruebas, etc.) sobre las vulnerabilidades más destacadas y frecuentes, y que ponen en riesgo la seguridad de las aplicaciones web. Este informe nos aporta una serie de técnicas que nos ayudarán a protegernos de estos problemas de seguridad.

Tal y como la propia fundación indica en OWASP Top 10 -2017: “OWASP Top 10 aborda los riesgos de seguridad de aplicaciones más impactantes que enfrentan las organizaciones en la actualidad.”

Informe OWASP

Riesgos de seguridad en aplicaciones web

Los atacantes pueden utilizar diferentes rutas a través de nuestra aplicación con el objetivo de explotar sus vulnerabilidades y utilizar dichos errores en su beneficio. En la siguiente imagen, la cual hemos obtenido del informe OWASP Top 10 -2017, podemos observar diferentes caminos, los cuales representan riesgos que pueden requerir acciones de mitigación o no, dependiendo de la gravedad del riesgo y de las repercusiones que supongan para nuestro software.

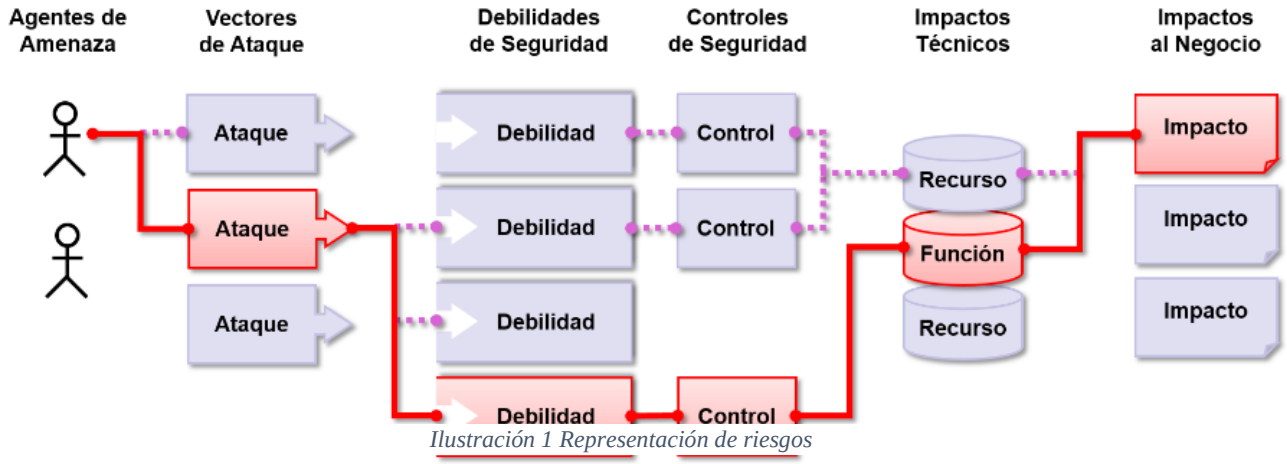


Ilustración 1 Representación de riesgos

Ilustración 1: Representación de riesgos

Inyección

Un ataque de inyección de comando pretende inyectar y ejecutar comandos que han sido especificados por un atacante en un software específico. La ausencia de una validación de los datos de entrada en una aplicación es la principal causa de este tipo de vulnerabilidades, ya que al no existir una validación un atacante puede manipular los datos que introduce a través de formularios, cookies, encabezados HTTP, etc.

Una de las inyecciones de comando más comunes es la inyección de comandos SQL. En una aplicación web estándar normalmente trabajaremos con una base de datos insertando, eliminando o consultando dichos datos, si alguien es capaz de alterar los comandos SQL que utilizamos en nuestra aplicación para comunicarnos con nuestra base de datos la información y seguridad que contiene quedarán comprometidas. Por ejemplo, si utilizamos consultas SQL en los controles de seguridad como las contraseñas o los usuarios los atacantes podrán modificar la consulta SQL con el objetivo de evitar el control de seguridad y acceder a datos confidenciales.

Otro tipo de vulnerabilidad relacionada con la inyección de comandos es la inyección de comando OS, cuando invocamos otro programa en el sistema operativo local, y se permite que una entrada no fiable construya la cadena de comandos que genera para ejecutar dicho programa, podemos ser vulnerables a que un atacante pueda ejecutar sus propios comandos en lugar de los nuestros.

Las vulnerabilidades de inyección de comando normalmente son fáciles de detectar al analizar el código, es interesante realizar pruebas de fuzz ya que este tipo de pruebas ayudan a identificar dichos errores.

Los errores derivados de una vulnerabilidad de inyección pueden causar divulgación, perdida o corrupción de información, así como la denegación del acceso a nuestro sistema.

Para prevenir estos errores es conveniente dividir los datos de los comandos SQL y las consultas.

- Una de las mejores opciones es usar una API segura, que evite utilizar un intérprete de los datos que son inyectados y que utilice una interfaz parametrizada.
- Deberemos realizar validaciones de los datos de entrada al servidor utilizando por ejemplo “listas blancas” que son los parámetros que se pueden admitir en la entrada, todo lo que no se incluya en dicha lista será rechazado.

Pérdida de Autenticación

La implementación incorrecta de las funciones que nuestra aplicación utiliza para la autenticación y la gestión de las sesiones puede permitir a un atacante acceder a la información relacionada con los usuarios y las contraseñas, y de esta forma comprometer los datos confidenciales, los tokens de las sesiones, o aprovecharse de otros fallos de implementación para adoptar la identidad de otros usuarios.

En este tipo de vulnerabilidades los atacantes pueden acceder a millones de credenciales debido a fugas de información de nuestra aplicación. También pueden obtener las credenciales mediante ataques de fuerza bruta o algoritmos.

Estos errores normalmente se deben al diseño y la implementación de los controles de acceso. Los atacantes pueden detectar el método de autenticación defectuosa utilizando métodos manuales y herramientas automatizadas.

Podemos prevenir este tipo de vulnerabilidades mediante las siguientes pautas:

- No utilizar las credenciales por defecto en el software que desarrollemos.
- Implementar controles de seguridad para evitar el uso de contraseñas débiles.
- Implementar controles de seguridad para controlar los parámetros de las contraseñas, como son los parámetros relacionados con la complejidad y la longitud.
- Controlar el tiempo de respuesta de cada intento fallido de inicio de sesión. Se deberán registrar todos los fallos y se deberá avisar a los administradores en el caso de detectar ataques de fuerza bruta.
- Establecer un límite de intentos en el inicio de la sesión. Si se supera ese límite la cuenta del usuario quedará bloqueada y se deberán usar métodos alternativos (asistencia de los administradores de la web) para desbloquearla.

Exposición de datos sensibles

Tal y como expone el informe OWASP Top 10 -2017: "Muchas aplicaciones web y APIs no protegen adecuadamente los datos sensibles, como pueden ser la información financiera, de salud o Información Personalmente Identificable (PII)". Debido a esto un posible atacante puede acceder a datos confidenciales, y llevar a cabo fraudes de tarjetas bancarias, suplantaciones de identidad, etc.

Es importante tener en cuenta que la información sensible debe ser protegida mediante métodos adicionales, como puede ser el cifrado de datos confidenciales.

Para prevenir este tipo de vulnerabilidades deberemos tener en cuenta los siguientes puntos:

- En primer lugar, deberemos clasificar los datos procesados, almacenados o trasmitidos por el sistema indicando que información es sensible de acuerdo con las regulaciones o leyes.
- Deberemos aplicar los controles de seguridad necesarios en función de las clasificaciones previamente realizadas.
- No deberemos almacenar datos sensibles innecesariamente, es necesario descartarlos tan pronto como sea posible.
- Es necesario cifrar los datos sensibles cuando sean almacenados.
- Deberemos cifrar los datos sensibles en tránsito utilizando protocolos seguros como TLS.
- Deberemos deshabilitar el almacenamiento en cache de los datos sensibles.
- Se deberá verificar la efectividad de las configuraciones y parámetros de forma independiente.

Entidades externas XML (XXE)

Tal y como indica el informe OWASP Top 10 -2017: "muchos procesadores XML desactualizados o mal configurados evalúan referencias a entidades externas en documentos XML." Estas entidades externas pueden utilizarse para revelar archivos internos, escanear puertos de la LAN, ejecutar código de forma remota y realizar ataques de denegación de servicio (DoS).

Para prevenir este tipo de vulnerabilidades podemos seguir los siguientes puntos:

- Deberemos utilizar formatos de datos menos complejos como son los ficheros con extensión JSON, y deberemos evitar serializar los datos confidenciales.
- Los procesadores y las bibliotecas XML deberán estar actualizadas. Utilizaremos validadores de dependencias.
- Implementación de validación de entrada positiva en el servidor, filtrando la información para evitar el ingreso de datos perjudiciales dentro de documentos, cabeceras y nodos XML.
- Deberemos verificar que la funcionalidad de carga de ficheros XML o XSL valide el XML entrante, usando validación XSD o similar.

Pérdida de control de acceso

Debemos aseguramos de que los usuarios de nuestro software están haciendo sólo lo que se les permite hacer, debemos tener en cuenta que un posible atacante tratará de explotar la autorización de acceso que le hemos proporcionado para realizar funcionalidades que no han sido autorizadas para el sino para usuarios restringidos.

Para prevenir este tipo de vulnerabilidades se tendrán en cuenta las siguientes pautas:

- El control de acceso debe estar restringido de forma predeterminada.
- Deberemos implementar los mecanismos de control de acceso una única vez y serán reutilizados en toda la aplicación.
- Los controles de acceso deberán controlar los permisos asignados a cada usuario evitando de esta forma que el usuario puede crear, leer, actualizar o eliminar cualquier registro.
- Los modelos de dominio deben hacer cumplir los requisitos exclusivos de los límites de negocio de las aplicaciones.
- Deshabilitar el listado de directorios del servidor web asegurándonos evitando que las copias de seguridad y los metadatos de los archivos se encuentren en carpetas públicas.
- Deberemos registrar los errores de control de acceso y deberemos alertar a los administradores de la web cuando corresponda.
- Limitar la tasa de acceso a las APIs para minimizar el daño de herramientas de ataque automatizadas.
- Los desarrolladores y los responsables de pruebas deben incluir pruebas de control de acceso en sus pruebas unitarias y de integración.

Configuración de seguridad incorrecta

Este tipo de vulnerabilidad normalmente se debe a que no existe una configuración de seguridad, o que se ha establecido una configuración de seguridad de forma manual, es un problema muy común. Como ejemplos podemos encontrar cabeceras mensajes de error que muestran contenido sensible, ausencia de actualizaciones, frameworks, dependencias y componentes desactualizados, etc.

Es necesario implementar procesos seguros de instalación:

- Deberemos implementar un proceso ágil que facilite la implementación de otro entorno asegurado. Los entornos de desarrollo, de control de calidad y de producción deben configurarse de manera idéntica y con diferentes credenciales para cada entorno.
- No se deberán instalar frameworks y funcionalidades que no vayan a ser utilizadas.

- La aplicación deberá tener una arquitectura segmentada que proporcione una separación efectiva y segura entre componentes y acceso a terceros, contenedores o grupos de seguridad en la nube.

Cross-Site scripting (XSS)

El Cross-Site scripting (XSS) es una vulnerabilidad muy común, y representa una gran amenaza para las aplicaciones web. Esta vulnerabilidad consiste en que los atacantes pueden inyectar código JavaScript o de otro lenguaje similar a través del navegador en una página Web que genere nuestra aplicación. Se trata de una inyección de código que no pertenece al original de la aplicación. Por ejemplo, podemos encontrar este tipo de vulnerabilidades en una página donde podamos insertar un comentario (hoy en día muchas tiendas web nos permiten que comentemos sus productos). El atacante puede llegar a obtener la entrada al servidor de la aplicación, o captar información del usuario que la está utilizando sin que se dé cuenta. Un atacante podría insertar un script que robe las cookies de la víctima pudiendo suplantar sus sesiones sin necesidad de su usuario y contraseña, también podría redirigir el tráfico de la web hacia sitios web de su conveniencia.

El XSS es un tipo de ataque que puede robar información, secuestrar sesiones de usuario, y comprometer el navegador, poniendo en riesgo la integridad del sistema.

Este tipo de vulnerabilidades surge al no validar correctamente los datos de entrada que son usados en la aplicación, o no tratar adecuadamente la información que nuestra aplicación muestra al usuario.

Del mismo modo para evitar este tipo de vulnerabilidad hoy en día podemos apoyarnos en diferentes plataformas de software de detección y explotación del XSS.

Tal y como indica el informe OWASP Top 10 -2017: “prevenir XSS requiere mantener los datos no confiables separados del contenido activo del navegador.” Para ello deberemos aplicar:

- La utilización de frameworks seguros que codifiquen el contenido para prevenir XSS.
- La codificación de los datos de requerimientos HTTP no confiables en los campos de salida HTML.

Deserialización insegura

Estos errores de software ocurren cuando una aplicación recibe objetos serializados que son incorrectos y suponen un riesgo, ya que pueden ser manipulados o borrados por un atacante. Este tipo de vulnerabilidades pueden provocar que un atacante realice ataques de repetición, inyecciones o elevar sus privilegios de ejecución, incluso podría permitir una ejecución remota del código del servidor.

Para evitar este tipo de errores no deberemos incorporar a nuestro software objetos serializados de fuentes no confiables o la utilización de medios de serialización que solo permitan tipos de datos primitivos. Si esto no es posible deberemos aplicar los siguientes puntos:

- Deberemos implementar verificaciones de integridad tales como firmas digitales en cualquier objeto serializado, con el fin de detectar modificaciones no autorizadas.
- Deberemos aislar el código que realiza la deserialización, de modo que se ejecute en un entorno con los mínimos privilegios posibles.

- Registrar las excepciones y fallos en la deserialización.
- Restringir y monitorizar las conexiones de entrada y salida de red desde contenedores o servidores que utilizan funcionalidades de deserialización.
- Monitorear los procesos de deserialización, alertando si un usuario deserializa constantemente.

Uso de componentes con vulnerabilidades conocidas

Tenemos que tener en cuenta que los componentes como los frameworks, bibliotecas y otros módulos se ejecutan con los mismos privilegios que la aplicación que los contiene. Por este motivo cuando nuestro software utiliza componentes que contienen vulnerabilidades puede ser susceptible a sufrir ataques y errores que debiliten nuestra aplicación, pudiendo poner en riesgo información confidencial y su correcto funcionamiento.

Para evitar este tipo de vulnerabilidades deberemos:

- Eliminar dependencias, funcionalidades, componentes, archivos y documentación innecesaria y no utilizada.
- Utilizar una herramienta para mantener un inventario de versiones de componentes tanto del cliente como del servidor.
- Obtener componentes únicamente de orígenes oficiales utilizando canales seguros. Utilizar preferentemente paquetes firmados con el fin de reducir las probabilidades de uso de versiones manipuladas maliciosamente.
- Supervisar bibliotecas y componentes que no poseen mantenimiento o no liberan parches de seguridad para sus versiones obsoletas o sin soporte.

Registro y monitoreo insuficientes

La ausencia de un registro y un monitoreo de los errores, unido con la falta de respuesta ante problemas de seguridad permiten a los atacantes prolongar su ataque, y por lo tanto manipular, extraer o destruir datos confidenciales durante un periodo más largo de tiempo. Según el informe OWASP Top 10-2017: “Los estudios demuestran que el tiempo de detección de una vulnerabilidad de seguridad es mayor a 200 días, siendo típicamente detectado por terceros en lugar de por procesos internos.”

Según el riesgo de los datos almacenados o procesados por la aplicación deberemos:

- Asegurarnos de que todos los errores de inicio de sesión, de control de acceso y de validación de entradas de datos del lado del servidor se pueden registrar para identificar cuentas sospechosas.
- Asegurarnos de que las transiciones de alto impacto tengan una pista de auditoria con controles de integridad para prevenir alteraciones o eliminaciones.
- Asegurarnos de que todas las transacciones de alto valor poseen una traza de auditoría con controles de integridad que permitan detectar su modificación o borrado, tales como una base de datos con permisos de inserción únicamente u similar.
- Deberemos establecer una monitorización efectiva de tal manera que las actividades sospechosas sean detectadas y solucionadas dentro de períodos de tiempo aceptables.

Otras vulnerabilidades

Como hemos explicado anteriormente el informe que desarrolla la comunidad de OWASP nos reporta las vulnerabilidades más comunes hasta la fecha, aunque el informe nos

presenta las diez más importantes, nosotros vamos a ampliar esta información con otras vulnerabilidades que también son frecuentes en el desarrollo y que consideramos importantes.

Manipulación del Path

Esta vulnerabilidad consiste en permitir que una entrada de usuario controle las rutas de acceso que se usan en operaciones sobre el sistema de archivos, con lo que se permitiría a un atacante acceder o modificar recursos del sistema protegidos.

Este tipo de vulnerabilidades surgen cuando se dan las siguientes condiciones:

1. Un atacante puede establecer una ruta de acceso que se utiliza en una operación sobre el sistema de archivos del servidor.
2. Al especificar el recurso, el usuario malintencionado consigue una capacidad que de otro modo no le estaría permitida.

El programa podría otorgar al usuario la capacidad de sobrescribir el archivo especificado o ejecutar una configuración controlada por él.

El desarrollador de la aplicación web deberá tener en cuenta la posibilidad de que un atacante pueda proporcionar un nombre de archivo como "../../apache/conf/httpd.conf", que provocará que la aplicación sustituya el archivo de configuración especificado por el del usuario, el cual lo ha configurado previamente para manipular la aplicación web que está atacando.

Por lo tanto, para evitar esta vulnerabilidad podemos establecer como plan de mitigación la comprobación del nombre de los ficheros de entrada a nuestro servidor.

Cross Site Request Forgery

El ataque CSRF es un ataque que utiliza una web de confianza a través de la cual el usuario va a ser víctima del ataque.

Un ataque de falsificación de solicitudes cruzadas (CSRF) obliga al navegador de una víctima a enviar una solicitud HTTP falsificada, la cual va a incluir la cookie de sesión de la víctima además de cualquier otra información de autenticación, a una aplicación web vulnerable. Esto permite al atacante obligar al navegador de la víctima a generar peticiones que la aplicación cree que son peticiones legítimas. CSRF aprovecha las aplicaciones web que permiten a los atacantes predecir todos los detalles de una transacción. Dado que los navegadores envían credenciales como cookies de sesión automáticamente, los atacantes pueden crear páginas web maliciosas que generan solicitudes que no se distinguen de las legítimas.

Podemos entender mejor este tipo de vulnerabilidades a través del siguiente ejemplo:

Si un usuario tiene en su navegador varias pestañas, y en una de ellas se encuentra la aplicación de su banco con su sesión iniciada, puede estar ejecutándose desde otra pestaña código destinado a provocar un ataque de tipo CSRF contra la aplicación de su banco, con el objetivo de acceder a la sesión ya iniciada y obtener información de dicho usuario sin que éste se dé cuenta. Esto se debe a que en ese momento la aplicación del banco se encuentra con la sesión iniciada y autenticada y no es capaz de distinguir las peticiones que realiza el usuario legítimo de las que puede realizar el código que se está ejecutando desde la otra pestaña del navegador.

Para evitar este ataque podemos utilizar las siguientes estrategias:

1. Envío doble de cookie:

   Esta técnica se basa en que, si bien el navegador envía toda la información relacionada a la sesión, si agregamos un campo oculto a cada formulario con el valor del identificador de sesión, podremos evitar el ataque, ya que el atacante debe conocer este valor para incluirlo en el enlace o el formulario oculto en su página. El navegador no se lo proporcionará por las políticas de seguridad de dominios cruzados.

2. Formulario único para cada solicitud:

   Se trata de generar un formulario con un campo oculto dinámico que es generado por cada solicitud. El contenido del campo debe ser generado aleatoriamente utilizando un buen sistema de PRNG (Pseudo Random Number Generator) para que no pueda ser adivinado mediante técnicas estadísticas avanzadas. Este campo se guardará en el servidor y será eliminado cuando el formulario sea utilizado ya que una nueva solicitud del formulario deberá ser acompañada por un nuevo contenido en el campo. Si se recibe un formulario con un contenido no válido, simplemente se rechaza la solicitud.

3. Requerir credenciales de autentificación nuevamente:

Es una técnica que se basa en requerir nuevamente los datos de ingreso del usuario al momento de generar una transacción.

Acceso al buffer con valor de longitud incorrecto

Este error de programación surge del acceso a un búfer de memoria con un valor de longitud incorrecto. Los programas que leen o escriben datos en un búfer tendrán resultados impredecibles una vez que se haya sobrepasado el límite del búfer en uso y el programa continúe leyéndolo o escribiéndolo. Para evitar esta vulnerabilidad deberemos comprobar las entradas de nuestro software que vayan a tener acceso a un buffer de memoria, con el objetivo de no provocar un desbordamiento.

Exposición de información a través de un mensaje de error

Cuando los programadores usan mensajes de error, pueden revelar sin quererlo información confidencial o crítica a cualquier atacante que desee aprovecharse de su software. Esta información podría abarcar una amplia gama de datos valiosos, incluyendo la información de la identificación personal del usuario, las credenciales de autenticación y la configuración del servidor. A veces, puede parecer información inofensiva que únicamente será útil para los usuarios y administradores de la aplicación, como puede ser la ruta de instalación completa de su software. Para evitar este tipo de vulnerabilidades es muy importante controlar la información que se muestra en nuestra aplicación en todo momento evitando mostrar información que comprometa la seguridad de nuestra aplicación.

Herramientas y métodos de detección de vulnerabilidades de software

Aunque hemos introducido las que consideramos que son las vulnerabilidades más comunes, existen muchas más que pueden afectar a nuestro software. Nuestro objetivo es evitar dichas vulnerabilidades, hoy en día existen múltiples herramientas, frameworks y métodos de análisis que nos ayudan a detectarlas, del mismo modo también existen métodos para verificar la calidad y resiliencia de nuestro código.

En este apartado vamos a introducir los métodos que se utilizan para verificar el código que desarrollamos.

Realización de pruebas de software

Uno de los métodos más eficaces para obtener un software sin vulnerabilidades y resiliente es la realización de pruebas. Existen varios tipos de pruebas de software:

Pruebas unitarias

Las pruebas unitarias son un proceso clave desde una perspectiva de seguridad y resiliencia. Este tipo de pruebas ayudan a evitar que los errores lleguen a la fase final. Los desarrolladores pueden validar ciertas condiciones de contorno e impedir vulnerabilidades tales como desbordamientos de búfer dentro de un módulo de la aplicación.

Las pruebas unitarias permiten comprobar el correcto funcionamiento de las unidades básicas de la aplicación. Estas pruebas permiten comprobar automáticamente aspectos que de otra forma serían inviables de probar, o bien requieren un tiempo excesivo de revisión. Nos permiten inspeccionar el código fuente para comprobar si se realiza un buen uso de las prácticas de programación, si los algoritmos implementados se comportan de manera adecuada, y para la detección de posibles errores o comportamientos inesperados en los módulos desarrollados.

Las pruebas unitarias son un método verificación de componentes individuales del código fuente. Para llevar a cabo esta verificación el desarrollador implementa casos de uso directamente en código fuente que es ejecutado durante el proceso de pruebas. Si la ejecución no obtiene los resultados esperados se dice la prueba ha fallado, y por lo tanto se considera que el código probado es incorrecto.

Uno de los Frameworks más importantes hoy en día para la realización de pruebas unitarias de código Java es JUnit. JUnit realiza una ejecución de las clases Java de nuestro software de forma controlada, de esta forma podemos evaluar si el funcionamiento de los métodos de la clase tiene un comportamiento esperado. En función de algún valor de entrada se evalúa el valor de retorno esperado, el cual hemos configurado previamente. Si la clase cumple con la especificación, nos indicará que el método pasó la prueba de forma satisfactoria, en caso contrario, JUnit nos indicará que se ha producido un error en el método evaluado. De esta forma podremos desarrollar nuestro código Java conociendo su comportamiento. Mediante la aplicación de JUnit seremos capaces de probar diferentes casos y situaciones en nuestro software, por ejemplo, podremos probar el comportamiento de nuestro código ante un error de tipo NullPointer, y controlar el comportamiento de la aplicación ante esta situación.

Pruebas de integración

La fase de pruebas es crítica para descubrir vulnerabilidades que no se descubrieron y solucionaron anteriormente. Se deberán crear casos de prueba de seguridad. En la fase de pruebas de integración es clave recurrir a la documentación de los requisitos del sistema.

El equipo de pruebas deberá utilizar todos los supuestos para crear varios casos de prueba que cubran con los requisitos de seguridad. Los encargados de probar la seguridad del sistema utilizan estos casos de prueba durante el análisis dinámico de la aplicación.

El software se cargará en un entorno de pruebas y se procederá a probar cada uno de los casos de prueba previamente diseñados. Estas revisiones manuales de seguridad son muy eficaces para descubrir fallos de la lógica de negocio en la aplicación.

Las pruebas de integración se pueden considerar como pruebas de caja negra en las que tenemos que probar una funcionalidad simplemente introduciéndole unos datos de entrada y en consecuencia deberemos obtener una salida esperada. De esta forma

podemos probar la funcionalidad de nuestro software. Se utilizan normalmente para comprobar que el sistema está integrado correctamente y que todas las funcionalidades del software desarrollado funcionan según los requisitos establecidos.

Pruebas de penetración

Una prueba de penetración trata de atacar un sistema informático con el fin de encontrar vulnerabilidades de seguridad, el objetivo de este tipo de pruebas es el de acceder al sistema y a la información que contiene.

Existen dos tipos de pruebas de penetración:

1. Pruebas de penetración de caja blanca, en las que un atacante tiene toda la información del sistema informático que trata de vulnerar.
2. Pruebas de penetración de caja negra, en las que el atacante no conoce nada acerca de la aplicación que esta intentando atacar salvo el nombre de la empresa a la que pertenece.

Este tipo de pruebas nos puede ayudar a determinar si un sistema informático es vulnerable a este tipo de ataques y si es capaz de responder solventando adecuadamente los mismos.

A través de los resultados que nos ofrecen las pruebas de penetración podemos evaluar el impacto que una organización puede sufrir ante este tipo de ataques y de esta forma se podrán sugerir medidas para reducir los riesgos.

Las pruebas de penetración son necesarias por varias razones:

- Nos muestran las posibilidades de éxito que puede tener un ataque en nuestro sistema.
- Identifican vulnerabilidades de alto riesgo que resultan de una combinación de vulnerabilidades de menor riesgo explotadas en una secuencia particular.
- Identifican vulnerabilidades que son difíciles de detectar mediante un software de análisis de vulnerabilidades.
- Comprueban la capacidad de respuesta y mitigación ante los posibles ataques a nuestro sistema.

