UNIVERSIDAD
SIGLO 21
La educación evoluciona

Especialidad en Cibercrimen

Evitar vulnerabilidades en el software por la falta de desarrollo seguro en los sistemas web

Resumen

El presente Trabajo Final tiene por finalidad crear una guía para implementar desarrollo seguro en las aplicaciones web para evitar vulnerabilidades en el software, las cuales pueden existir desde el mismo momento en que se encuentren operativas afectando los pilares básicos de la seguridad de la información; Se abordara la necesidad de contar con un modelo de seguridad para prevenir y disminuir los riesgos que se presenta en una aplicación web, y la implementación de mecanismos de seguridad que tienen que existir durante todas las fases del desarrollo de software para mitigarlos. Para ello se elaborará una guía con distintas cuestiones sobre desarrollo seguro dentro del ciclo de vida del software, mencionando posibles soluciones para mitigarlas y que podrá ser aplicada por completo o en parte por los que se dediquen al desarrollo de software y deseen verificar los requerimientos de pruebas de seguridad.
Por último, se concluirá este trabajo con una breve reseña acerca de la concientización necesaria de mitigar vulnerabilidades en el software, necesaria hoy en día ya que uno de los activos más valiosos es la información.

Autor: Carlos Pacheco

Junio 2024

Justificación

Actualmente, la seguridad de la información representa una preocupación crítica para las organizaciones. A medida que avanza la tecnología, también evolucionan los ciberataques que son cada vez más sofisticados (Forbesargentina, 2024), lo cual también afecta a la industria del software que desempeña un papel fundamental, ya que la tecnología y el software son componentes esenciales en la mayoría de las actividades empresariales y de la vida cotidiana.

Teniendo en cuenta el crecimiento del cibercrimen, el cual aprovecha todas las vulnerabilidades existentes en la red, el software no es ajeno a este, existiendo gran cantidad de ciberdelitos sobre sistemas web a causa de vulnerabilidades que estos presentan. El problema es que las vulnerabilidades de los sistemas son causadas porque los procesos de desarrollo muchas veces no contemplan dentro del ciclo de vida de desarrollo del Software (ISO12207, 2024) la comprobación de seguridad que debe ser parte de su proceso estándar.

Actualmente existen estándares y normativas adecuadas a las circunstancias en lo que refiere a cuestiones de seguridad durante el proceso del ciclo de vida del software estipulado por las normas ISO (ISO27001:2022-Anexo-A8.25, 2022), las que complementan la evaluación del ciclo de vida del software, mencionadas en las ISO 33061 (ISO33061, 2021); Y combinando las normas anteriores con distintas fuentes de información existentes de otras organizaciones que se dedican al estudio y análisis de vulnerabilidades existentes como OWASP (OWASP, owasp, 2024), Mitre ATT&CK (Mitre, s.f.) y NIST (Nist, s.f.) se pueden crear técnicas que ayuden a mitigar el crecimiento de la ciberdelincuencia existente en la actualidad en el ámbito de ciberataques web por falencias existentes durante su desarrollo.

El objetivo del presente trabajo es contar con una guía clara que proponga buenas prácticas sobre el desarrollo seguro en el ciclo de vida del software que contribuirá a reducir las vulnerabilidades en los sistemas web, ayudando a mejorar la seguridad de la información.

Objetivos

A) Objetivos Principal

- Desarrollar una guía que impacte en el ciclo de vida del desarrollo de software para generar buenas prácticas sobre desarrollo seguro.

B) Objetivos Específicos

- Explorar los principales ciberataques existentes, que afecten el software desde la perspectiva de vulnerabilidades en este.
- Describir principales técnicas para detectar vulnerabilidades durante el ciclo de vida del software.
- Plantear posibles formas de mitigar estos ciberataques desde la perspectiva del desarrollo seguro.

Marco Teórico

La Seguridad de la Información es el conjunto de objetivos, políticas, procesos y procedimientos que una organización define con la finalidad de preservar su información.

Los principios fundamentales de la Seguridad de la Información son la disponibilidad, integridad y confidencialidad de la información. La disponibilidad establece que la información puede ser accedida al momento que se necesite. En segundo lugar, la confidencialidad se refiere a que la información podrá ser consultada únicamente por personas que estén autorizadas. Y, por último, el principio de integridad de la información garantiza que la misma se resguarda en forma tal que es inalterada, exacta y total. (ISO/IEC., 2005)

Por otra parte, las organizaciones dependen en gran medida del uso de productos y servicios de tecnología de la información (TI) para ejecutar sus actividades diarias. Por lo cual garantizar la seguridad de estos productos y servicios es de suma importancia para el éxito de la organización, como menciona la publicación de M. Niels, K. Dempsey y Yan Pillitteri sobre los principios de seguridad de la información, en que las organizaciones pueden aprovechar para comprender las necesidades de seguridad de la información de sus respectivos sistemas. (Niels, Dempsey, & Pillitteri, 2017)

Y teniendo en cuenta que los procesos de auditorías se definen como un proceso sistemático, independiente y documentado para obtener evidencias y evaluarlas de manera objetiva con el fin de determinar la extensión en que se cumplen los criterios de auditoría (ISACA., 2019)(ISO 19011); Y en particular, las auditorías de Tecnologías Informáticas recolectan y analizan evidencia con la finalidad de corroborar que los activos se encuentren protegidos y se cumplan la integridad, disponibilidad y confidencialidad de la información, obtenemos como finalidad el obtener información que permita evaluar los controles mediante la implementación de técnicas que permitan identificar vulnerabilidades en el proceso auditable.

Entonces por lo mencionado anteriormente nos remitiremos a las distintas organizaciones que nos pueden ayudar y orientar sobre la mitigación de esta problemática teniendo como referencia a las siguientes:

OWASP - Open Web Application Security Project (OWASP, owasp, 2024), la cual es un proyecto de código abierto dedicado a determinar y combatir las causas que hacen que el software sea inseguro y determinar la forma de asegurarlo.

Esta entidad investiga y analiza cada determinado periodo de tiempo las vulnerabilidades más comunes encontradas en aplicaciones web y los errores de programación que generan dichas vulnerabilidades, generando una lista denominada TOP10, y además nos provee una guía para probar y corregir las aplicaciones (Top10, 2021). Entre los principales objetivos de OWASP top 10 podemos mencionar los siguientes:

- Basado en the owasp risk rating methodology.
- El objetivo principal de owasp top 10 es educar a desarrolladores, diseñadores, arquitectos, gerentes y organizaciones sobre las consecuencias de las vulnerabilidades de seguridad en aplicaciones web.
- Proveer información sobre como evaluar los riesgos en las aplicaciones web.
- Describe la probabilidad general de ocurrencia y sus consecuencias.
- Presenta una orientación sobre como verificar los problemas y como evitarlos.

NIST - National Institute of Standards and Technology (Nist, s.f.): Es una agencia de la Administración de Tecnología del Departamento de Comercio de Estados Unidos y la misión de este instituto es promover la innovación y la competencia industrial en Estados Unidos mediante avances en metrología, normas y tecnología de forma que mejoren la estabilidad económica. A continuación, mencionamos los siguientes estándares que se encuentran relacionados:

Estándar **Nist 800-64** el cual dispone entre sus características:

- Security Considerations in the SDLC: guía para integrar seguridad en el SDLC desde la concepción del sistema.
- Se describen las entradas y salidas del sistema, se realiza in BIA y se determinan las contingencias.
- Se busca maximizar el retorno de inversión y manejar el riesgo en cualquier tipo de proyecto.

Estándar **Nist 800-268** el cual dispone entre sus características:

- Nist samate (software assurance metrics and tool evaluation): proyecto dedicado a mejorar la seguridad del software mediante el desarrollo de métodos, herramientas de evaluación y medición de calidad.

- El alcance del proyecto va desde sistemas operativos hasta sistemas SCADA y aplicaciones web.

- Principalmente basado en CWE.

CWE - Top25 (cwe-mitre, s.f.)**:** Se trata de una lista de las 25 debilidades de software más peligrosas del año 2023, las cuales son publicadas por la organización **Mitre ATT&CK** (Mitre, s.f.).

Esta lista muestra las debilidades de software más comunes e impactantes en la actualidad, que son fáciles de encontrar y explotar, y pueden dar lugar a vulnerabilidades explotables que permiten a los adversarios tomar el control total de un sistema, robar datos o impedir que las aplicaciones funcionen. Entre sus principales características tenemos:

- CWE/Sans top25 es una lista de los errores de programación más comunes, difundidos y críticos que pueden conducir a serias vulnerabilidades en el software.

- CWE sitio contiene datos sobre más de 800 errores de programación, arquitectura y diseño que pueden dar lugar a vulnerabilidades explotables.

Las características de esta lista son:

- Adaptando estas mitigaciones a cada aplicación, la misma serán más segura.

- Se dividen en mitigaciones específicas para cada error y, en generales, aplicables a todas las vulnerabilidades.

- Se clasifican en alta, moderadas, limitadas, y defensa in Depth.

Esta lista ayudara a crear técnicas de mitigación aplicables y efectivas para prevenir y solucionar las vulnerabilidades del owasp top 10 y del Sans top 25 contribuyendo a la guía a desarrollar en nuestro documento.

Otras de las características consideradas importantes a tener en cuenta para el desarrollo de nuestro documento es el paradigma de programación al cual pertenece el sistema, como así también las nuevas tendencias sobre Security Software Development Life Cycle(SSDLC), el cual es un enfoque del desarrollo de software que integra prácticas y consideraciones de seguridad a lo largo de todo el ciclo de vida del desarrollo de software, desde el diseño y el desarrollo hasta la implementación y el mantenimiento el cual desarrollaremos (ISO27001:2022-Anexo-A8.25, 2022).

Por lo cual es necesario sentar las bases de algunas definiciones sobre los siguientes conceptos, por la temática tratada y para ello definiremos los mismos:

Lenguajes de programación: Un lenguaje de programación es un lenguaje formal (o artificial, es decir, un lenguaje con reglas gramaticales bien definidas) que proporciona a una persona, en este

caso el programador, la capacidad y habilidad de escribir (o programar) una serie de instrucciones o secuencias de órdenes en forma de algoritmos con el fin de controlar el comportamiento físico o lógico de un sistema informático, para que de esa manera se puedan obtener diversas clases de datos o ejecutar determinadas tareas. A todo este conjunto de órdenes escritas mediante un lenguaje de programación se le denomina programa informático (Wikipedia, Wikipedia, s.f.).

Paradigmas de programación: Se denominan paradigmas de programación a las formas de clasificar los lenguajes de programación en función de sus características. Los lenguajes se pueden clasificar en múltiples paradigmas. Algunos paradigmas se ocupan principalmente de las implicancias para el modelo de ejecución del lenguaje, como permitir efectos secundarios o si la secuencia de operaciones está definida por el modelo de ejecución. Otros paradigmas se refieren principalmente a la forma en que se organiza el código, como agrupar un código en unidades junto con el estado que modifica el código. Sin embargo, otros se preocupan principalmente por el estilo de la sintaxis y la gramática (Wikipedia, Wikipedia, s.f.).

Con respecto a los paradigmas de programación, concepto que interviene en este documento se aclara que los mismos no solo tienen relevancia sobre cómo se escribe el código, sino que también en la forma en la que se manipulan, gestionan y protegen los datos y recursos de la información. Por lo tanto, tienen una estrecha relación con la Seguridad de la Información y se considera apropiado que se identifiquen los riesgos asociados a cada uno de los paradigmas para generar contramedidas oportunas sobre el mismo en caso de que sea el correspondiente al sistema a evaluar.

Por lo mencionado anteriormente, es necesario tener en cuenta el paradigma de programación que se aplicara en el desarrollo del software para analizar las posibles vulnerabilidades que existen sobre el mismo y como mitigarlas. Y teniendo en cuenta que, desde los paradigmas iniciales hasta los paradigmas modernos o más actuales, cada uno de estos presentan sus propias características, así como riesgos que influyen significativamente en la seguridad del software.

