1. El Problema de la Seguridad en el Software

1.1. Introducción y objetivos

Actualmente, las tecnologías de seguridad de red pueden ayudar a aliviar y mitigar los ciberataques, pero no resuelven el problema de seguridad real del software. Estas defensas no están destinadas, normalmente, a defender la capa de aplicación, que, según los últimos informes de amenazas, es donde más están aumentado los ataques.

Además, mediante ingeniería social, un atacante puede comprometer una máquina desde el interior; a través de esta podrá atacar a las demás empezando por las que dispongan de aplicaciones vulnerables. Se hace necesario, por lo tanto, disponer de un software seguro que funcione en un entorno agresivo y malicioso.

Los objetivos del presente tema son los siguientes:

* Introducir al alumno en los principales conceptos que abarca la seguridad del software en cuanto a los beneficios que produce y su importancia en la seguridad global de un sistema.
* Conocer los estándares de gestión de vulnerabilidades.
* Estudiar las propiedades de un software seguro.
* Estudiar los principios de diseño seguro para tener en cuenta en la arquitectura y el diseño de cualquier aplicación.
* Introducir una serie de estándares de seguridad aplicables a los procesos de desarrollo seguro de software.

1.2. El problema de la seguridad en el software

Hoy en día, los ataques cibernéticos son cada vez más frecuentes, organizados y costosos en el daño que infligen a las administraciones públicas, empresas privadas, redes de transporte, redes de suministro y otras infraestructuras críticas, desde la energía a las finanzas hasta el punto de poder llegar a ser una amenaza a la prosperidad, la seguridad y la estabilidad de un país.

La sociedad está cada vez más vinculada al ciberespacio. Un elemento importante del mismo lo constituyen el software o las aplicaciones que proporcionan los servicios, utilidades y funcionalidades. Sin embargo, estas aplicaciones presentan defectos, debilidades de diseño o configuraciones inseguras que originan vulnerabilidades que pueden ser explotadas por atacantes de diversa índole, desde aficionados hasta organizaciones de cibercrimen o, incluso, estados en acciones de ciberguerra, utilizándolas como plataformas de ataque al comprometer los sistemas y las redes de la organización.

Nadie quiere un software defectuoso, especialmente los desarrolladores, cuyo código incorrecto es el problema. En un informe de Klocwork (2004) se indica que las principales causas y aspectos que influyen en la aparición de vulnerabilidades son las siguientes:

* Tamaño excesivo y complejidad de las aplicaciones.
* Mezcla de código proveniente de varios orígenes, como compras a otra compañía, reutilización de otros existentes, etc., lo que puede producir comportamientos e interacciones no esperados de los componentes del software.
* Integración de los componentes del software defectuoso, lo que establece, por ejemplo, relaciones de confianza inadecuadas entre ellos.
* Debilidades y fallos en la especificación de requisitos y diseño no basados en valoraciones de riesgo y amenazas.
* No realizar pruebas de seguridad basadas en riesgo.
* Uso de entornos de ejecución con componentes que contienen vulnerabilidades o código malicioso embebido, como pueden ser capas de middleware, sistema operativo u otros componentes COTS.
* Falta de herramientas y un entorno de pruebas adecuado que simule correctamente el entorno real de ejecución.
* Cambios de requisitos del proyecto durante la etapa de codificación.
* Mezcla de equipos de desarrolladores, entre los que podemos tener equipos propios de desarrollos, asistencias técnicas, entidades subcontratadas, etc.
* Falta de conocimiento de prácticas de programación segura de los desarrolladores en el uso de lenguajes de programación.
* No controlar la cadena de suministro del software, lo que puede dar lugar a la introducción de código malicioso en origen.
* No realizar un seguimiento, por parte de los desarrolladores, de guías normalizadas de estilo en la codificación.
* Fechas límite de entrega de proyectos inamovibles.
* Cambio en la codificación en base al requerimiento de nuevas funcionalidades.
* Tolerancia a los defectos.
* No tener actualizadas las aplicaciones en producción con los parches correspondientes, configuraciones erróneas, etc.

Las aplicaciones son amenazadas y atacadas no solo en su fase de operación, sino que, también, en todas las fases de su ciclo de vida (Goertzel, K. M., 2009): desarrollo, distribución e instalación, operación, mantenimiento o sostenimiento.