Fuzzing

El fuzzing consiste en la realización de pruebas semiautomáticas o automáticas donde se introducen en la aplicación que estamos probando datos aleatorios, que no son válidos y que no han sido controlados en el software. De esa forma podemos comprobar si nuestra aplicación valida correctamente los datos de entrada.

La herramienta que permite ejecutar una prueba de fuzzing se la conoce, como fuzzer. El Fuzzer se puede dividir en tres componentes: generador, mecanismo de entrega y sistema de monitoreo.

- Generador: este componente es el encargado de formar las diferentes cadenas de texto para realizar las pruebas en la entrada del software que estamos probando. Normalmente, el generador da la posibilidad de crear entradas que se adaptan al sistema.
- Mecanismo de entrega: este componente se encarga de obtener las entradas que ha generado el componente generador y mandarlo al sistema que se está sometiendo a prueba. Este componente está ligado al sistema de prueba, ya que los mecanismos de entrada a un sistema de software pueden variar notablemente.

- Sistema de monitoreo: este componente se encarga de monitorear la respuesta del sistema a partir de los valores de entrada que le hemos proporcionado. Su función se basa principalmente en buscar errores y en evaluar los posibles impactos que puedan provocar sobre el sistema que está siendo probado. A partir de los errores obtenidos se podrá determinar cuáles de esos errores son críticos y pueden ser explotados.

Existen diferentes formas de llevar a cabo las pruebas sobre el sistema. Deberemos tener en cuenta tres conceptos:

- Caja negra: las pruebas de caja negra son totalmente opacas, es decir, únicamente veremos la respuesta de nuestro sistema ante las entradas que le proporcionemos.
- Caja blanca: Para la realización de este tipo de pruebas se tienen en cuenta otro tipo de datos, como, por ejemplo, el código fuente del software que está siendo probado. En este caso, el proceso de pruebas es mucho más eficiente ya que se tiene en cuenta información adicional relativa al software que se está evaluando por lo que los resultados serán más acertados.
- Caja gris: mezcla los conceptos de las pruebas de caja negra y de caja blanca. Esta alternativa no tiene en cuenta el acceso al código fuente, pero utiliza otros mecanismos tales cómo el análisis reversivo o la utilización de un debugger.

Como herramientas para la realización de pruebas de Fuzz destaca:

- Peach Fuzzer

Peach Tech estableció el estándar para la tecnología fuzzing hace más de diez años con la herramienta Peach Fuzzer Community, la versión de código abierto de Peach Fuzzer.

Peach es uno de los frameworks de fuzzing de código abierto más avanzados.

Peach tiene un framework para hacer su propio fuzzing o ampliarlo añadiéndole código. Incluye muchas interfaces externas para funciones de cifrado, bibliotecas de compresión y de tipos de codificación. Se trata de una plataforma automática de pruebas que se encarga de prevenir ataques encontrando vulnerabilidades en los sistemas de software.

- Autodafe

Autodafe es un framework de fuzzing capaz de descubrir desbordamientos de búfer.

Autodafe admite fuzzing de estándares de red, de archivos, y de funciones inversas de fuzzing, donde es capaz de realizar fuzzing contra un cliente que se conecta a él. A diferencia de otros fuzzers, Autodafe puede tomar un archivo pdml (un volcado de paquetes basado en XML) y generar la estructura que utilizará el fuzzer para realizar el fuzzing.

- Scratch

Scratch realiza un análisis complejo de archivos binarios para determinar qué datos hay que utilizar para realizar una prueba de fuzz correctamente.

Análisis Estático

Este tipo de análisis de software se caracteriza porque se ejecuta sobre el código de la aplicación sin necesidad de que dicho código sea ejecutado.

El análisis se puede realizar tanto sobre el código fuente como sobre el código compilado. El análisis estático se encarga de buscar de vulnerabilidades de seguridad, y es uno de los puntos más importantes para un correcto desarrollo de software.

Uno de los puntos débiles del análisis estático es que puede dar como resultado "falsos positivos" o "falsos negativos". Los falsos positivos ocurren cuando el programa contiene errores que el análisis no detecta, y que los da por buenos, mientras que los falsos negativos surgen cuando el análisis detecta errores que el programa no tiene. Es conveniente realizar un análisis estático ayudándonos de alguna herramienta que automatice el proceso, ya que realizar este tipo de análisis de forma manual resulta muy ineficiente.

Un aspecto que tenemos que tener en cuenta en cuenta en un analizador estático es el alcance del análisis. Un análisis local examina una sola función cada vez y no tiene en cuenta las relaciones que pueden existir entre las funciones que componen el programa. Un analizador modular considera una unidad de compilación cada vez, de forma que tiene en cuenta relaciones entre funciones en el mismo modulo, y las propiedades aplicables a las clases, pero no las relaciones entre los distintos módulos. Un análisis global involucra un programa entero de modo que tiene en cuenta todas las relaciones entre funciones. El alcance del análisis determina también la cantidad de contexto que el analizador considera. Cuanto mayor es el contexto menor es el número de "falsos positivos" que encontramos, sin embargo, necesitaremos mayor capacidad de computación.

A lo largo de los años se han desarrollado varias herramientas para realizar el análisis estático con el fin de encontrar vulnerabilidades.

El análisis estático se puede realizar de forma manual, existen diferentes métodos de integración continua, de los cuales hablaremos en la sección correspondiente más adelante, que permiten realizar un análisis estático manual de forma sencilla, sin embargo, la utilización de herramientas de software evita que se comentan errores y aceleran el proceso. Existen en el mercado multitud de herramientas de análisis estático, tanto de código libre como comerciales, deberemos buscar aquellas herramientas que estén certificadas si necesitamos de una seguridad crítica.

A continuación, describimos algunas de las herramientas más destacadas y utilizadas que se encargan de realizar el análisis estático:

Sonarqube

Sonarqube es una herramienta de software libre y gratuita que permite gestionar y analizar la calidad del código fuente. Al instalarla podremos recopilar, analizar, visualizar y llevar un histórico de la calidad del código fuente desarrollado.

Usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.

Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, errores potenciales, comentarios y diseño del software.

Sonarqube es una herramienta que se utiliza principalmente para el análisis del código desarrollado en Java. Aunque también da soporte a los siguientes lenguajes: ABAP, C, Cobol, C#, Delphi/Pascal, Flex/ActionScript, Groovy, JavaScript, Natural, PHP, PL/SQL, Visual Basic 6, Web y XML.

Google CodePro Analytix

CodePro Analytix es un conjunto de herramientas de apoyo al desarrollador totalmente integrada en entorno de desarrollo de Eclipse, que implementa los siguientes módulos:

- Análisis de código.
- Generador de pruebas unitarias JUnit.
- Métricas.
- Reportes del código ejecutado.
- Análisis de dependencias.
- Análisis de código muerto.

Coverity

Coverity es una marca de productos de desarrollo de software de Synopsys, que consiste principalmente en el análisis de código estático y herramientas de análisis de código dinámico. Las herramientas permiten a los desarrolladores encontrar defectos y vulnerabilidades de seguridad en el código fuente escrito en C, C ++, Java, C # y JavaScript.

Coverity ofrece los siguientes servicios:

- Coverity Code Advisor es una herramienta de análisis de código estático para C, C ++, C #, Java y JavaScript.
- Coverity Scan es un servicio gratuito de análisis estático basado en la nube para la comunidad de código abierto. La herramienta analiza más de 3900 proyectos de código abierto y se integra con GitHub y Travis CI.
- Coverity Test Advisor es una serie de productos destinados a identificar las debilidades en las pruebas de software de un proyecto.
- Seeker es un producto de prueba de seguridad para aplicaciones interactivas.

En nuestro caso nos hemos registrado en la web oficial de Coverity para poder ver cómo trabaja esta herramienta de análisis.

Nada más registrarnos en la aplicación, vemos que Coverity trabaja con Travis y con GitHub, podemos ver que hay multitud de proyectos que podemos visualizar y ver cómo se han realizado los análisis correspondientes, ya que como hemos explicado anteriormente Coverity Scan es un servicio gratuito de análisis estático basado en la nube cuya comunidad es de código abierto.

Por lo tanto, podemos asignarnos un proyecto ya existente, podremos colaborar con su desarrollo o simplemente ver como se ha realizado su análisis y también podremos subir nuestro propio proyecto a la plataforma (GitHub) y posteriormente registrar el proyecto en

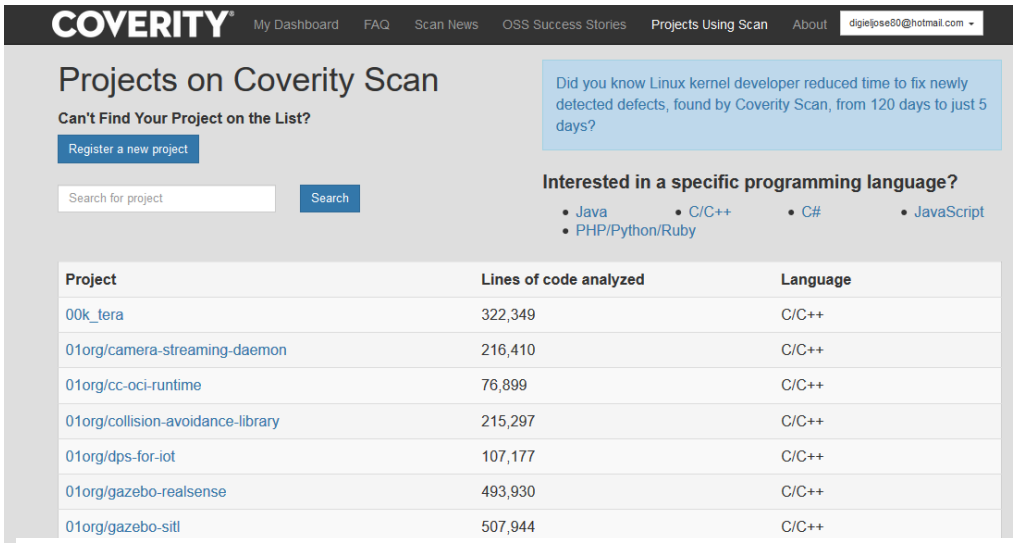


Ilustración 2 Coverity

Coverity para realizarle el correspondiente análisis.

Como vemos en el panel de control podemos buscar proyectos según nuestras preferencias de lenguaje de programación, como pueden ser C++, C#, JavaScript, etc.

En nuestro caso, no hemos registrado nuestro proyecto, sino que hemos elegido un proyecto de C/C++, simplemente seleccionando uno de los que aparecen en la lista, ya que nuestro objetivo es ver y comprender como trabajan estas herramientas de análisis estático.

En nuestro caso hemos elegido el proyecto 2LGC:

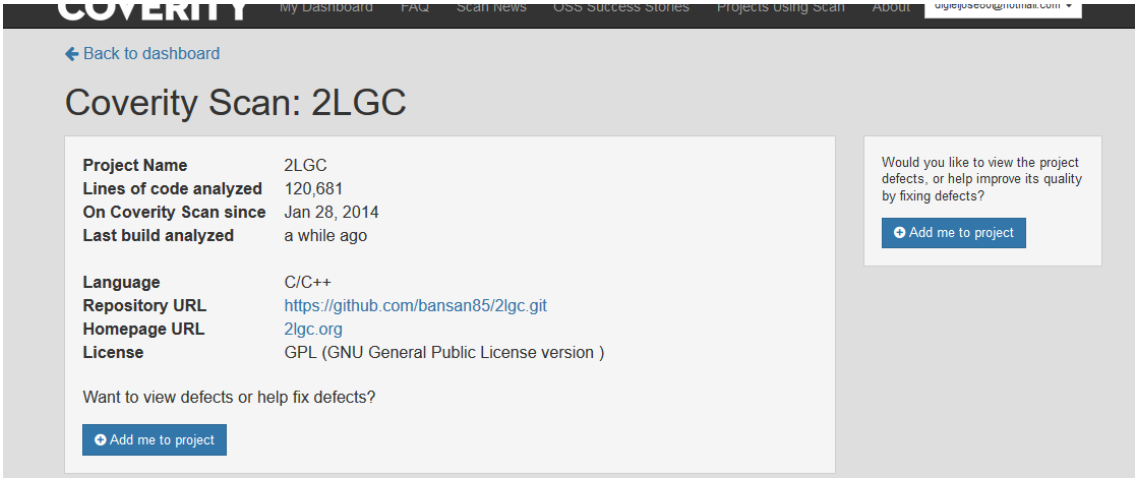


Ilustración 3Menú de Coverity

En este menú se pueden ver las estadísticas relacionadas con el análisis de código:

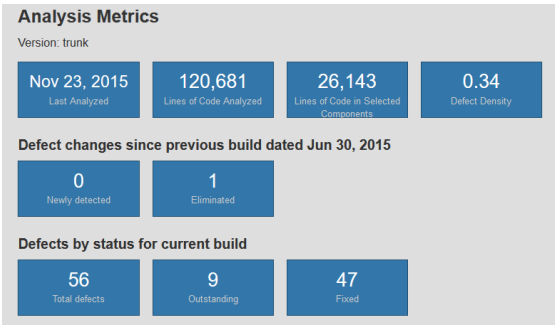


Ilustración 4 Análisis de métricas

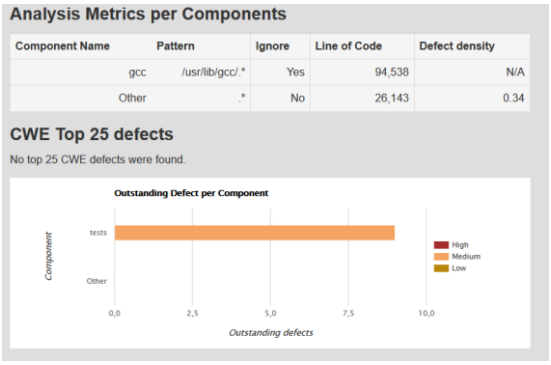


Ilustración 5 Análisis de métricas por componentes

Como podemos ver muestra datos estadísticos acerca de la detección de vulnerabilidades a lo largo del desarrollo de la aplicación.

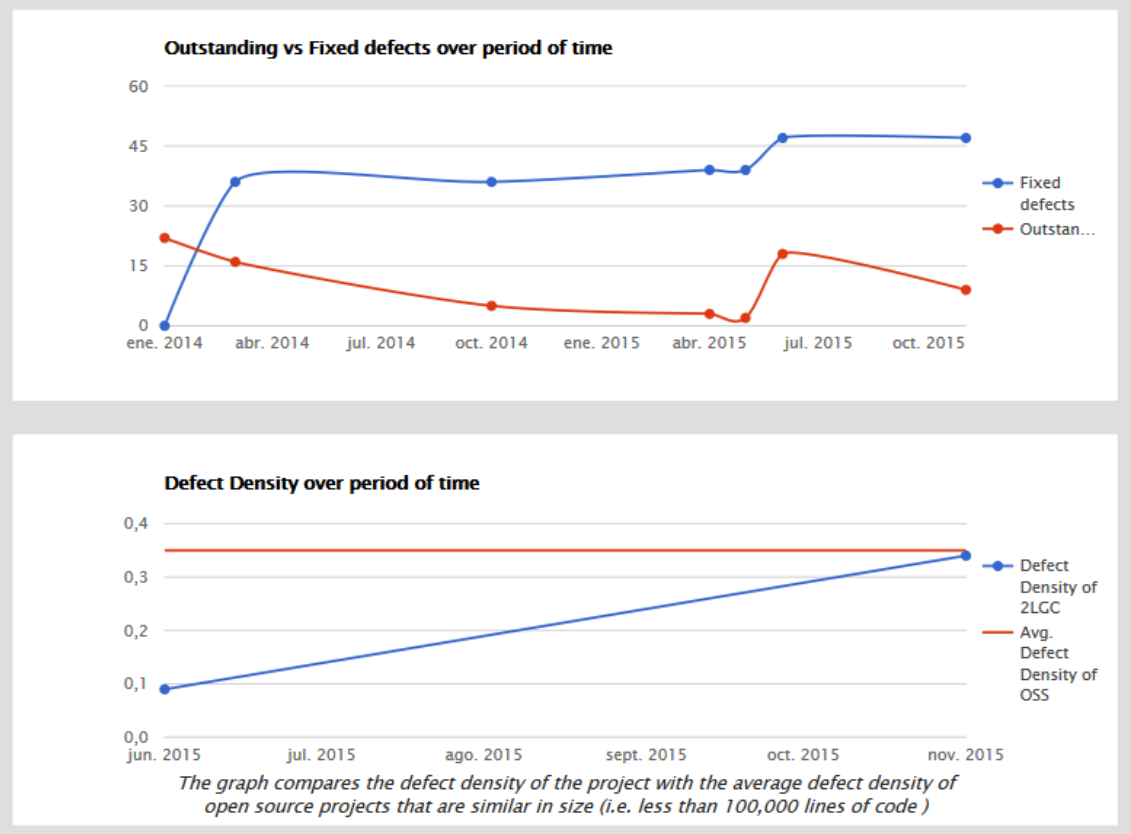


Ilustración 6 Estadísticas de Coverity

El análisis de código nos va a mostrar los errores detectados de la siguiente manera:

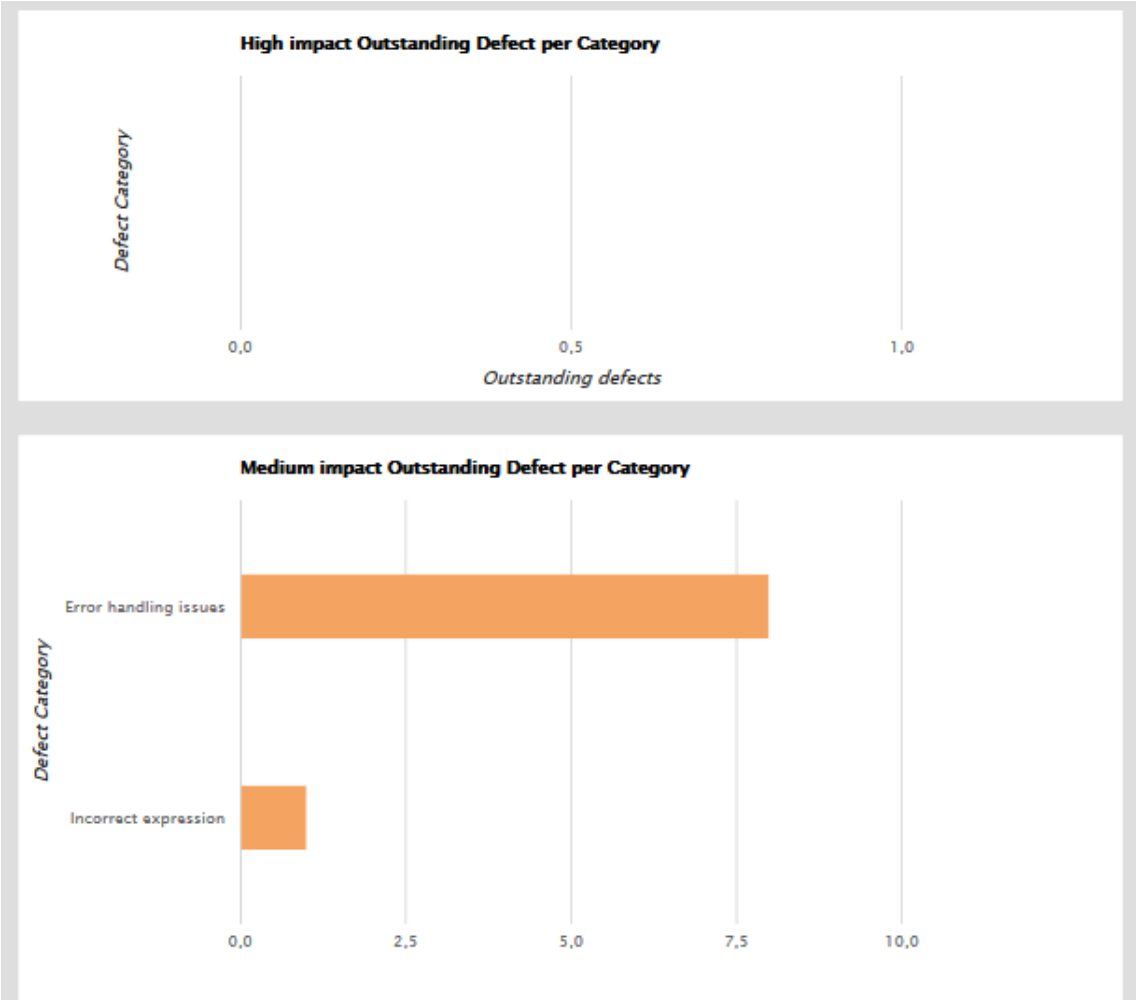


Ilustración 7 Representación de errores detectados en Coverity

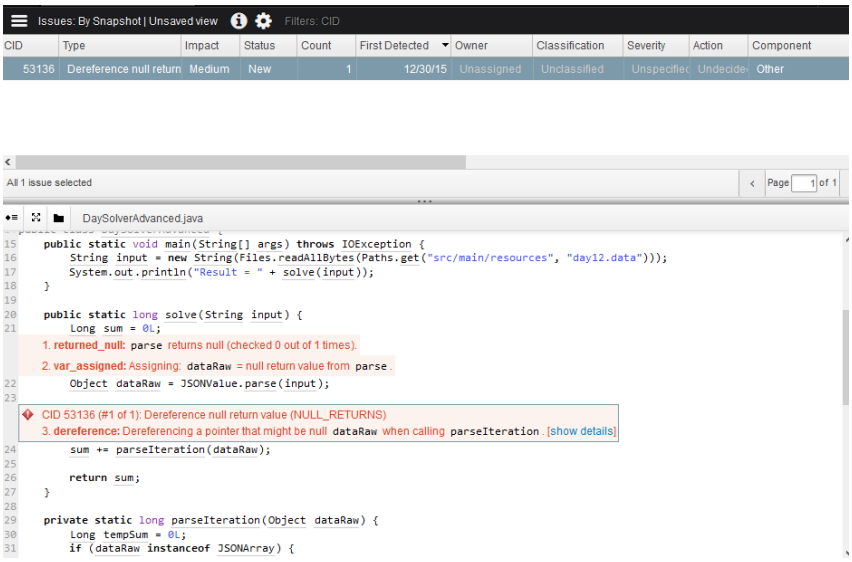


Ilustración 8 Identificación de código vulnerable

Como podemos observar se nos está indicando que el código analizado puede hacer referencia a un valor Null, y por lo tanto lo identifica como código vulnerable. Este analizador es capaz de indicarnos la parte del código que se considera incorrecta.

Fortify Static Code Analyzer

Fortify Static Code Analyzer es un conjunto de analizadores de seguridad de software que busca violaciones de reglas y pautas de codificación específicas de seguridad en una variedad de lenguajes de programación. El analizador de código estático Fortify proporciona datos enriquecidos que permiten a los analizadores identificar y priorizar las violaciones para que las correcciones sean rápidas y precisas. Fortify Static Code Analyzer realiza un análisis de software que nos ayuda a mejorar nuestro código, así como para hacer que las revisiones de código sean más eficientes, coherentes y completas.

El uso de Fortify Static Code Analyzer implica:

- Ejecución de Fortify Static Code Analyzer como un proceso independiente o la integración de Fortify Static Code Analyzer en una herramienta de compilación.
- Traducir el código fuente a un formato traducido intermedio.
- Escaneando el código traducido y produciendo informes de vulnerabilidad de seguridad.
- Auditoría de los resultados de la exploración, ya sea abriendo los resultados (archivo FPR) en Fortify Audit Workbench o cargándolos en Fortify Software Security Center para su análisis, o directamente con los resultados mostrados en la pantalla.

En cuanto a los errores debidos a falsos positivos basándome en la experiencia profesional, una estrategia de mitigación podría ser la revisión del código de desarrollo por parte del equipo de trabajo, es decir, la revisión del código por parte de otros compañeros de desarrollo que formen parte del equipo, de este modo, si el desarrollo o el proyecto en global presenta fallos que no han sido detectados por el análisis estático pueden solucionarse antes de que el código erróneo pase a ser definitivo. Del mismo modo en el proyecto en el que me encuentro trabajando ahora mismo, ya que implementa muchas funcionalidades basadas en servicios y que cada servicio es implementado por un

equipo diferente, se proponen pruebas “de caja negra” para conocer las respuestas de los servicios, de este modo podemos comprobar las posibles respuestas y comprobar si el comportamiento no es correcto, también se realizan pruebas unitarias y pruebas de integración del sistema completo, de este modo también es posible detectar errores que las herramientas de análisis no han sido capaces de detectar o que han detectado erróneamente.

Por lo tanto, podemos concluir que una herramienta de análisis estático proporciona una gran ayuda para implementar un código de mayor calidad y más seguro, pero deberemos tener en cuenta que también se producen casos de falsos positivos y falsos negativos, que podremos detectar y corregir a través de las pruebas que implementemos para nuestras aplicaciones.

Las herramientas de análisis estático son un elemento más para obtener un software óptimo, pero no sustituyen otros elementos de comprobación como pueden ser las pruebas de este, o las revisiones del software que realizan otros miembros del equipo, sino que son un elemento complementario.

Herramientas de apoyo en la detección de vulnerabilidades de software

Proyecto OWASP

Una vez más consideramos importante hacer referencia a la fundación OWASP ya que esta vez nos proporciona la herramienta OWASP Xenotix XSS Exploit Framework que se trata de un Framework de detección y explotación de vulnerabilidades avanzado del tipo Cross Site Scripting (XSS), es de código abierto y de licencia libre. Xenotix realiza el escaneo de vulnerabilidades dentro de los motores del navegador. Incorpora en su escáner 3 fuzzers para reducir el tiempo de escaneo y producir mejores resultados. Xenotix se incorpora con un módulo de recopilación de información rico en características para reconocimiento de objetivos. El Framework incluye módulos de explotación XSS para el entorno real que permite realizar pruebas de penetración y la creación de prueba de concepto.