Entonces, ahora se hará una exploración resumida sobre los principales paradigmas y como estos afectan la Seguridad de la Información, identificando los riesgos asociados a cada uno para que sea tenido en cuenta sobre los controles y estrategias de auditorías que resultarían efectivos para mitigarlos.

Entre los mismos describiremos los siguientes:

- Programación imperativa:

  Estos son los que describe una secuencia de instrucciones ordenadas para resolver un problema, las mismas van modificando el estado del programa a medida que se ejecutan. Entre sus principales posibles fallas tenemos:

- La manipulación incorrecta de memoria genera desbordamientos de buffer, permitiendo la ejecución de código malicioso.
- La ejecución concurrente sin sincronización causa condiciones de carrera y corrupción de datos.
- La falta de validaciones robustas puede producir inyecciones de código.
- El uso inadecuado de punteros puede causar acceso a memoria no autorizados y fallos de seguridad.

- Programación Funcional:

  La programación funcional tiene sus orígenes en el cálculo de raíces y la teoría de funciones recursivas. Las bases de este modelo de programación son las funciones, el cual es su concepto central. Con este paradigma, aparece el concepto de modularidad, técnica que permite dividir el problema en partes más pequeñas, hasta alcanzar un problema tan simple que pueda resolverse directamente. Entre sus principales posibles fallas tenemos:

  - Si las funciones no se validan apropiadamente las entradas, pueden sufrir inyecciones de código.
  - La exposición de funciones mediante interfaces públicas puede explotarse si no se encuentran asegurada.
  - Si los efectos laterales no se gestionan adecuadamente, puede introducirse vulnerabilidades.
  - La dificultad para gestionar el estado funcional puede ser aprovechado por atacantes.

- Programación Lógica:

  En este modelo, los programas son declarativos: se construyen a partir de la definición de relaciones lógicas entre hechos y reglas. Entre sus principales posibles fallas tenemos:

  - Las consultas mal protegidas pueden ser manipuladas maliciosamente.
  - El procesamiento de inferencias puede explotarse para generar ataques de Denegación de Servicio (DoS).
  - La manipulación no autorizada de los hechos y reglas puede ocasionar comportamientos indeseados.

- La exposición de detalles sobre la lógica se puede aprovechar para entender y explotar el sistema.

- Programación Orientada a Objetos:

  Este paradigma propone un mayor nivel de abstracción con el objetivo de mejorar las características del código final a partir de la introducción de conceptos como clases y objetos. Otros aspectos destacables de este paradigma son la composición, herencia y polimorfismo, los cuales permiten la posibilidad de la reutilización de código.
  Entre sus principales posibles fallas tenemos:

  - Si los métodos no están protegidos, la herencia y el polimorfismo pueden ser explotados.

  - La exposición de atributos y métodos permite el acceso no autorizado.

  - Los procesos inseguros de serialización y deserialización pueden sufrir inyección de código malicioso.

  - La ausencia de encapsulamiento permitiría la manipulación directa de los objetos.

- Programación Orientada a Servicios:

  Esta metodología define a los comportamientos de los sistemas como servicios y entiende a los servicios como unidades autónomas que realizan una función específica y se comunican con otros por medio de protocolos estándar. Entre sus principales posibles fallas tenemos:

  - Las comunicaciones entre servicios pueden interceptarse si no están adecuadamente cifradas.

  - Los servicios mal diseñados, son vulnerables a inyecciones de código a través de las interfaces de servicio.

  - La ausencia de controles robustos de autenticación permite accesos no autorizados.

  - Las APIs expuestas pueden sufrir ataques de Denegación de Servicio (DoS) o el uso indebido de recursos.

Como se observa, cada uno de los paradigmas de programación, con sus características y diversos enfoques, están expuestos a riesgos de seguridad. Y otros como por ejemplo el orientado a servicios están expuestos a riesgos propios de su modelo, sin embargo, también están expuestos a los riesgos de los otros paradigmas afirmando que cada paradigma de programación tiene riesgos inherentes únicos.

Como conclusión respecto a esto podemos decir que cada paradigma presenta desafíos de seguridad únicos que requieren enfoques de mitigación específicos.

Ciclo de vida de desarrollo de software (Wikipedia, Wikipedia-CicloVidaDesarrolloSoftware, s.f.): El Proceso para el desarrollo de software, también denominado ciclo de vida del desarrollo de software es una estructura aplicada al desarrollo de un producto de software. Hay varios modelos a seguir para el establecimiento de un proceso para el desarrollo de software, cada uno de los cuales describe un enfoque diferente para diferentes actividades que tienen lugar durante el proceso. Algunos autores consideran un modelo de ciclo de vida un término más general que un determinado proceso para el desarrollo de software. Y existen distintos modelos de desarrollo de software como por ejemplo el modelo en Espiral, Incremental, Cascada, etc.

DevSecOps (RedHat, s.f.): DevSecOps significa desarrollo, seguridad y operaciones. Se trata de un enfoque que aborda la cultura, la automatización y el diseño de plataformas, e integra la seguridad como una responsabilidad compartida durante todo el ciclo de vida de la TI.
Actualmente, en el marco de trabajo en colaboración de DevOps, la seguridad es una responsabilidad compartida e integrada durante todo el proceso. Es un enfoque tan importante que llevó a que se acuñara el término "DevSecOps" para enfatizar la necesidad de diseñar una base de seguridad en las iniciativas de DevOps. DevSecOps implica pensar desde el principio en la seguridad de las aplicaciones y de la infraestructura. También supone automatizar algunas puertas de seguridad para evitar que se ralenticen los flujos de trabajo de DevOps. A fin de cumplir con estos objetivos, es necesario seleccionar las herramientas adecuadas para integrar la seguridad de manera permanente, como acordar el uso de un entorno de desarrollo integrado (IDE) con funciones que permitan proteger los sistemas. Sin embargo, la seguridad efectiva de DevOps requiere más que nuevas herramientas, ya que se construye sobre los cambios culturales de este enfoque para integrar el trabajo de los equipos de seguridad lo antes posible.
Ya sea que utilice los términos "DevOps" o "DevSecOps", siempre es conveniente incluir a la seguridad como parte integral de todo el ciclo de vida de la aplicación. Con DevSecOps, la seguridad es una función integrada y no un perímetro que rodea a las aplicaciones y los datos.

En parte, DevSecOps destaca la necesidad de que los equipos y los partners de seguridad adopten las iniciativas de DevOps desde el comienzo para incorporar medidas de protección de la información y establecer un plan para automatizar la seguridad. Resalta la necesidad de ayudar a los desarrolladores a escribir el código teniendo en cuenta la seguridad, lo cual implica que los equipos encargados de esta área compartan el conocimiento, los comentarios y la información valiosa sobre las amenazas conocidas. Además, se centra en identificar los riesgos de la cadena de suministro del software y hace hincapié en la seguridad de los elementos del software open source y las dependencias al principio del ciclo de vida del desarrollo. Para tener éxito, un enfoque efectivo de DevSecOps debe incluir también capacitaciones de seguridad nuevas para los desarrolladores, ya que no siempre ha sido el centro del desarrollo de aplicaciones más tradicional.

Security Software Development Life Cycle (Microsoft, Microsoft-SSDCL, s.f.)**:** El ciclo de vida de desarrollo de seguridad (SDL) es un enfoque para integrar la seguridad en los procesos de DevOps (denominado enfoque DevSecOps). Las prácticas descritas en el enfoque SDL se pueden aplicar a todo tipo de desarrollo de software y a todas las plataformas, desde el enfoque clásico en cascada hasta los enfoques modernos.

Ahora teniendo en cuenta que el software se encuentra expuestos a distintos tipos de ataques informático que son un factor común respecto a vulnerabilidades existentes a causa de falencias existentes en el código de programación podemos mencionar como unos de los principales el ataque el de inyección de código interpretado o inyección de scripts. Esta inyección de código hace uso de los errores al procesar información errónea, lo cual puede ser usado por un atacante al introducir código en un programa para cambiar la ejecución normal.

Bajo la denominación de inyección de scripts se agrupan distintas técnicas que comparten el mismo sistema de explotación pero que persiguen distinto fin. El hecho de separarlas en distintas categorías atiende únicamente a facilitar la fijación de los objetivos perseguidos. Por lo cual el "Cross Site Scripting" es la base de distintas técnicas que conforman distintos tipos de ciberataques de la misma naturaleza (XSS, CSRF, XFS, XZS, XAS, XRS, Image Scripting, IMAP3 XSS).

En el caso del Cross-Site Scripting (XSS), los atacantes insertan scripts maliciosos, generalmente JavaScript, en sistemas web que son vistas por otros usuarios. Estos scripts se ejecutan en el navegador de las víctimas, lo que permite al atacante robar cookies de sesión, redirigir a los usuarios a páginas falsas o comprometer su información personal y una amplia gama de acciones maliciosas que incluso incluye la distribución de malware. También cabe aclarar que la forma en que afecta este tipo de ataque dependerá en buena medida de la habilidad que tiene el atacante para "saltarse" (hacer un "bypass") los filtros que pueda tener una determinada aplicación.

Entre sus principales características podemos mencionar en forma de resumen que:

- Cross site scripting (XSS) es una ejecución de script y comandos no deseados a través de aplicaciones web, que explotan la confianza del usuario.

- Se originan por la validación incorrecta de variables que permiten ejecutar scripts en campos de entrada.

- El atacante inyecta código malicioso (HTML y Scripts) que son ejecutados en el entorno del navegador del cliente afectado.

- Mediante el control del navegador del usuario es posible realizar ataques de:

  1-Robo de sesión e identidad, mediante la manipulación de las cookies.

  2-Phishing, mediante la modificación de la interfaz normal del sitio.

3-Redireccion a sitios dañinos.

Otra característica de este es que podemos distinguir tres grandes grupos (Kaspersky2024, s.f.):

• Persistente: El XSS almacenado, también conocido como XSS persistente, se considera el tipo de ataque de XSS más perjudicial. Un XSS almacenado se produce cuando en cada visita al sitio se carga el XSS y el navegador web ejecuta la carga en el lado del cliente a través del navegador.

Las características de esta variante del Cross Site Scripting son:

- Es posible almacenar el código del XSS en la base de datos del servidor.
- Una vez almacenado, cada usuario que visualice la página será víctima del código malicioso.
- El script se ejecuta en el entorno del navegador de la víctima.

• No Persistente: Es el tipo más habitual, en este caso, la carga del atacante debe formar parte de la solicitud enviada al servidor web. Esta carga de XSS reflejado se ejecuta en el navegador del usuario, por lo que el atacante debe enviar la carga a cada víctima.

• Scripting entre sitios basado en DOM: Un XSS basado en DOM hace referencia a una vulnerabilidad de scripting entre sitios que aparece en el DOM (Modelo de Objetos del Documento) y no en una parte del HTML. En los ataques de scripting entre sitios reflejados y almacenados, se puede ver la carga de la vulnerabilidad en la página de respuesta; sin embargo, en el scripting entre sitios basado en DOM, el código fuente HTML del ataque y la respuesta serán los mismos, es decir, no se puede encontrar la carga en la respuesta. Solo se puede observar en el tiempo de ejecución o analizando el DOM de la página.

Herramientas: Entonces conociendo algunas de las principales características de esta metodología utilizada por ciberdelincuentes, podemos incursionar en herramientas y técnicas que ayuden a mitigar esta problemática en un sistema web. Por lo cual para comenzar es necesario explorar algunas de las herramientas que existen actualmente en el mercado, las cuales pueden llegar a incluirse dentro de la guía propuesta en este documento.

Entre las herramientas propuestas se incluyen las siguientes:

• Acunetix: Es un escáner de vulnerabilidades web desarrollado por un equipo especializado en el año 2005, el mismo utiliza una interfaz web con tecnologías que ayudan a descubrir más vulnerabilidades como AcuMonitor y AcuSensor, proporcionando ayuda para poder encontrar la vulnerabilidad en el código fuente. Actualmente solo existe la versión privativa, la cual tiene un costo (Acunetix, s.f.).