En base a lo expuesto, se considera necesario que las diferentes organizaciones públicas o privadas dispongan de un software fiable y resistente a los ataques; es decir, de confianza, con un número de vulnerabilidades explotables que sea el mínimo posible. En respuesta a esto, nace la seguridad del software, que en el documento de referencia de SAFECode se define como: «La confianza que el software, hardware y servicios están libres de vulnerabilidades intencionadas o no intencionadas y que funcionan conforme a lo especificado y deseado» (SafeCode, 2010).

En este sentido, se puede definir la seguridad del softwarecomo:

El conjunto de principios de diseño y buenas prácticas a implantar en el SDLC para detectar, prevenir y corregir los defectos de seguridad en el desarrollo y adquisición de aplicaciones; de forma que se obtenga un software de confianza y robusto frente a ataques maliciosos, que realice solo las funciones para las que fue diseñado, que esté libre de vulnerabilidades, ya sean intencionalmente diseñadas o accidentalmente insertadas durante su ciclo de vida, y se asegure su integridad, disponibilidad y confidencialidad.

Para conseguir lo anterior y minimizar al máximo los ataques en la capa de aplicación y, por lo tanto, en número de vulnerabilidades explotables, es necesario incluir la seguridad, desde un principio, en el ciclo de vida de desarrollo del software (SDLC), incluyendo buenas prácticas de seguridad en todas y cada una de las fases de este, como, por ejemplo, requisitos de seguridad, casos de abuso, análisis de riesgo, análisis de código, pruebas de penetración dinámicas, etc.

Un beneficio importante que se obtendría de incluir un proceso sistemático que aborde la seguridad desde las primeras etapas del SDLC sería la reducción de los costes de corrección de errores y vulnerabilidades, pues estos son más altos conforme más tarde son detectados. Acorde a lo publicado por NIST (National Institute of Standards and Technology), el coste que tiene la corrección de código o vulnerabilidades después de la publicación de una versión es hasta treinta veces mayor que su detección y corrección en etapas tempranas del desarrollo.

A graph showing a bar graph

Description automatically generated with medium confidence

Figura 2. Coste relativo de corrección de vulnerabilidades en función de la etapa de desarrollo. Fuente: ImagineX consulting, 2018.

La empresa Microsoft, una de las primeras empresas en aplicar un modelo S-SDLC, presentó los siguientes datos al año de tener implantado el citado modelo (Caro, 2016):

* El 75 % de las vulnerabilidades están relacionadas con las aplicaciones.
* Se genera alrededor de un 40-50 % de reducción en el número de vulnerabilidades tras la implantación de un S-SDLC en el primer año (un 75-80 % sobre vulnerabilidades críticas).
* Una reducción de un 50 % en el número de vulnerabilidades implica una reducción de un 75 % en los costes de gestión de la configuración y respuesta a incidentes.

A graph of data on a white background

Description automatically generated with medium confidence

Figura 3. Datos de la empresa Microsoft. Fuente: Caro, 2016.

1.3. Vulnerabilidades y su clasificación

Se define vulnerabilidad como un fallo de programación, configuración o diseño que permite a los atacantes, de alguna manera, alterar el comportamiento normal de un programa, violar la política de seguridad del software y realizar algo malicioso, como alterar información sensible, interrumpir o destruir una aplicación o tomar su control.

Se puede decir que es un subconjunto del fenómeno más grande que constituyen los *bugs*(errores de programación) y *flaws* (errores de diseño) del software.

Sus fuentes se deben a:

* Fallos de implementación. Fallos provenientes de la codificación de los diseños del software realizados, por ejemplo: desbordamientos de búfer, formato, condiciones de carrera, *path traversal*, *cross-site scripting*, inyección SQL, etc.
* Fallos de diseño. Los sistemas hardware o software contienen, frecuentemente, fallos de diseño o debilidades *(flaws)* que pueden ser utilizados para realizar un ataque. Por ejemplo, TELNET no fue diseñado para su uso en entornos hostiles, para eso se implementó SSH.
* Fallos de configuración. La instalación de software, por defecto, implica, generalmente, la instalación de servicios que no se usan, pero que pueden presentar debilidades que comprometan la máquina.

A blue and white box with black text

Description automatically generated

Figura 4. Tipos de vulnerabilidades del software. Fuente: Elaboración propia.

Casi todos los fallos que se producen en un software provienen de fallos de implementación y diseño,pero solamente algunos resultan ser vulnerabilidades de seguridad. Un fallo debe tener algún impacto o característica relevante para ser considerado un error de seguridad; es decir, tiene que permitir a los atacantes la posibilidad de lanzar un *exploits* que les permita hacerse con el control de un sistema.