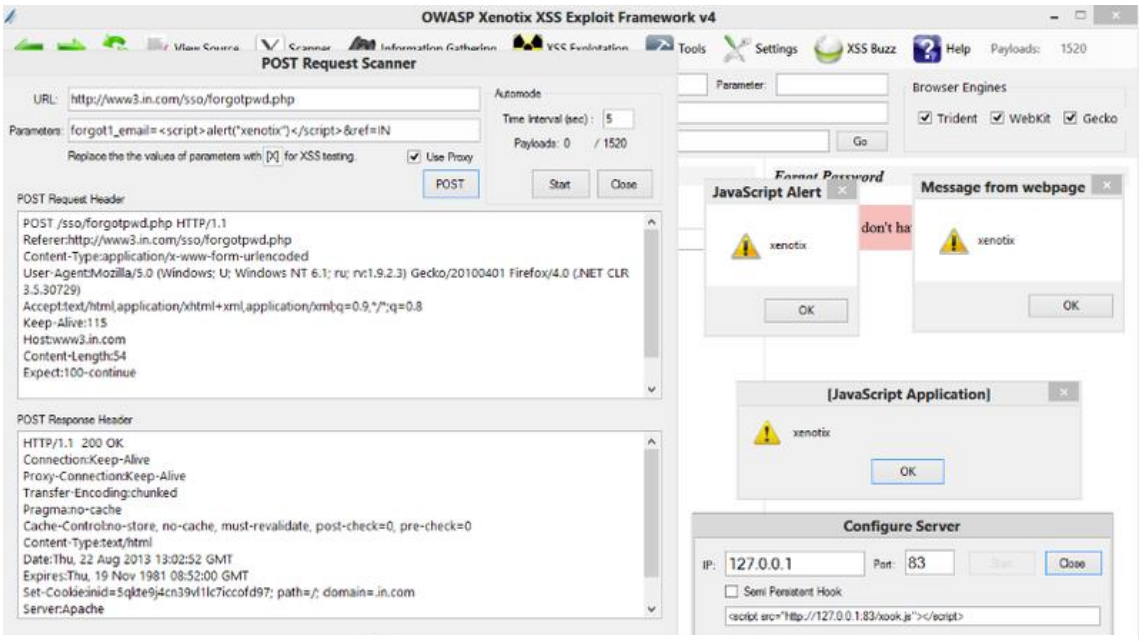


Ilustración 9 Owasp Framework

Vega

Vega es un escáner de seguridad web gratuito, de código abierto, y una plataforma de pruebas que se encarga de probar la seguridad de las aplicaciones web. Vega puede ayudarnos a encontrar y validar vulnerabilidades como el Cross-Site Scripting (XSS). Está escrito en Java, basado en GUI, y se ejecuta en Linux, OS X y Windows. Vega se ejecuta en dos modos de operación: como un escáner automático y como un proxy que intercepta vulnerabilidades de este tipo.

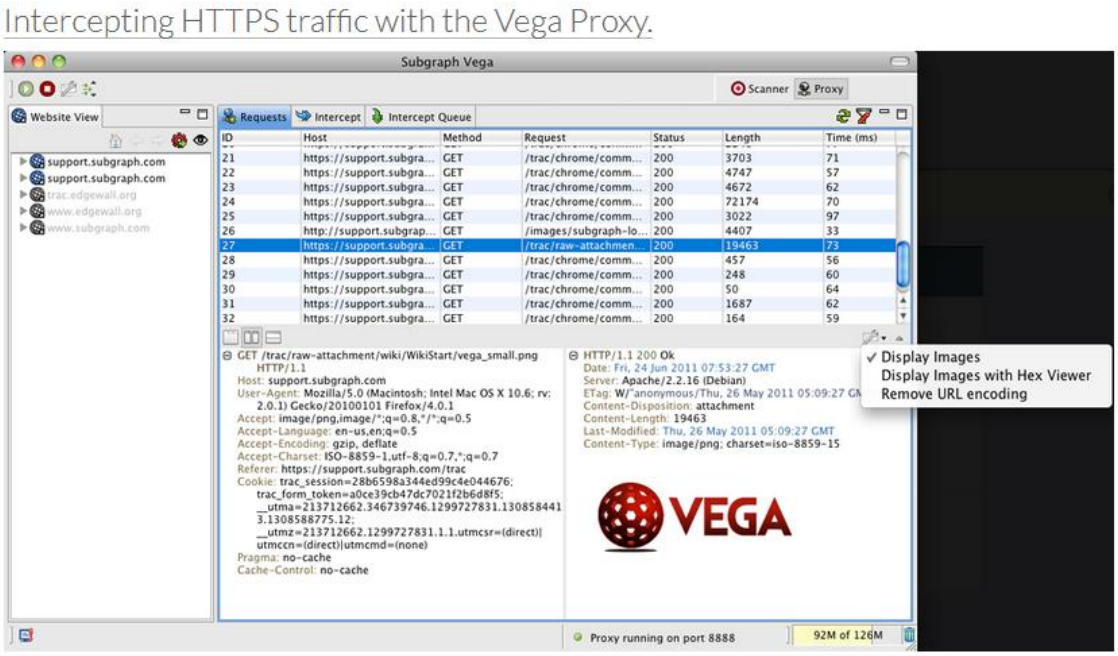


Ilustración 10 Vega

Zed Attack Proxy (ZAP)

El Zed Attack Proxy (ZAP) de OWASP es una herramienta muy utilizada a nivel mundial y además es de licencia libre. Esta herramienta se mantiene gracias a la colaboración de cientos de voluntarios internacionales. Su función principal consiste en ayudarnos a encontrar vulnerabilidades de seguridad en nuestras aplicaciones web de forma automática mientras las desarrollamos y probamos. ZAP puede ser utilizada como servidor proxy, permitiendo a los usuarios manipular todo el tráfico que pasa a través de éste, incluyendo el tráfico del protocolo seguro HTTPS. Esta herramienta puede ser utilizada para detectar múltiples vulnerabilidades.

Entre las características disponibles se encuentran:

- Servidor proxy de interceptación.
- Rastreadores web tradicionales y por AJAX.
- Escáner automatizado.
- Escáner pasivo.
- Navegación forzada.
- Fuzzer
- Soporte para WebSocket.
- Lenguajes de scripting y compatibilidad con Plug-n-Hack.

ZAP es una herramienta muy sencilla de utilizar y que nos proporciona información muy valiosa para conseguir un software seguro.

Acunetix

Es uno de los escáneres de vulnerabilidades web más importantes del mercado, la licencia de Acunetix es privada, aunque dispone de una versión de prueba. Incluye la detección de vulnerabilidades como la inyección SQL y la tecnología de escaneo de XSS. Es capaz de rastrear automáticamente los sitios web y replicar técnicas de hackeo detectando de esta forma vulnerabilidades peligrosas que pueden comprometer nuestros sitios web y sus datos.

Acunetix realiza pruebas para posibles ataques de inyección SQL, XSS, XXE, SSRF, inyección de encabezado de host y más de 4500 vulnerabilidades web. Tiene las técnicas de escaneo más avanzadas que generan los menos falsos positivos posibles. Simplifica el proceso de seguridad de la aplicación web a través de sus características integradas de administración de vulnerabilidades que lo ayudan a priorizar y administrar la resolución de vulnerabilidades.

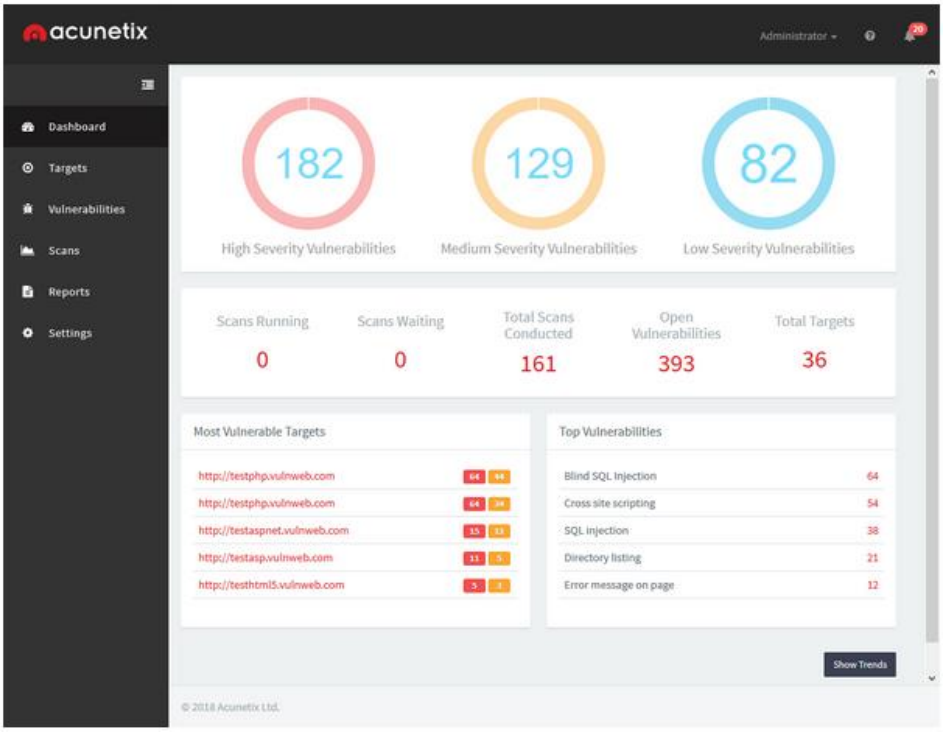


Ilustración 11 Acunetix

Nessus

Nessus es un programa de escaneo de vulnerabilidades, cuya licencia es privada, que funciona en diversos sistemas operativos. Consiste en un daemon que realiza un escaneo en el sistema y muestra el avance de dicho escaneo e informa sobre el estado de este.

Nessus permite escaneos para los siguientes tipos de vulnerabilidades:

- Vulnerabilidades que permiten a un hacker controlar o acceder a datos confidenciales en un sistema.

- Vulnerabilidades provocadas por una configuración incorrecta.
- Vulnerabilidades provocadas debido al uso de contraseñas predeterminadas, uso de contraseñas comunes y ausencia de estas.
- Denegaciones de servicio TCP / IP mediante el uso de paquetes mal formados.

Procedimientos para la mitigación de vulnerabilidades de software

En este apartado introduciremos los procedimientos que consideramos necesarios para la mitigación de vulnerabilidades durante el desarrollo del software.

Dado que el desarrollo del software se puede dividir en varias fases, hemos decidido dividir estos procedimientos según las fases del desarrollo del software en las que nos encontremos, ya que de esta forma consideramos que será más sencillo aplicar y diseñar dichos procedimientos, del mismo modo que será más metódico y fácil de recordar para el desarrollador o para los equipos involucrados en el desarrollo del software (equipos de diseño, de desarrollo, de pruebas, etc.).

Podemos diferenciar el ciclo de vida del desarrollo del software en:

- Fase de diseño.
- Fase de desarrollo.
- Fase final o de despliegue.

El primero de los procedimientos que vamos a presentar está enfocado en la fase de diseño de nuestro software, mientras que el segundo está enfocado en la fase de desarrollo del código, el tercer procedimiento está enfocado a una fase final del desarrollo del software, la que sería la fase de despliegue, y mediante el procedimiento que vamos a presentar pretendemos evitar que las vulnerabilidades que puedan existir lleguen a producción o al destino que le corresponda a dicho software.

En este apartado nuestro objetivo es proponer una metodología que minimice las vulnerabilidades de software en cada una de las fases presentadas.

Procedimiento de mitigación de vulnerabilidades durante la fase de diseño

Introducción

La fase de diseño del software es una fase crítica para la generación de un software seguro y resiliente ya que es la fase en la que definiremos nuestro sistema.

Los atacantes parecen estar particularmente motivados para atacar sistemas de software que contienen datos confidenciales, con el objetivo de comprometer los datos. Ya que de esta forma pueden obtener información crítica acerca de ganancias financieras, manipulación social o política e incluso poner en evidencia a la organización que se encarga de gestionar los datos. Dependiendo de la ubicación de los datos confidenciales pueden surgir vulnerabilidades en el software, lo cual supone un riesgo para dicha información.

En este documento, basándonos en el estudio realizado por G. Yee (2018), proponemos un método para identificar y eliminar durante la fase de diseño las vulnerabilidades

asociadas a los datos confidenciales con los que trabaje nuestra aplicación. Aplicando el método que proponemos a continuación podemos eliminar dichas vulnerabilidades.

El método proporcionará protección a los datos confidenciales en sus correspondientes ubicaciones o nos propondrá cambiarlas. Este método realiza estas modificaciones aplicando un modelo grafico del sistema de software que es fácil de usar y posteriormente traduce las modificaciones al sistema real.

El método utilizaremos se basa en la observación de que estas vulnerabilidades surgen debido a la ubicación de los datos confidenciales en el sistema y los riesgos para los datos en esos lugares. El método emplea un modelo gráfico fácil de usar, el Modelo para la Identificación y Eliminación de Vulnerabilidades (MVIR), para representar el sistema de software. El método primero utiliza el modelo MVIR para identificar las vulnerabilidades, posteriormente modifica el modelo MVIR para eliminar vulnerabilidades ya sea protegiendo los datos en sus ubicaciones o cambiando las ubicaciones de los datos. Finalmente, el método traduce estas modificaciones al diseño del sistema real. Aunque en principio, también se puede aplicar el método a un sistema de software ya implementado, hacerlo sería más costoso, ya que pueden requerirse grandes cambios en el sistema para cambiar las ubicaciones de los datos confidenciales.

En este apartado explicaremos los siguientes puntos:

a) Describir MVIR,
b) Presentar un método para la identificación y eliminación de vulnerabilidades en el momento del diseño utilizando MVIR
c) Ilustrar el método con los ejemplos.

Datos sensibles, ataques y vulnerabilidades

Los datos confidenciales son aquellos que incluyen información privada o personal, que contienen información sobre una persona, que pueden identificarla y que son de su propiedad. Por ejemplo, la altura, el peso o el número de la tarjeta de crédito pueden usarse para identificar a una persona y, por lo tanto, se consideran información personal o información confidencial personal. La privacidad de la persona se refiere entonces a su capacidad para controlar la recopilación de sus datos (qué información personal y qué tipo de información se ha recopilado), el propósito de esta recopilación, la retención y la divulgación de esa información por terceras partes, según lo establecido en las preferencias de privacidad de la persona. Para mayor comodidad, deberemos etiquetar estos parámetros de preferencia de privacidad como qué información se recopila, quién recopila dicha información, el propósito, el tiempo que dicha información va a estar retenida y la divulgación de dicha información, respectivamente. Los datos confidenciales también incluyen información no personal que puede comprometer la competitividad de una empresa si dicha información se divulga, como secretos comerciales o algoritmos. Para las organizaciones gubernamentales, los datos confidenciales no personales pueden incluir información que es vital para la seguridad del país del cual es responsable la organización gubernamental.

DEFINICIÓN 1: Los datos confidenciales (SD de aquí en adelante) son información que debe protegerse del acceso no autorizado para salvaguardar la privacidad de una persona, el bienestar o el funcionamiento esperado de una organización, o el bienestar y el funcionamiento esperado de una entidad.

DEFINICIÓN 2: Un ataque es cualquier acción que se lleve a cabo contra los datos confidenciales y que se lleva a cabo en el sistema informático de una organización que, si tiene éxito, hace que se comprometa la información protegida.

DEFINICIÓN 3: Una vulnerabilidad de un sistema de software es cualquier debilidad en el sistema que pueda ser atacada. Una vulnerabilidad de datos confidenciales se caracteriza por la ubicación de los datos confidenciales dentro del sistema y los riesgos de ataque a la dicha información en esa ubicación.

Identificación y eliminación de vulnerabilidades

El método para la identificación y eliminación de vulnerabilidades consiste en:

1. Modelar el sistema de software usando MVIR, identificar todas las ubicaciones en el sistema de software donde puede residir la SD.

2. Evaluar el modelo MVIR para los riesgos de SD en estas ubicaciones, lo que lleva a identificar las ubicaciones como vulnerabilidades.

3. Eliminar las vulnerabilidades en el modelo MVIR protegiendo la SD en su ubicación, o cambiando la ubicación de la SD.

El primer paso requiere un buen conocimiento del sistema de software, mientras que el segundo paso requiere conocimientos sobre las vulnerabilidades de seguridad. El tercer paso necesita creatividad, con conocimiento de mitigación de explotaciones de seguridad.

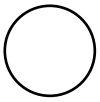
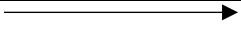
MVIR asume las siguientes características comunes de los sistemas de software:

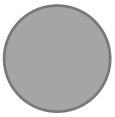
a) El sistema de software utiliza SD como parte de llevar a cabo su función (por ejemplo, usar valores secretos en un algoritmo).

b) El sistema de software puede transmitir el SD (por ejemplo, moverlo de un módulo a otro dentro del sistema de software) y almacenar el SD (por ejemplo, almacenarlo en una base de datos).

Basado en las vulnerabilidades que surgen de las ubicaciones de SD, MVIR tiene elementos que sirven para representar estas ubicaciones, como se muestra en la Tabla 1. En esta tabla, los elementos de dibujo se utilizan para modelar el diseño del sistema de software. Los elementos de seguridad se utilizan para modelar las mitigaciones de seguridad.

Tabla 1 Representación de ubicaciones y modelado de vulnerabilidades

Elementos de dibujo	Descripción
Círculo de uso 	Identifica donde se usa el SD, se etiqueta con un texto y se describe en dicha leyenda.
Almacén de datos 	Identifica donde se guarda la información confidencial, se etiqueta con un texto y se describe en dicha leyenda.
Flujo de los datos confidenciales 	Identifica el movimiento de los datos confidenciales de una localización a otra,

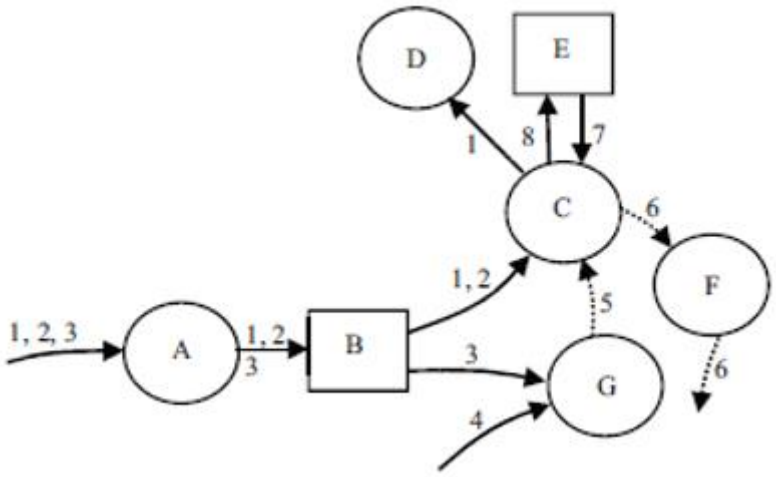
	se etiqueta con un texto y se describe en dicha leyenda.
Flujo de datos no confidenciales	Identifica el movimiento de los datos no confidenciales de una localización a otra, se etiqueta con un texto y se describe en dicha leyenda.
Elementos de seguridad	Descripción
Protección antimalware mediante uso de círculo 	Identifica donde se utiliza la información confidencial con protección antimalware, se etiqueta con un texto y se describe en dicha leyenda.
Almacén de datos ofuscado 	Identifica donde el almacén de la información confidencial ha sido ofuscado, se etiqueta con un texto y se describe en dicha leyenda.
Flujo de datos SD ofuscado	Identifica el movimiento ofuscado de los datos no confidenciales de un sitio a otro, se etiqueta con un texto y se describe en dicha leyenda.
Reducción de la superficie de ataque 	encierra círculos y almacenes de datos que se ejecutan en la misma plataforma de computación, lo que reduce la superficie de ataque de flujo de datos transversal
Descripción de elemento	Descripción
leyenda	Descripciones correspondientes a los textos o números, se colocan encima de los elementos donde están etiquetados.

Identificación de vulnerabilidades con MVIR

1. Dibujar el modelo: deberemos dibujar los círculos de uso y los almacenes de datos en el sistema junto con las rutas de los flujos de datos, según los puntos a) y b) definidos anteriormente, es decir, teniendo en cuenta que los datos se pueden transmitir, almacenar y usar. Etiquetaremos los círculos con letras, y los flujos de datos con números. Cada una de estas etiquetas corresponde a una descripción sobre su función, el tipo de almacenamiento o el tipo de datos. La siguiente imagen de ejemplo ilustra este paso para el sistema de software de un vendedor de productos en línea, como podría ser Amazon u otro gran almacén, este almacén requiere almacenar información como el nombre del usuario, la dirección, el artículo seleccionado y la información de la tarjeta de crédito. Estos se consideran como tres elementos SD personales (1, 2, 3), donde el nombre y la dirección juntos se consideran como un solo elemento. La figura también muestra tres flujos de SD no personales (propiedad de la empresa) (4, 7, 8) y dos flujos de no-SD (5, 6).

2. Evaluar el modelo: deberemos inspeccionar el modelo resultante del paso anterior y, para cada ubicación (flujo de datos, almacén de datos y círculo de uso) y cada elemento de SD personal, identificaremos los riesgos de dicho SD enumerando las posibles formas en que la información puede ser atacada, colector, propósito, tiempo de retención y divulgación. De manera similar, para cada ubicación de la SD (todas las SD), identificaremos los riesgos de la SD enumerando las posibles formas en que se puede acceder, robar o modificar la SD.

   Estas enumeraciones se basan en la experiencia y el conocimiento tanto de la seguridad como del sistema de software, siempre podemos ayudarnos haciéndonos preguntas sobre "cómo", como se propone en la Tabla 2. Registraremos los resultados en una Tabla de Vulnerabilidades que contiene tres columnas: la primera columna la utilizaremos para enumerar las vulnerabilidades, la columna central para los registros del formulario "(SD1, SD2, ... / ubicaciones que sean vulnerables)" y la tercera columna para describir los riesgos correspondientes de la SD. Una violación una colección de datos, del propósito de dichos datos, del tiempo de retención y de divulgación se denomina una violación de CPRD. La Tabla de Vulnerabilidades es el objetivo de este paso. La Tabla 3 ilustra este paso para el vendedor en línea de la Ilustración. 12. En esta tabla, las vulnerabilidades en la parte superior están numeradas con un prefijo "P" (de Personal) (por ejemplo, "P.1"); los que están en la parte inferior están numerados con un prefijo "A" (de Todos).



Leyenda	
A: recibir y guardar información	1: nombre y dirección (SD)
B: base de datos del cliente (SD)	2: ítem seleccionado (SD)
C: comprobar inventario y envío	3: información de la tarjeta de crédito (SD)
D: imprimir etiqueta de envío	4: número de cuenta de la compañía (SD)
E: base de datos del inventario	5: estado del pago
F: notificar al cliente el estado del envío	6: estado del envío
G: cargo a la tarjeta de crédito	7: datos del inventario (SD)
	8: actualización del inventario (SD)

Ilustración 12 Modelo MVIR de un vendedor online

Tabla 2 Preguntas de ayuda

Ataque a la información SD personal	Preguntas de riesgo para la información personal SD
Que	¿Cómo se le puede pedir al usuario otra SD personal, ya sea intencional o involuntariamente?
Colector	¿Cómo se puede recibir la SD personal por parte de un recolector involuntario, además de o en lugar del recolector deseado?
Propósito	¿Cómo se puede utilizar la SD personal para otros fines?
Tiempo de retención	¿Cómo se puede violar el tiempo de retención de la SD personal?
Revelar a	¿Cómo se puede divulgar la SD personal de forma intencional o inadvertida a un destinatario no deseado?
Todos los ataques SD	**Preguntas de riesgo para todo SD**
Robo de datos	¿Cómo se puede acceder y robar la información SD?
Modificación de datos	¿Cómo puede la SD ser accedida y modificada?
Robo y modificación de datos	¿Cómo puede ser accedida la SD, robada y modificada en el sistema?

Tabla 3 Vulnerabilidades

p	Vulnerabilidades (Personal SD/Localizaciones vulnerables)	Riesgos y localizaciones vulnerables para la SD personal
1	(1,2,3/ruta en A)	Al usuario se le puede pedir una SD personal que viole Que, por ejemplo, una SD personal diferente a la que el usuario esperaba.
2	(1,2,3/ruta en A), (1,2,3/ruta en B), (1,2 /ruta en C), (1 /ruta en D), (3/ruta en G)	Un tercero realiza un ataque y roba los datos, violando el CPRD
3	(1,2,3/ruta en A), (1,2 /ruta en C), (1 /ruta en D), (3/ruta en G)	Un troyano o hacker realiza un ataque, y consigue robar datos violando el CPRD
4	(1,2,3/ruta en B)	Ataque de SQL en el almacenamiento de datos, robando información y violando el CPRD
5	(1,2,3/ruta en B)	Los datos en la base de datos están guardados más tiempo del que se tenía previsto
A	**Vulnerabilidades (Todas SD/localizaciones vulnerables)**	**Riesgos en las localizaciones vulnerables para todos los SD**
1	(1,2,3/ruta en A), (1,2,3/ruta en B), (1,2 /ruta en C), (1 /ruta en D), (3/ruta en G)	Un tercero realiza un ataque modificando la SD
2	(1,2,3/ruta en A), (1,2 /ruta en C), (1 /ruta en D), (3/ruta en G)	Un ataque troyano o hacker realiza un ataque en el punto marcado con un circulo modificando la SD

3	(1,2,3/ruta en B)	Un ataque interno o de SQL en los datos almacenados modifica la SD
4	(4/ruta en G), (7/ruta en C), (8/ruta en E)	Un tercero ataca, roba o modifica la información SD
5	(4/ruta en G), (7/ruta en C)	Un troyano o atacante realiza un ataque en el punto marcado con un circulo robando o modificando la información
6	(8/E)	Un ataque SQL o interno en la información guardada roba o modifica dicha información

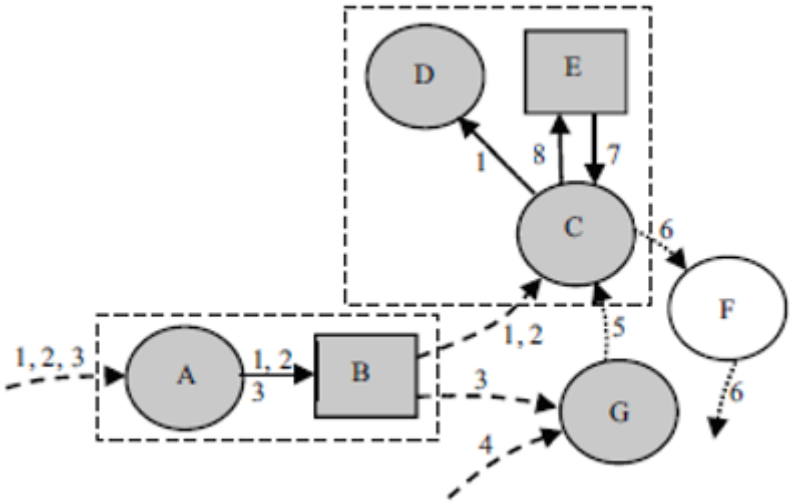
Un modelo MVIR muestra qué SD se requiere, dónde va la SD, dónde se almacena la SD y dónde se usa la SD, lo que corresponde a la idea de que la ubicación de los datos confidenciales y los riesgos de ataque en esos lugares son clave para identificar las vulnerabilidades. Es importante mostrar todos los flujos de datos en el modelo, incluidos los flujos de datos que no son SD. Hacerlo ayuda a garantizar que no falten flujos y validar que los flujos que no son SD realmente no lo son.