• Burp Suite: Es una herramienta de seguridad de software, de código abierto que se utiliza para hacer pruebas de pentesting y descubrir vulnerabilidades en aplicaciones web. Fue desarrollado por PortSwigger Web Security. Actualmente existen las versiones “community” para uso gratuito y la versión “empresarial” con mayores cantidades de herramientas respecto a la versión community, la cual tiene un costo (Suite, s.f.).

• Zaproxy: Se trata de una herramienta diseñada para identificar vulnerabilidades en aplicaciones web. Es parte de un proyecto de OWASP (OWASP, owasp, 2021), por lo cual es de código abierto y gratuito existiendo una gran comunidad para su mantenimiento y actualización. Su principio de funcionamiento se basa en un proxy de intermediario, disponiendo también de otras características como escaneo activo y pasivo, interceptación y modificación de tráfico, automatización, api y Spidering automáticos (zaproxy, s.f.).

• Marco de explotación XSS de OWASP Xenotix: se trata de un framework open source con un marco avanzado de detección y explotación de vulnerabilidades Cross Site Scripting (XSS). Proporciona resultados de escaneo de cero falsos positivos con su exclusivo escáner integrado Triple Browser Engine (Trident, WebKit y Gecko). Se afirma que tiene la segunda carga útil XSS más grande del mundo, con más de 1500 cargas útiles XSS distintivas para la detección efectiva de vulnerabilidades XSS y la omisión de WAF (Web Application Firewall). Incorpora un módulo de recopilación de información rico en funciones para el reconocimiento de objetivos. El marco de explotación incluye módulos de explotación XSS altamente ofensivos para pruebas de penetración y creación de prueba de concepto (OWASP-Xenotix-XSS-Exploit-Framework, s.f.).

• Fortify: Fortify es un Analizador de código estático y pertenece a OpenText Fortify el cual tiene una licencia open source. Este analiza el código fuente y señala la causa raíz de las vulnerabilidades de seguridad, localizando la causa raíz de las vulnerabilidades de seguridad en el código fuente y priorizando los problemas más graves y proporciona orientación detallada sobre cómo solucionarlos.

Este Analizador de código estático, dispone de varios analizadores de vulnerabilidades, como Buffer, Contenido, Flujo de control, Flujo de datos, Semántico, Configuración y Estructural. Cada uno de estos analizadores acepta un tipo diferente de regla adaptada para ofrecer la información necesaria para el tipo de análisis realizado. Entre sus principales características mencionamos que soporta múltiples lenguajes de programación, se integra fácilmente con herramientas CI/CD y a los principales IDE como Visual Studio y Eclipse por mencionar algunos y que dispone de alertas en tiempo real, y un asistente de auditoría impulsado por aprendizaje automático (OpenText, s.f.).

• Proyecto OWASP-dependency-check: Dependency-Check es una herramienta de análisis de composición de software (SCA) que intenta detectar vulnerabilidades divulgadas públicamente contenidas en las dependencias de un proyecto. Para ello, determina si existe un identificador de enumeración de plataforma común (CPE) para una dependencia determinada. Si lo encuentra,

generará un informe que vinculará las entradas CVE asociadas. Esta herramienta tiene una interfaz de línea de comandos, y su actualización es automática (OWASP P.-d.-c. , s.f.).

• Proyecto OWASP-dependency-track: Dependency-Track es una plataforma de análisis de componentes inteligente que permite a las organizaciones identificar y reducir los riesgos en la cadena de suministro de software. Dependency-Track adopta un enfoque único y sumamente beneficioso al aprovechar las capacidades de la lista de materiales de software (SBOM). Este enfoque proporciona capacidades que las soluciones tradicionales de análisis de composición de software (SCA) no pueden lograr. La misma monitorea el uso de componentes en todas las versiones de cada aplicación de su cartera para identificar de manera proactiva los riesgos en todo el software. La plataforma tiene un diseño que prioriza las API y es ideal para su uso en entornos CI/CD (OWASP P.-t. , s.f.).

• Nessus: Herramienta de ciberseguridad multiplataforma que incluye diferentes configuraciones (Templates) que permiten hacer escaneos a medida, y dispone de otros Templates de uso común como el Advanced Scan y el Web Application Tests para la detección de vulnerabilidades simple, escalable y automatizada de aplicaciones web. Esta proporciona una evaluación de vulnerabilidades completa y precisa, desde los 10 principales riesgos de OWASP hasta los componentes vulnerables de las aplicaciones web y las API permitiendo obtener visibilidad unificada de las vulnerabilidades de TI y las aplicaciones web para operar con eficacia su mitigación. Los resultados del escaneo pueden ser exportados como informes en varios formatos, como texto plano, XML, HTML, y LaTex y/o ser guardados en una base de conocimiento para referencia en futuros escaneos de vulnerabilidades. Se trata de una herramienta privativa, pero registrándose se puede conseguir una versión de prueba por 7 días (Nessus, s.f.).

• SDL Threat Modeling Tool (TMT): La herramienta de modelado de amenazas de Microsoft facilita el modelado de amenazas para todos los desarrolladores a través de una notación estándar para visualizar los componentes del sistema, los flujos de datos y los límites de seguridad. También ayuda a los modeladores de amenazas a identificar las clases de amenazas que deben tener en cuenta en función de la estructura del diseño de su software. Diseñamos la herramienta teniendo en cuenta a los expertos no especializados en seguridad, lo que facilita el modelado de amenazas para todos los desarrolladores al proporcionar una guía clara sobre la creación y el análisis de modelos de amenazas (Microsoft, threatmodeling, s.f.).

• WAPITI: Wapiti es una herramienta que permite auditar la seguridad de los sitios web o aplicaciones web. Realiza escaneos de "caja negra" (no estudia el código fuente) de la aplicación web rastreando las páginas web de la aplicación implementada, buscando scripts y formularios donde se pueda inyectar datos. Una vez que obtiene la lista de URL, formularios y sus entradas, Wapiti actúa como un fuzzer, inyectando cargas útiles para ver si un script es vulnerable (Wapiti, s.f.).

• Nikto: Nikto es un escáner de servidores web de código abierto que realiza pruebas exhaustivas en servidores web para detectar múltiples elementos, incluidos más de 7000 archivos y programas potencialmente peligrosos, verifica versiones obsoletas de más de 1250 servidores y problemas específicos de la versión en más de 270 servidores. También verifica elementos de configuración del servidor, como la presencia de múltiples archivos de índice, opciones de servidor HTTP e intentará identificar servidores web y software instalados. Los elementos y complementos de escaneo se actualizan con frecuencia y se pueden actualizar automáticamente. Esta herramienta no está diseñada como una herramienta sigilosa, por lo cual no es adecuada para pruebas de pentest, pero si probará un servidor web en el menor tiempo posible (Cirt, s.f.).

• AntiXSSLibrary: AntiXSS es una biblioteca de Microsoft para proteger a los usuarios de ataques de cross-site scripting (XSS), que utiliza un enfoque de lista segura para la codificación. Proporciona métodos de codificación HTML, XML, URL, formulario, LDAP, CSS, JScript y VBScript para permitirle evitar ataques de secuencias de comandos entre sitios. Esta biblioteca forma parte de las herramientas SDL de Microsoft ya que fue creada por la compañía Microsoft Corporation y dispone de licencia publica "Microsoft Public License (Ms-PL)" (Microsoft, Biblioteca-anti-CrossSiteScripting, s.f.).

• URLScan: herramienta de seguridad que restringe los tipos de peticiones http en Internet Information Server IIS (Microsoft, FxCop, s.f.).

• IIS Lockdown Tol: templates de configuración y seguridad para Internet Information Server IIS (Microsoft, Herramienta-IIS, s.f.).

• Veracode: Veracode es una plataforma privativa que ofrece una versión trial por 14 días, que sirve para escanear superficies de ataque de aplicaciones web, proporcionándote un análisis de seguridad con una serie de herramientas para detectar y analizar vulnerabilidades en el código fuente de las aplicaciones.
Esta herramienta utiliza una amplia variedad de técnicas de análisis estático y dinámico para identificar vulnerabilidades de seguridad en el código fuente de las aplicaciones. Esto permite a los desarrolladores identificar y corregir vulnerabilidades de seguridad reduciendo el riesgo de que las aplicaciones sean explotadas por atacantes malintencionados (Veracode, s.f.).

• SonarQube: SonarQube es una plataforma de código abierto para la inspección continua de la calidad del código a través de diferentes herramientas de análisis estático de código fuente. Proporciona métricas que ayudan a mejorar la calidad del código de un programa permitiendo a los equipos de desarrollo hacer seguimiento y detectar errores y vulnerabilidades de seguridad para mantener el código limpio. Entre sus principales características podemos mencionar que se encuentra preparada para admitir los lenguajes de programación más populares permitiendo ampliar sus funcionalidades con el uso de distintos complementos (SonarQube, s.f.).

• AppVerifer: Application Verifier es una herramienta que nos brinda Microsoft en forma gratuita que sirve para la comprobación en tiempo de ejecución y permite encontrar errores de programación sutiles, problemas de seguridad y problemas de privilegios de cuenta de usuario limitados que pueden ser difíciles de identificar con técnicas de prueba de aplicaciones normales. Este comprobador de aplicaciones busca errores de programación, supervisando las acciones de la aplicación mientras se ejecutan, y somete la aplicación a una variedad de distintos tipos de pruebas que son parametrizables, generando un informe sobre posibles errores en la ejecución o el diseño de la aplicación. La finalidad de AppVerifier es ayudar a crear aplicaciones confiables y seguras (Microsoft, AppVerifier, s.f.).

Marco Metodológico

La ciberseguridad es una prioridad fundamental para las organizaciones de todos los sectores. Particularmente, las vulnerabilidades en el software que ponen en juego la confidencialidad, disponibilidad e integridad de la información, representan un riesgo importante que debe ser identificado, evaluado y mitigado ya que en caso de que estos riesgos se materialicen, pueden tener consecuencias devastadoras para la organización.

Teniendo en cuenta que la tecnología avanza rápidamente, adaptarse a nuevas necesidades y desafíos es fundamental, ya que las superficies de ataque también se incrementan, y según el informe “Cyberthreat Minute” de Microsoft (Microsoft, Microsoft, 2022), el cual expone que actualmente el cibercrimen es una actividad en expansión, produciéndose 34740 ataques de credenciales por minutos y un ataque de Inyección SQL por minuto (se aclara que en este documento solo se indican las métricas de ciberataques relacionados a vulnerabilidades del software, ya que el mismo incluye métricas de otros tipos de ciberataques producidos), se puede afirmar que nuestro espacio del problema es causado por la falta de mecanismos de control durante el desarrollo del software incrementando de esta forma las vulnerabilidades en el desarrollo de este, lo cual demuestra la falta de procesos que consideran la seguridad de los componentes del software en el ciclo de vida de este.

Por consiguiente, para mitigar la problemática existente por la falta de desarrollo seguro, se pueden implementar pruebas exhaustivas de seguridad para asegurar que el sistema esté protegido contra ciberataques y accesos no autorizados. Estas pruebas también denominadas auditorías y pruebas de seguridad que requiere una gestión eficiente servirán para identificar y corregir vulnerabilidades en el código durante una fase previa antes que el software se encuentre en producción.

Otro aspecto importante para mencionar es que actualmente la seguridad de las aplicaciones es un aspecto fundamental y prioritario teniendo como su principal variable el desarrollo seguro de estas, considerando la seguridad durante todo su ciclo de vida. Por lo cual muchas empresas que desarrollan software pueden intentar añadir controles y procedimientos en cada una de las fases con el objetivo de incorporar seguridad a sus aplicaciones. No obstante, existen metodologías e

iniciativas que aportan las pautas necesarias y pueden ser utilizadas como referencia para facilitar esta tarea. Algunos ejemplos de estas metodologías que podemos mencionar son las de las empresas Oracle con su implementación de Oracle Software Security Assurance (Oracle, Oracle-OSSA, s.f.) (Oracle, Oracle-GuiaSeguridad, s.f.), Comprehensive Lightweight Application Security Process (CLASP) de la organización Open Web Application Security Project (OWASP-Clap, s.f.) , el Ciclo de vida de desarrollo de seguridad de Microsoft (Microsoft, SDL-Microsoft, s.f.) (Microsoft, Microsoft-SSDLC, s.f.) entre algunas. Y en este caso nos enfocaremos en describir las características de esta última, la de Microsoft.