*Exploits:* es una instancia particular de un ataque a un sistema informático que aprovecha una vulnerabilidad específica o un conjunto de ellas.

Una vulnerabilidad se define (INCIBE, 2011), básicamente, por cinco factores o parámetros que deben identificarla.

* Producto: productos a los que afecta, versión o conjunto de ellas.
* Dónde: componente o módulo del programa donde se localiza la vulnerabilidad.
* Causa y consecuencia: fallo técnico concreto que cometió el programador a la hora de crear la aplicación que es el origen de la vulnerabilidad.
* Impacto: define la gravedad de la vulnerabilidad, indica lo que puede conseguir un atacante que la explotase.
* Vector: técnica del atacante para aprovechar la vulnerabilidad, se le conoce como «vector de ataque».

Gestión de vulnerabilidades

Dada la gran cantidad de vulnerabilidades descubiertas, se hace necesario disponer de estándares que permitan referenciarlas unívocamente para poder conocer su gravedad de forma objetiva y obtener el conocimiento necesario para mitigarlas. Existen varios estándares, catálogos o bases de datos que, a continuación, pasamos a comentar:

* Common Vulnerabilities and Exposures[(CVE)](http://cve.mitre.org/). Es un diccionario o catálogo público de vulnerabilidades, administrado por MITRE, que normaliza su descripción y las organiza desde diferentes tipos de vista (vulnerabilidades web, de diseño, implementación, etc.).

MITRE: organización sin ánimo de lucro de carácter público que trabaja en las áreas de ingeniería de sistemas, tecnologías de la información, concepto de operación y modernización de empresas.

Cada identificador CVE incluye:

* Identificador con el siguiente formato:

A screenshot of a phone

Description automatically generated

Figura 7. Identificador CVE. Fuente: Elaboración propia.

* Breve descripción de la vulnerabilidad.
* Referencias.
* Common Vulnerability Scoring System [(CVSS)](https://nvd.nist.gov/vuln-metrics/cvss). Es, básicamente, un sistema que escalona la severidad de una vulnerabilidad de manera estricta a través de fórmulas, proporcionando un estándar para comunicar las características y el impacto de una vulnerabilidad identificada con su código CVE. Su modelo cuantitativo asegura una medición exacta y repetible, a la vez que permite ver características subyacentes que se usaron para generar su puntuación.

También permite organizar la priorización de las actividades de remediación o parcheo de las vulnerabilidades. En la figura se muestra el proceso de cálculo de una severidad:

A diagram of a company

Description automatically generated

Figura 8. Cálculo puntuación CVSS. Fuente: Elaboración propia.

El cálculo se realiza en base a tres tipos de métricas: base, temporales y ambientales, las dos últimas son opcionales. En cuanto a las métricas base, se tienen dos subconjuntos:

* Explotabilidad: vectores de acceso, complejidad de acceso y autenticación.
* Impacto: confidencialidad, integridad y disponibilidad.

El Instituto Nacional de Estándares y Tecnología de los Estados Unidos posee una página web donde se puede realizar el cálculo de la severidad de una vulnerabilidad conforme al estándar CBSS. Accede a ella a través del siguiente enlace: <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>

* Common Weakness Enumeration[(CWE)](http://cwe.mitre.org/). Estándar internacional y de libre uso que ofrece un conjunto unificado de debilidades o defectos del software medibles, que permite un análisis, descripción, selección y uso de herramientas de auditoría de seguridad de software y servicios que pueden encontrar debilidades en el código fuente y sistemas, así como una mejor comprensión y gestión de los puntos débiles de un software relacionados con la arquitectura y el diseño. Sus principales objetivos son:
  + Proporcionar un lenguaje común para describir los defectos y debilidades de seguridad del software en su arquitectura, diseño y codificación.
  + Proporcionar un estándar de comparación de herramientas de auditoría de la seguridad del software.
  + Proporcionar una línea base para la identificación de vulnerabilidades, su mitigación y los esfuerzos de prevención.

Incluye los siguientes tipos de debilidades del software: desbordamientos del *buffer,*formato de cadenas, estructura y problemas de validación, errores de ruta, interfaz de usuario, autenticación, gestión de recursos, manipulación de datos, verificación de datos, inyección de código, etc.

Cada identificador CWE incluye los siguientes campos de información:

A screenshot of a computer error message

Description automatically generated

Figura 9. Campos de información de cada entrada CWE. Fuente: Elaboración propia.