Eliminar las vulnerabilidades

1. Para cada vulnerabilidad de MVIR registrada la Tabla de Vulnerabilidades, tendremos que llevar a cabo un proceso de mitigación añadiendo protección para la SD en dicha ubicación o cambiando la ubicación de la SD. Esto último puede requerir cierta creatividad, por ejemplo, quizá sea necesario cambiar la funcionalidad para que la SD ya no sea necesaria en una ubicación particular, siempre que la nueva funcionalidad sea aceptable y cumpla con los requisitos del sistema.

2. Deberemos proponer cambios al equipo de desarrollo del sistema de software, en función del modelo modificado resultante del paso anterior. El equipo de desarrollo identificará y descartará los cambios que no son factibles debido a otros factores, como el rendimiento, las restricciones presupuestarias, la escalabilidad y la facilidad de implementación.

Al aplicar el paso 1 de este apartado a la Ilustración 12, se obtiene la Ilustración 13, en la que se ve que las vulnerabilidades de ataque de un tercero se eliminaron al aplicar el elemento del modelo de “Reducción de la superficie de ataque” y el elemento del modelo “Flujo de datos SD ofuscado”. Las vulnerabilidades de ataque debido a un troyano se han eliminado utilizando el elemento “Protección antimalware mediante uso de círculo”. Las vulnerabilidades de ataque de SQL se han eliminado utilizando el elemento de “Almacenamiento de datos ofuscados” (cifrado). Las vulnerabilidades restantes son: violación sobre la petición de información SD al usuario (“Que”, descrito en la Tabla 2) personal en el flujo de SD personal en el punto A, ataques de piratas informáticos en las ubicaciones donde se utilizan los datos, ataques internos en los almacenes de datos y “Violación de tiempo de retención”, definido en la Tabla 2. Estas vulnerabilidades se pueden controlar mejor con medidas que son externas al diseño, como el uso de firewalls, y los mecanismos de auditoría para el tiempo de retención de la información confidencial.



Leyenda	
A: recibir y guardar información	1: nombre y dirección (SD)
B: base de datos del cliente (SD)	2: ítem seleccionado (SD)
C: comprobar inventario y envío	3: información de la tarjeta de crédito (SD)
D: imprimir etiqueta de envío	4: número de cuenta de la compañía (SD)
E: base de datos del inventario	5: estado del pago
F: notificar al cliente el estado del envío	6: estado del envío
G: cargo a la tarjeta de crédito	7: datos del inventario (SD)
	8: actualización del inventario (SD)

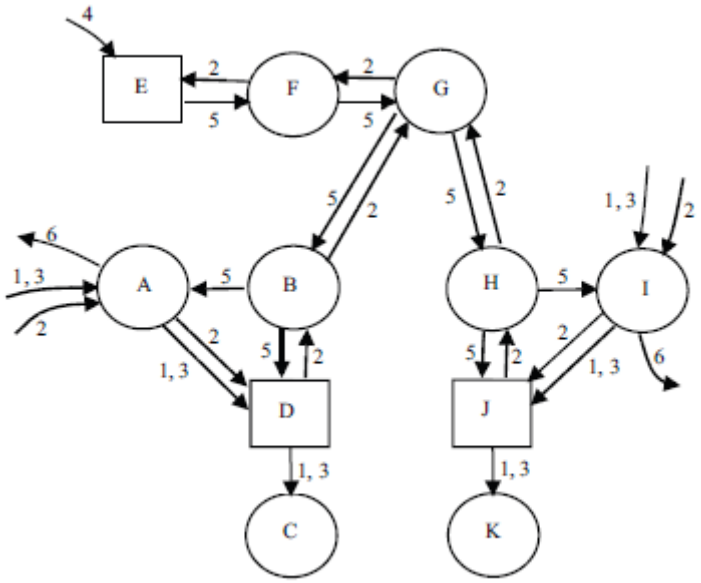
Ilustración 13 Modelo MVIR de la Ilustración 12 después de eliminar las vulnerabilidades

Aplicar el paso 2 descrito anteriormente significa recomendar al equipo de desarrollo que se cambie el diseño real del sistema de software de acuerdo con la Ilustración 13, lo que se considera factible para este ejemplo.

Los procedimientos de identificación de vulnerabilidades con MVIR y de eliminación de vulnerabilidades explicados anteriormente pueden ser aplicados por un equipo de mitigación, compuesto por tres o cuatro personas, seleccionados por sus conocimientos técnicos del sistema de software y dirigido por un analista de seguridad y privacidad. Los candidatos para el equipo deben incluir un arquitecto de diseño y el arquitecto de pruebas del sistema de software. El analista de seguridad y privacidad también debe conocer el sistema de software y contar con el apoyo de los arquitectos de software para llevar a cabo el cambio de requisitos.

Otro ejemplo

Consideremos un sistema de reserva de vuelos llamado AccuReserve, que tiene dos módulos de ejecución, uno de ellos Main en Canadá y el otro Mod-EU en Alemania.



Leyenda	
A, I: recibir y guardar información	1: detalles de identificación (personal SD)
B, H: comunicación con G	2: petición de detalles de vuelos (no personal SD)
C, K: cobrar a la tarjeta de crédito	3: detalles de pago (personal SD)
E: bases de datos de vuelos, no personal SD	4: disponibilidad de vuelos y actualizaciones (no personal SD)
D: bases de datos de clientes, personal SD	5: detalles de vuelos asignados
F: gestor de disponibilidad de vuelos	6: itinerario de viaje
G: comunicación con módulos	

Ilustración 14 Model MVIR

Como hemos descrito anteriormente en la fase de identificación de vulnerabilidades con MVIR deberemos diseñar el modelo MVIR para el sistema AccuReserve, en este caso nos quedaría un diagrama como el que vemos en la Ilustración 14.

El siguiente paso de la fase de identificación de vulnerabilidades sería el de proporcionar una tabla de vulnerabilidades para las diferentes partes que vemos en la Ilustración 14. La Tabla 4 nos proporciona dicha tabla de vulnerabilidades, en la que vemos los riesgos de SD.

Los riesgos personales de SD se han obtenido al examinar las ubicaciones de SD personales de la Ilustración 14, también se han identificado considerando las preguntas de riesgo para SD que mostrábamos en la Tabla 2. Todos los riesgos de SD que mostramos

en la Tabla 4 se obtuvieron al considerar las preguntas de riesgo para todos los SD en la Tabla 2.

Tabla 4 Vulnerabilidades obtenidas a partir de las preguntas de riesgo

p	**Vulnerabilidades: (Personal SD/localizaciones vulnerables)**	**Riesgos en las localizaciones vulnerables, para el SD personal**
1	(1,3 / ruta en A), (1,3/ ruta en I), (6/ ruta de A), (6/ruta de I)	Un tercero ataca y roba datos violando el CPRD.
2	(1,3 / ruta entre A y D), (1,3 / ruta entre I y J), (1,3/ ruta entre D y C), (1,3/ruta entre J y K)	Un tercero ataca y roba datos violando el CPRD.
3	(1,3 / ruta en A), (1,3 / ruta en I)	El usuario podría ser preguntado por información personal que viola la norma "Que", vista en la tabla 2.
4	(1,3 / A, I), (1,3 / C, K)	Un troyano o hacker ataca en A, I, C o K robando información y violando el CPRD.
5	(1,3 / D, J)	Un ataque de SQL o interno en D y J roba información. La información personal en D y J se mantiene retenida pasado el tiempo definido de retención de información.
A	**Vulnerabilidades (Todo SD/localizaciones de vulnerabilidades)**	**Riesgos en las localizaciones vulnerables para todo SD**
1	(1,3 / ruta en A), (1,3/ ruta en I), (6/ ruta de A), (6/ruta de I)	Un tercero ataca y modifica los datos.
2	(1,3 / ruta entre A y D), (1,3 / ruta entre I y J), (1,3/ ruta entre D y C), (1,3/ruta entre J y K)	Un tercero ataca y modifica los datos.
3	(2 / ruta en A), (2 / ruta en I), (2 / todas las rutas con el flujo 2)	Un tercero ataca, roba o modifica los datos.
4	(5 / todas las rutas con el flujo 5)	Un tercero ataca, roba o modifica los datos.
5	(Todo SD / D), (Todo SD / J), (Todo SD / E)	Un ataque SQL o interno en D, J o E roba o modifica información.
6	(4 / ruta de E), (6 / ruta de A), (6 / ruta de I)	Un tercero ataca, roba o modifica los datos.
7	(Todo SD / todo uso en los círculos)	Un troyano o hacker ataca en el uso del circulo robando o modificando información.

Fase de eliminación de vulnerabilidades:

Al examinar cada vulnerabilidad de la Tabla 4 y al considerar cómo se podría cambiar el modelo en de la Ilustración 14 para eliminar las vulnerabilidades, se obtiene la versión que contiene menos vulnerabilidades que podemos ver en la Ilustración 15. Las vulnerabilidades de las bases de datos D y J (filas P.) 5 y A.5 en la Tabla 4) se eliminaron al no tener estas bases de datos (un ejemplo de cambio de ubicación de SD), es decir, no se mantienen los historiales de vuelos de clientes anteriores, así como la información de pago. Las vulnerabilidades de ataque de terceros se eliminaron aplicando el elemento del modelo de "Reducción de la superficie de ataque" o utilizando el elemento del modelo de "Flujo de datos SD ofuscado". La vulnerabilidad de la base de datos E fue eliminada por el elemento "Almacén de datos ofuscado" (cifrado). Las vulnerabilidades del círculo de

uso se resolvieron utilizando el elemento “Protección antimalware mediante uso de círculo”. Deberemos tener en cuenta que las vulnerabilidades de los ataques de hackers en los círculos de uso, los ataques de información privilegiada en los almacenes de datos, la información personal que se mantiene más allá del tiempo de retención establecido y la solicitud de otra información personal permanecen.

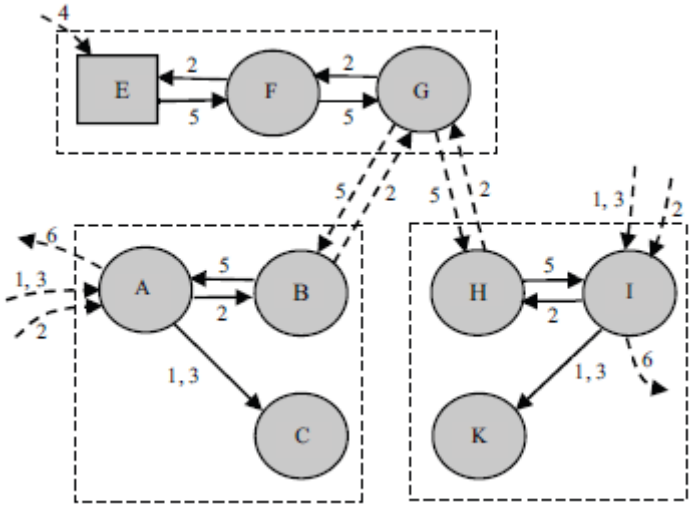
Nuestro objetivo aquí es solo eliminar cualquier vulnerabilidad que podamos con el método propuesto, no tienen por qué ser todas, posteriormente, en otra sección, aplicaremos otros métodos de mitigación.

Como resultado, obtenemos que las recomendaciones para el desarrollo del software y para la administración de esta aplicación son:

i) no almacenar solicitudes de vuelo de clientes, información de pagos y asignaciones de vuelo en bases de datos.

ii) ejecutar módulos en plataformas individuales como se indica en Fig. 4.

iii) emplear software de ofuscación de datos y antimalware como se indica en la Fig. 4. Además, utilizaremos el cifrado sin rechazo para los flujos de datos ofuscados de la Fig. 4. Todas las recomendaciones se consideran factibles.



Leyenda	
A, I: recibir y guardar información	1: detalles de identificación (personal SD)
B, H: comunicación con G	2: petición de detalles de vuelos (no personal SD)
C, K: cobrar a la tarjeta de crédito	3: detalles de pago (personal SD)
E: bases de datos de vuelos, no personal SD	4: disponibilidad de vuelos y actualizaciones (no personal SD)
D: bases de datos de clientes, personal SD	5: detalles de vuelos asignados
F: gestor de disponibilidad de vuelos	6: itinerario de viaje
G: comunicación con módulos	

Ilustración 15 Modelo MVIR de la Ilustración 14 después de haber eliminado las vulnerabilidades

Puntos fuertes y débiles

Algunos de los puntos fuertes del método son:

a) Es un enfoque de "diseño para la seguridad" que aborda la etapa de diseño del software, lo que significa que la seguridad está diseñada en el software desde el inicio del desarrollo.

b) Proporciona un procedimiento paso a paso fácil de entender para identificar y eliminar las vulnerabilidades de SD.

c) Emplea un modelo gráfico fácil de usar.

d) Se enfoca en las ubicaciones que tienen vulnerabilidades de SD, que son razonables y análogas a las físicas seguridad.

e) Permite eliminar vulnerabilidades al cambiar la ubicación de SD (por ejemplo, la eliminación de las bases de datos en AccuReserve descrito previamente), que puede ser más eficaz que el uso de medidas de seguridad tradicionales como el cifrado.

Algunos puntos débiles del enfoque son:

a) Dibujar el modelo MVIR es una tarea manual propensa a errores.

b) Nunca puede identificar todas las vulnerabilidades de SD.

c) requiere experiencia en privacidad, seguridad y sistemas de software.

Conclusiones

Este método agrega protección a nuestro software mediante el uso de elementos de MVIR que representan mitigaciones de seguridad. Aunque un equipo de desarrollo solo aceptará cambios factibles para el diseño real, entre todos los cambios que podamos encontrar, simplemente con la eliminación de una sola vulnerabilidad de SD puede ser suficiente para justificar el uso de este método, ya que nuestra aplicación será mejor y más resistente. Las vulnerabilidades no eliminadas por este método pueden mitigarse utilizando medidas externas al diseño.

Este procedimiento nos permite:

- Abordar el tratamiento de vulnerabilidades de software desde la etapa de diseño, por lo que habremos tenido en cuenta su seguridad desde el inicio del desarrollo.

- Nos proporciona un procedimiento sencillo para identificar y eliminar las vulnerabilidades de información crítica o confidencial.

- Utiliza un modelo gráfico, que es intuitivo y fácil de usar.

- Es un procedimiento que se enfoca en las ubicaciones que tienen vulnerabilidades de información crítica.

- Permite eliminar vulnerabilidades al cambiar la ubicación de la información confidencial, esta metodología puede ser más eficaz que el uso de medidas de seguridad tradicionales como el cifrado.

- Es aplicable a cualquier lenguaje de programación, por lo que es un método que podemos utilizarlo de forma universal en todos nuestros desarrollos.

Dada nuestra experiencia personal considero que este método es especialmente útil, ya que no siempre se tiene en cuenta la seguridad del software durante la fase de diseño, muchas veces nos encontramos con que necesitamos proteger información que es crítica durante la fase de desarrollo, debido a un mal diseño o planteamiento en la fase de diseño y necesitamos proteger la información crítica mediante métodos de codificación como puede ser el cifrado, aumentando el riesgo de exposición de la información clave.

Este método permite realizar un diseño con seguridad teniendo en cuenta la información más importante y crítica con la que va a trabajar nuestro software. Si tenemos en cuenta esta metodología durante la fase de diseño del código estaremos protegiendo la información crítica desde el principio, y como resultado nuestro software será más seguro y estará más preparado ante posibles ataques.

Procedimiento de mitigación de vulnerabilidades durante el desarrollo

La fase del desarrollo del software es una fase crítica, por este motivo deberemos apoyarnos de herramientas y metodologías para evitar errores que deriven en un mal funcionamiento del software por un mal desarrollo.

En primer lugar, una de las principales metodologías para evitar errores en el software que estamos desarrollando es el diseño de unas pruebas unitarias que se encarguen de testar nuestro código.

Implementación de pruebas unitarias

Como hemos explicado anteriormente, en el apartado de estado del arte, las pruebas unitarias nos ayudan a evitar que los errores lleguen a la fase final. A través de este tipo de pruebas podemos validar condiciones de contorno e impedir que vulnerabilidades tales como el desbordamiento de búfer lleguen a la fase final del desarrollo.

Nuestro objetivo como desarrolladores será realizar un correcto diseño de las pruebas unitarias, de tal forma que se compruebe el correcto funcionamiento de las unidades básicas del código. De esta forma comprobaremos automáticamente aspectos que de otra forma serían inviables de probar, o bien podrían requerir de un tiempo excesivo de revisión, en el caso de tener que realizar una revisión visual. Este tipo de pruebas nos van a permitir inspeccionar el código fuente para verificar que se está realizando un correcto uso de las buenas prácticas de programación, si los algoritmos implementados se comportan de manera adecuada, y además nos permiten detectar posibles errores o comportamientos inesperados en los módulos desarrollados.

Deberemos implementar casos de uso directamente en código fuente que será ejecutado durante el proceso de pruebas de forma automática. Si la ejecución no obtiene los resultados esperados se dice la prueba ha fallado, y por lo tanto se considera que el código probado es incorrecto. Los casos de pruebas que implementemos deberán ser tanto positivos como negativos. Todo el código desarrollado deberá ser probado de tal forma que sea ejecutado en su totalidad, con el objetivo de comprobar que funciona de forma correcta. Las pruebas deben ser independientes unas de otras, de tal forma que una prueba no pueda interferir en otra.

Existen multitud de Frameworks con los que podemos realizar las pruebas unitarias, podemos encontrar Typemock para el lenguaje C++ y C#, PHPUnit para pruebas unitarias

de PHP, JUnit para pruebas unitarias en Java, PyUnit para pruebas unitarias en Python, etc.

En este documento nos vamos a centrar en JUnit ya que, a nivel particular, normalmente utilizamos Java para realizar los desarrollos de software. A modo de ejemplo vamos a mostrar cómo funcionaría el Framework de JUnit y como se implementa.

JUnit es un framework de código abierto que permite realizar la ejecución de clases Java de forma controlada, con el objetivo de evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta de manera esperada. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que devolvió el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente, de esta forma podremos desarrollar nuestro código Java conociendo su comportamiento. Mediante la aplicación de JUnit seremos capaces de probar diferentes casos y situaciones en nuestro software, por ejemplo, podremos probar el comportamiento de nuestro código ante un error de tipo NullPointer, y controlar el comportamiento de la aplicación ante esta situación.

Cada una de las clases Java que desarrollemos deberán ir acompañadas por una clase tipo test, como buena práctica dicha clase test deberá tener el mismo nombre que la clase que estamos probando seguida por el sufijo Test y además deberá estar guardada en un package aparte que agrupe todos los test de Junit de nuestro proyecto.

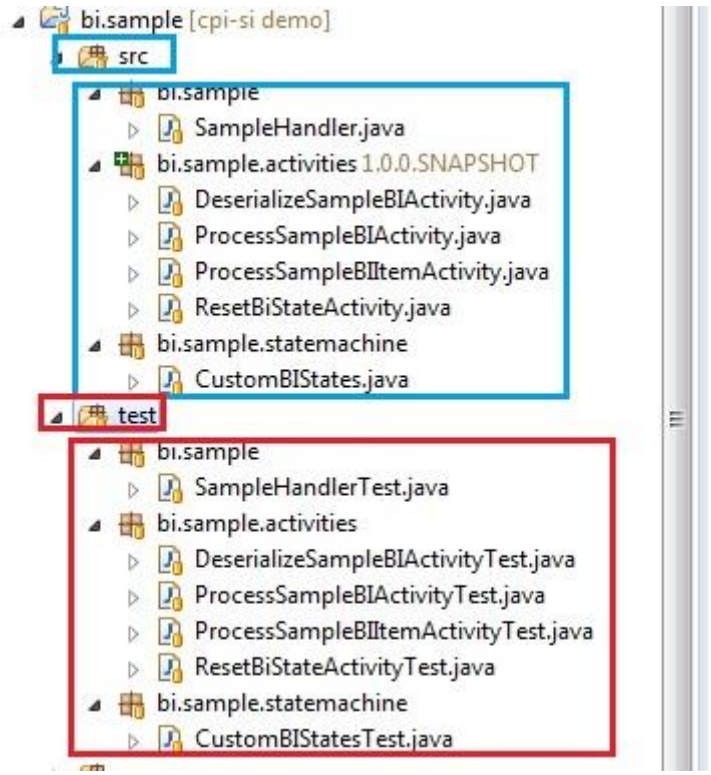


Ilustración 16 Pruebas unitarias

Para la realización de las pruebas unitarias también es interesante utilizar otras herramientas que nos ayudan a probar el código de forma correcta:

- **Mockito:** es un framework para emular objetos en test de Java. Este framework es muy útil, ya que se encarga de simular parte del comportamiento de una clase. Podremos definir la respuesta del objeto ante la llamada de un método de dicho objeto. Suelen ser configurados para simular un comportamiento determinado durante una prueba. Con Mockito emulamos las respuestas de los componentes que pudiera necesitar el método a probar de forma que solo se pruebe este último. Por ejemplo: emular la lectura de un archivo .properties a través de una clase.

- **ECLEmma:** es un plugin del IDE Eclipse para determinar la cobertura que las pruebas unitarias tienen sobre el código. Se recomienda que el 100% de la cobertura del código sea probada.

Ejemplo Junit:

```
@Test
public void prepare_withoutCatalogService()
{
    contenedorDatos = new DataContainerBuilder<>(null, null, null).addCPMService(servicioCliente).build();
    controladorRecargas.prepare(contenedorDatos);

    assertThat(contenedorDatos.getErrorCode().getStatus().getStatusCode(), equalTo(Status.INTERNAL_SERVER_ERROR.getStatusCode()));
}

@Test
public void prepare_withoutCpmService()
{
    contenedorDatos = new DataContainerBuilder<>(null, null, null).addCPMService(servicioCliente).build();
    controladorRecargas.prepare(contenedorDatos);

    assertThat(contenedorDatos.getErrorCode().getStatus().getStatusCode(), equalTo(Status.INTERNAL_SERVER_ERROR.getStatusCode()));
}
```

Ilustración 17 JUnit

Cada uno de los métodos Junit lleva la etiqueta @Junit, y son métodos de prueba independientes cuya función es probar una parte del código determinada.

A continuación, mostramos un ejemplo del uso de los mockitos:

```
@Mock
private CatalogService servicioCatalogo;

@Mock
private CPMService servicioCliente;

private RefillSpecificationHandler controladorRecargas = new RefillSpecificationHandler();

@Test
public void conclude() throws DataAccessException
{
    @SuppressWarnings ("rawtypes")
    DataContainer container = Mockito.mock(DataContainer.class);
    Context context = Mockito.mock(Context.class);
    CatalogService service = Mockito.mock(CatalogService.class);
    Mockito.when(container.getCatalogService()).thenReturn(service);
    Mockito.when(container.get(CatalogService.Key.CONTEXT_KEY.name())).thenReturn(context);

    controladorRecargas.conclude(container);
    verify(service).destroyContext(context, false);
}
```

Ilustración 18 Mocks

Como podemos observar estamos simulando el comportamiento de un objeto, de tal forma que cuando llamemos a un método en concreto del objeto que estamos simulando podremos devolver una respuesta que nosotros hemos simulado. De esta forma podremos continuar con la ejecución de la prueba, y podremos cubrir la ejecución completa de nuestro código simulando las respuestas de los componentes que hemos implementado.

Finalmente, mediante el plugin del IDE de Eclipse EclEmma podremos comprobar el porcentaje del código que hemos cubierto con nuestras pruebas, es decir la cantidad del código que se ha ejecutado y que hemos probado. Esta herramienta es de gran utilidad ya que si todo nuestro código ha sido ejecutado sin errores estaremos verificando que ninguna de nuestras implementaciones falla de forma individual y que el software implementado funciona como se espera.

Ejemplo de uso de EclEmma:

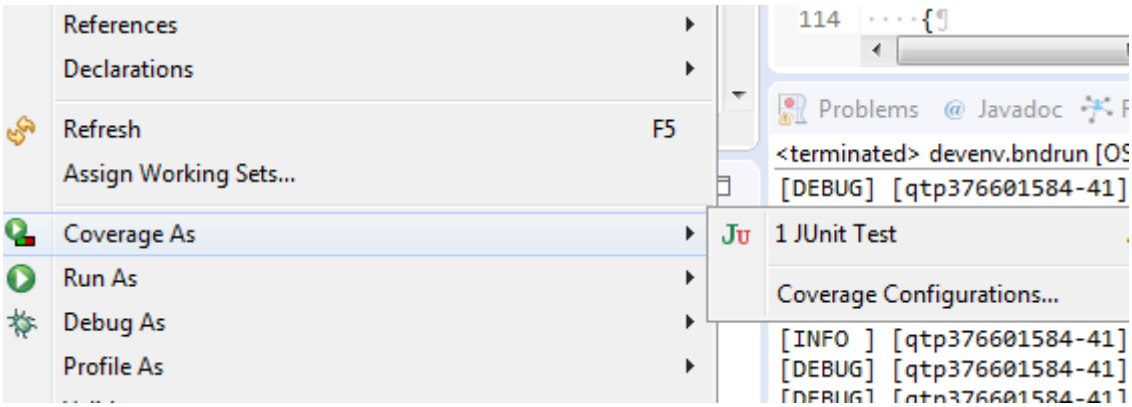


Ilustración 19 EclEmma

Nos mostrará en color verde la parte del código (tanto de la prueba como de la clase que estamos probando) que se ha ejecutado satisfactoriamente, o como esperaba la prueba:

```
@Test
public void execute_whenThrowsDataAccessException_shouldSetDataAccessExce
{
    when(veonChargingService.refill(any(RefillInformation.class), any(Req

    executeVoucherBasedRefill.execute(businessInteraction, container);

    assertThat(container.getErrorCode().getCode(), equalTo(ErrorCode.Code
}
```

Ilustración 20 Funcionamiento EclEmma

Y nos mostrará el porcentaje de código que se ha ejecutado con la prueba que hemos realizado:

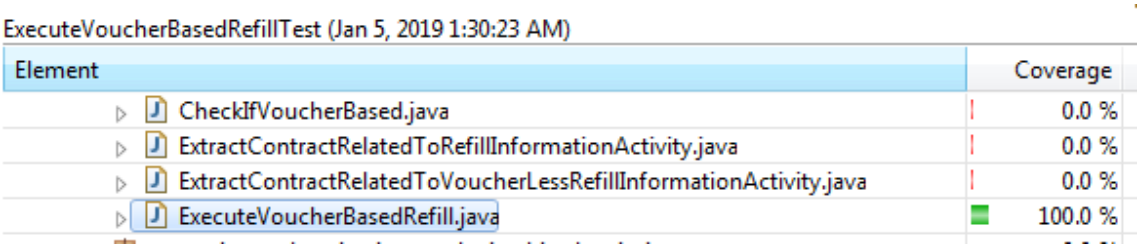


Ilustración 21 Porcentaje de ejecución

Si la prueba hubiera sido incorrecta o se hubiera quedado parte del código sin probar nos lo mostraría en color rojo, y nos devolvería un porcentaje de "Coverage" o de cobertura de código menor (en este caso podemos ver que una de las clases tiene el 100% del código ejecutado, mientras que el resto tienen un 0%, es decir, ese código no tendría unas pruebas unitarias establecidas o no se habrían ejecutado).