• Microsoft SDL (Microsoft Security Development Lifecycle) (Microsoft, SDL-Microsoft, s.f.): Microsoft tiene una larga historia de implementación de seguridad de software, con proyectos como la compatibilidad con criterios comunes. Sin embargo, antes de la adopción del proceso SDL en Microsoft, los procesos de seguridad se aplicaban de modo dispar. Los equipos de los productos implementaban protecciones a la seguridad y privacidad en gran medida siguiendo su propio criterio. Algunos equipos de desarrollo sometían sus aplicaciones a un examen exhaustivo, mientras que otros daban prioridad a las nuevas funciones y la funcionalidad por encima de la seguridad y privacidad.

Posterior a esto, y luego de una serie de incidentes de alto perfil relacionados con software malintencionado llevaron a Microsoft a reformular sus procesos y estrategia de seguridad para los desarrolladores. Y se presentó una propuesta de proceso obligatorio a principios de 2004 ante el equipo de dirección principal de Microsoft; Tal propuesta fue aprobada como directiva de Microsoft, con el resultado de que todos los productos y servicios en línea que estaban expuestos a un riesgo de seguridad significativo o la información confidencial de ese proceso debían estar de acuerdo con los requisitos del proceso SDL.

Esta guía es un proceso de control de seguridad dedicado al desarrollo de software y la introducción de seguridad y privacidad en todas las fases del proceso de desarrollo. Como directiva obligatoria para toda la empresa desde el año 2004, el proceso SDL ha tenido un papel fundamental en la incorporación de seguridad y privacidad en el software y la cultura de Microsoft. El proceso SDL combina un enfoque holístico y práctico para reducir el número y la severidad de las vulnerabilidades de los productos y servicios de Microsoft, limitando de esta manera las oportunidades para que los atacantes pongan en riesgo los equipos. Esta metodología de procesos SDL es compartida libremente por Microsoft con las organizaciones de desarrollo de los clientes y la industria del software, donde se lo ha usado para desarrollar software más seguro.

(EnriqueDutra, s.f.)

Esta guía tuvo distintas evoluciones las cuales perfeccionaban esta metodología a través de distintas versiones, las cuales mencionaremos a continuación:

- 2004 – SDL 2.0 Principales características:

Era fundamentalmente una lista codificada de perfeccionamientos de las técnicas y procesos que se habían usado anteriormente durante las tareas de .NET y los movimientos de seguridad de Windows. Algunos elementos figuraban como requisitos y otros se incluyeron como recomendaciones. La versión inicial del proceso SDL garantizaba que las acciones como el modelado de amenazas, análisis estático y la revisión final de seguridad fueran obligatorias para el software de Microsoft que estuviera expuesto a un riesgo de seguridad significativo.

- 2005 – SDL 2.1 y 2.2 Principales características:

Límite de errores: El límite de errores (bug bar) es un criterio de calidad que se aplica a un proyecto completo de desarrollo de software. Se agregó al proceso SDL como una manera de establecer los criterios de calidad de la versión según qué tan críticos fueran los errores. Se define al principio de un proyecto para mejorar la comprensión de los riesgos asociados con problemas de seguridad y permite que los equipos identifiquen y corrijan errores de seguridad durante el desarrollo.

Exploración de vulnerabilidades (archivo y llamada a procedimiento remoto (RPC)): Se empezó a aplicar la introducción deliberada de datos aleatorios o no válidos en un programa (lo que se denomina “exploración de vulnerabilidades”) para ayudar a detectar errores de aserción, pérdidas de memoria y bloqueos.

Normas criptográficas. Se implantó una serie de requisitos en torno al uso de criptografía en las aplicaciones de Microsoft para exigir que los equipos de productos usen normas criptográficas establecidas en lugar de intentar desarrollar sus propios algoritmos no estándar.

Comprobación en tiempo de ejecución. Además de la exploración de vulnerabilidades, el proceso SDL exigía pruebas adicionales de los programas en tiempo de ejecución.

- 2006 – SDL 3.0 y 3.1 Principales características:

Exploración de vulnerabilidades (ActiveX). La exploración de vulnerabilidades se amplió a los controles ActiveX incluidos en las aplicaciones de Microsoft. Los controles ActiveX se pueden usar como una forma de inyectar datos no confiables desde sitios web y otros lugares en un equipo host. Una saturación del búfer en un control ActiveX por script puede llevar a la ejecución de código desde la zona de Internet en el contexto de un usuario registrado.

Interfaces de programación de aplicaciones (API) vedadas. La biblioteca en tiempo de ejecución C se había creado hacia más de 25 años (respecto a esta versión), cuando la conectividad de red y el entorno de amenazas era mucho menos problemático. Microsoft decidió dejar de utilizar un subconjunto de la biblioteca en tiempo de ejecución C para eliminar funciones que, según se detectó, presentaban amenazas de seguridad, especialmente saturaciones del búfer.

Normas de privacidad para el desarrollo. Microsoft estableció amplias directivas internas para los desarrolladores que se concentraban en la protección de la privacidad de clientes y usuarios.

Requisitos de servicios en línea. Antes de SDL 3.1, había dos conjuntos diferenciados de requisitos de seguridad: el proceso SDL para el desarrollo de aplicaciones cliente/servidor (aplicado a los equipos de productos tradicionales) y un conjunto diferente de requisitos de seguridad que se aplicaban al desarrollo de MSN y otros servicios en línea. El requisito de seguridad en los servicios en línea se agregó para unificar los elementos de los dos procesos en un proceso SDL coherente.

- 2007 – SDL 3.2 Principales características:

Defensas de scripts de sitios. Los procedimientos obligatorios del proceso SDL para mitigar o prevenir el uso de ataques de scripts de sitios (XSS), entre ellos validación de entrada, codificación de salida y exploración de vulnerabilidades de caja negra.

Defensas contra ataques por inyección de código SQL. Microsoft incorporó orientación específica de SDL en los ataques por inyección de código SQL, lo que incluye el uso de procedimientos almacenados y consultas con parámetros, además de exploraciones de seguridad de caja negra.

Defensas de análisis de XML. Se agregó un requisito para enfrentar los ataques de análisis de XML. Debido a que un servicio web por lo general intenta analizar cualquier XML válido que

se le transfiere, es fundamental hacer una exploración de vulnerabilidades de todas las interfaces para garantizar la validación de entrada correcta.

Primer lanzamiento público del proceso Microsoft SDL. En respuesta a consultas de los clientes, usuarios y otros interesados, Microsoft publicó el proceso SDL para aumentar la transparencia y permitir una mejor comprensión de los procesos y tecnologías usados para proteger el software de Microsoft.

- 2008 – SDL 4.0 y 4.1 Principales características:

Selección aleatoria del diseño de espacio de direcciones (ASLR). La selección aleatoria del diseño de espacio de direcciones, que era una recomendación, pasó a ser un requisito. Este requisito especificaba que se habilitara ASLR en todos los archivos binarios de código nativo (C/C++) para protegerse contra los ataques de volver a biblioteca libc (return-to-libc).

Defensas contra falsificación de solicitudes entre sitios (CSRF). Se agregó un requisito (ViewStateUserKey) para garantizar que se usen tokens únicos por sesión aleatorios para prevenir los ataques de CSRF contra las aplicaciones de web.

- 2009 – SDL 5.0 Principales características:

Exploración de vulnerabilidades (red). Requisitos ampliados de exploración de vulnerabilidades para todos los analizadores e interfaces de red. Aumento de los umbrales de aceptabilidad para las exploraciones de vulnerabilidades.

Revisiones de seguridad operativa. Todas las aplicaciones destinadas a ejecutarse en centros de datos de Microsoft deben pasar por una revisión de seguridad operativa adicional antes de implantarse, además de cumplir todos los criterios de desarrollo de seguridad del proceso SDL.

Requisitos de seguridad para licencias de terceros. Se ampliaron los procedimientos existentes de desarrollo de SDL y los criterios de servicios de seguridad a todo el código de terceros con licencia para usarse en productos y servicios de Microsoft.

- 2010 – SDL 5.1 Principales características:

Compatibilidad del "código muestra" con SDL. El código muestra se usa como plantilla para varios proyectos de desarrollo. Los errores de seguridad en el código muestran, a menudo significan errores de seguridad en un código desarrollado por terceros que aprovecha el código muestra.

Lanzamiento público de la implementación simplificada de Microsoft SDL se basa en conceptos de seguridad comprobados y adaptados para satisfacer las necesidades comerciales y técnicas específicas de Microsoft.

En este ciclo de vida de desarrollo de software seguro que presenta Microsoft, también existen técnicas que se pueden usar para contrarrestar las técnicas de explotación, las cuales son aplicar nuevos invariables que invaliden las suposiciones implícitas de una o más técnicas de explotación. Esta idea simple se ha plasmado en forma de varias tecnologías de mitigación, como las mencionadas a continuación (Microsoft, DefensaSeguridadSoftware, s.f.):

- Prevención de ejecución de datos (DEP): Una de las suposiciones que las vulnerabilidades de seguridad hacen a menudo es que los datos se pueden ejecutar como un código. El origen de esta suposición deriva del procedimiento común en que las vulnerabilidades de seguridad inyectan un código máquina personalizado (que a menudo se denomina código shell) y posteriormente lo ejecutan, el proceso conocido como ejecución de código arbitrario. En la mayoría de los casos, las vulnerabilidades de seguridad guardan ese código máquina personalizado en partes de la memoria de un programa, como la pila o el montón, que tradicionalmente están destinados a contener únicamente datos.

  Cuando se habilita DEP, ya no es posible que una vulnerabilidad de seguridad inyecte y ejecute directamente un código máquina personalizado desde regiones de memoria que están destinadas para datos.

- Protección contra sobreescritura del controlador de excepciones estructurado (SEHOP): Determinados tipos de vulnerabilidades pueden permitir que un atacante use una técnica de explotación conocida como sobreescritura del controlador de excepciones estructurado (sobreescritura de SEH). Esta técnica involucra dañar una estructura de datos que se usa cuando se administran condiciones excepcionales que se presentan mientras se ejecuta un programa. El daño a esta estructura de datos puede permitir que el atacante ejecute código desde cualquier parte de la memoria. Esta técnica se mitiga mediante SEHOP, que comprueba que la integridad de las estructuras de datos que se usan para administrar excepciones esté intacta. Este nuevo invariable hace que sea posible detectar el daño ocurrido cuando una vulnerabilidad de seguridad usa la técnica de sobreescritura de SEH y es en definitiva lo que hace posible contrarrestar las vulnerabilidades de seguridad que la usan. SEHOP es una tecnología de mitigación y requisito en SDL.

- Adición de diversidad artificial: La existencia de diversidad dentro de una población ayuda a minimizar el número de suposiciones universales que se pueden hacer acerca de los miembros de la población. Una analogía adecuada para esto se da a menudo en términos de biodiversidad: si surge un nuevo virus, la presencia de biodiversidad puede ayudar a garantizar que no toda la población se vea afectada. Este principio también se aplica en el mundo digital, donde los atacantes a menudo suponen que la configuración de un equipo será idéntica a la otro. La introducción de diversidad artificial en los equipos puede invalidar estas suposiciones y, por lo tanto, evitar que un atacante explote una vulnerabilidad de manera confiable. Un buen ejemplo de la adición de diversidad artificial se puede observar en el contexto de una mitigación de vulnerabilidades de seguridad conocida como selección aleatoria del diseño de espacio de direcciones (ASLR).