1.4. Propiedades del software seguro

Básicamente se tienen dos conjuntos de propiedades que definen a un software seguro del que no lo es. Las primeras son las esenciales, comunes a la seguridad de cualquier sistema, cuya ausencia afecta gravemente a la seguridad de una aplicación; y, un segundo conjunto, las complementarias a las anteriores, que no influyen en su seguridad pero que ayudan a mejorarla en gran medida.

Las principales propiedades esenciales que distinguen al software de confianza del que no lo es, son:

* Integridad. Capacidad que garantiza que el código, los activos manejados, las configuraciones y el comportamiento no puedan ser, o no hayan sido, modificados o alterados por personas, entidades o procesos no autorizados, tanto durante la fase de desarrollo como en la fase de operación. Entre las modificaciones que se pueden realizar, tenemos: sobreescritura, inclusión de puertas traseras, borrado, corrupción de datos, etc.
* Disponibilidad. Capacidad que garantiza que el software es operativo y accesible por personas, entidades o procesos autorizados de forma que se pueda acceder a la información y a los recursos o servicios que la manejan conforme a las especificaciones de estos.
* Confidencialidad. Capacidad de preservar que cualquiera de sus características, activos manejados, están ocultas a usuarios no autorizados, de forma que se garantice que solo las personas, entidades o procesos autorizados pueden acceder a la información.

Un ejemplo de ataque podría ser un desbordamiento de *buffer (buffer overflow)* algo que consigue el control total de la máquina y permite violar las tres propiedades anteriores al poder robar información del sistema, cuentas de usuario, corromper ficheros del sistema e incluso apagar la máquina y borrar los ficheros necesarios para que no vuelva a arrancar.

Estas tres primeras serían las propiedades esenciales mínimas fundamentales que debería de disponer todo software, a las que habría que añadir las siguientes propiedades complementarias:

* Fiabilidad. Capacidad del software de funcionar de la forma esperada en todas las situaciones a la que estará expuesto en su entorno de funcionamiento; es decir, que la posibilidad de que un agente malicioso pueda alterar la ejecución o resultado de una manera favorable para el atacante está significativamente reducida o eliminada.
* Autenticación. Capacidad que permite a un software garantizar que una persona, entidad o proceso es quien dice ser, o bien que garantiza la fuente de la que proceden los datos.
* Trazabilidad. Capacidad que garantiza la posibilidad de imputar las acciones relacionadas en un software que la ha originado.
* Robustez. Capacidad de resistencia a los ataques realizados por los agentes maliciosos (*malware, hackers,* etc.).
* Resiliencia. Capacidad del software de aislar, contener y limitar los daños ocasionados por fallos causados por la explotación de una vulnerabilidad de este y recuperarse reanudando su operación en (o por encima de) cierto nivel mínimo predefinido de servicio aceptable en tiempo oportuno.
* Tolerancia. Capacidad del software para tolerar los errores y fallos que resultan de ataques con éxito y seguir funcionando como si los ataques no se hubieran producido.

Las propiedades que distinguen al software de confianza se ilustran en la siguiente figura:

A diagram of software components

Description automatically generated

Figura 11. Propiedades seguridad del software*.* Fuente: Elaboración propia.

Existe una serie de factores que influyen en la probabilidad de que un software sea consistente con las propiedades anteriormente mostradas (Mead y Woody, 2016).

Estos incluyen:

* Principios de diseño y buenas prácticas de desarrollo. Las prácticas utilizadas para desarrollar el software y los principios de diseño que lo rigen.
* Herramientas de desarrollo. El lenguaje de programación, bibliotecas y herramientas de desarrollo utilizadas para diseñar, implementar y probar el software y la forma en la que fueron utilizados por los desarrolladores.
* Componentes adquiridos. Tanto los componentes de software comercial como libre en cuanto a cómo fueron evaluados, seleccionados e integrados.
* Configuraciones desplegadas. Cómo el software se configuró durante la instalación en su entorno de producción.
* Ambiente de operación. La naturaleza y configuración de las protecciones proporcionadas por el entorno de ejecución o producción.
* Conocimiento profesional. El nivel de concienciación y conocimiento de seguridad que poseen los analistas, diseñadores, desarrolladores, probadores y mantenedores del software, o su falta de este.

1.5. Principios del diseño de seguridad del software

La adopción de estos principios de diseño constituye una base fundamental de las técnicas de programación segura, tanto para la protección de aplicaciones web, normalmente más expuestas a los ciberataques al estar desplegadas masivamente en Internet, como otras más tradicionales del tipo cliente-servidor.