Para el caso de condicionales (como if, switch case, etc.) que no hayan sido probados en todos sus casos nos mostrará esa parte del código en amarillo y nos indicará que faltan condiciones por probar.

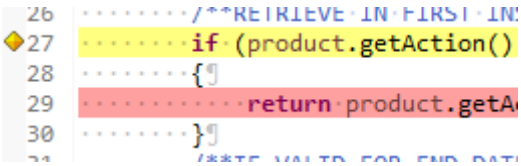


Ilustración 22 Condicionales en EclEmma

Como vemos estas herramientas resultan de gran utilizad para probar nuestro código de manera unitaria, verificando y asegurándonos de que todo nuestro software ha sido probado de forma satisfactoria. Esta es una de las formas mediante la cual podemos verificar y corregir posibles errores que vayan surgiendo durante la fase de desarrollo de nuestro código.

Como hemos dicho anteriormente en este caso concreto y por experiencia propia nos hemos centrado en explicar el uso de las pruebas unitarias con Junit y con otras herramientas asociadas, pero existen multitud de frameworks que nos permiten realizar las pruebas unitarias. Del mismo modo que existen multitud de herramientas que se encargan de medir el porcentaje de código cubierto por las pruebas unitarias, como son Cobertura, CodeCover, Emma, etc. Nuestro objetivo con este estudio es indicar que mediante el uso de estas herramientas y frameworks podemos realizar un código más seguro, siendo necesario para ello la realización de pruebas unitarias unidas con herramientas de medida de cobertura de código.

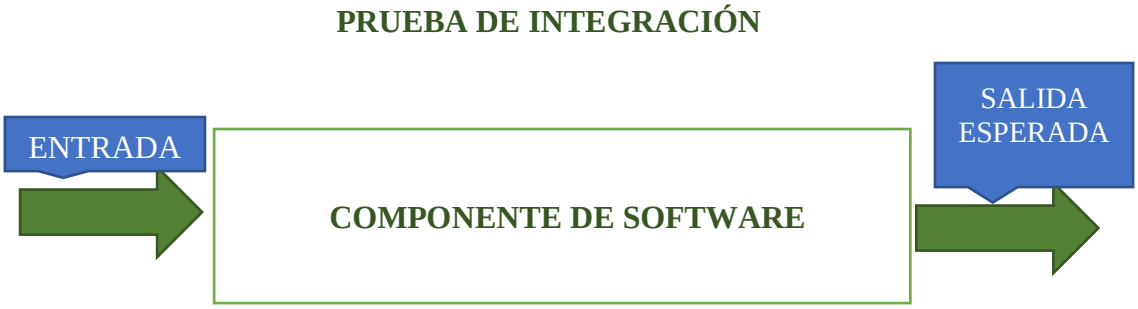
En las pruebas unitarias deberemos tener en cuenta la destreza del desarrollador en probar todo el software con cierto sentido y con ayuda de las herramientas descritas, el desarrollador además será responsable de realizar los casos de prueba oportunos, cubriendo todos los posibles casos que el software pueda tener. Ya que, aunque todo el código haya sido ejecutado y validado por las herramientas previamente descritas no debemos olvidar que son herramientas de apoyo que nos ayudan en nuestro trabajo. Es

nuestra responsabilidad realizar unas pruebas unitarias útiles que prueben todos los casos posibles, sobre todo aquellos en los que pensamos que nuestro software puede fallar (por ejemplo, un caso de desbordamiento de buffer, un nullpointer, etc.) ya que es la única forma mediante la cual podremos "blindar" nuestro software y controlar dichos errores. Las pruebas unitarias nos permiten realizar también test negativos para probar aquellos casos en los que nuestro software va a fallar de forma controlada.

Implementación de pruebas de integración

Como hemos definido previamente las pruebas de integración son las pruebas que se realizan en el ámbito del desarrollo de software una vez que han pasado todas las pruebas unitarias y su función principal es comprobar que todos los elementos unitarios de los que se compone el software funcionan juntos correctamente haciendo las pruebas correspondientes en grupo.

Este tipo de pruebas se centran en probar la comunicación entre los componentes ya sea hardware o software. Como hemos comentado en apartados anteriores estas pruebas se pueden considerar como pruebas de caja negra en las que tenemos que probar una funcionalidad simplemente introduciéndole unos datos de entrada y en consecuencia deberemos obtener una salida esperada. De esta forma podemos probar la funcionalidad de nuestro software. Se utilizan normalmente para comprobar que el sistema está integrado correctamente y que todas las funcionalidades del software desarrollado funcionan según los requisitos establecidos.

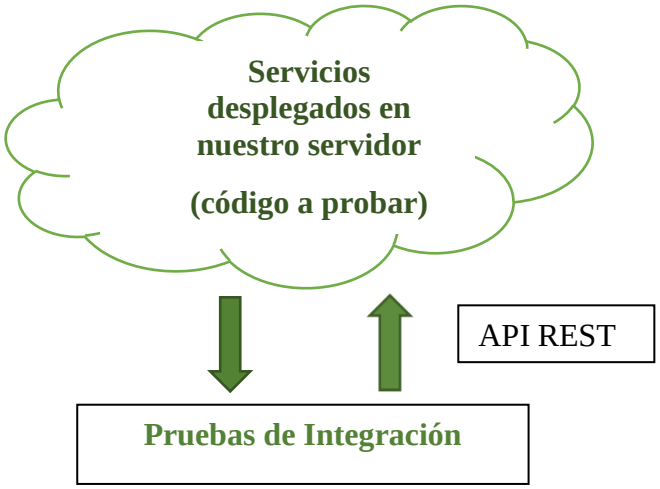


Para las pruebas de integración podemos utilizar las mismas herramientas que hemos visto para las pruebas unitarias, la diferencia entre las pruebas unitarias y las de integración es puramente conceptual, a nivel técnico utilizamos de la misma forma Junit o el framework de pruebas que hayamos elegido, así como las herramientas de apoyo para medir la cobertura de código ejecutado y las simulaciones de clases como hemos explicado anteriormente con mockito. El concepto fundamental de este tipo de pruebas es el de comprobar que el sistema de software funciona como corresponde según las especificaciones establecidas en los requisitos y que los diferentes componentes de software (en el caso de que los tenga) funcionan correctamente entre ellos.

En nuestro caso tenemos la experiencia de haber desarrollado microservicios con OsGi, por lo que una prueba de integración adecuada sería comprobar que todos los microservicios desarrollados (cada uno es un componente de software (bundle) independiente) funciona correctamente entre sí, devuelve los datos de forma adecuada y a partir de una entrada de datos establecida el sistema devuelve una salida esperada y que se adapta a nuestros requisitos.

Las pruebas unitarias explicadas anteriormente no se centraban en la funcionalidad del sistema, su función principal era comprobar que el software desarrollado de forma unitaria e independiente funcionaba correctamente, así como que todo el código desarrollado era ejecutado, para comprobar que ningún elemento daba errores. El tipo de pruebas de integración se preocupa de la funcionalidad de los componentes de software desarrollado, no se centra en el código de forma individual, sino que prueba el sistema en conjunto verificando que funciona.

A modo de ejemplo ilustrativo mostraremos como haríamos una prueba de nuestro sistema compuesto por diversos microservicios y componentes. Tendremos varios servicios, los cuales hemos desarrollado previamente, y que queremos probar de forma funcional. Nuestra prueba consistiría en primer lugar en desplegar los servicios que hemos desarrollado en un servidor correspondiente y posteriormente realizaríamos las pruebas sobre dichos servicios.



En este caso como estamos probando servicios web nos tendremos que comunicar con ellos a través de protocolos de comunicación como API REST, pero como vemos el concepto de pruebas de integración es que a través de unos test que definimos previamente probamos el comportamiento del código desarrollado (en nuestro caso los microservicios), y deberemos analizar los resultados obtenidos en función de los datos enviados tal y como está establecido en los requisitos del sistema en los que nos hemos basado para el desarrollo de nuestro software.

Este tipo de pruebas de integración se ejecutarán de forma automática una vez que las hayamos definido, igual que ocurría con las pruebas unitarias. Pero durante el desarrollo de los servicios, o de cualquier otro sistema de software, podremos realizar pruebas a la vez que vamos haciendo el desarrollo de nuestro código, de esta forma podremos afinar nuestros desarrollos y probablemente nos resultará más fácil trabajar, es decir, no tendremos que esperar a tener el sistema de software completamente terminado para empezar a probarlo. En nuestra opinión es una buena práctica ir realizando ciertas pruebas de apoyo mientras vamos desarrollando el sistema. Para esto también nos podemos apoyar en diversas herramientas.

Para probar en nuestro caso la funcionalidad del software que hemos descrito podemos apoyarnos, además de en las mismas herramientas que usábamos para el desarrollo de los JUnit, en otras herramientas como pueden ser Postman y SOAP UI, estas dos herramientas nos permiten comunicarnos con los servicios web que queramos y enviarles peticiones de tipo XML o JSON y recibir respuestas en el mismo formato de datos. De

esta forma podremos verificar antes del desarrollo de las pruebas de integración que nuestro desarrollo está funcionando de forma correcta de manera visual, aunque estas dos herramientas también permiten implementar pruebas automáticas.

En este caso hemos escogido el uso de Postman ya que es una herramienta de software libre que permite elaborar o importar colecciones con distintas pruebas configuradas. Una colección puede contener distintas pruebas que a su vez están formadas por distintos pasos que sirven para probar el código. Por lo que para el caso de Postman podremos definir un conjunto de pruebas de integración y verificar de forma automática el correcto funcionamiento de nuestro sistema de software con esta herramienta.

Una vez que nuestros servicios están desplegados en el servidor correspondiente procederemos a probarlos mediante esta herramienta:

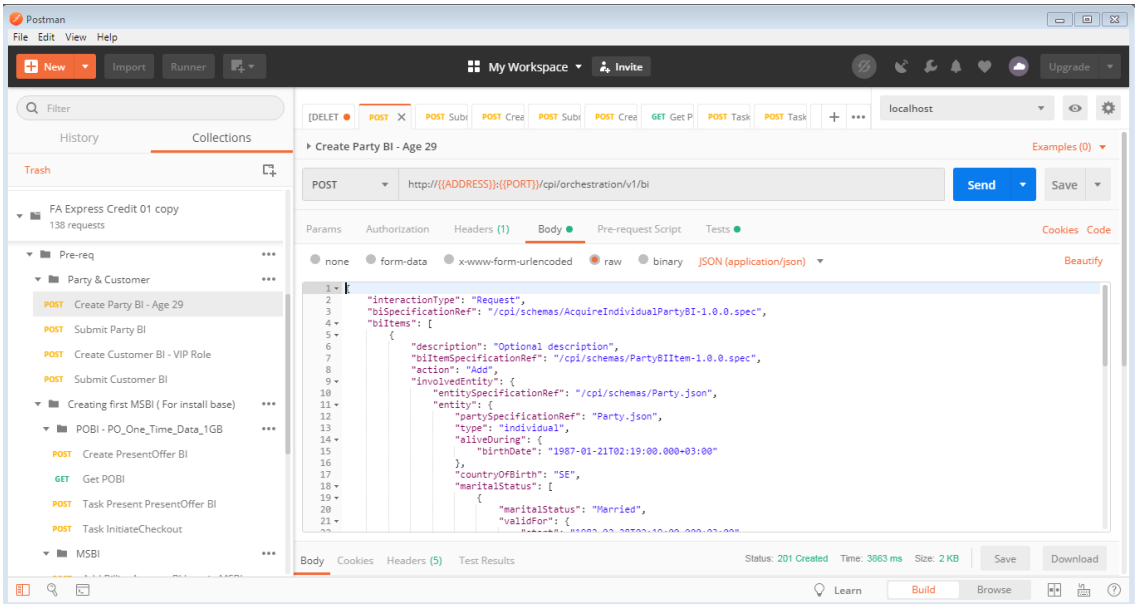


Ilustración 23 Postman

Como vemos en la Ilustración 23, Postman nos permite enviar una petición a nuestro servidor mediante el método de comunicación GET o POST que escojamos. Definiremos el cuerpo de la petición en el Body de la misma y en este caso hemos escogido enviarla en el formato JSON que es el formato que hemos definido en nuestros servicios. Una vez enviamos en la URL establecida, la que hayamos definido en nuestro servicio, mandamos la petición y recibiremos una respuesta, la cual la tendremos que validar y de esta forma comprobaríamos si nuestro desarrollo está siendo correcto o no. Como hemos indicado anteriormente también podremos definir pruebas en Postman que validen las respuestas que recibamos de nuestro servidor.

Lo que acabamos de describir sería una prueba de integración, ya que estamos comprobando que el servicio que hemos desarrollado funciona según los requisitos establecidos, enviamos una petición con unos datos determinados al sistema y éste nos devuelve una respuesta que nosotros mediante la herramienta de Postman validamos en consecuencia con los requisitos de nuestro software, es lo que se llama una prueba de caja negra. Como vemos Postman tiene un apartado de Tests donde podemos definir las pruebas que nuestro sistema tiene que pasar.

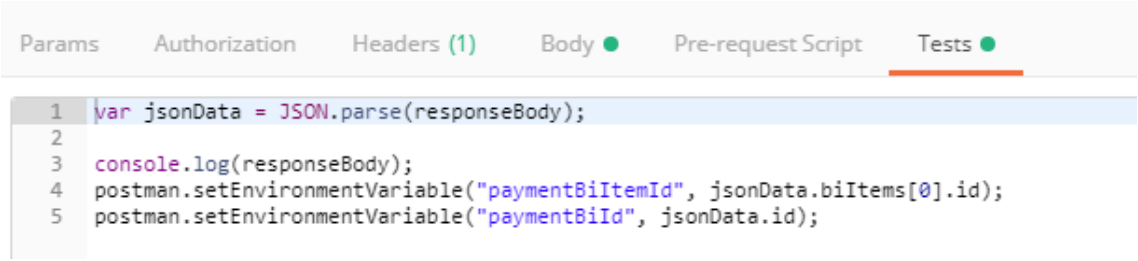


Ilustración 24 Pruebas en Postman

Aparte de las pruebas de integración de Postman, también realizamos pruebas de integración en Java que se ejecutan y se validan de forma automática. A continuación, mostramos un ejemplo de cómo desarrollamos este tipo de pruebas en nuestros sistemas:

```
@Test
public void post_minimalCustomerWithCharacteristic() throws Exception
{
    GenericResponse customerBIResp = CustomerReferences.createActiveAcquireCustomerBIWithCharacteristics(createTwoCharacteristicValues(), partyId);
    String customerBIId = customerBIResp.getId();

    String expectedCustomer = createParamsForExpectedResponse(customerBIResp.getResponse(), true);
    TextHttpResponse expected = new TextHttpResponse(Response.Status.CREATED).withPayload(expectedCustomer);
    expected.addHeader(CONTENT_TYPE, CONTENT_TYPE_VALUE + SchemaNames.getContentTypeProfile(SCHEMA_CPI_ACQUIRE_CUSTOMER_BI));

    TextHttpRequest request = new TextHttpRequest(ORCHESTRATION_URI + customerBIId + Task.SUBMIT, HttpMethod.POST);
    validateAndGetResponse(request, expected, SCHEMA_CPI_ACQUIRE_CUSTOMER_BI);
}
```

Ilustración 25 Pruebas de Integración

Como vemos en la Ilustración 25 hemos creado una prueba de la misma forma que previamente lo hacíamos para los casos de las pruebas unitarias, utilizando Junit, podemos ver que estamos comunicándonos mediante el protocolo http con el servicio correspondiente, enviándole unos datos determinados y en su respuesta estamos validando los datos que hemos recibido con unos datos esperados que hemos definido previamente. Si los datos que nos devuelve el servicio no son los que esperábamos la prueba fallará.

En conclusión, vemos que las pruebas de integración son de gran utilidad para probar nuestros sistemas, ya que nos permiten comprobar que nuestro software funciona de forma correcta de acuerdo con los requisitos establecidos. Vemos que existen múltiples herramientas como Postman y Junit que nos permiten el desarrollo y la implementación de este tipo de pruebas y que nos ayudan a realizar un software más seguro y resistente. Pero tal y como hemos indicado previamente estas herramientas y frameworks son solo de apoyo para la realización de las pruebas, es responsabilidad del desarrollador de dichas pruebas que éstas tengan sentido y que prueben el sistema de forma correcta teniendo en cuenta todos los datos y casos posibles.

Herramientas de apoyo al desarrollo

Una vez que hemos explicado las principales pruebas que una aplicación o desarrollo de software debe pasar, para tratar que nuestro software sea más seguro y detectar a tiempo algunos tipos de vulnerabilidades, junto con los principales frameworks y herramientas que nos ayudan al desarrollo de dichas pruebas, procedemos a explicar algunas de las herramientas que consideramos de utilidad para desarrollar el software de forma segura.

Modo depuración

Un depurador de código nos va a permitir ejecutar el software paso a paso, dándonos la posibilidad de parar la ejecución de este en cada una de sus instrucciones, y nos mostrará que es lo que ocurre en la memoria del ordenador en cada momento, por ejemplo, nos mostrará los valores de las variables que hayamos definido en el código en tiempo de

ejecución. De esta forma nosotros como desarrolladores podremos comprobar en tiempo real si nuestro programa está funcionando como deseamos y podremos ver más en detalle aquellos puntos por los que nuestro software va pasando. Este modo nos puede ser de gran utilidad para nuestros desarrollos. Existen diferentes tipos de depuradores, nosotros generalmente utilizamos el depurador que incorpora por defecto el IDE de eclipse para desarrolladores JAVA.

Para ejecutar el modo Debug en Eclipse simplemente tendremos que ejecutar nuestras pruebas unitarias o de integración en modo Debug:

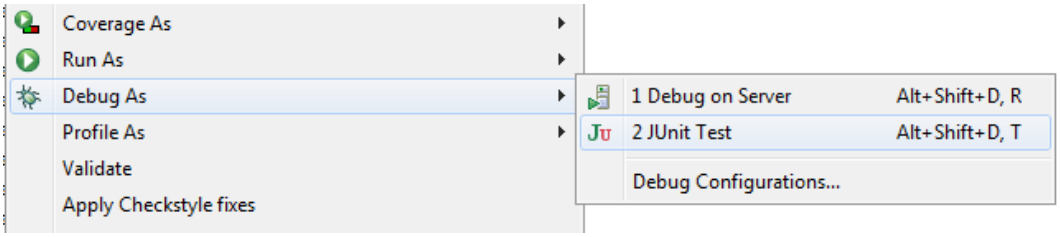


Ilustración 26 Ejecución en modo Debug

Una vez que hemos iniciado nuestro modo Debug podremos poner en las clases que nos interesen puntos de ruptura o "Breakpoints" que serán aquellos puntos donde se detendrá la ejecución y podremos examinar el código en tiempo real mientras la ejecución aún se encuentra activa, aunque detenida en ese punto.

```
77
78        invalidAccountId.withTypeOfIdentifier("MSISDN");
79        invalidAccountId.withIdentifier(UNKNOWN_ACCOUNT_IDENTIFIER);
80
```

Ilustración 27 Punto de ruptura

Como podemos ver en la Ilustración 27, hemos puesto un punto de ruptura en la línea 78 de esa clase, por lo que cuando la ejecución llegue a ese punto se detendrá mostrándonos el valor de las variables que se han inicializado y configurado en ese momento de la ejecución.

```
64    @Override
65    public void prepare(DataContainer<?, ?> container)
66    {
67        LOG.debug("Getting MSV properties:");
68        LOG.debug(MSV_MESSAGE, ID_NOTIFICATION_DU_ACTIVE_150001,
69        container.put(FlowConstants.IS_ID_NOTIFICATION_DU_ACTIVE_
70        if (container.getCPMService() == null)
71        {
72            container.setErrorCode(ErrorCode.newBusinessLogicFail
73            return;
74        }
75    }
```

Ilustración 28 Inspección de código en modo Debug

Como vemos en la Ilustración 28, el sombreado en verde indica donde se encuentra la ejecución y podemos ver los valores que van tomando las variables simplemente situando el cursor encima de ellas o seleccionando la opción con el botón derecho de Watch:

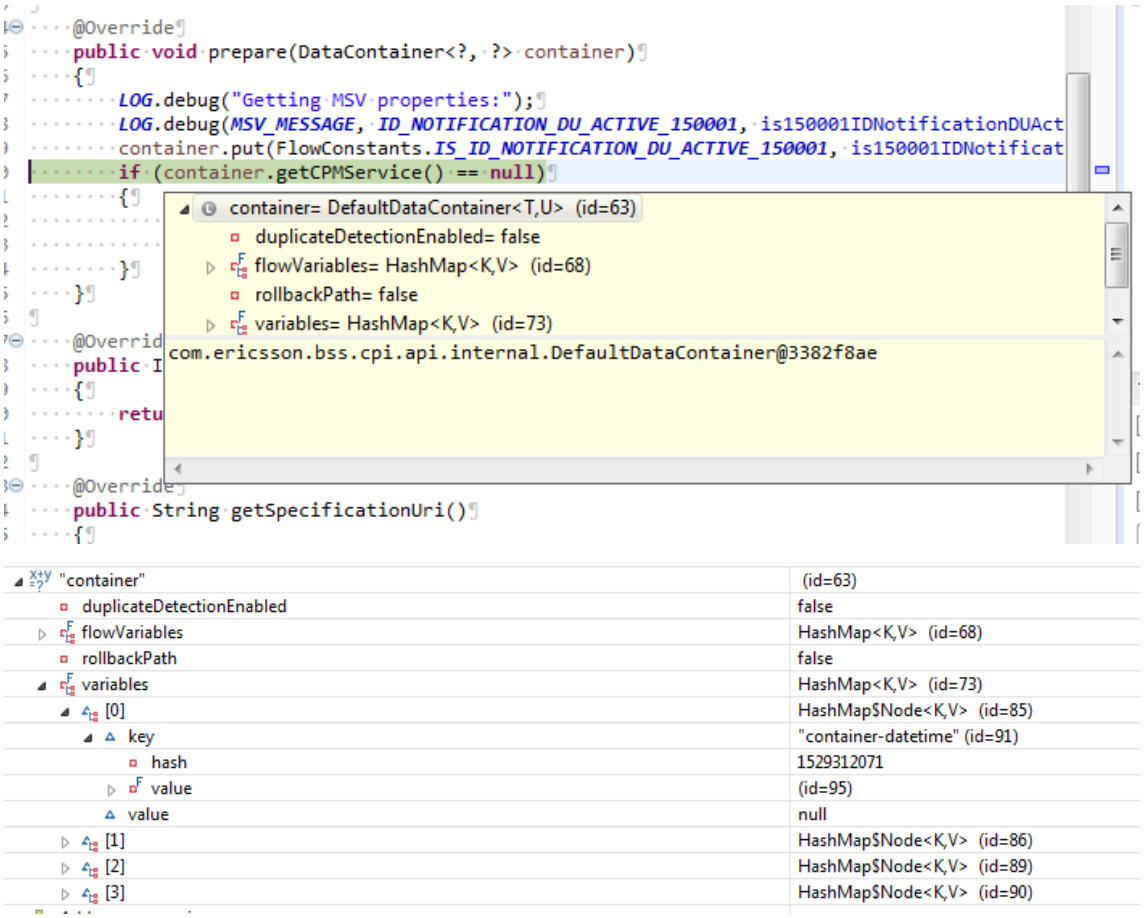


Ilustración 29 Análisis de variables en modo Debug

Como vemos en la Ilustración 29, se muestra de forma detallada el valor de la variable container y los valores que tiene en el momento en el que hemos detenido la ejecución.

El modo Debug de Eclipse nos permite controlar la ejecución de nuestro código mediante una serie de controles muy intuitivos como vemos en la Ilustración 30:

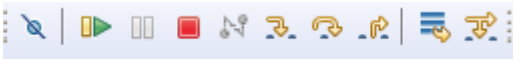


Ilustración 30 Controles del modo Debug

A través de estos controles podremos detener la ejecución actual, continuar hasta el siguiente punto de ruptura establecido, continuar a la siguiente sentencia de código, continuar dentro del método de ejecución en el que nos encontremos, etc. Podemos encontrar las siguientes opciones:

- Step Into (o pulsar F5): "paso en". Significa que se ejecutará instrucción por instrucción, pero si el depurador encuentra una función (y/o una subrutina-procedimiento-método), al pulsar Step Into se irá a la primera instrucción de dicha función, entrando en ella.
- Step Over (o pulsar F6): "paso sobre". Significa que se ejecutará el código instrucción por instrucción, pero si el depurador encuentra una función (y/o

subrutina-procedimiento-método), al pulsar Step Over se irá a la siguiente instrucción del código sin entrar en la subrutina.

- Step Return (o pulsar F7): "paso retorno". Imaginemos que nos encontramos dentro de una función y queremos salir de ella. Una forma sería encontrar la última instrucción, poner ahí un punto de ruptura, pulsar Resume y finalmente Step Into, sin embargo, tenemos la opción de pulsar Step Return y el depurador automáticamente encontrará el final de la función y nos situará fuera de ella.
- Resume (o pulsar F8): Reanudar. Se utiliza para depurar el código. El depurador parará en el siguiente punto de ruptura que hayamos puesto. Esto es muy útil cuando no queremos analizar instrucción por instrucción y queremos que el depurador se pare directamente en una línea donde tenemos puesto un punto de ruptura.
- Run to Line (o pulsar Ctrl+R). Ejecutar hasta la línea. Se reanuda la ejecución del código hasta la línea que hayamos seleccionado.

Mediante el depurador de código podemos detectar y solucionar los errores de software que podamos cometer ya que vamos viendo instrucción a instrucción la ejecución de nuestro código y los valores que van tomando las variables en cada momento.

Existen muchos tipos de depuradores de código que podemos utilizar como SoftICE, IDA Pro, OllyDbg, etc. Pero utilizamos el depurador que trae Eclipse por defecto ya que es un depurador muy completo que cumple con los requisitos que necesitamos y está incorporado en el IDE que utilizamos en nuestros proyectos.

Plataformas de evaluación del código fuente

Podemos encontrar diferentes plataformas de análisis estático del código fuente, en el apartado previo del estado del arte hemos introducido algunas de ellas, y aunque existen muchas más de las que hemos explicado, dos de las más famosas hoy en día son Sonarqube y Kiuwan.