- Selección aleatoria del diseño de espacio de direcciones (ASLR): Los atacantes a menudo dan por sentado que determinados objetos (como los archivos DLL) estarán ubicados en la

misma dirección en la memoria cada vez que se ejecuta un programa (y en cada equipo en el que se ejecuta el programa). Las suposiciones de este tipo son convenientes para un atacante porque a menudo se requieren fundamentalmente para que la vulnerabilidad de seguridad tenga éxito. La imposibilidad de que un atacante codifique estas direcciones de forma rígida puede hacer que sea difícil o imposible escribir una vulnerabilidad de seguridad confiable que funcione en todos los equipos. Esta idea es la que impulsa la motivación por la ASLR. La ASLR puede contrarrestar varias técnicas de explotación introduciendo diversidad en el diseño de espacio de direcciones de un programa. En otras palabras, la ASLR realiza una selección aleatoria de la ubicación de los objetos en la memoria para evitar que un atacante pueda hacer suposiciones sobre su ubicación de manera confiable. Esta táctica tiene el efecto de hacer que el diseño de espacio de direcciones de un programa sea distinto en todos los equipos y es lo que en definitiva evita que un atacante haga suposiciones universales acerca de la ubicación de los objetos en la memoria.

- Aprovechamiento de los déficits de conocimiento: En algunas situaciones, las técnicas de explotación se pueden contrarrestar aprovechando secretos que el atacante desconoce o no puede predecir con facilidad. Una analogía simple para esta táctica puede ser una puerta con una cerradura con combinación. El enorme número de combinaciones posibles evita que el atacante pueda abrir sin dificultad la puerta, simplemente porque es poco viable que el atacante pueda adivinar la combinación a tiempo. El mismo concepto se aplica también para contrarrestar las técnicas de explotación. El uso de esta táctica en la práctica se demuestra más claramente con el soporte de la seguridad de generación de código (GS) que existe en el compilador de Microsoft Visual C++.

- GS: El ejemplo más conocido de una vulnerabilidad de software es la saturación del búfer basado en pilas. Este tipo de vulnerabilidad se explota tradicionalmente mediante la sobreescritura de datos críticos que se usa para ejecutar un código después de que se ha completado una función. Desde el lanzamiento de Visual Studio 2002, el compilador de Microsoft Visual C++ ha incluido soporte para el modificador del compilador /GS que, cuando está habilitado, introduce una comprobación de seguridad adicional que está diseñada para mitigar esta técnica de explotación. Esta táctica funciona colocando un valor aleatorio (conocido como cookie) antes de los datos críticos que un atacante podría querer sobrescribir. Luego se comprueba esta cookie cuando la función se completa para asegurarse de que sea igual al valor esperado. Si hay un error de coincidencia, se supone que se ha producido un daño y el programa puede terminar de manera segura. Este concepto simple demuestra cómo un valor secreto (en este caso la cookie) se puede usar para contrarrestar determinadas técnicas de explotación detectando daños en puntos críticos del programa. El hecho de que el valor sea secreto presenta un déficit de conocimiento que al atacante le resulta difícil resolver.

Como se puede apreciar por sus distintas versiones, Microsoft ha estado usando el proceso SDL para inyectar procedimientos comprobados de seguridad en el desarrollo de nuestro software, permitiendo una mejora continua que permite que el mismo sea una metodología de seguridad para reducir vulnerabilidades e introducir mitigaciones de las amenazas contra el software.

Por otra parte, también haremos alusión a los modelados de amenazas para la identificación de estas. Por lo cual definimos que estas son técnicas formales, estructuradas y repetibles que permiten determinar y ponderar los riesgos y amenazas a los que estará expuestas una aplicación. En pocas palabras es una forma de visualizar el software enfatizando las amenazas que pueden sufrir sus componentes.

Entonces definiremos el riesgo como la posibilidad de daño y sus posibles consecuencias. Por lo cual, para realizar el modelado de amenazas, primero realizaremos el análisis de riesgo. A este tenemos que entenderlo en termino de impacto en el negocio. Y para gestionar este tendremos que:

- Comprender los objetivos del negocio.
- Identificar riesgos del negocio y riesgos técnicos.
- Priorizar riesgos.
- Definir mínimamente alguna mitigación de los riesgos.
- Realizar correcciones y validaciones en un circuito cíclico.

A su vez también tenemos que conocer cuál es la superficie de ataque, al que definiremos como el entorno definido por los puntos de entrada y salida de una aplicación. Estos delimitaran la superficie de ataque de esta aclarando que la reducción de la superficie de ataque está estrechamente relacionada con los modelos de riesgos. Y al reducir el riesgo dado a los atacantes menos oportunidades para aprovechar una vulnerabilidad existirá. Para ello podemos implementar los árboles de amenazas que son un método grafico utilizado para identificar vulnerabilidades asociadas a una amenaza y determinar los caminos validos de ataque. Entonces tenemos que:

- Un árbol consiste en un nodo raíz e hijos.
- Cada hijo representa una condición necesaria para que el adversario halle una amenaza.
- Una vez completado se analiza "Botton-Up" para identificar caminos de ataques.
- Esos puntos deben ser accesibles a los usuarios autorizados.

Entonces ya teniendo el análisis de riesgos podremos continuar con el modelado, y decir que el mismo se puede observar desde distintas perspectivas y que no existe un modelo único para modelar amenazas, y estas pueden ser:

- Centrado en el activo.
- Centrado en el atacante.
- Centrado en el software.

Y las etapas del modelado son las siguientes:

- Conformar un grupo de análisis de riesgos.
- Descomponer la aplicación e identificar componentes claves.

- Determinar las amenazas a cada componente de la aplicación.
- Asignar un valor a cada amenaza.
- Decidir cómo responder a las amenazas.
- Identificar las técnicas y tecnologías necesarias para mitigar los riesgos.

Con este modelo buscaremos identificar los agentes de amenazas potenciales y para ello existen dos (2) técnicas. STRIDE y DREAD.
A continuación, describiremos cada una de estas en forma resumida, para entenderlas.

- STRIDE: Es una forma de garantizar que las aplicaciones sean seguras. Sus siglas significan (Spoofing – Tampering – Repudiation - Information Disclosure - Denial of Service - Elevation of Privilege) y para aplicar esta técnica tenemos que:

1-Descomponer el sistema con DFDs (Data Flow Diagrams) o UML (languaje model unifiest).

2-Actualizar el diagrama cuando el sistema cambia.

3-Analizar los procesos y subprocesos.

4-Analizar cada amenaza sobre los procesos.

5-Buscar patrones de ataque.

6-Mitigar las amenazas.

Y con su mitigación se pretende:

- Proteger a los usuarios
- Identificar cada proceso
- Identificar cada amenaza
- Aplicar estándares de la industria

- DREAD: Es un esquema de clasificación para cuantificar, comparar y dar prioridad a los riesgos que presenta una amenaza especifica. Esta se utiliza para clasificar y ordenar las amenazas en base a su riesgo. Sus siglas significan lo siguiente:

Damage potential: Que tan alto es el daño si la vulnerabilidad es explotada.

Reproducibility: Que tan fácil se puede reproducir el ataque.

Explotability: Que tan fácil es lanzar el ataque.

Affected User: Cuantos usuarios(porcentual) son afectados.

Discoverability: Que tan fácil es encontrar la vulnerabilidad de terceros.

Y a su vez su calificación puede ser Alto, Medio y Bajo.
Con esta metodología podremos disponer de una métrica realizando el siguiente cálculo:

Riego = (D + R + E + A + D) / 5

Respecto a ambas metodologías mencionadas anteriormente, STRIDE es un sistema de clasificación centrado en el atacante y ayuda a identificar amenazas en los componentes de un sistema. En cambio, DREAD es un modelo que facilita dar valor y puntuación y nos ayuda a ponderar las amenazas identificadas y los riesgos. Este último es centrado en el defensor.

Es así que el desarrollo de un marco de auditoría específico, la implementación de herramientas automatizadas de análisis de vulnerabilidades, adopción de mejores prácticas y una evaluación constante que contemple las características y riesgos específicos de cada uno e incluya listas de verificación detalladas para evaluar la seguridad del código y los sistemas, permite asegurar que los sistemas sean más seguros y resistentes a amenazas y puedan identificar vulnerabilidades en etapas tempranas del ciclo de desarrollo.

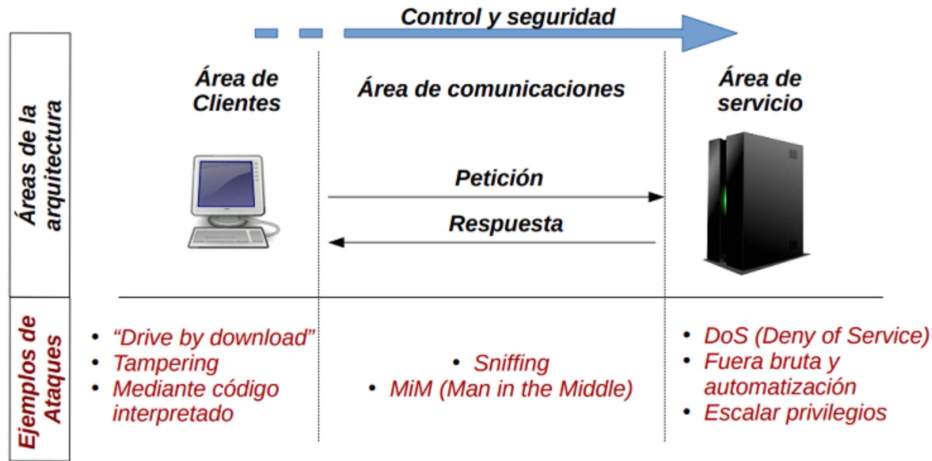
Por lo cual para un análisis preliminar tenemos que, una arista fundamental de este proceso es analizar los paradigmas de programación, el cual puede incluir la facilidad con la que pueden explotarse nuevas vulnerabilidades según al paradigma al que pertenezca el software; Y a su vez teniendo en cuenta la perspectiva de Shreyas, D. (2002) la cual señala en su documento "Software Engineering for Security: Towards Architecting Secure Software" que la seguridad en el software es un concepto multidimensional, y cuyas dimensiones están conformado por los procesos de autenticación, control de acceso, confidencialidad, integridad, disponibilidad registro cronológico para auditorías y no repudio. (Shreyas, 2002)

Y, por consiguiente, teniendo en cuenta que nuestro trabajo aplica sobre sistemas web, nuestra auditoria se aplicara en relación con el código fuente del sistema y sus posibles tipos de ciberataques que puede recibir.

Hipótesis: La arquitectura Cliente-Servidor utilizadas en sistemas web permite explotar vulnerabilidades típicas que pueden ser mitigadas con la implementación de una guía de desarrollo seguro.

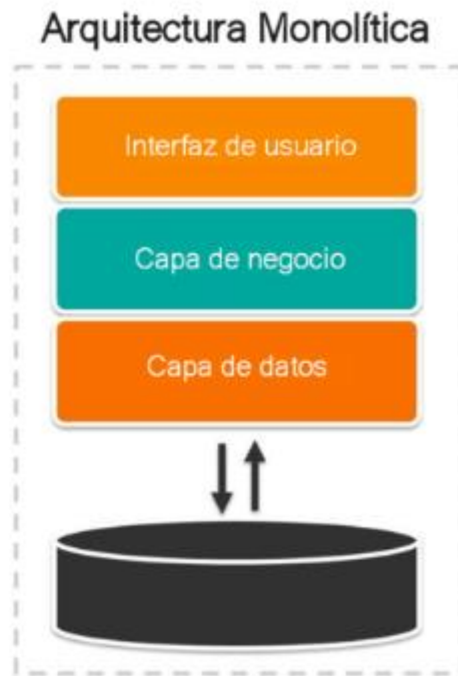
Los sistemas web por lo general son implementados con la arquitectura cliente-servidor, pudiendo ser estas de 2 capas o 3 capas, la cuales cuentan a grandes rasgos con 3 áreas específicas; Área de clientes, área de comunicaciones y el área de servicios.