Los principales principios de diseño para tener en cuenta en el desarrollo de toda aplicación son:

A blue and white list with black text

Description automatically generated

Tabla 1. Objetivos y principios de seguridad. Fuente: elaboración propia.

Defensa en profundidad

Uno de los principios más importantes de una estrategia defensiva efectiva es la defensa en profundidad,que se define en la guía CCN-STIC-400 (CCN, 2013) como:

«Estrategia de protección consistente en introducir múltiples capas de seguridad, que permitan reducir la probabilidad de compromiso en caso de que una de las capas falle y en el peor de los casos minimizar el impacto». (CNN, 2013).

La arquitectura del software y hardwarede base, que constituirá el entorno de ejecución donde la aplicación vaya a ser instalada, debería contar con una variedad de servicios de seguridad y protecciones que reduzcan y dificulten la probabilidad de que una acción maliciosa alcance el software, ya que:

* Se minimiza la exposición de las propias vulnerabilidades al mundo exterior.
* Se reduce la visibilidad externa de los componentes confiables principales.
* Se aíslan los componentes no confiables, de forma que su ejecución se vea limitada y sus malos comportamientos no afecten o amenacen la operación confiable de los demás.

El aislamiento significa que el software o componente no confiable dispone de recursos específicos para su ejecución, como memoria, espacio en disco duro, interfaz de red virtual, etc., en un entorno aislado. Para su implementación, se suelen utilizar máquinas virtuales que, además, pueden proporcionar otras características que ayudan a mejorar la fiabilidad, confiabilidad y resistencia, como el balanceo de carga, soporte para la restauración de imágenes, etc.

Objetivo: introducir múltiples capas de seguridad para reducir la probabilidad de compromiso del sistema.

Este principio propone un enfoque defensivo que implanta protecciones o mecanismos de seguridad en todos los niveles del sistema o capas del modelo Open Systems Interconnection (OSI). Las medidas de seguridad a implementar en cada capa podrán variar en función del entorno de operación del sistema. Sin embargo, el principio base, o general, permanece inalterable; por ejemplo, para las capas siguientes tendríamos:

A blue and white table with text

Description automatically generated

A blue and white table with text

Description automatically generated

Tabla 2. Principio de defensa en profundidad. Fuente: elaboración propia.

Otro aspecto importante es la verificación de la cadena de suministros mediante la comprobación del *hash,*código de firma, aplicados al código ejecutable mediante la validación de la integridad de esta en el momento de la entrega, la instalación o en el tiempo de ejecución, para determinar:

* Si el código se originó a partir de una fuente de confianza.
* Si la integridad del código de se ha visto comprometida.

Simplicidad del diseño

La realización de un diseño tan sencillo como sea posible, y la redacción de unas especificaciones de este fácilmente comprensibles y simples, es una forma de obtener un nivel de seguridad mayor. Esto disminuye la probabilidad de que se incluyan debilidades de diseño y errores de programación que resulten en vulnerabilidades que comprometan la seguridad de la aplicación. Algunas de las opciones específicas de diseño del software que lo simplifican son (Mead y Woody, 2016):

* Limitar el número de estados posibles en el software.
* Favorecer los procesos deterministas sobre los no deterministas.
* Se debe evitar la funcionalidad innecesaria o los mecanismos de seguridad innecesarios.
* Utilizar una sola tarea en lugar de realizar múltiples tareas, siempre que sea práctico.
* El uso de técnicas de sondeo en lugar de interrupciones.
* Diseñar los componentes del software con el conjunto mínimo de características y capacidades que se requieran para realizar sus tareas en el sistema.
* La descomposición en subsistemas o componentes de un programa debería adaptarse a su descomposición funcional, para permitir una asignación uno a uno de los segmentos de programa a sus fines previstos.
* Desacoplar los componentes y procesos para minimizar las interdependencias entre ellos impedirá que un fallo o anomalía en un componente o proceso afecte a los estados de otros.
* No implementar características o funciones innecesarias.
* Facilidad de uso. Por ejemplo, Single Sign On (SSO) es un buen ejemplo que ilustra la simplificación de la autenticación de usuario.

Como podemos ver, el implementar arquitecturas complejas cuando se puede resolver el diseño de forma simple puede perjudicar la seguridad del sistema.

Objetivo: reducir la complejidad del diseño para minimizar el número de vulnerabilidades explotables por el atacante y debilidades en el sistema.