El análisis estático del código es el proceso de evaluación de software que se lleva a cabo sin ejecutarlo. La idea de este análisis es que teniendo como entrada nuestro código fuente, podemos obtener información y métricas que nos permita mejorar el código detectando errores de programación. Las herramientas de análisis de código estático nos proporcionarán sugerencias sobre que partes del código son mejorables.

Sonarqube es una de las herramientas más populares para realizar el análisis estático de código. Es de código abierto, por lo que es gratuito. Eso sí, tendremos que instalarlo en una máquina, y mantenerlo actualizado. Además, deberemos tener en cuenta que determinados plugins son de pago.

Kiuwan es otro analizador estático de código, pero en este caso se trata de un servicio que podemos usar para analizar nuestro código sin preocuparnos por instalaciones ni actualizaciones. Podemos subir nuestro código a la nube para analizarlo, o descargar una aplicación que analizará nuestro código localmente y subirá los resultados a Kiuwan.

Tanto Sonarqube como Kiuwan tienen integración con distintos IDEs, que nos permiten detectar incidencias en nuestro código mientras lo escribimos, sin tener que esperar a análisis posteriores. También en ambos casos existen plugins para utilizarlo con herramientas de integración continua como Jenkins.

En nuestros desarrollos trabajamos con Sonarqube, por lo que vamos a mostrar a modo de ejemplo, sus principios básicos para ilustrar cómo funcionaría una herramienta de análisis de código estático.

Sonarqube emplea diversos plugins de análisis estático de código fuente como son Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un desarrollo de software y entender que problemas tenemos en nuestro código fuente.

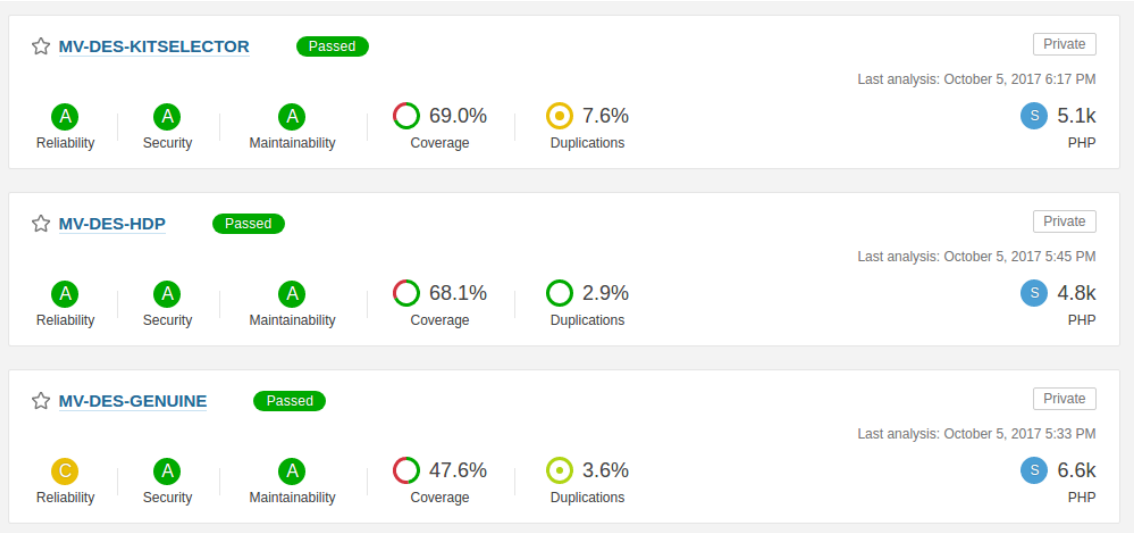


Ilustración 31 Sonarqube

Entre sus funciones están la de dar métricas acerca de código duplicado, pruebas unitarias, cobertura del código, indicar posibles errores potenciales o estándares. Inicialmente, la herramienta estaba pensada para el lenguaje Java, pero acepta extensiones para otros lenguajes. Al poder integrarse con Jenkins, podemos automatizar estos análisis cada vez que un desarrollador sube código al repositorio.

Al realizar un análisis estático del código podemos detectar:

- Problemas de Diseño: Podemos detectar problemas en el diseño y arquitectura del software analizando las dependencias entre las clases del proyecto. Esto nos permite detectar el desarrollo clases totalmente acopladas entre ellas y difícilmente reutilizables.

- Duplicidad de Código: Sonarqube nos proporciona métricas de código duplicado, pudiendo detectar partes de nuestro software son similares, pudiendo tomar decisiones como desacoplar componentes o aplicar técnicas de refactorización utilizando métodos como herencia y la reutilización de componentes.

- Detección de Vulnerabilidades: Sonarqube cuenta con una base de datos de código vulnerable y errores típicos de programación que detectan si alguna línea de código puede estar provocando algún problema que pueda vulnerar la seguridad. Por ejemplo, a la hora de cómo recoger los parámetros o cómo usarlos en nuestras consultas para evitar vulnerabilidades del tipo SQL Injection.

- Standard de codificación: Sonarqube nos indica si existen malas prácticas a la hora de definir constantes, variables, llamadas a métodos estáticos, etc.

- Monitorización de la cobertura de código: En nuestro caso también lo usamos para poder monitorizar si la cobertura de las pruebas es aceptable y de esta manera tener una visión global del estado de la cobertura de todos los proyectos.

El análisis de código estático es un análisis totalmente recomendable para el desarrollo de un software seguro, que nos ayuda a detectar vulnerabilidades y a realizar un mantenimiento correcto del código. Vemos necesario el uso de herramientas como Sonarqube, o la herramienta que el desarrollador elija para llevar a cabo estos análisis.

Implementar un modelo de trabajo basado en la integración continua

La Integración Continua es una práctica de software expuesta por Martin Fowler, que permite la reducción de riesgos y la realización de tareas repetitivas, automatizando al máximo los procesos involucrados en el desarrollo de software. Este proceso permite la generación de software listo para desplegar, con una alta calidad.

En este modelo de trabajo los miembros de un equipo integran su trabajo con frecuencia, por lo general, cada persona integra su trabajo de forma diaria, lo que lleva a múltiples integraciones por día (si tenemos en cuenta que puede haber muchos miembros en el mismo equipo de desarrollo trabajando en el mismo proyecto). Cada integración se verifica mediante una compilación automatizada (incluida la prueba) para detectar errores de integración lo más rápido posible.

Cuando se cumplen los principios que rigen la integración continua se obtiene una herramienta eficaz y útil que ayuda al desarrollo de los proyectos de software.

Más allá del control de versiones, un servidor de integración continua es una de las herramientas más importantes que un equipo de desarrollo puede usar. Su propósito es validar los cambios que se han producido en los repositorios donde está el código, también se encarga de su verificación y de ejecutar los comandos que activan la compilación de este.

Para poder utilizar los principios de la integración continua debemos tener en cuenta que todo el código debe mantenerse en un repositorio de control de versiones. En este caso nosotros vamos a poner el ejemplo de Git (en el siguiente apartado explicaremos más en detalle estas herramientas), aunque cada desarrollador deberá escoger la que más se adapte a sus necesidades. El uso de un repositorio de código y la creación de versiones de este deberían ser una práctica común. En un ciclo de vida de integración continua, cuando alguien modifica su código alojado y revisado en el repositorio, un sistema automatizado recoge el cambio que acaba de ocurrir, verifica el código nuevo y ejecuta un conjunto de comandos, que incluyen la compilación, para verificar que el cambio es correcto y no introduce ningún error en el sistema. La herramienta que realiza estas evaluaciones pretende adoptar el papel de ser el juez imparcial de si un cambio funciona o no, evitando así el típico comentario de "funciona en mi máquina" antes de que el código llegue a la producción.

Para que todo esto funcione, la compilación debe ser automática, y gracias a los frameworks web y herramientas de compilación, esta tarea es fácil de realizar. La compilación también debe incluir un conjunto de pruebas para verificar que el código funciona con cada cambio introducido. El objetivo es aumentar la cobertura de la prueba con cada cambio. Si las pruebas duran más de 10 minutos, la productividad del

desarrollador disminuye, ralentizando el proceso de envío de nuevas características o correcciones de errores al cliente.

Muchos equipos encuentran que este enfoque reduce los problemas de integración y que permite desarrollar software cohesivo más rápidamente.

Basándome en mi experiencia puedo decir que la integración continua aporta las siguientes características:

- Todos los desarrolladores ejecutan compilaciones privadas en sus propias estaciones de trabajo antes de enviar su código al repositorio de control de versiones para asegurarse de que sus cambios no rompen la compilación en el sistema completo.
- Los desarrolladores envían su código a un repositorio de control de versiones al menos una vez al día.
- Las compilaciones de integración ocurren varias veces al día en una máquina de compilación separada.
- Todas las pruebas (unitarias y de integración al menos) deben ser positivas en cada compilación.
- Se genera un producto (por ejemplo, WAR, ensamblaje, ejecutable, etc.) que puede probarse funcionalmente.
- La reparación de las compilaciones erróneas es la máxima prioridad.
- Algunos desarrolladores revisan los informes generados por la compilación, como los estándares de codificación y los informes de análisis de dependencia, para buscar áreas de mejora.

En un nivel alto, el valor de la integración continua aporta a nuestro proyecto:

- Reducción de riesgos.
- Reducción de procesos manuales repetitivos.
- Generación de software desplegable en cualquier momento y en cualquier lugar.
- Permite una mejor visibilidad del proyecto.
- Establece una mayor confianza en el producto de software del equipo de desarrollo.

Herramientas de control de versiones en un sistema de integración continua

Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que podemos recuperar versiones específicas más adelante. Aunque un control de versiones está pensado y diseñado para su uso en archivos de código. Podemos hacer un control de versiones con casi cualquier tipo de archivo que encontremos en nuestro ordenador.

Este tipo de sistemas nos permite regresar a versiones anteriores de nuestros archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un error en el sistema y cuándo, y mucho más. Usar un sistema de control de versiones también significa generalmente que, si perdemos nuestros archivos, será posible recuperarlos fácilmente.

Podemos encontrar multitud de herramientas de control de versiones actualmente, como Subversion, o Git, etc. En nuestros proyectos nosotros utilizamos Git debido a la forma en la que este último guarda sus datos.

- Git

La diferencia principal entre Git y el resto de los sistemas de control de versiones es que, a diferencia de Git, el resto de los sistemas manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo, mientras que Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos. Cada vez que confirmamos un cambio, o guardamos el estado de nuestro proyecto en Git, el sistema básicamente toma una foto del aspecto de todos nuestros archivos en ese momento, y guarda una referencia a esa copia. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino que crea un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

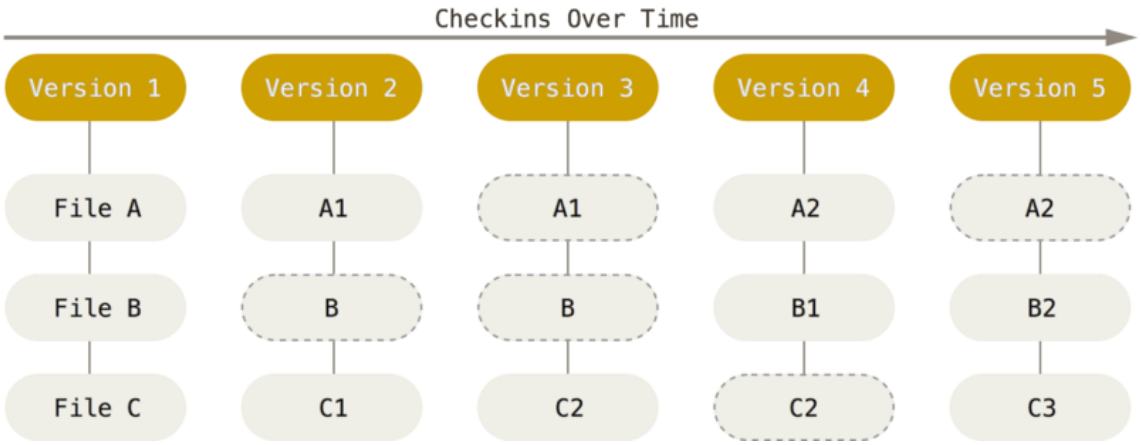


Ilustración 32 Funcionamiento de Git

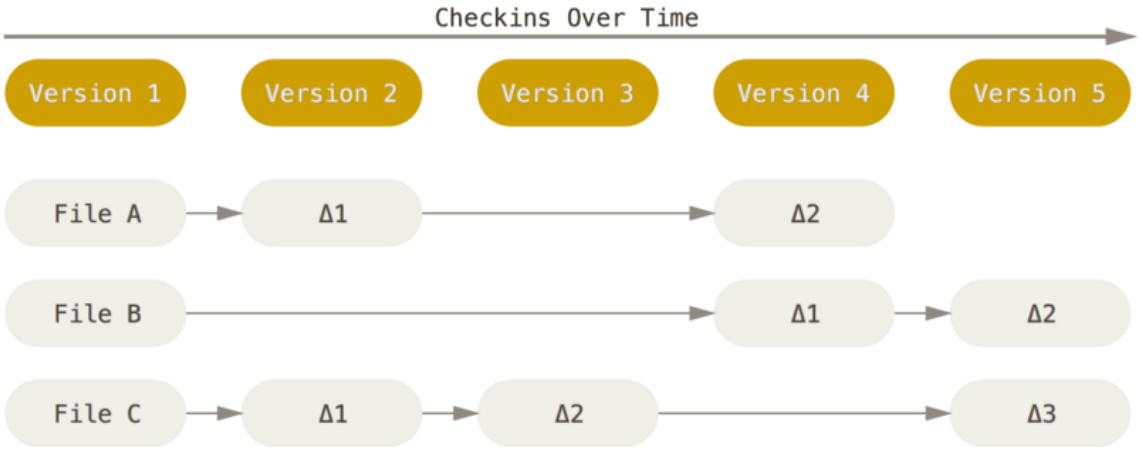


Ilustración 33 Funcionamiento de otros sistemas

Git nos aporta:

- Software de control de versiones.
- Gestiona versiones mediante instantáneas.
- Rapidez: casi cualquier operación es local.
- Proporciona integridad de archivos.
- “Hard to break”: generalmente sólo añade información.
- Distribuido: eficiente en proyectos grandes.

➢ Subversion

Apache Subversion es una herramienta de control de versiones de código abierto basada en un repositorio cuyo funcionamiento se asemeja al de un sistema de ficheros. Es software libre.

Subversion presenta las siguientes características:

- Nos permite recopilar el historial de los archivos y directorios a través de copias y renombrados.
- Las modificaciones (incluyendo cambios a varios archivos) son atómicas.
- La creación de ramas y etiquetas son operaciones eficientes.
- Se envían sólo las diferencias entre ficheros.
- Es capaz de controlar eficientemente archivos binarios.
- Permite selectivamente el bloqueo de archivos.

➢ Perforce

Perforce es un sistema de control de versiones comercial.

Presenta las siguientes características:

- El servidor mantiene el historial completo y los metadatos de los ficheros.
- El servidor mantiene el historial completo de revisiones de ficheros ramificados, renombrados, movidos, copiados y borrados.
- Presenta una visión gráfica de históricos y ramificaciones.
- El interfaz administrativo funciona de modo gráfico.
- Proporciona soporte para control distribuido de versiones
- Los ficheros modificados pueden agruparse y manejarse como unidades lógicas
- Las listas de cambio se actualizan de manera indivisible.
- Se puede almacenar temporalmente el trabajo en curso para cambiar de tarea.

➢ Mercurial

Mercurial es un sistema de control de versiones multiplataforma, para desarrolladores de software.

Las principales metas de desarrollo de Mercurial incluyen un gran rendimiento y escalabilidad, desarrollo completamente distribuido, sin necesidad de un servidor. Gestión robusta de archivos tanto de texto como binarios. Finalmente proporciona capacidades avanzadas de ramificación e integración, todo ello manteniendo sencillez conceptual. También incluye una interfaz web integrada.

Dada su filosofía de funcionamiento nos hemos decantado por el uso de Git en lugar de utilizar otro sistema de control de versiones. Aunque es decisión del desarrollador escoger el control de versiones que mejor se adapte a su proyecto. Git tiene la ventaja además de que es un software de licencia libre y que cuenta con el apoyo de una gran comunidad de desarrolladores, por lo que es sencillo encontrar documentación acerca de su funcionamiento.

Nosotros vemos de gran utilidad la utilización de un control de versiones en el desarrollo de nuestro software ya que nos va a permitir tener un control sobre nuestro código y poder solventar los errores que puedan surgir durante el desarrollo teniendo un control de las versiones que han fallado, pudiendo por ejemplo volver a una versión del código estable en poco tiempo en el caso de que la instalación de una nueva versión cause alguna vulnerabilidad. Del mismo modo nos va a permitir trabajar en paralelo al software que se encuentra en producción sin una gran dificultad, ya que podemos generar ramas paralelas al código de producción y trabajar sobre ellas sin necesidad de estar trabajando sobre el código final. Posteriormente podremos incluir el código de las diferentes ramas que hemos ido desarrollando en el código de producción controlando el código que estamos añadiendo y pudiendo volver a la versión estable rápidamente en el caso de que surja algún error.

Un control de versiones también nos proporciona la seguridad de que no vamos a perder nuestro código, aunque nuestro ordenador falle, ya que estará guardado en un repositorio seguro.

Aunque proponemos estas herramientas como forma de trabajo en un sistema de integración continua podemos utilizarlas de forma independiente con cualquier desarrollo.

Aplicaciones de apoyo a la revisión del código en un sistema de integración continua

Hoy en día existen multitud de herramientas que permiten a los desarrolladores realizar revisiones de código con el objetivo de mejorarlo o de evitar vulnerabilidades. Es una buena práctica de desarrollo que se realicen revisiones de código entre los diferentes miembros del equipo. Para llevar a cabo esta práctica nosotros en nuestros desarrollos utilizamos Gerrit, ya que es una herramienta de software libre que permite la integración con Git, que es el sistema de control de versiones que utilizamos, aunque, existen muchas más y cada desarrollador o equipo de desarrollo deberá escoger la herramienta que mejor se adapte a sus necesidades.

En este apartado vamos a mostrar cómo funcionaría una herramienta de revisión de código basándonos en Gerrit. Mostraremos también las ventajas que tiene el uso de este tipo de herramientas.

- Gerrit

Gerrit es:

- Es una herramienta web de revisión de código de licencia libre.
- Capa intermedia entre las ramas locales y remotas.
- Permite revisar el código y aprobar o rechazar los cambios que se han realizado en el código.
- Totalmente integrado con Git.

- Permite operaciones como clonar el proyecto, coger todo el histórico de cambios que se ha realizado en un desarrollo, coger únicamente el cambio aplicado a un fichero concreto, etc.

Cuando entramos a visualizar alguno de nuestros cambios de software, en Gerrit, podemos ver una pantalla como la siguiente:

Aquí podemos ver los ficheros que han sido modificados, quien ha subido dicho cambio, el proyecto donde se ha subido el código, así como la rama donde nos encontramos trabajando. También podemos ver si algún compañero de equipo nos ha dejado algún comentario sobre nuestro código. Gerrit también nos muestra si existen cambios relacionados con nuestro código.

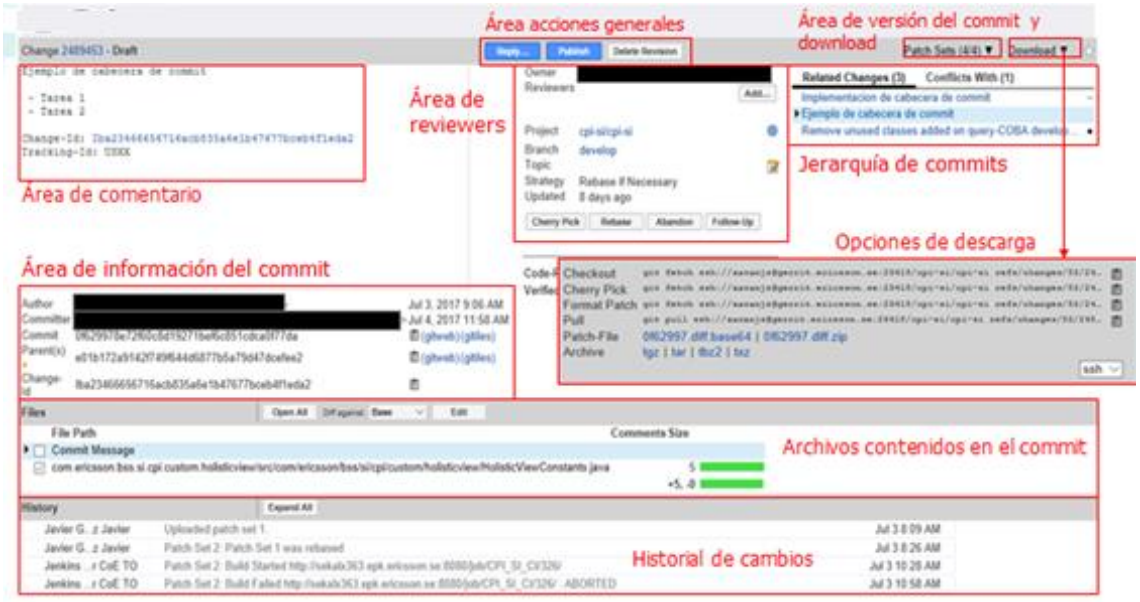


Ilustración 34 Gerrit

En Gerrit podemos ver el historial de revisiones que tenemos, ya sean revisiones en las que estamos colaborando con otros equipos de desarrollo o nuestros propios cambios.

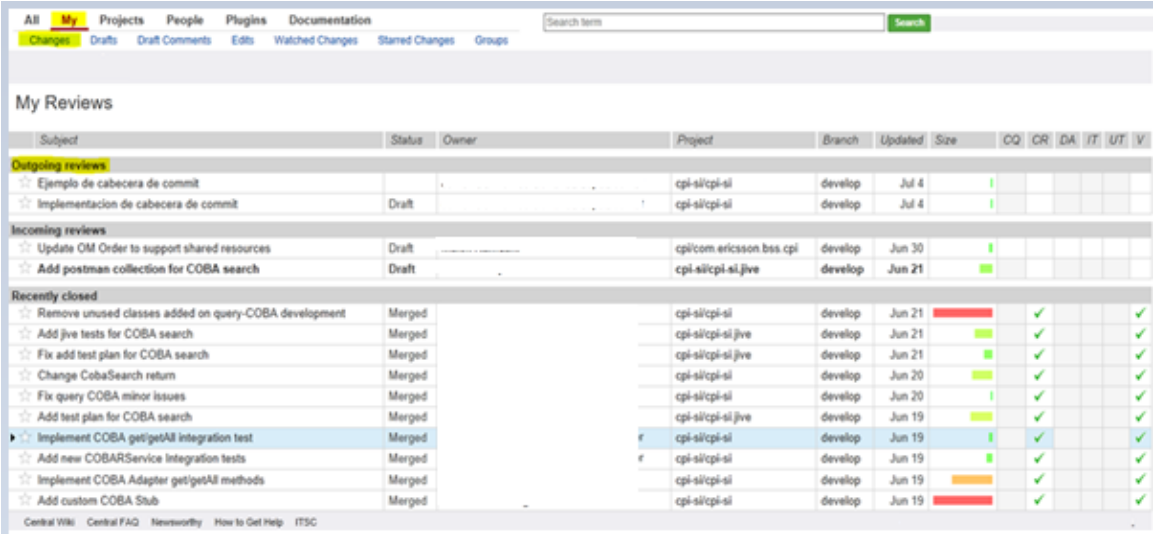


Ilustración 35 Historial de revisiones en Gerrit

Cuando un compañero de equipo nos deja comentarios en el software nos aparecerá una entrada como la siguiente en la interfaz de Gerrit:

Ilustración 36 Notificación de comentarios en Gerrit

Se nos indicará la versión de nuestro código en la que se han realizado los comentarios y además podremos inspeccionar si nos han comentado nuestro propio código:

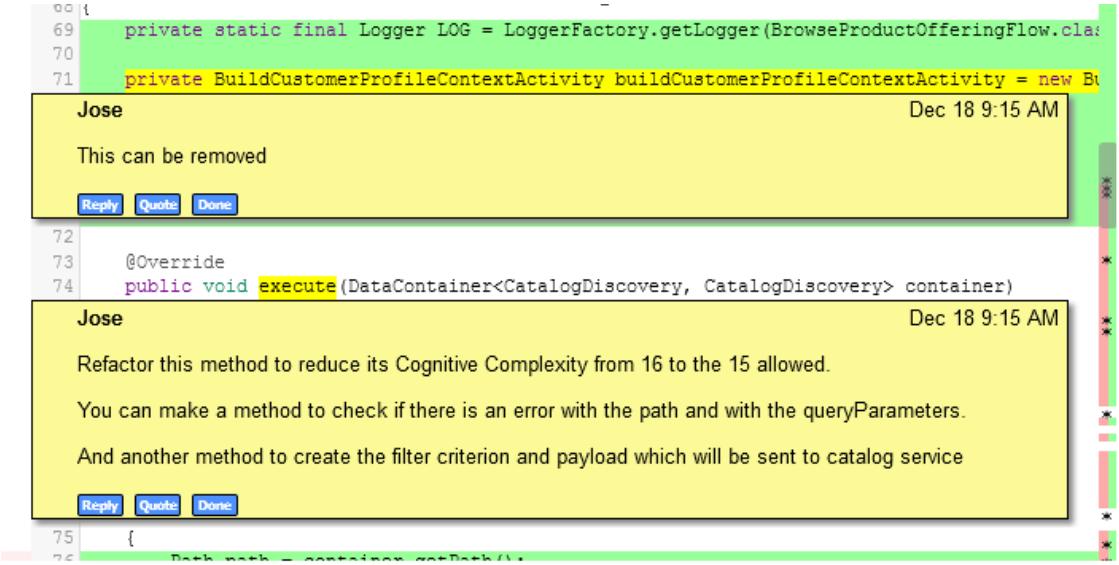


Ilustración 37 Comentarios en Gerrit

Como vemos, los compañeros nos pueden sugerir cambios en nuestro código con el objetivo de mejorarlo, o de eliminar fallos y vulnerabilidades que nosotros no hayamos visto durante el desarrollo.

Gerrit al ser una herramienta de revisión de código que está totalmente integrada con Git nos va a permitir trabajar a través de la interfaz web con el control de versiones, podremos ir una versión atrás en nuestro código, eliminar versiones erróneas, unificar cambios con otras ramas, solucionar conflictos entre versiones de código, etc.