A su vez cada área presenta distintas superficies para explotación de vulnerabilidades, las cuales permiten diferentes tipos de ciberataques como muestra la imagen a continuación:



Fuente propia

Por otra parte, también se aclara que, dentro de la arquitectura presentada para nuestra guía de mitigación, nos enfocaremos las que pertenecen a una "estructura monolítica", en la que se accede con una lógica secuencial como muestra la siguiente imagen:



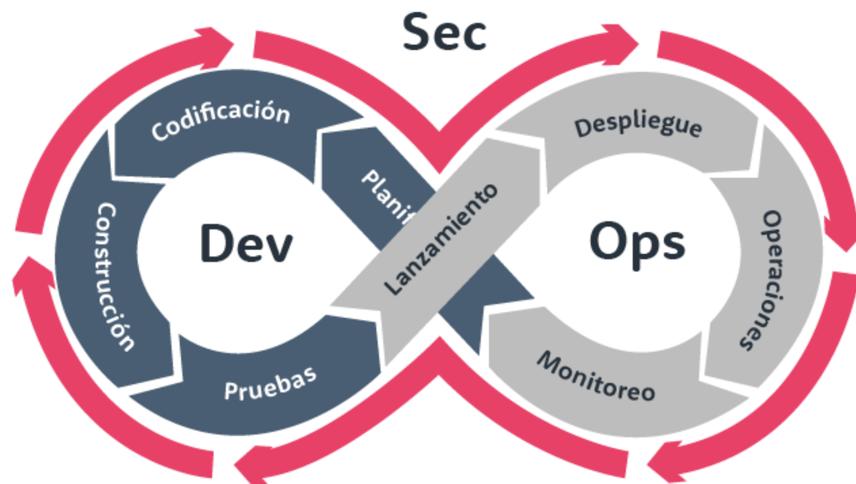
(EstructuraMonolitica, s.f.)

Entonces haciendo referencia que nuestra guía se basara en la mitigación de vulnerabilidades en el área de clientes, disponemos de una aproximación de cuál es nuestra superficie expuesta y cuales las vulnerabilidades a mitigar; Siendo las de código interpretado en las que nos enfocaremos. Teniendo en cuenta que entre las de código interpretado se encuentra el ciberataque de tipo Cross-Site-Scripting (XSS), que es causado por errores de programación, se indicaran las posibles pruebas a realizar y herramientas automatizadas que se pueden utilizar para detectar este tipo de vulnerabilidad y eliminarlas antes que el mismo se encuentre en producción.

En conclusión, la Seguridad de la Información en el contexto del desarrollo de software representa un desafío multifacético y requiere de un enfoque direccionado entre las cuales realizar auditorías especializadas para cada paradigma permite abordar estos desafíos de manera efectiva, mejorando la seguridad global del software.

Desarrollo del tema

Un aspecto fundamental para el ámbito del desarrollo del software es que las organizaciones que desarrollan software, migren de los ciclos de vida de software tradicionales y aborden S-SDCL (Secure Software Development Life Cycle) junto a las practicas DevSecOps (Developer Security Operation) las cuales integran las prácticas de seguridad y protección en cada fase del ciclo de vida del desarrollo de software, desde el diseño inicial hasta la integración, las pruebas, la entrega y el despliegue del software.



(https://sentrio.io, s.f.)

En las practicas del S-SDCL el software debe ser planeado, diseñado, construido e implementado de acuerdo con principios de ingenierías que deben ser respetados durante todo el proceso y se deben incluir métricas, modelado, métodos y técnicas de desarrollo y seguimiento de progresos. Con esta metodología el software debe estar basado en principios de seguridad y estándares reconocidos teniendo en cuenta los siguientes aspectos:

- Privacidad de la información.
- Enfoque de riesgos, cumplimiento y QA.
- Actividades destinadas a garantizar el cumplimiento de las normas.
- Medidas relacionadas con la seguridad en los procedimientos de desarrollo de las aplicaciones.
- Pruebas automatizadas.
- Pruebas de regresión.
- Pruebas de integración.
- Pruebas de rendimiento.
- Auditorías internas y externas de amenazas y vulnerabilidades.

Una vez abordada la cultura DevSecOps, y con la finalidad de securizar las estructuras monolíticas, podemos indicar que el manejo de errores se puede realizar en algunas de estas etapas o en todas:

1) En la aplicación web

2) Controlar en el web Server

3) Controlar en el WAF (web aplication firewall)

En nuestro caso para la guía a desarrollar nos basaremos dentro de la aplicación web para lo cual es necesario identificar expuestos de seguridad, aclarando que no existe una metodología exacta, pero si muchas guías que pueden orientarnos, las cuales coinciden en su mayoría e indican que las principales técnicas son las siguientes:

1) Análisis Manuales.

2) Herramientas automatizadas.

3) Técnicas de Pentesting.

Una vez definidas las técnicas a emplear y para tener un dimensionamiento de la seguridad en las aplicaciones es necesario comenzar desde el principio para poder llevar a cabo un análisis y desarrollo más seguro. Para lo cual describiremos que tener en cuenta en cada fase del desarrollo del software:

1) Análisis:

Es recomendable disponer desde el principio definiciones sobre cuestiones técnicas que ayudaran a definir la forma de desarrollo a realizar, además de los propósitos primarios de la aplicación para lo cual se está desarrollando.

En la fase inicial es el momento más apropiado para definir los requisitos de fiabilidad de un proyecto de software, analizando cuáles son los posibles vectores de ataque y la necesidad de considerar la seguridad y la privacidad desde el primer instante, lo cual es un aspecto fundamental para el desarrollo de un sistema seguro.

También es importante aclarar que la mitigación de los problemas de seguridad y de privacidad es mucho menos costosa si se planifica y realiza desde las etapas iniciales del ciclo de vida del proyecto, mediante la aplicación de un control de seguridad (NIST, s.f.).

En esta fase inicial podemos planificar y/o tener en cuenta los siguientes aspectos:

- Umbrales de calidad y límites de errores: se usan umbrales de calidad y límites de errores para establecer niveles mínimos aceptables de calidad en materia de seguridad y privacidad. Al definir estos criterios al comienzo de un proyecto, se comprenderán mejor los riesgos asociados a los problemas de seguridad durante el desarrollo.

- Definiciones sobre cuestiones de seguridad técnica como ser:

    - Cuál será la forma de autenticación. (desarrollada o implementar alguna tecnología externa).
    - Como serán las validaciones por realizar dentro de cada entrada y salidas (input - output).
    - Como persistirá el usuario dentro de la aplicación.

- Evaluaciones de los riesgos de seguridad y de privacidad son procesos obligatorios que identifican los aspectos funcionales del software que requieren una revisión exhaustiva. Dichas evaluaciones pueden incluir:

    - Parte del proyecto que van a requerir modelos de riesgos antes de su puesta a producción.
    - Partes del proyecto que van a requerir revisiones del diseño de seguridad antes del lanzamiento.
    - Partes del proyecto que van a requerir pruebas de intrusión por parte de un grupo externo al equipo del proyecto y acordado mutuamente.
    - Requisitos de análisis o de pruebas adicionales que se consideran necesarias para mitigar los riesgos de seguridad.
    - Ámbitos específicos para pruebas de exploración de vulnerabilidades mediante datos aleatorios.
    - Analizar cuál es el nivel de impacto sobre la privacidad.

Otra cuestión, también importante en esta fase tener en cuenta los patrones de diseños (relacionado a los lenguajes de programación) a utilizar, debido a que un patrón de diseño es el encargado de eliminar problemas que se presentaban con frecuencia en situaciones particulares del diseño con el fin de proponer soluciones a estos. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

2) Diseño:

Es importante disponer de un equipo de diseño que tengan instrucción o capacitación en seguridad del software, para que su diseño no solo se encuentre basado en la máxima optimización de la arquitectura y cómo van a fluir los datos dentro de esta, sino que también apliquen seguridad en la misma desde la concepción del mismo diseño. Esto para optimizar las funcionalidades que tienen que disponer el software sin dejar de lado la seguridad ya que esta es directamente proporcional sobre las funcionalidades.

La seguridad en el diseño plantea y tiene en cuenta los siguientes aspectos:

- Diseño y arquitectura segura: la seguridad es considerada en el diseño y se piensa en mitigar las amenazas (enfoque proactivo).
- Modelamiento de amenazas: se crean los modelos de amenazas.
- Eliminación de vulnerabilidades de seguridad: se analiza el diseño para hallar protocolos antiguos y se los remplaza por estándares actuales de la industria.

En esta fase del desarrollo, se pueden implementar las técnicas relacionadas al análisis de riesgos aplicando su correspondiente metodología. Este análisis nos permitirá identificar el vector de ataque y su posible riesgo, ofreciéndonos un dimensionamiento de la seguridad que tendremos que aplicar para reducir la superficie expuesta respecto a las vulnerabilidades a mitigar, en este caso el “Cross Site Scripting”.

Estas técnicas ayudaran a determinar cuál es la superficie de ataque expuesta que tenemos, la cual nos brindara una serie de ítems a controlar para reducirlas. Entre ellas hay que tener en cuenta que:

- El entorno definido por los puntos de entrada y salidas(respuestas) de una aplicación definen su superficie de ataque.
- Esos puntos deben ser accesibles solo a los usuarios autorizados.
- La reducción de la superficie de ataque está estrechamente relacionada con los modelos de riesgos.
- Se reduce el riesgo dando a los atacantes menos oportunidades para aprovechar una vulnerabilidad.

- Se cierra o se restringe el acceso a los servicios del sistema, se aplica el principio de mínimo privilegio y se utiliza defensas por capas.

3) Desarrollo:

La parte de desarrollo del proyecto es donde mayor énfasis hay que hacer, ya que se trata específicamente de la codificación del software, donde se producen los errores que permiten generar las vulnerabilidades que estamos tratando de evitar. Y por tal motivo es adecuado que los desarrolladores cuenten con conocimientos de "Desarrollo Seguro", a fin de evitar errores comunes en la codificación.

Mas allá de los conocimientos técnicos del desarrollador para generar el código fuente de la aplicación, y así también prevenir posibles errores en el código, siempre existe la posibilidad que estos pasen inadvertidos y no sean detectados, incluso hasta que se encuentre el software en producción. Y es por ello por lo que también existen técnicas para prevenir estas situaciones. Desde el punto de vista del desarrollador el mismo puede seguir estas reglas, que lo guíen:

- Todas las validaciones se deben realizar en ambos lados: cliente y servidor.
- Si los datos no cruzan un límite, es menor el cuidado a tener.
- Si el código necesita acceso a algo que ya tenía previamente, también es menor el cuidado a tener (ley inmutable).
- Si se ejecuta código con privilegios elevados, hay que tener precaución y cuidado.
- Si el código invalida las suposiciones hechas por otras entidades, hay que tener precaución y cuidado.
- Si su código está en la red, hay que tener precaución y cuidado.
- Si el código recupera información de internet, hay que tener precaución y cuidado.
- Si se tratan datos provenientes de un archivo, hay que tener precaución y cuidado.
- En código fuente, precaución con olvidarse comentarios que no correspondan.

Con respecto a codificaciones erróneas:

- Cada sentencia o comando puede ser escrito de varias maneras y ser interpretados de la misma.
- Si no controla o valida la codificación de las entradas, un atacante puede ejecutar código.

También se pueden utilizar herramientas que ayuden a los programadores a probar su código mientras se escriben o cuando terminaron de escribir una parte de este. Para ello el desarrollador y/o programador se puede dotar de alguna extensión y/o herramientas en su entorno de desarrollo (IDE) como ser:

- AntiXSSlibrary (Micrososft, s.f.): plugins para visual estudio 2010 premium que ayuda a detectar xss en el código.
- FxCop (Microsoft, FxCop, s.f.): parte de visual estudio 2010 premium para detectar vulnerabilidades
- Fortify (OpenText, s.f.): detectan cualquier tipo de vulnerabilidades en una amplia cantidad de lenguajes de programación.

En esta fase también se acostumbra a realizar pruebas adoptando técnicas para analizar nuestro desarrollo de manera continua. Estas pueden ser en estado de reposo o estático, o en un estado de actividad, el tipo dinámico. Las características de cada uno son las siguientes:

El Análisis Estático: Este tipo de análisis permite revisar el código del programa de forma escalable y contribuye a asegurar que se observan las directivas de codificación segura. En general, por sí solo, el análisis de código estático no puede remplazar una revisión de código manual, pero esta técnica ayuda a analizar el programa, su estructura y sus variables sin ejecutarlo y permitiendo obtener información que será válida para todas las posibles ejecuciones. Es importante aclarar que la misma se puede ampliar con una revisión humana.