Mínimo privilegio

De acuerdo con el Software Assurance (2012), se define el mínimo privilegio como:

«Mínimo privilegio es un principio según el cual se concede a cada entidad (usuario, proceso o dispositivo) el conjunto más restrictivo de privilegios necesarios para el desempeño de sus tareas autorizadas. La aplicación de este principio limita el daño que puede resultar de un accidente, error o el uso no autorizado de un sistema. También reduce el número de interacciones potenciales entre los procesos privilegiados o programas, por lo que se minimiza la probabilidad de ocurrencia de usos maliciosos de privilegios, no deseados o inapropiados» (Software Assurance Pocket Guide Series, 2012).

Una de las principales razones por las que es necesario que una entidad se ejecute con los mínimos privilegios posibles es debido a que, si un ciberatacante consigue comprometer una máquina, o es capaz de inyectar código malicioso en un proceso del sistema, este se debería ejecutar con los mismos privilegios que tuviera el usuario en la máquina o el proceso.

Este principio requiere que el diseñador realice una lista de las entidades de softwarecon los recursos que utiliza y las tareas que debe realizar en el sistema, especificando para cada una los privilegios realesestrictamente necesarios. Es normal cometer el error de asignar un usuario general con un conjunto de privilegios que le permitirá realizar todas las tareas, incluidas las no necesarias.

La programación modular ayuda a implementar menos privilegios, además de hacer que el código sea más legible, reutilizable y fácil de mantener. Cada unidad de código (clase, método, etc.) tiene un único propósito y las operaciones que puede realizar están limitadas solo a las que están diseñadas. Ejemplos de errores comunes son:

* Aplicación con derechos de administrador.
* Instalación de aplicaciones y servicios con el usuario de administrador.
* Usuarios con derechos de administrador.
* Servicios y procesos con privilegios por tiempo indefinido.

Objetivo: lo que no está expresamente permitido está prohibido.

Separación de privilegios

Es un principio relacionado con el anterior mínimo privilegio que implica la asignación a las diferentes entidades de un rol de las siguientes propiedades:

* Asignación de un subconjunto de funciones o tareas de todas las que ofrece el sistema.
* Acceso a los datos necesarios que debe gestionar para llevar a cabo su función en base a una serie de roles definidos.

Se evita así que todas las entidades sean capaces de acceder a la totalidad o llevar a cabo todas las funciones del sistema con privilegios de «superusuario» y, por lo tanto, que ninguna entidad tenga todos los privilegios necesarios para modificar, sobrescribir, borrar o destruir todos los componentes y recursos de la aplicación. Como ejemplo tenemos el servidor web, en donde el usuario final solo requiere de la capacidad de leer el contenido publicado e introducir datos en los formularios HTML. El administrador, por el contrario, tiene que ser capaz de leer, escribir y eliminar contenido y modificar el código de los formularios HTML.

Objetivo: asignación a las diferentes entidades de un rol que implique el acceso a un subconjunto de funciones o tareas y a los datos necesarios.

Separación de dominios

Es un principio que en unión con los dos anteriores, mínimo privilegio y separación de privilegios, permite minimizar la probabilidad de que actores maliciosos obtengan fácilmente acceso a las ubicaciones de memoria u objetos de datos del sistema. Para que el diseño cumpla con este principio, deben utilizar técnicas de compartimentación de los usuarios, procesos y datos de forma que las entidades:

* Solo deben ser capaces de realizar las tareas que son absolutamente necesarias.
* Llevarlas a cabo solamente con los datos que tengan permiso de acceso.
* Utilizar el espacio de memoria y disco que tengan asignado para la ejecución de esas funciones.

Objetivo: minimizar la probabilidad de que actores maliciosos obtengan fácilmente acceso a las ubicaciones de memoria u objetos de datos del sistema.

El aislar las entidades de confianza en su propia área de ejecución (con recursos dedicados a la misma) de otras menos confiables (procesadores de texto, software de descarga, etc.), permite reducir al mínimo su exposición a otras entidades e interfaces externas susceptibles de ser atacadas por agentes maliciosos.

Las técnicas que tenemos para llevar a cabo lo anterior son:

* Sistema operativo confiable.
* Máquinas virtuales. Además, pueden proporcionar otras características que ayudan a mejorar la fiabilidad, confiabilidad y resistencia, como el balanceo de carga, soporte para la restauración de imágenes, etc.
* Funciones *sandboxing*de lenguajes de programación como Java, Perl, .NET (Code Access Security), etc.
* Sistemas Unix: *chroot jails.*
* Trusted Processor Modules (TPM).