Para mayor seguridad a la hora de integrar los cambios en producción, Gerrit puede ser configurado de tal forma que ningún cambio podrá pasar a ser código definitivo de producción hasta que al menos un número determinado de usuarios validen el código que se propone. Esto podemos verlo en el apartado de verificaciones:

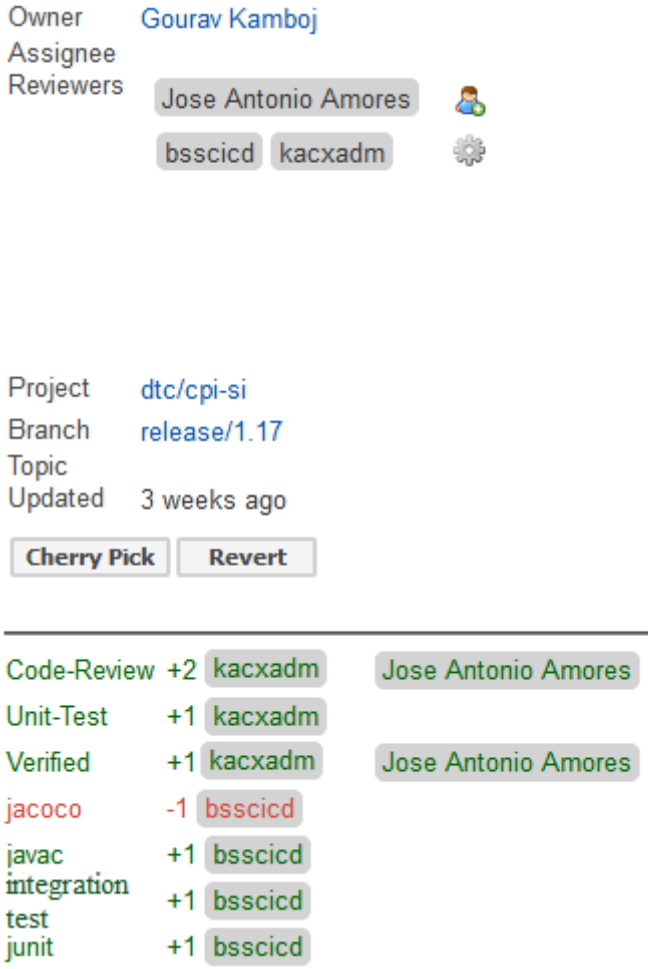


Ilustración 38 Verificaciones en Gerrit

En la Ilustración 38 podemos ver que hay una serie de usuarios que son los revisores y a la vez, son miembros del equipo de desarrollo. Para que el cambio que hemos propuesto pueda ser integrado con el código de producción, al menos dos usuarios (ninguno de ellos el propietario del desarrollo) han tenido que validar dicho cambio. Además, nuestra configuración de Gerrit también incluye que para que los cambios puedan ser integrados todos los test unitarios tienen que haber sido satisfactorios, así como las pruebas de integración.

Gerrit, si nosotros lo configuramos de esta manera, nos proporciona la seguridad de que nuestro software va a ser revisado por un equipo de personas, que a su vez pueden sugerir cambios con el objetivo de mejorar el desarrollo realizado. También nos ofrece la posibilidad de configurarlo de tal manera que nos de la seguridad de que nuestro código no va a pasar a una fase final de producción a no ser que se hayan pasado una serie de pruebas y de revisiones, lo que nos garantiza una mayor seguridad.

Aunque para ilustrar el funcionamiento de esta metodología de revisión hemos escogido Gerrit, también existen otras herramientas de revisión de código que explicamos a continuación:

- GitHub

Es una plataforma de desarrollo colaborativo, que permite alojar proyectos utilizando el sistema de control de versiones Git. El código de los proyectos alojados en GitHub se almacena por defecto de forma pública en la versión gratuita, aunque la versión de pago nos permite hospedar nuestro software en un repositorio privado.

- SourceForge

SourceForge una plataforma de desarrollo colaborativo para proyectos de software. Es una central de desarrollos de software que controla y gestiona proyectos de software libre y actúa como un repositorio de código fuente.

- BitBucket

BitBucket es un servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de versiones Mercurial y Git. BitBucket ofrece planes comerciales y gratuitos. Se ofrece cuentas gratuitas con un número ilimitado de repositorios privados.

- SourceTree

SourceTree simplifica la forma de interactuar con los repositorios de Git. A través de SourceTree podremos visualizar y administrar los repositorios mediante la interfaz de usuario de Git de SourceTree. SourceTree nos va a permitir revisar los conjuntos de cambios que hayan tenido lugar en el código fuente, realizar cambios provisionales, elegir entre ramas de forma selectiva entre otras funcionalidades.

En definitiva, consideramos que la utilización de herramientas de revisión de código son un elemento imprescindible en el desarrollo de software, ya que nos permiten revisar el código de forma conjunta con el resto de los miembros del equipo de desarrollo en el que nos encontremos trabajando, evitando vulnerabilidades de software. Además, es un método de trabajo que permite intercambiar conocimiento, ya que podemos comentar el código de los compañeros sugiriendo posibles mejoras y alertando de posibles errores.

Servidores de integración continua

Más allá del control de versiones, un servidor de integración continua es una de las herramientas más importantes que un equipo de desarrollo puede usar. Su único propósito es verificar los cambios que se producen en el repositorio donde está el código, verificar el código si se detecta algún cambio y ejecutar una lista de comandos para activar la compilación.

Un servidor de integración continua es imparcial. Sus tareas consisten en comunicar al equipo si los cambios más recientes superan las pruebas que están configuradas. Hay una

gran cantidad de herramientas disponibles, podemos encontrar sistemas como Jenkins, TeamCity y Bamboo. Todos ellos son muy fáciles de configurar.

- Jenkins

Como servidor de integración continua destacamos en especial Jenkins ya que es muy popular en la comunidad de desarrolladores. Jenkins es un servidor autónomo de código abierto, multiplataforma que se puede usar para automatizar todo tipo de tareas relacionadas con la construcción, prueba y entrega o implementación de software.

Jenkins puede instalarse a través de paquetes de sistema nativos, Docker, o incluso ejecutarse de manera independiente en cualquier máquina con un Java Runtime Environment (JRE) instalado. Aunque podríamos hacer con él todo tipo de tareas automáticas, Jenkins se suele utilizar como servidor de integración continua.

Como sistema de automatización, es capaz de realizar cientos de tareas necesarias para asegurar la calidad del software y facilitar su despliegue. Algunas de las tareas más habituales que Jenkins es capaz de hacer son las siguientes:

- Probar el software, Jenkins permite realizar pruebas automatizadas de nuestro software.
- Revisar las métricas de calidad del software establecidas por el equipo de desarrollo.
- Enviar las modificaciones del software, una vez pasadas todas las validaciones, al repositorio principal.
- Automatizar la compilación del software o su despliegue, una vez se hayan integrado nuevos cambios en el proyecto.
- Notificar debidamente a los desarrolladores o al equipo encargado de la gestión de la calidad del software cuando se encuentre cualquier tipo de error, ya sea en base a las pruebas del software o a las métricas de calidad definidas.
- Generar o visualizar la documentación del proyecto.

Como servidor de automatización, además, Jenkins es capaz de extender sus funcionalidades con centenares de plugins para realizar múltiples tareas.

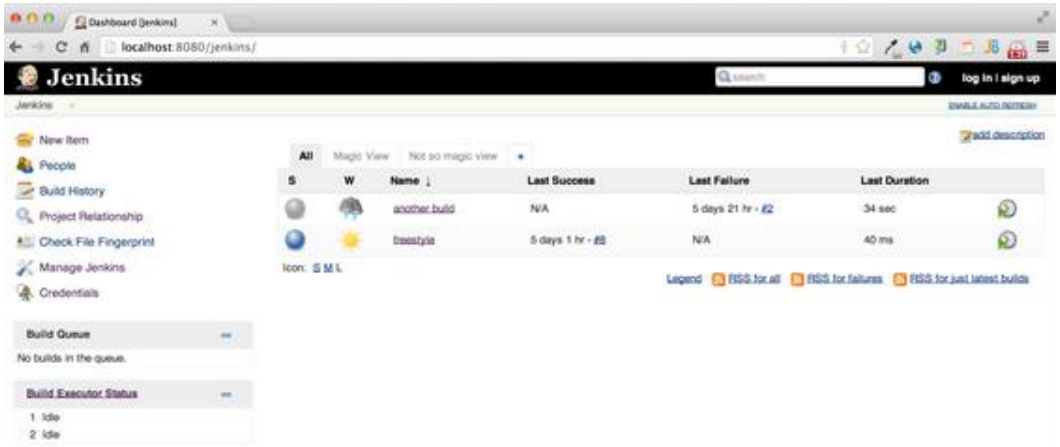


Ilustración 39 Jenkins

➢ TeamCity

TeamCity es un servidor de integración continua (CI) fácil de usar, la licencia de este servidor no es libre, aunque existe una versión de prueba limitada.

TeamCity nos permite:

- Ejecutar varias compilaciones simultáneamente en diferentes plataformas y entornos.
- Optimizar el ciclo de integración del código y asegurarnos de que nuestro código no contiene errores.
- Revisar los informes de resultados de las pruebas.
- Revisar la cobertura del código y buscar código duplicado para Java y .NET
- Personalizar las estadísticas sobre la duración de la compilación, la tasa de éxito, la calidad del código y las métricas personalizadas.

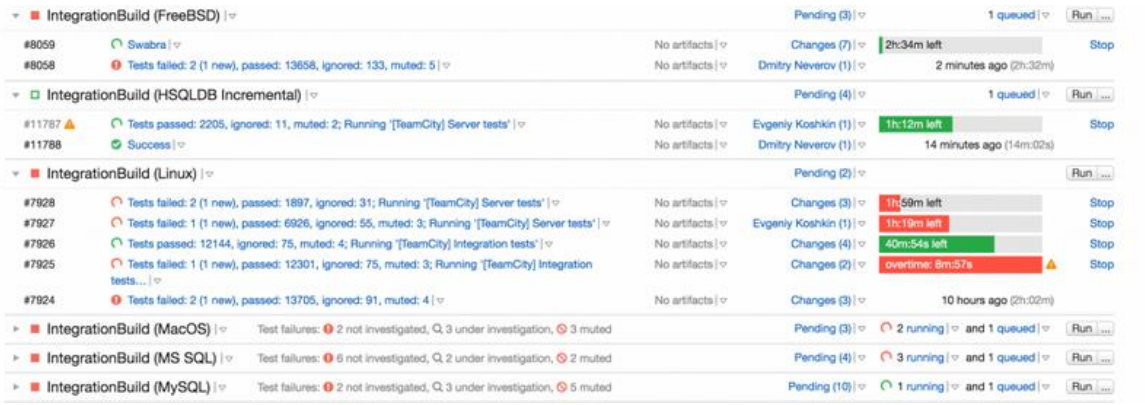


Ilustración 40 TeamCity

➢ Bamboo

Bamboo es un servidor de integración continua cuya licencia no es libre, permite crear planes de compilación de varias fases, configurar activadores para iniciar compilaciones tras cada commit y permite asignar agentes a las compilaciones y despliegues. Bamboo también permite realizar pruebas automatizadas.

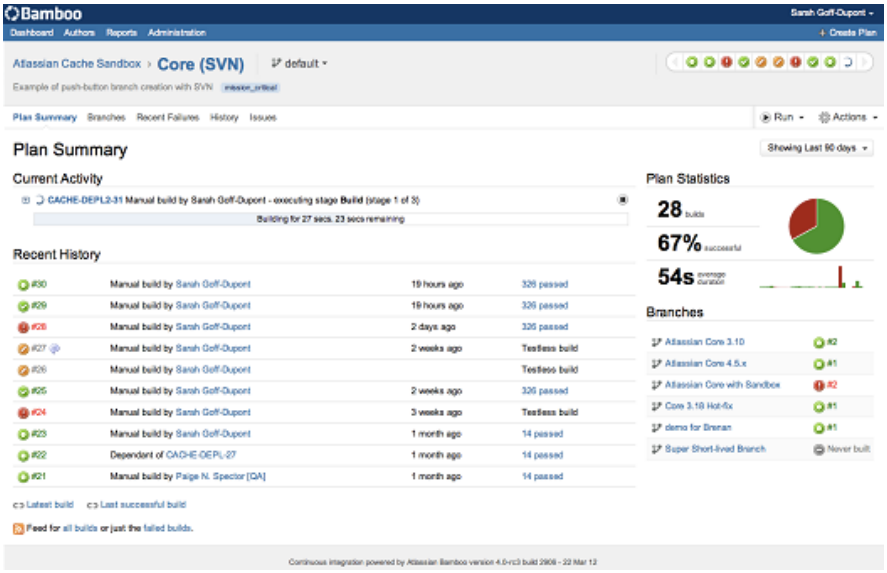


Ilustración 41 Bamboo

Permite realizar las pruebas automatizadas de forma paralela y de esta forma se desata el potencial del desarrollo ágil, además de facilitar y agilizar la detección de errores. Bamboo ofrece soporte para la fase de "entrega". Los proyectos de despliegue automatizan la tediosa tarea de efectuar el lanzamiento en todos los entornos disponibles, además de ofrecer control sobre el flujo con permisos propios de cada entorno.

Una vez hemos estudiado en qué consisten los servidores de integración continua y haber visto algunos ejemplos destacables debemos tener en cuenta que el último paso de una compilación totalmente automatizada es la capacidad de implementación en producción, que requiere de un proceso automatizado que cada desarrollador debe poder ejecutar al igual que el servidor de integración continua.

Revisiones de código en un modelo de integración continua

Como hemos explicado previamente existen herramientas de revisión y análisis del código fuente, como Sonarqube que se pueden integrar con un servidor de integración continua como es Jenkins y llevar a cabo análisis del código de forma periódica.

Mantener el entorno sin errores

Con todo el equipo de desarrollo comprometido en trabajar con la rama principal del proyecto, una correcta compilación es una responsabilidad prioritaria. Una compilación correcta es el resultado de un cambio de código que supera todos los pasos implicados en la compilación. Cuando los desarrolladores provocan una compilación errónea, están impidiendo que todos los demás en el equipo puedan implementar sus cambios con confianza. Una compilación correcta debería ser una prioridad en el equipo de desarrollo.

Más allá de tener buenas herramientas y un conjunto de pruebas, la integración continua trata de fomentar la responsabilidad del equipo de desarrollo y de proporcionar valor al cliente. El servidor para ejecutar la compilación es solo una pequeña parte de esta filosofía. Los principios de la integración continua abarcan un concepto aún mayor: un equipo de desarrollo debe garantizar que las nuevas funciones en el código se envíen rápidamente a producción y al cliente, y podemos asegurarlo mediante la integración continua ya que todos los miembros pueden realizar su trabajo y verificar el nuevo código de forma común.

Cuando los nuevos cambios se fusionan de forma continua en la rama principal, hay una mayor responsabilidad para todos en el equipo, deben asegurarse de que esos cambios no solo realicen la compilación satisfactoriamente, sino que también tengan un impacto mínimo en el entorno de producción. Los desarrolladores deben pensar en cómo pueden diseñar los nuevos cambios de manera que les permitan fusionar su trabajo en la rama principal de forma continua.

Una buena supervisión es esencial para ver que los cambios no tienen un impacto negativo en el entorno de producción: el registro, las métricas, las alertas y el seguimiento de excepciones ayudarán a realizar los cambios con confianza.

Beneficios de integrar el código en el repositorio de forma continua

Una práctica central de integración continua es que todos los desarrolladores trabajen con la rama principal del proyecto. Con sistemas modernos de control de versiones distribuidas, como es Git, es relativamente sencillo fusionar los cambios realizados por cada desarrollador con el proyecto que está en el repositorio. Si todos los desarrolladores realizasen sus cambios en la rama principal del proyecto podríamos saber que nuestro código no está comprometido, y que funciona correctamente al finalizar la jornada, es importante que este hábito de trabajo se mantenga, de esta forma llevaremos un control sobre los cambios que cada desarrollador ha incorporado y como han afectado dichos cambios al sistema completo.

Cuando el equipo de desarrollo realiza pequeños cambios y los integra en la rama principal con regularidad, incrementa los beneficios de esta metodología de trabajo ya que estos cambios son mucho más fáciles de detectar cuando algo falla. La forma de trabajo y la esencia de la integración continua se basa en que, en lugar de integrar los cambios tras largos periodos de tiempo, los cambios se integran continuamente, proporcionando los beneficios previamente mencionados.

Procedimiento de mitigación de vulnerabilidades durante la fase final

Basándonos en los estudios de J. Jurn, T. Kim y H. Kim (2018), proponemos una metodología que se basa en la realización de las ejecuciones fuzzing y simbólica en función de la complejidad binaria del fichero que queramos analizar con el objetivo de detectar automáticamente las vulnerabilidades que nuestro software pueda contener. También proponemos una técnica de parcheo automático para eliminar las vulnerabilidades que tiene el fichero binario de nuestro código, nuestro objetivo será ofrecer una respuesta rápida ante un ataque tras la detección de vulnerabilidades.

En este apartado analizaremos las tendencias y las técnicas para la detección automatizada de vulnerabilidades y la reparación automática de las mismas.

Basándonos en el estudio realizado por J. Jurn, T. Kim y H. Kim (2018), presentamos un método que detecta y elimina automáticamente vulnerabilidades utilizando un procedimiento de detección llamado fuzzing híbrido, basado en un análisis de complejidad binario. Y posteriormente generamos los correspondientes parches para eliminar dichas vulnerabilidades.

Detección automática de vulnerabilidades

Fuzzing

Como hemos introducido anteriormente el Fuzzing es un método de prueba de software que hace que el código que estamos probando se bloquee ante la generación de entradas aleatorias. El Fuzzing fue introducido por primera vez por el Profesor Miller de la Universidad de Wisconsin en 1988. Se realizó un estudio en el cual se introdujeron

valores de entrada aleatorios en un ordenador con el objetivo de intentar iniciar sesión de forma remota. El programa finalizó debido a un valor de entrada no permitido y esta experiencia evolucionó hacia el concepto de fuzzing, que inyecta valores de entrada aleatorios en el software y causa errores.

El Fuzzing se divide en el Fuzzing simple y en el Fuzzing inteligente, dependiendo del modelado de entrada y se divide también en Fuzzing de mutación y Fuzzing de generación, dependiendo del método de generación de los casos de prueba.

El Fuzzing sencillo es el más simple que existe porque se basa en la generación de errores al cambiar aleatoriamente los valores de entrada en el software que estamos probando. El caso de prueba se genera de forma muy rápida ya que es sencillo cambiar el valor de la entrada, sin embargo, es difícil encontrar un bloqueo válido, ya que son muchos los valores con los que se realizan las pruebas y solamente unos valores concretos podrían provocar el error en nuestro software.

Por otro lado, el Fuzzing inteligente es una tecnología que genera valores de entrada que se adaptan al código que estamos probando a través del análisis del software y la generación de errores. El fuzzing inteligente tiene la ventaja de saber dónde pueden ocurrir los errores a través de un análisis de software. Una persona encargada de probar el software es capaz de crear un caso de prueba para provocar un bloqueo en el código que estamos probando, sin embargo, existe la desventaja de que para generar caso de prueba se requiere un conocimiento experto para analizar el software que estamos probando. La supresión de mutaciones es una técnica de prueba que modifica las muestras de datos que se insertan en el software de destino. La generación de casos difusos es una tecnología que modela el formato de los valores de entrada que se aplicarán al software de destino y crea un nuevo caso de prueba para ese formato. Recientemente, se ha introducido una técnica de fuzzing evolutiva que genera nuevos valores de entrada al proporcionar información sobre la respuesta del software.

Ejecución simbólica

En el paradigma de ejecución simbólica, no se asume ningún valor inicial para las variables del entorno; en su lugar, se asigna un valor simbólico a cada una de ellas. Acorde con el estándar, el valor simbólico asociado a la variable X se mostrará bajo la forma ?X. Los valores simbólicos son expresiones de contenido indeterminado, es decir, no pueden identificarse con un valor concreto (salvo que las restricciones del programa indiquen lo contrario). Por este motivo, las dependencias de control se resuelven explorando todos los caminos posibles y acumulando las condiciones que se satisfacen en cada uno de ellos. Al término de la ejecución, se obtiene una lista con todos los estados finales posibles y los correspondientes conjuntos de restricciones sobre los datos de entrada que permiten alcanzarlos. Estos grupos de propiedades asociados a cada camino ejecutable reciben el nombre de “path conditions”.

La ejecución simbólica se divide en la ejecución simbólica fuera de línea y la ejecución simbólica en línea. La ejecución simbólica fuera de línea se resuelve eligiendo solo una ruta para crear un nuevo valor de entrada. El programa debe ejecutarse desde el principio para explorar otros caminos, por lo que la principal desventaja es que causa sobrecarga debido a la re-ejecución. La ejecución simbólica en línea es la forma en que se replican los estados y se generan predicados de ruta en cada punto donde el ejecutor de símbolos encuentra la declaración de rama. No hay gastos generales asociados con la reemisión usando el método en línea, pero la desventaja es que requiere el almacenamiento de toda la información de estado y el procesamiento simultáneo de múltiples estados, lo que lleva

a un consumo significativo de recursos. Para resolver este problema, se sugiere la ejecución simbólica de forma híbrida.

La ejecución simbólica híbrida guarda la información del estado a través de la ejecución simbólica en línea cada vez que se ejecuta una declaración y continúa hasta que se agotan los recursos de memoria del sistema. Cuando no hay más espacio para guardar la información, se produce un cambio a la ejecución simbólica fuera de línea y se realiza una búsqueda de ruta.

Por lo que podemos resolver el problema de desbordamiento de memoria de la ejecución simbólica en línea mediante la aplicación del método de guardar la información de estado y su uso posterior a través de la ejecución simbólica híbrida.

Además, resuelve la sobrecarga de la ejecución simbólica fuera de línea porque no necesita ejecutarse de nuevo desde el principio.

En los últimos años, se ha propuesto la ejecución simbólica concólica, que es un método de prueba que sustituye un valor real (valor concreto) y prueba una mezcla de ejecuciones simbólicas. Esta metodología es utilizada por muchas de las herramientas de ejecución simbólica actuales.

Fuzzing híbrido

El método de fuzzing híbrido es una técnica en la que la explotación automática de vulnerabilidades combina las ventajas de la metodología fuzzing para generar valores de entrada aleatorios con la ejecución concólica para rastrear la ruta de ejecución del programa. El fuzzing híbrido resuelve el estado incompleto del fuzzer y el problema de la trayectoria de la ejecución concólica.

Se usa el fuzzer para buscar el segmento inicial del programa. Si la sentencia condicional detiene el proceso, el motor simbólico se usa para guiar la siguiente sección y entonces el Fuzzer toma el control nuevamente y busca vulnerabilidades con mayor rapidez.

Driller es una herramienta de fuzzing híbrida que utiliza AFL (American Fuzzy Lop) y Angr. AFL es un fuzzer que genera y transforma valores de entrada a través de un algoritmo genético y Angr es un motor que realiza la ejecución simbólica al convertir códigos binarios en el VEX IR de Valgrind, que también es conocido por Mayhem y S2E como el motor de ejecución de simbólica más optimizado.

En cuanto al funcionamiento de esta herramienta tenemos que el perforador realiza el fuzzing a través de la AFL y llama al motor de ejecución concólica Angr con el fin de encontrar una nueva ruta de transición de estado, en el caso de que el fuzzer ya no pueda encontrar transiciones de estado adicionales. En este caso, la razón principal por la que el fuzzer no puede encontrar la ruta de transición de estado adicional es que no puede generar valores de entrada específicos para satisfacer declaraciones condicionales complejas en el software. El motor de ejecución simbólico que recibe el control en ese punto genera el valor de entrada, satisfaciendo la condición compleja utilizando el solucionador de restricciones. El valor generado se pasa a la cola del fuzzer y el control también vuelve al fuzzer para realizar el fuzzing. El perforador puede realizar este proceso repetidamente para buscar un camino rápido y profundo. Un factor importante para determinar la eficiencia en este flujo analítico es evitar la explosión del camino, que es un punto limitante inherente a la ejecución concólica. Esto se debe a que la ruta de ejecución limitada se analiza mediante el valor de entrada generado a través del fuzzing. Las herramientas y características relacionadas con las técnicas automatizadas de detección de vulnerabilidades se muestran en la Tabla 5.

Tabla 5. Herramientas de detección automática de vulnerabilidades

Técnica	Herramienta	Pruebas	Generación de entrada estratégica
Fuzzing	Zzuf	Blackbox	Mutation Algorithm
	AFL	Blackbox	Genetic Algorithm
	Peach	Blackbox	Format Modeling
Ejecución simbólica	Angr	Whitebox	Stepping
	KLEE	Whitebox	Random Path
	S2E	Whitebox	Search Heuristics
	Mayhem	Whitebox	Hybrid
Fuzzing híbrido	Driller	Greybox	Selective

Generación automática de parches

Una tecnología importante para la generación automática de parches es la generación de parches utilizando algoritmos genéticos.

Genprog, es la tecnología que ha tenido el mayor impacto en los estudios sobre parches automatizados. Genprog es una técnica para parchear automáticamente los programas basados en el lenguaje C.

Después de convertir la estructura del código fuente a AST (árbol de sintaxis abstracta), parchea el nodo que contiene una vulnerabilidad con tres modificaciones; eliminar, agregar y reemplazar. Para modificar los nodos, utiliza plantillas para cada error. El analizador automático de parches es capaz de corregir vulnerabilidades como "Desbordamiento de búfer", "Desbordamiento de enteros" y "Vulnerabilidad de cadena de formato", entre otras.

Otra tecnología para la generación automática de parches es la generación automatizada de parches utilizando la información que se recoge en los informes de errores. R2Fix es una de las herramientas más típicas para generar un parche automáticamente utilizando esta información; esta tecnología surge a partir de la idea de que hay muchos errores en el software desarrollado que no se han modificado debido a la falta de recursos para corregir errores ya conocidos. R2Fix realiza parches automáticamente al combinar patrones de corrección de errores, aprendizaje automático y técnicas de generación de parches. R2Fix consta de tres módulos: clasificador, extractor y generador de parches. El clasificador aplica el aprendizaje automático para recopilar la información de los informes de errores y categorizarlos automáticamente, los extractores extraen los parámetros generales (nombre de archivo, versión, etc.) y los parámetros de información detallada (nombre del búfer, tamaño, condición de verificación de límite, etc.) para cada tipo de error, y, por último, el generador de parches se encarga de generar y aplicar el código de los parches usando patrones de modificación.