El análisis Dinámico: Este análisis es necesario para recolectar información del programa conforme se está ejecutando (real-time y preciso). Es útil para eliminar componentes innecesarios, incompatibilidad con otros software y errores de entrada/salida. Esta comprobación ayudara a supervisar el comportamiento del programa ayudando a detectar problemas existentes como por ejemplo cuestiones de seguridad critica y/o cuestiones de privilegios de usuarios por nombrar solo algunas. En este tipo de análisis también podemos realizar pruebas de exploración de vulnerabilidades mediante datos aleatorios, para lograr los niveles de cobertura deseados de la prueba de seguridad.

Algunas herramientas sugeridas para estos tipos de análisis son:

- La herramienta VeraCode (Veracode, s.f.) puede ayudarnos en el análisis estático.
- Zap Proxy (zaproxy, s.f.): Herramienta de análisis dinámico para integrar al proyecto.
- Burp Suite (Suite, s.f.): Herramienta que sirve para realizar análisis dinámico.
- Escáner de vulnerabilidades web Acunetic (Acunetix, s.f.) el cual me ayudara a encontrar vulnerabilidades en el código fuente.

- La herramienta AppVerifer (Microsoft, AppVerifier, s.f.) puede ayudarnos en el análisis dinámico.

- Dependency-Check (OWASP P.-d.-c. , s.f.) Analizará la composición de software para detectar vulnerabilidades divulgadas públicamente contenidas en las dependencias del proyecto. Si encuentra alguna vulnerabilidad, generará el informe que vinculará las entradas CVE asociadas.

4) Preproducción y QA:

Para comenzar tenemos que dejar en claro que las personas involucradas en hacer testing de aplicaciones, no realizan actividades de ciberseguridad relacionadas al software. Estas se ocupan de verificar que el software realice las funcionalidades que tiene que cumplir el mismo y que asimismo se respeten las características de calidad que tiene que disponer el software según los estándares de la industria. Pero si pueden ayudar a securizar las aplicaciones verificando cuestiones relacionadas a las funcionalidades que pueden aportar medidas preventivas en la seguridad del software.

Ejemplo de esto puede ser la validación de datos basándose en procedimientos definidos, como los siguientes:

- Verificar que se validen las entradas de datos en el software, para que acepte solo el tipo dato para el que está preparado y no otro tipo de dato.

- Validar las cookies de sesión, y el tiempo de vida de estas. Corroborando su inutilización, cuando el usuario cierra la sesión.

- Controlar que los mensajes de errores que devuelve el software no comprometan la seguridad, solo entregue información descriptiva y útil para el usuario o en su defecto redirija a una página del tipo 404 directamente.

Ahora disponiendo de personal especifico de la seguridad del software en esta área, el mismo puede basarse en los distintos “CheckList” que nos brinda OWASP (OWASP, OWASP Web Security Testing Guide, s.f.) para realizar un análisis simple, realizar la comprobación de errores cometidos en desarrollo ofrecida por Top25 de Mitre Att&ck (cwe-mitre, s.f.) e incluso realizar análisis interactivos con herramientas, o preparar pruebas unitarias específicas con frameworks de pruebas unitarias tales como JUnit, Nunit, Cunit que pueden ser adaptados para verificar los requerimientos de pruebas de seguridad. Esto con la finalidad de entregar un código un poco más seguro (OWASP, OWASP Web Application Penetration Checklist, s.f.).

En esta fase algunas de las principales cuestiones a controlar es el manejo de sesiones, ya que es un método ampliamente utilizado en las aplicaciones web. Esta secuencia de páginas se genera en un sitio cuando el usuario ingresa hasta que lo abandona, y en el proceso de cualquier lenguaje de programación debe responder a las siguientes consideraciones:

- ¿Existe la sesión?
- Si existe la sesión se retorna
- Si no existe la sesión se crea una nueva
- se genera un identificador único

Y este manejo de sesiones tiene como objetivo primordial asegurar que los usuarios autenticados posean una amplia y criptográfica asociación segura con sus sesiones, haciendo cumplir los controles de autorización y previniendo los ataques web como la reutilización, falsificación e intercepción de sesiones.

Entonces para que la seguridad de sesiones sea efectiva, este manejo de sesiones tiene que permitir la vinculación de información a un usuario en concreto durante el proceso de visita a un sitio permitiendo las tareas de control y supervisión de accesos con su respectiva privacidad.

Otra cuestión relacionada a las sesiones y considerada importante para su control en esta fase son las cookies. Este fragmento de información se almacena en el navegador del usuario que visita el sitio (a petición del servidor de este) y a su vez luego es recuperada por el servidor en posteriores visitas para que se pueda conservar información entre una página y otra ya que el protocolo HTTP es incapaz de mantener información por sí mismo. Estas cookies existen de varios tipos, pero nos enfocaremos en las cookies de sesión.

Entonces cuando una cookie se usa para autenticación normalmente se hace mediante la utilización de sesiones. En la cookie se almacena el identificador de una sesión que es asociado al usuario que accede a la aplicación. Y por ello podemos afirmar que para que una cookie de sesión se considere segura debe cumplir una serie de condiciones, entre ellas:

- La única información que debe contener es el identificador de la sesión asociada. El resto de las variables se almacenan internamente en el array que se encuentra en el servidor.

- El identificador de sesión debe ser único, aleatorio y no predecible. Con el fin de evitar suplantaciones de identidad.

- Las variables almacenadas en la sesión, deben permanecer lo más protegidas posible del exterior. El usuario no debe conocer su nombre ni su valor, y tampoco puede modificarlas a voluntad.

- Cuando el usuario ha terminado su actividad o ha transcurrido un periodo de tiempo prudencial, la sesión debe eliminarse.

Por lo tanto, se sugiere para el manejo de sesiones durante el envió de información en las entradas y salidas de los formularios existentes en el sitio, que los mismos al ser enviados de vuelta al usuario solo sea posible este proceso cuando:

• Existan controles de integridad para prevenir la manipulación.

• Cuando los datos son validados luego de cada envío del formulario, o al menos al final del proceso de envío.

También mencionaremos que en esta etapa los tipos de testing relacionados a la seguridad del software que se pueden realizar son los siguientes:

- Testing White box: En este caso se requiere acceso al código fuente y comprender los objetivos del software y del negocio. Se realiza con base en el conocimiento de cómo se implementa el sistema, e incluyen el análisis de flujo de datos, prácticas de decodificación, el control de errores y las excepciones.

  En este tipo de análisis se prueban comportamiento intencional y el no intencional. Y su objetivo es para validar si la aplicación sigue el diseño previsto, y así mismo validar las funcionalidades implementadas y descubrir vulnerabilidades.

- Testing Black box: Se basa en obtener las especificaciones del software y comprender su funcionamiento, sin referencia ni conocimiento interno. Se lo puede utilizar para encontrar vulnerabilidades, y puede ser útil para detectar errores en el run-time y en el código compilado.

La herramienta que se propone para la fase de testing es Wapiti (Wapiti, s.f.), ya que permite auditar la seguridad de los sitios y aplicaciones web realizando escaneos de "caja negra" buscando scripts y formularios donde se pueda inyectar datos.

Otra herramienta propuesta puede ser SonarQube (SonarQube, s.f.), ya que es una herramienta Open Source para analizar la calidad del software, pero puede orientarse para seguridad.

En esta fase también se revisarán los modelos de riesgos y la superficie de ataque ya que, en muchas ocasiones, una aplicación se desvía de manera significativa de las especificaciones funcionales y de diseño creadas durante las fases iniciales del proyecto de desarrollo de software. Por ello, es muy importante que se revisen los modelos de riesgos y la medición de la superficie de ataque de una aplicación una vez completado su código. De este modo, se asegura que se toman en consideración los cambios de diseño o de implementación realizados en el sistema y que se revisan y se mitigan los nuevos vectores de ataques creados como resultado de los cambios.

5) Implementación / Pase a producción:

Una vez q tenemos el reporte del área de desarrollo y del área de QA, se realizará una revisión de seguridad final donde se inspeccionará deliberada todas las actividades de seguridad realizadas en la aplicación de software antes de su puesta a producción.

Esta revisión no consiste en parchear, ni tampoco en realizar las actividades de seguridad que se han omitido o se han olvidado durante fases previas, la misma suele consistir en verificar las existencias de funciones inseguras prohibiendo su uso en caso de haberlas, estudio de los modelos de riesgos, solicitudes de excepciones, resultados de las herramientas y el rendimiento teniendo en cuenta los umbrales de calidad y los límites de errores previamente determinados.

En esta etapa podemos afirmar que nuestro análisis de seguridad va a ser mucho más corto; Porque ya se tiene evidencia de cosas que fueron mitigadas o resueltas y nuestro pase a producción será menor ya que el set de pruebas a realizar dentro de esta etapa será mucho más chico. En esta etapa como pruebas, se sugiere realizar un análisis de escaneo automatizado para finalizar.

Algunas herramientas que se sugiere utilizar en forma independiente o en combinación de las mencionadas, según la disponibilidad de los tiempos del proyecto son:

- El escáner de servidores web Nikto (Cirt, s.f.), para realizar pruebas exhaustivas en el servidor web el cual ayudara verificar los elementos de configuración del servidor, con sus distintos complementos de escaneos.

- La herramienta de seguridad Burp Suite (Suite, s.f.) para hacer pruebas de pentesting y descubrir vulnerabilidades en aplicaciones web.

- El framework Xenotix de OWASP (OWASP-Xenotix-XSS-Exploit-Framework, s.f.) para la detección de vulnerabilidades, con el cual también se podrá realizar pruebas de pentesting sobre el sitio.

- El scanner web de Nessus (Nessus, s.f.) para ayudar en la detección de vulnerabilidades ya que el mismo es una herramienta escalable y automatizada para aplicaciones web, la cual permitirá la visibilidad de las vulnerabilidades existentes.

Para la realización del despliegue a producción se sugiere involucrar herramienta de mitigación en otra etapa como ser web aplication firewall (WAF), ya que el mismo actúa en la capa de aplicación relacionado al modelo OSI aplicando al mismo solo un “set de reglas” que se hayan preconfigurado y probadas previamente en forma local.

En esta fase podemos aplicar seguridad relacionada al despliegue de la aplicación en las cuales existen guías de implementación con prescripciones de como implementar cada función e información que permita al usuario evaluar los riesgos al activarlas, herramientas de administración que ayude a los administradores determinar y configurar los niveles óptimos de seguridad según sus necesidades y/o herramientas que provean mayor seguridad.

Otras cuestiones para tener en cuenta en esta fase es que incluso programas sin vulnerabilidades conocidas en el momento de su lanzamiento pueden estar expuestos a nuevas amenazas, por lo

cual puede ayudar de forma preventiva el incluir un plan de respuesta a incidentes, tema que se puede considerar como futura línea de investigación.

Conclusiones

Como conclusiones sobre nuestro trabajo podemos afirmar que la evolución del software seguirá incrementándose, como así también los ciberataques existentes los cuales serán más sofisticados y difíciles de evitar. Por lo cual la concientización de las organizaciones para aplicar desarrollo seguro y una cultura DevSecOps será una necesidad prioritaria para aquellos que realicen desarrollo del software. Y, por lo tanto, podemos afirmar que:

- Las empresas que desarrollan software tendrán que evolucionar a un enfoque de aplicar las buenas prácticas del desarrollo seguro durante el ciclo de vida del software.

- Una gestión adecuada de pruebas y auditoria durante el ciclo de vida del software permitirá reducir vulnerabilidades existentes en el software.

- El desarrollo de un marco de auditoría específico, la capacitación de los profesionales, la implementación de herramientas automatizadas, adopción de mejores prácticas y una evaluación constante, permite asegurar que los sistemas sean más seguros y resistentes a amenazas.