Separación de código, ejecutables y datos de configuración y de programa

Este principio pretende reducir la probabilidad de que un ciberatacante que haya accedido a los datos del programa fácilmente pueda localizar y acceder a los archivos ejecutables y datos de configuración de este, lo que le daría la posibilidad de manipular el funcionamiento del sistema a su interés e incluso el escalado de privilegios.

La mayoría de las técnicas de separación de los datos de programa, configuración y archivos ejecutables se realizan en la plataforma de ejecución (procesador más sistema operativo). A continuación, vemos las principales:

* Utilizar plataformas con arquitectura Harvard.
* Establecer permisos de escritura y lectura de los datos de programa y sus metadatos al programa que los creó, a menos que exista una necesidad explícita de otros programas o entidades para poder leerlos.
* Los datos de configuración del programa solo deben poder ser leídos y modificados por el administrador.
* Los datos utilizados por un *script* en un servidor web deben ser colocados fuera del árbol de documentos de este.
* Prohibir a los programas y *scripts*escribir archivos en directorios escribibles, como el de UNIX/TMP.
* Almacenar los archivos de datos, configuración y programas ejecutables en los directorios separadosdel sistema de archivos.
* Los programas y *scripts*que están configurados para ejecutarse como servidor web de usuario*nobody*(debe suprimirse) deben ser modificados para funcionar bajo un nombre de usuario específico.
* Cifrar todos los archivos ejecutables e implementar un módulo de decodificación que se ejecute como parte del inicio del programa para desencriptarlos al iniciar su funcionamiento.
* Incluir técnicas de cifrado de archivos y firma digital o almacenamiento en un servidor de datos externos con conexión cifrada (por ejemplo, mediante Secure Sockets Layer ─SSL— o Transport Layer Security —TLS—) para aislar los datos de configuración del software de la manipulación y eliminación, en caso de que las técnicas de control de acceso al sistema no sean lo suficientemente fuertes. Ello requerirá que el software incluya la lógica de cifrado para descifrar y validar la firma del archivo de configuración al inicio del programa.
* Implantar clonado de sistemas como una medida de recuperación desde un servidor remoto (que guardaría las imágenes y los datos de configuración) mediante una red de comunicaciones fuera de banda específica y cifrada.

Objetivo: reducir la probabilidad de que un ciberatacante que haya accedido a los datos del programa fácilmente pueda localizar y acceder a los archivos ejecutables y datos de configuración del programa.

Entorno de producción o ejecución inseguro

Al asumir que todos los componentes del entorno de producción y sistemas externos son inseguros o no confiables, se pretende reducir la exposición de los componentes del software a agentes potencialmente maliciosos que hayan podido penetrar en el perímetro de defensa (los límites de los dispositivos de protección perimetral) de la organización, y comprometer una máquina desde la que puedan expandirse e iniciar otros ataques a otras pertenecientes a la red *(pivoting).*

El software debe ser diseñado con una mínima dependencia de los datos externos. Se debe tener control completo sobre cualquier fuente de entrada de datos, tanto los proporcionados por la plataforma de ejecución como los de los sistemas externos, y se deberán validar todos los datos provenientes de las diferentes fuentes del entorno antes de utilizarlos, pues implican una posibilidad de ataque.

Hay que realizar una descomposición del sistema en sus componentes principales y realizar una lista de las diferentes fuentes de datos externas, entre las que podemos tener las siguientes:

* Llamadas a sistema operativo. Se deben evitar realizándolas a través de middleware o API’s.
* Llamadas a otros programas en la capa de aplicación.
* Llamadas a una capa de middleware intermedia.
* API’s a los recursos del sistema. Las aplicaciones que utilizan las API no deberían ser utilizadas por usuarios humanos.
* Referencias a objetos del sistema.
* En aplicaciones cliente-servidor, el flujo de datos entre los mismos.
* En aplicaciones web, el flujo de datos ente el cliente y el servidor.

Gran parte de los ataques a los sistemas TIC actualmente se deben afallos o carencias en la validación de los datos de entrada, el confiar en la fuente que las originó hace a la aplicación vulnerable a ataques originados en el cliente al modificar los datos en el origen o durante su transporte. Los atacantes pueden manipular diferentes tipos de datos de entrada con el propósito de encontrar vías de compromiso de la aplicación.