Entre las tecnologías de generación automática de parches, hay una técnica que utiliza información de parches escrita por expertos. Esta tecnología se llama PAR, es una herramienta representativa que se encarga de generar parches automáticamente basándose en informes de parcheo de vulnerabilidades escritos por expertos. Analiza más de 60,000

parches para encontrar patrones comunes, y genera automáticamente parches basados en dichos patrones. PAR crea parches a través de tres pasos: localización de fallos, generación de candidatos a parches basados en las plantillas que solventan las vulnerabilidades y evaluación de los parches. La fase de localización de fallos utiliza un algoritmo de posicionamiento estadístico de fallos que asigna ponderaciones a cada sintaxis de ejecución en base a un caso de prueba Positivo / Negativo. En este paso, la suposición clave es que la sintaxis en la que se ejecuta el caso de prueba Negativo es probable que sea defectuosa. Primero, ejecuta ambos casos de prueba y registra la ruta. La ruta se divide en cuatro grupos:

1. La sintaxis de ejecución ejecutada por ambos grupos (positivo/negativo).
2. Sintaxis de ejecución solo para casos positivos.
3. Sintaxis de ejecución solo para casos negativos.
4. Sintaxis de ejecución no ejecutada por ninguno de estos casos.

Los pesos son 1.0 para (3), 0.1 para (1) y 0 para otros. Dependiendo de los pesos asignados, este sistema se encarga de crear un parche candidato en la posición evaluada. Durante la fase de generación de candidatos de parches basados en las plantillas, se aplican un total de 10 plantillas de modificación, incluidos el reemplazo de parámetros, el reemplazo de métodos y el reemplazo de expresiones, para cada ubicación de fallo y para cada parche candidato. La generación de parches candidatos se lleva a cabo en tres etapas:

1. Análisis AST.
2. Verificación de contexto.
3. Edición del programa.

El paso (1), consiste en el escaneo AST del software objetivo y se encarga de analizar la ubicación del error. El Paso (2) examina si es posible modificar el error a través de la plantilla de modificación y, si corresponde, crear un parche candidato volviendo a escribir el AST para el programa de destino en función del script predefinido en la plantilla de modificación. En el Paso (3) finalmente se lleva a cabo la fase de evaluación de los parches aplicados. En esta fase se evalúa la idoneidad de los parches candidatos creados utilizando los diversos casos de prueba recopilados.

Detección automática de vulnerabilidades y generación de parches

El Fuzzing funciona mucho más rápido que la ejecución simbólica y es capaz de explorar un rango más extenso de código. Sin embargo, es difícil para un fuzzer explorar código generalizado. La ejecución simbólica puede descubrir posibles caminos del programa que son vulnerables. Hay muchos tipos de fuzzers híbridos que implementan lo mejor de cada una de estas metodologías. La mayoría de los fuzzers híbridos eligen un motor de detección de vulnerabilidades sin analizar el programa que tratamos de evaluar. Un programa tiene partes complejas, como los bucles que son difíciles de analizar y explorar, por el contrario, otras partes del programa son accesibles y no resultan tan complicadas. Por consiguiente, es posible identificar qué partes son favorables para la ejecución simbólica o para el fuzzing. En este documento, presentamos un método de análisis binario para identificar las áreas favorables para la ejecución simbólica y para la ejecución de fuzzing, como se muestra en la Ilustración 42:

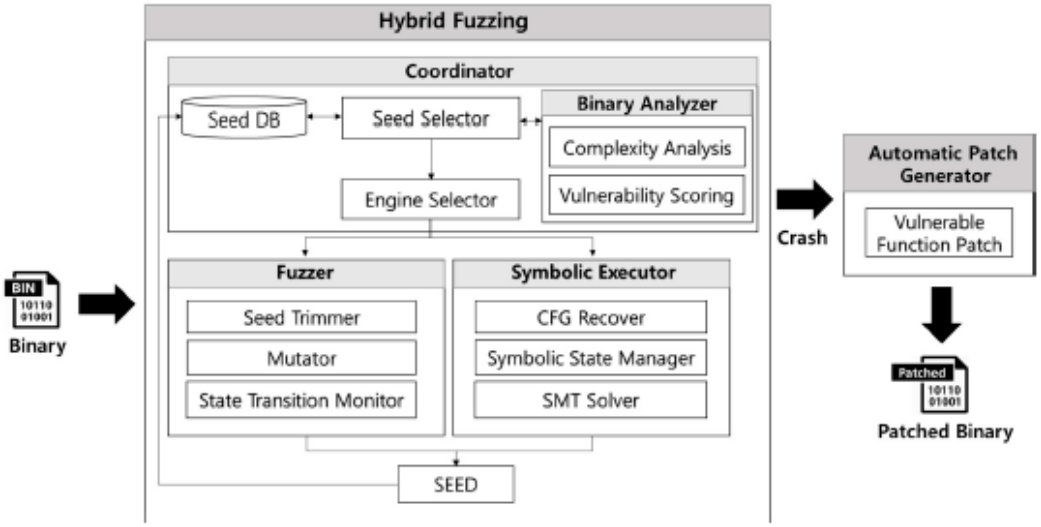


Ilustración 42 Fuzzing Híbrido

El Fuzzing híbrido basado en análisis binario se basa en el fuzzing y en la ejecución simbólica a través de métricas de complejidad binarias. Usando el motor de búsqueda de vulnerabilidades seleccionado, crea una nueva semilla que provoca una transición de estado. La semilla recién creada se usa para analizar nuevamente la complejidad binaria y repite el mismo proceso. El fuzzing híbrido basado en un análisis binario aplicará los siguientes pasos:

1. **Selección de semillas:** la selección de semillas es un paso para seleccionar el valor de entrada que se utilizará para la detección de vulnerabilidades. En una primera iteración, un programador de semillas selecciona una semilla definida por el usuario para detectar la vulnerabilidad. Desde una segunda iteración, se selecciona una nueva semilla creada por el motor de generación de semillas para el siguiente paso.

2. **Análisis binario:** un módulo de análisis binario ejecuta el archivo binario de destino en un entorno de prueba, de la ejecución y del análisis del fichero se puede extraer la información que necesitamos. La complejidad binaria y las puntuaciones de vulnerabilidad se analizan utilizando la información extraída. Los resultados del análisis de complejidad determinan si ejecutar el método fuzzer o la ejecución simbólica para detectar vulnerabilidades. Los resultados de la puntuación de vulnerabilidad se utilizarán para determinar si un bloqueo detectado se puede explotar en futuras investigaciones.

3. **Selección de motor:** el módulo de selección de motor se encarga de seleccionar uno de los motores de ejecución simbólica y de fuzzing según el resultado del análisis de complejidad. El motor de ejecución simbólico se ejecuta si el resultado del análisis de complejidad es más pequeño que un umbral específico; de lo contrario, se selecciona el motor de fuzzing.

4. **Generación de semillas:** la ejecución simbólica o el fuzzer se ejecutan para generar las semillas que se utilizarán en la siguiente iteración. El proceso de generación de semillas se realiza hasta que se produce una transición de estado.

Cuando se produce una transición de estado, el valor semilla que causó la transición de estado se almacena en una base de datos.

5. **Repetir.**

Para la corrección automática de vulnerabilidades, la técnica de parche basado en funciones vulnerables se utiliza para crear una lista de funciones débiles y modificar el PLT / GOT con respecto a las funciones correspondientes, para inducir una llamada a una función segura. Tal y como se muestra en la Tabla 6, la lista de funciones vulnerables está compuesta por 50 funciones que pueden causar cada uno de los cuatro tipos de vulnerabilidad, como desbordamiento de búfer, cadena de formato, condición de carrera y ejecución de múltiples comandos.

Tabla 6 Tipos de vulnerabilidades y funciones vulnerables

Tipo de vulnerabilidad	Funciones vulnerables
Desbordamiento de Buffer	Sstrcpy, wcscpy, stpcpy, wcpcpy, strepcy, memcpy, strcat, wcscat, streadd, strtrns, sprintf, vsprintf, vprintf, vfprintf, gets, scanf, fscanf, vscanf, vsscanf, sscanf, vfscanf, getwd, realpath
Formato de String	Syslog, vsyslog, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, snprintf, vsnprintf, vasprintf, asprintf, vdprintf, dprintf
Condición de carrera	Tmpnam, tmpnam_r, mktemp
Ejecución de multiple comando	System, popen, execve, fexecve, execv, execle, execl, execvp, execlp, exevpe

Análisis binario

1. **Instrumentación**

Para extraer información binaria, aprovechamos una herramienta de instrumentación llamada QEMU. QEMU es una herramienta de virtualización que se utiliza a menudo para el análisis binario dinámico. QEMU produce un TCG (generador de código pequeño) que convierte el código binario a IL (lenguaje intermedio). Después de la conversión, QEMU inserta un valor en el IL para extraer la información que queremos analizar. Utilizamos esta función de instrumentación para obtener la información que necesitamos.

2. **Análisis de complejidad**

Utilizamos las métricas de complejidad de Halstead para analizar la complejidad binaria. Las métricas fueron publicadas en 1977 por Maurice Howard Halstead y están destinadas a determinar la relación entre las propiedades medibles y la complejidad del software. Las métricas se desarrollaron originalmente para medir la complejidad del código fuente, pero las aplicaremos para medir la complejidad del código de ensamblaje. Analizamos la complejidad binaria midiendo el número de operadores y operandos en el código de ensamblaje utilizando las métricas definidas en la Tabla 7.

Tabla 7 Definición de métricas

Clasificación	Variable	Descripción
Variable	n_1	Número de operadores
	n_2	Número de operandos
	N_1	Operadores totales
	N_2	Operandos totales
	n	$n_1 + n_2$
	N	$N_1 + N_2$
Complejidad de fórmula	$V = N \times log_2 n$	Volumen del programa
	$\mathrm{D} = \frac{n_1}{2} \times \frac{N_2}{n_2}$	Dificultad del programa
	$\mathrm{E} = \mathrm{D} \times \mathrm{V}$	Esfuerzo
	$\mathrm{B} = \frac{E^{\frac{2}{3}}}{3000}$	Errores estimados

3. Puntuación de vulnerabilidades

Al ejecutar el binario, se rastrean todas las llamadas de función de la ruta ejecutada. En las llamadas a funciones rastreadas, verificamos si se llama a una función de las funciones vulnerables que hemos mostrado en la tabla anterior.

Asignamos una puntuación de vulnerabilidad por separado para funciones peligrosas y para funciones prohibidas como se muestra en la Tabla 8. Acumularemos puntuaciones cada vez que una función peligrosa o prohibida se llama. Hacemos referencia a la investigación sobre la detección de vulnerabilidades a través de la complejidad del software y la puntuación de la vulnerabilidad.

Tabla 8 Puntuación de vulnerabilidades

Puntuación de vulnerabilidad	Funciones
0,5 (Peligrosa)	Scanf, fscanf, vscanf, vsscanf, vsscanf, vfscanf, snprintf, vsnprintf, strtok, wcstok, itoa
1.0 (Prohibida)	Strcpy, wcspy, stpcpy, wcpcpy, strecpy, memcpy, stract, wcscat, streadd, strims, sprintf, vsprintf, vprintf, vfprintf, gets, getwd, realpath, syslog, vsyslog, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, vasprintf, asprintf, vdprintf, dprintf

Selección del motor

El módulo de selección de motor se encarga de elegir entre el motor de ejecución simbólica y de fuzzing a través de un análisis de complejidad del fichero binario. Para el análisis de complejidad, el usuario ha tenido que definir el valor de umbral como valor de entrada de la selección del motor. La cantidad de operadores, de operandos, los operadores totales y los operandos totales extraídos de la instrumentación binaria

dinámica se utilizan para medir la dificultad del programa. Si la dificultad del programa es inferior al umbral, el módulo de selección del motor elige el ejecutor simbólico. Si no, se selecciona el fuzzer. El algoritmo para la selección del motor de detección de vulnerabilidad es el siguiente:

Algoritmo: Selección del motor de detección de vulnerabilidades

```
Input: Threshold T
Data: Dificultad D, n1, n2, N1, N2
Result: Seed S
function Selección del motor (T)
begin
    D ← Complejidad (n1, n2, N1, N2);
    if D < T then
        S ← Ejecutor simbólico
    else
        S ← Fuzzer();
end
return S;
end
```

Ilustración 43 Algoritmo de detección de vulnerabilidades

Fuzzing

En este estudio se ha utilizado el American Fuzzy Lop (AFL), el cual realiza fuzzing en tres pasos. Primero, la AFL muta el valor semilla con varias estrategias de mutación de bits.

En segundo lugar, la AFL continúa supervisando el estado de ejecución y registra la transición de estado para ampliar la cobertura del código. Tercero, el AFL omite las partes que no afectan la transición de estado para minimizar el rango de mutación. Al repetir los tres pasos anteriores, el valor de entrada se desarrolla en la dirección de la transición de estado. No modificamos el motor central del AFL Fuzzer y consideramos solo su coordinación con el ejecutor simbólico.

Ejecutaremos el análisis de Fuzzer cuando el resultado del análisis binario genera una puntuación mayor que la que el usuario registró como umbral.

Ejecución simbólica

La herramienta Angr se ha utilizado en este estudio para la ejecución simbólica. Angr utiliza el VEX IR de Valgrind para crear un entorno de ejecución simbólico. SimuVEX, que es un emulador VEX IR, produce un entorno para ejecutar un estado simbólico traducido de binario a VEX IR.

El ejecutor simbólico opera cuando el resultado del análisis binario es menor que el marcado como umbral.

Generación automática del parche binario

El archivo de ejecución del entorno Linux tiene el formato ELF (archivo ejecutable y enlazable).

La estructura del archivo ELF se muestra brevemente en la Ilustración 44. En esta sección, el código es la función main() y es la inicialización para configurar el entorno de

ejecución, que consiste en el procesamiento de argumentos argc / argv, la configuración de la pila, la carga de la biblioteca para la ejecución de la función main() y el código de procesamiento.

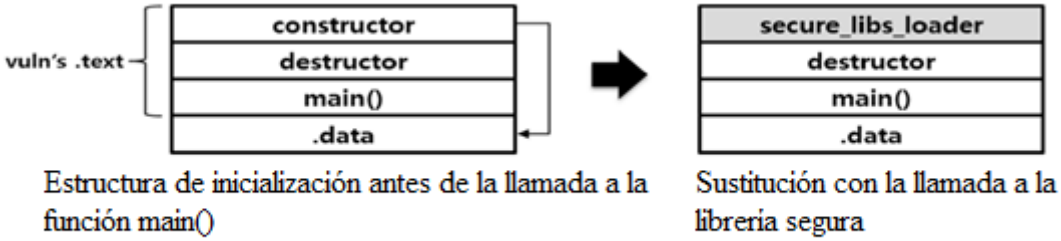


Ilustración 44 Generación automática de parches

Se reemplazará el área del constructor del binario con una biblioteca segura (secure_libs_loader) que se encarga de que GOT pueda cambiarse antes de que la función main() llame a alguna de las funciones vulnerables. La función de secure_libs_loader es, en última instancia, cargar libsecu.so, que es una biblioteca segura que se encuentra en el mismo espacio de memoria que el binario.

El secure_libs_loader encuentra la dirección de la primera función dlopen() como se muestra en la figura siguiente, y carga libsecu.so en la memoria. Después de cargar libsecu.so, el compilador llama a una función que realiza un parche PLT / GOT dentro de la biblioteca.

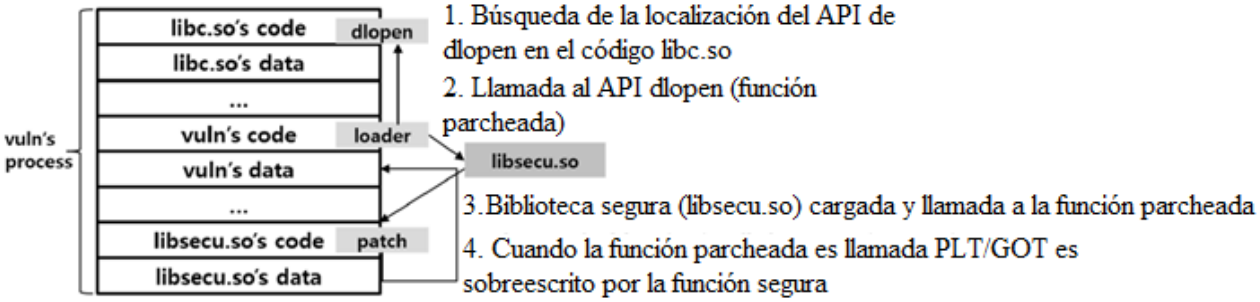


Ilustración 45

Todos los procesos ocurren antes de que se ejecute la función main(). Por lo tanto, el proceso de parches accede al GOT a través del PLT de la función de vulnerabilidad a la que la dirección real aún no está vinculada. Como nunca se ha llamado, el parche se ejecuta sobrescribiendo el GOT de la función vulnerable con la dirección para el enlace (generalmente ubicación de PLT + 6) con la dirección de la función segura.

Esta técnica está dirigida a minimizar la modificación binaria y reemplazar indirectamente las funciones vulnerables por funciones seguras. En otras palabras, es necesario modificar el constructor para cargar la biblioteca. Además, como la biblioteca cargada tiene la forma de una biblioteca compartida, se puede mantener y reparar de forma independiente.

Resultados experimentales

Resultado de los parches binarios

Se ha utilizado Peach Fuzzer para desencadenar bloqueos en nuestro software y se han comparado la cantidad de bloqueos antes y después de aplicar los parches para evaluar cuántos bloqueos se pueden eliminar a través del método de parches automáticos.

Sea B el número de bloqueos antes del parche y A el número de bloqueos después del parche. El método de medición de la tasa de eliminación de bloqueos de software se evaluó como R como se muestra en la siguiente ecuación.

$$R(\%) = (B - A)/B \times 100$$

Como resultado, la tasa promedio de eliminación de vulnerabilidades fue del 52%, ya que se producían otros bloqueos aparte de los que provocaban las funciones vulnerables que se habían sustituido con este método.

Mediante la aplicación de esta técnica las funciones vulnerables que se muestran en la Tabla 8 se convirtieron en funciones seguras a través de la carga segura de la biblioteca descrita anteriormente, disminuyendo de esta forma el número de vulnerabilidades de nuestro código.

Conclusiones

El número de vulnerabilidades está aumentando rápidamente debido al desarrollo de nuevas técnicas de piratería. Por este motivo es importante seguir unas pautas o tener unos procedimientos de actuación que traten de prevenir o detectar vulnerabilidades de software.

En este documento hemos diferenciado en tres partes los procedimientos que deberemos seguir para detectar y mitigar las vulnerabilidades que nos puedan surgir dependiendo de la fase del desarrollo del software en la que nos encontremos, ya sea la fase de diseño, desarrollo o la fase final. De esta forma podremos tener en cuenta los problemas que pueden surgir en cada una de las fases. Así sabremos cómo actuar siempre, independientemente de cuál sea nuestra función en el desarrollo del software, ya sea como analistas que tengamos que realizar un diseño, desarrolladores que tengamos que desarrollar el código que se nos propone o testers y tengamos que probar un desarrollo final.

Consideramos que para la realización de un software seguro deberemos seguir estas recomendaciones desde la fase de diseño, siguiendo las pautas que hemos marcado. Posteriormente durante la fase de desarrollo hemos visto que es una práctica indispensable el diseño y la implementación de unas pruebas unitarias y de integración, con ayuda de las herramientas explicadas, que prueben nuestro software a medida que se desarrolla y nos aseguren que no existen errores ni en el código ni en la integración en el sistema.

En la fase de desarrollo desde nuestra experiencia y punto de vista consideramos muy importante la implementación del código siguiendo una metodología de integración continua, que realice tareas de despliegue y pruebas de forma automática, mostrándonos en todo momento que el código que vamos introduciendo no daña el sistema ya desarrollado, así como llevar un control de versiones que nos permita responder ante posibles fallos de forma rápida y eficaz. También consideramos interesantes las herramientas de revisión de código estático que nos permiten la revisión de varios miembros del equipo de nuestro código, de tal forma que es más sencillo detectar posibles errores o sugerir mejoras en el software ya desarrollado.

Como procedimiento de actuación final hemos propuesto una metodología de análisis del código ya finalizado que no habíamos probado antes, esta metodología prueba sobre nuestro código ya definitivo nuestro software realizando pruebas y correcciones automáticas, buscando cual es el sistema que mejor se adapta a nuestro código según su complejidad.

El procedimiento propuesto para la fase de diseño del software lo consideramos indispensable y siguiendo nuestra experiencia, no siempre se aplican estas consideraciones que proponemos. Muchas veces la fase de diseño no es analizada como es debido siguiendo una metodología que tenga en cuenta los posibles errores que puedan surgir debido a un mal diseño del sistema, por este motivo consideramos que esta metodología puede ayudarnos mucho en nuestro día a día.

En cuanto a la fase del desarrollo del software sucede que no siempre se prueba de forma correcta ni con las herramientas necesarias el código que se está desarrollando provocando errores futuros que se podrían haber prevenido siguiendo las pautas que proponemos.

La fase final del desarrollo del software es una de las más delicadas y no siempre se prueba como tal, debido a la complejidad que puede tener el código se aplica una metodología para todo el software desarrollado, que puede basarse en pruebas humanas si intervención de herramientas automáticas de análisis, en lugar de identificar que metodología de pruebas automáticas puede ser mejor en función de la complejidad del código. Mediante este estudio podemos ver que existen metodologías que analizan el código tratando de identificar qué sistema de pruebas puede ser mejor o peor según su complejidad, y finalmente generar los parches necesarios para solucionar el problema de forma automática.

Como conclusión creemos que es importante seguir las tres metodologías propuestas en forma rutinaria en el desarrollo del software, es decir, tenerlas siempre en cuenta y aplicarlas en cada uno de los desarrollos que implementemos.

Deberemos tener en cuenta las vulnerabilidades que pueden existir en nuestro código desde las fases iniciales, y por lo tanto deberemos aplicar metodologías y procedimientos para evitarlas. En este documento se han propuesto tres metodologías en función de la fase de desarrollo en la que nos encontremos, que consideramos que nos ayudarán a generar un código más seguro.

Trabajos futuros

Como hemos indicado previamente deberemos tener en cuenta que las técnicas de piratería se encuentran en continuo cambio y evolución, y por lo tanto las vulnerabilidades aumentan gradualmente. Por este motivo deberemos estar continuamente buscando nuevas formas de detección y análisis de vulnerabilidades de software, adaptándonos a la a las nuevas vulnerabilidades, siendo capaces de responder con precisión y mitigando los problemas de nuestro software en un tiempo aceptable.

Debemos tener en cuenta que los computadores cada vez tienen una mayor capacidad y velocidad de procesamiento, también están surgiendo continuamente nuevos lenguajes y frameworks de programación, y con ellos nuevas vulnerabilidades, por lo que deberemos ser capaces de adaptar nuestras metodologías de detección de errores de software a dichos cambios y evolucionar nuestros procedimientos y técnicas de detección.

Aunque en este documento se han dado unas técnicas prevención y mitigación de vulnerabilidades de software que incluyen unas herramientas y frameworks de análisis recientes, y además nos hemos basado en estudios actuales sobre prevención de vulnerabilidades, no debemos olvidar que estas metodologías pueden quedar obsoletas en un periodo de tiempo relativamente corto.

Ya sea como analistas, desarrolladores o personal de pruebas de software deberemos estar continuamente investigando sobre nuevas técnicas de detección, mitigación y prevención de vulnerabilidades, ya que pueden surgir nuevas metodologías, herramientas y frameworks que sean más eficientes que los procedimientos que proponemos en este estudio, y que además se adapten a las nuevas vulnerabilidades que vayan surgiendo.

Bibliografía

[1] M. C. Tumino and Juan Manuel Bournissen Reinhard Schmidlin Emanuel Irrazábal, "Prácticas de Desarrollo Software: Un Estudio Exploratorio con Herramientas de Análisis Estático," *Revista Latinoamericana De Ingeniería De Software*, vol. 3, *(4)*, pp. 155-160, 2015

[2] E. Lluna, "Análisis estático de código en el ciclo de desarrollo de software de seguridad crítica," *REICIS: Revista Española De Innovación, Calidad e Ingeniería Del Software*, vol. 7, *(3)*, pp. 26-38, 2011.

[3] G. Evron, *Open Source Fuzzing Tools.* (1st ed.) 2007;2011;.

[4] K. Tsipenyuk, B. Chess and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security & Privacy*, vol. 3, *(6)*, pp. 81-84, 2005.

[5] R. C. Seacord, *Secure Coding in C and C++*. (2nd ed.) 2013.

[6] M. S. Merkow, L. Raghavan and I. Books24x7, *Secure and Resilient Software Development.* (1st ed.) 2010.

[7] https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html

[8] U. Sarmah, D. K. Bhattacharyya and J. K. Kalita, "A survey of detection methods for XSS attacks," *Journal of Network and Computer Applications*, vol. 118, pp. 113-143, 2018.

[9] M. Meyer, "Continuous Integration and Its Tools," IEEE Software, vol. 31, (3), pp. 14-16, 2014.

[10] P. M. Duvall, S. Matyas and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk. (First ed.) 2007.

[11] https://es.atlassian.com/software/bamboo

[12] https://gerrit-review.googlesource.com/Documentation/

[13] https://www.mercurial-scm.org/

[14] https://git-scm.com/

[15] https://jenkins.io/

[16] The OWASP Foundation, The Ten Most Critical Web Application Security Risks, OWASP Top 10 -2017.

[17] J. Jurn, T. Kim and H. Kim, "An Automated Vulnerability Detection and Remediation Method for Software Security," *Sustainability*, vol. 10, *(5)*, pp. 1652, 2018.

[18] G. Yee, "Removing software vulnerabilities during design," in 2018. DOI: 10.1109/COMPSAC.2018.10284.

[19] OWASP Top 10 -2017. Los diez riesgos más críticos en Aplicaciones Web.

Siglas, abreviaturas y acrónimos

- SOA: Arquitectura Orientada a Servicios.
- AFL: American Fuzzy Loop.
- OWASP: Open Web Application Security Project.
- OS: Sistema Operativo.
- API: Interfaz de programación de aplicaciones.
- HTTP: Protocolo de transferencia de hipertexto.
- JRE: Java Runtime Environment.
- WAR: Archivo de aplicación web.
- SD: Datos confidenciales.
- MVIR: Modelo para la Identificación y Eliminación de Vulnerabilidades.
- AST: Árbol de sintaxis abstracta.
- IL: Lenguaje intermedio.
- GOT: Global Offsets Table.
- PLT: Procedure Linkage Table. Tabla de vinculación de procedimientos.
- SQL: Structured Query Language.
- XML: Extensible Markup Language.
- XSL: Extensible Stylesheet Language.
- XSD: XML Schema Definition.
- URI: Uniform Resource Identifier.
- JSON: JavaScript Object Notation.
- LAN: Local Area Network.
- XSS: Cross Site Scripting.
- CSRF: Cross Site Request Forgery.
- HTML: HyperText Markup Language.
- HTTPS: Hypertext Transfer Protocol Secure.
- AJAX: Asynchronous JavaScript And XML.
- PRNG: Pseudo Random Number Generator.
- REST: Transferencia de estado representacional.