Por lo mencionado anteriormente se debe dar la importancia necesaria e implementar un modelo de seguridad en el desarrollo de software dentro de las organizaciones, ya que existe una dependencia hacia la tecnología cada vez mayor, y si bien las amenazas, en cuanto a TI, siempre van un paso más adelante que las soluciones, entonces se debe procurar que estas soluciones sean implementadas adecuadamente. Esto permitirá mejorar la seguridad técnica y promover una cultura organizacional en la cual la Seguridad de la Información, abordando la misma de forma proactiva y continúa.

Referencias

Acunetix. (s.f.). *Acunetix*. Obtenido de Acunetix: https://www.acunetix.com/

Cirt. (s.f.). *Cirt-Nikto*. Obtenido de Cirt-Nikto: https://www.cirt.net/nikto2

cwe-mitre. (s.f.). *cwe-mitre*. Obtenido de cwe-mitre: https://cwe.mitre.org/top25/

EnriqueDutra. (s.f.). *enriquedutra*. Obtenido de SDL-Microsoft: https://enriquedutra.wordpress.com/2012/01/18/microsoft-sdl-security-development-lifecycle/

EstructuraMonolitica. (s.f.). *EstructuraMonolitica*. Obtenido de https://blog.feedback-it.com/2020/10/13/microservicios-que-son-pros-contras-y-companias-que-la-utilizan/

feedback-it. (s.f.). *ArquitecturaMonolitica*. Obtenido de feedback-it: https://blog.feedback-it.com/2020/10/13/microservicios-que-son-pros-contras-y-companias-que-la-utilizan/

Forbesargentina. (2024). *ForbesArgentina*. Obtenido de ForbesArgentina: https://www.forbesargentina.com/innovacion/ciberataques-argentina-registraron-262-millones-intentos-intrusion-primer-trimestre-n53913

https://sentrio.io. (s.f.). *DevSecOps*. Obtenido de DevSecOps: https://sentrio.io/blog/que-es-devsecops-vs-devops/

ISACA. (2019). *Information Systems Audit and Control Association.*

ISO/IEC., I. 2.–T.–S.–R. (2005). *ISO/IEC 27001:2005.*

ISO12207. (2024). *ISO*. Obtenido de Normas ISO: https://normasiso.org/norma-iso-12207/

ISO27001:2022-Anexo-A8.25. (2022). *ISO*. Obtenido de https://www.iso.org: https://www.iso.org/es/search.html?PROD_isoorg_es%5Bquery%5D=27001%3A2022&PROD_isoorg_es%5Bmenu%5D%5Bfacet%5D=standard

ISO33061. (2021). *ISO*.

Kaspersky2024. (s.f.). *https://www.kaspersky.es*. Obtenido de https://www.kaspersky.es: https://www.kaspersky.es/resource-center/definitions/what-is-a-cross-site-scripting-attack

Microsoft. (2022). *Microsoft*. Obtenido de microsoft-cyberthreat-minute-2022: https://www.microsoft.com/en-us/security/security-insider/emerging-threats/cyberthreat-minute-2022

Microsoft. (s.f.). *AppVerifier*. Obtenido de AppVerifier: https://learn.microsoft.com/es-es/windows-hardware/drivers/devtest/application-verifier

Microsoft. (s.f.). *Biblioteca-anti-CrossSiteScripting*. Obtenido de Biblioteca-anti-CrossSiteScripting: https://learn.microsoft.com/en-us/previous-versions/dotnet/articles/aa973813(v=msdn.10)?redirectedfrom=MSDN

Microsoft. (s.f.). *DefensaSeguridadSoftware*. Obtenido de DefensaSeguridadSoftware: https://learn.microsoft.com/en-us/previous-versions/bb430720(v=msdn.10)?redirectedfrom=MSDN

Microsoft. (s.f.). *FxCop*. Obtenido de FxCop: https://learn.microsoft.com/en-us/previous-versions/dotnet/netframework-3.0/bb429362(v=vs.80)?redirectedfrom=MSDN

Microsoft. (s.f.). *Herramienta-IIS*. Obtenido de Herramienta-IIS: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/dd450372(v=ws.10)?redirectedfrom=MSDN

Microsoft. (s.f.). *Microsoft-SDL*. Obtenido de Microsoft-SDL: https://www.microsoft.com/en-us/securityengineering/sdl/

Microsoft. (s.f.). *Microsoft-SSDCL*. Obtenido de Microsoft-SSDCL: https://www.microsoft.com/en-us/securityengineering/sdl

Microsoft. (s.f.). *Microsoft-SSDLC*. Obtenido de Microsoft-SSDLC: https://learn.microsoft.com/es-es/compliance/assurance/assurance-microsoft-security-development-lifecycle

Microsoft. (s.f.). *SDL-Microsoft*. Obtenido de SDL-Microsoft: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/cc307891(v=msdn.10)?redirectedfrom=MSDN

Microsoft. (s.f.). *threatmodeling*. Obtenido de threatmodeling: https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

Micrososft. (s.f.). *AntiXSSLibrary*. Obtenido de AntiXSSLibrary: https://learn.microsoft.com/es-es/dotnet/api/system.web.security.antixss.antixssencoder?view=netframework-4.8.1

Mitre. (s.f.). *Attack Mitre*. Obtenido de attack mitre: https://attack.mitre.org/

Nessus. (s.f.). *https://es-la.tenable.com*. Obtenido de https://es-la.tenable.com: https://es-la.tenable.com/products/web-app-scanning

Niels, M., Dempsey, K., & Pillitteri, V. Y. (2017). *NIST Publicación especial 800-12)*. Obtenido de nist: https://csrc.nist.gov/

NIST. (s.f.). *Impacto Economico*. Obtenido de Impacto Economico: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf

Nist. (s.f.). *Nist*. Obtenido de Nist: https://www.nist.gov/

OpenText. (s.f.). *Fortify*. Obtenido de Fortify: https://www.opentext.com/es-es/productos/fortify-static-code-analyzer

Oracle. (s.f.). *Oracle-GuiaSeguridad*. Obtenido de https://docs.oracle.com/en/industries/financial-services/banking-liquidity-management/14.6.0.0.0/secgu/index.html

Oracle. (s.f.). *Oracle-OSSA*. Obtenido de Oracle-OSSA: https://www.oracle.com/corporate/security-practices/assurance/

OWASP. (2021). *owasp*. Obtenido de owasp - project-web-security-testing-guide/v42/: https://owasp.org/www-project-web-security-testing-guide/v42/

OWASP. (2024). *owasp*. Obtenido de https://owasp.org

OWASP. (s.f.). *OWASP Web Application Penetration Checklist*. Obtenido de OWASP Web Application Penetration Checklist: https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Web_Application_Penetration_Checklist_v1_1.pdf

OWASP. (s.f.). *OWASP Web Security Testing Guide*. Obtenido de OWASP Web Security Testing Guide: https://owasp.org/www-project-web-security-testing-guide/

OWASP, P.-d.-c. (s.f.). *https://owasp.org*. Obtenido de https://owasp.org/www-project-dependency-check/: https://owasp.org/www-project-dependency-check/

OWASP, P.-t. (s.f.). *owasp*. Obtenido de https://owasp.org: https://owasp.org/www-project-dependency-track/

OWASP-Clap. (s.f.). *OWASP-Clap*. Obtenido de https://wiki.owasp.org/index.php/CLASP_Concepts

OWASP-Xenotix-XSS-Exploit-Framework. (s.f.). *GitHub*. Obtenido de OWASP-Xenotix-XSS-Exploit-Framework: https://github.com/ajinabraham/OWASP-Xenotix-XSS-Exploit-Framework

Rapid7. (s.f.). *rapid7*. Obtenido de rapid7: https://www.rapid7.com/products/nexpose/

RedHat. (s.f.). *DevSecOps*. Obtenido de DevSecOps: https://www.redhat.com/es/topics/devops/what-is-devsecops

Shreyas, D. (2002). *citeseerx.ist.psu.edu*. Obtenido de citeseerx.ist.psu.: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9def6b259212de597cf91f31f2b24ada3904abe8

SonarQube. (s.f.). *SonarQube*. Obtenido de SonarQube: https://www.sonarsource.com/

Suite, B. (s.f.). *PortSwigger*. Obtenido de https://portswigger.net/burp: https://portswigger.net/burp

Top10. (2021). *owasp*. Obtenido de https://owasp.org/Top10/

Veracode. (s.f.). *Veracode*. Obtenido de Veracode: https://www.veracode.com/

Wapiti. (s.f.). *Wapiti-GitHub-Scanner*. Obtenido de Wapiti-GitHub-Scanner: https://wapiti-scanner.github.io/
Wikipedia. (s.f.). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Paradigma_de_programaci%C3%B3n
Wikipedia. (s.f.). *Wikipedia*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n
Wikipedia. (s.f.). *Wikipedia-CicloVidaDesarrolloSoftware*. Obtenido de Wikipedia-CicloVidaDesarrolloSoftware: https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software
zaproxy. (s.f.). *Zaproxy*. Obtenido de https://www.zaproxy.org/: https://www.zaproxy.org/download/

Bibliografía

"Cross Site Scripting Attacks: XSS Exploits and Defense", Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - 2007, Syngress, ISBN-10: 1-59749-154-3
Hacking Expose Web Applications. Ed. Mc-Graw-Hill/Osborne Media. Scambray, J.; Shema, M. and Sima, C. (2006).
Professional Pen Testing for Web Applications. Ed. Wrox. Andreu, A. (2006).
SQL Injection Attacks and defense. E. Syngress. Clarke, J. (2009).
The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. Ed. Wiley. Stuttard, D. (2007).
Xss Attacks: Cross Site Scripting Exploits And Defense. Ed. Syngress. Grossman, J. et al. (2007).
"La evolución de los Paradigmas de Programación", Devon M. Simmons (2012, Universidad de Carolina del Norte Wilmintong).
"The Open Web Aplication Security Project. Una guía para construir aplicaciones y servicios web seguros" Segunda Edicion. Editorial Black Hat EE. UU. 2005.
"Professional Pen Testing for Web Applications" Andreu, A**.** (2006). Ed. Wrox.
"Xss Attacks: Cross Site Scripting Exploits And Defense" **Grossman, J. et al** Ed. Syngress.
*"*Hacking Expose Web Applications*",* Scambray, J.; Shema, M. and Sima, C. Ed. Mc-Graw-Hill/Osborne Media (2006).
"The Web Application Hacker's Handbook: Discovering and Exploiting Security" Stuttard, D. (2007).
"Web Hacking", Sheila Berta, RedUser ISBN 978-987-1949-31-1.
"Ciclo de vida de desarrollo ágil de software seguro", Miguel Hernández, Bejarano Luis Eduardo, Baquero Rey ISBN relacionados 9789585478411. 9789585478442.
"Amenazados: seguridad e inseguridad en la web", Barría Huidobro Cristian - Rosales Guerrero Sergio, PRINT ISBN: 9789566086048, E - ISBN: 9789566086055, Editorial e-books Patagonia - Ediciones UM.

“Auditoría de seguridad informática”, Gómez Vieites Álvaro, PRINT ISBN: 9788492650743, E - ISBN: 978849964328, Editorial: RA-MA Editorial.
“Auditoría de seguridad informática” IFCT0109 (2a. ed.), Chicano Tejada Ester, E-ISBN:9788411035286, Editorial: IC Editorial.
“Normativa de Ciberseguridad”, Gómez Hervás Nuria del C., E - ISBN:9788418971389, Editorial RA-MA Editorial.
“Manual de un CISO”, Cano M. Jeimy J, PRINT ISBN: 9789587625820, E - ISBN: 9789587625837, Editorial: Ediciones de la U.
“Puesta en producción segura”, Fernández Riera Máximo, PRINT ISBN: 9788418971952, E - ISBN: 9788419444325, Editorial RA-MA Editorial.
“Seguridad en aplicaciones Web Java”, Ortega Candel - José Manuel, PRINT ISBN: 9788499647326, E-ISBN: 9788499647722 Editorial RA-MA Editorial.
“Pentesting con Kali Linux” Pablo González Pérez German Sánchez Garces José Miguel Soriano de la Cámara, ISBN 978-84-616-7738-2 E: 0xWORD COMPUTING S.L. 2013.