Todos los tipos de entrada deben ser validados y verificados mediante pruebas específicas. Según Goertzel, K. M. y Winograd, T. (2008), entre los diferentes tipos de entradas a las aplicaciones, tenemos:

* Parámetros de línea de comandos.
* Variables de entorno.
* Localizadores de recursos universales (direcciones URL) e identificadores (URI).
* Referencias a nombres de archivo.
* Subida de contenido de archivos.
* Importaciones de archivos planos.
* Cabeceras Hyper Text Transfer Protocol (HTTP).
* Parámetros HTTP GET.
* Campos de formulario (especialmente los ocultos).
* Las listas de selección, listas desplegables.
* *Cookies.*
* Comunicaciones mediante Java *applets.*

Los tipos de aplicaciones que más probabilidad presentan de sufrir este tipo de ataques son las del tipo cliente-servidor, portales web y agentes *proxy.*Para evitar este tipo de vulnerabilidades, se deben aplicar una serie de principios de validación de las entradas, entre los que tenemos (Goertzel, K. M., Winograd, T., 2008):

* Centralizar la lógica de validación de las entradas.
* Asegurar que la validación de la entrada no puede ser evitada.
* Confiar en listas blancas validadas y filtrar las listas negras.
* Validar todas las entradas de usuario, incluidas las realizadas a través de *proxies*y agentes que actúen en nombre de los usuarios.
* Rechazar todos los contenidos ejecutables en las entradas provenientes de fuentes no autorizadas.
* Verificar que los programas que solicitan las llamadas tienen derecho (por la política) a emitirlas.
* Definir reacciones significativas a errores de validación de entrada.
* Validar los datos de salida de la aplicación antes de mostrarlos al usuario o redirigirlos a otro sistema.

Objetivo: evitar vulnerabilidades aplicando una serie de principios de validación de las entradas.

Registro de eventos de seguridad

Tradicionalmente, los sistemas de auditoría se dedicaban solo a grabar las acciones realizadas por los usuarios de estos. Sin embargo, hoy en día es necesario dotar a las aplicaciones de la capacidad de generar eventos *(logs)* de seguridad para garantizar que todas las acciones realizadas por un agente malicioso se observan y registran, contribuyendo a la capacidad de reconocer patrones y aislar o bloquear la fuente del ataque de modo que se evite su éxito; en definitiva, dotarle de cierta capacidad de detección de intrusiones.

Las principales diferencias que distinguen los sistemas con registros de auditoría de seguridad de registro de los registros de eventos estándar son:

* El tipo de información capturada en el registro de auditoría: eventos de seguridad.
* Capacidad de gestión de los incidentes relacionados con los eventos de seguridad.
* Posibilidad de que los eventos de seguridad registrados en la aplicación o sistema puedan ser utilizados en procesos reactivos después de la ocurrencia de un incidente.
* El nivel de protección de integridad.

Objetivo: generar eventos *(logs)*de seguridad para garantizar que las acciones realizadas por un ciberatacante se observan y registran.

Fallar de forma segura

El propósito de este principio es el de reducir la probabilidad de que un fallo en el software pueda saltarse los mecanismos de seguridad de la aplicación, dejándolo en un estado de fallo inseguro, vulnerable a los ataques. Es imposible implementar una aplicación perfecta que nunca falle, la solución consiste en saber en todo momento cuál es su estado y tener implementado un mecanismo, en caso de que falle.

Su objetivo es mantener la resiliencia (confidencialidad, integridad y disponibilidad) del software por defecto, para que sea seguro. Algunas de las características específicas del diseño que minimizan la probabilidad de que el software pase a un estado inseguro son:

* Implementar temporizadores tipo *watchdog.*
* Implementar una lógica de control de excepciones.
* El software siempre debe comenzar y terminar su ejecución en un estado seguro.
* Evitar problemas de sincronización y secuenciación.

Objetivo: reducir la probabilidad de que un fallo en el software pueda saltarse los mecanismos de seguridad de la aplicación, dejándolo en un modo de fallo inseguro, vulnerable a los ataques.

Diseño de software resistente

Dos propiedades importantes del software seguro es la robustez y la resiliencia, el siguiente principio pretende aumentar la resistencia de las aplicaciones reduciendo al mínimo la cantidad de tiempo que un componente de software defectuoso o fallido sigue siendo incapaz de protegerse de los ataques. Algunas de las características de diseño que aumentarán la resistencia del software incluyen (Goertzel, K. M. y Winograd. T., 2008):

* Capacidad de autocontrol y de limitar el uso de los recursos.
* Uso de técnicas de monitorización.
* Detección de estados anómalos.
* Proporcionar una realimentación que permita que todos los supuestos y modelos en los que el programa tome decisiones sean validados antes de ejecutarlas.
* Aprovechar cualquier redundancia y funciones de recuperación.
* Uso de plataformas virtuales.
* Uso de técnicas de recuperación.
* Determinar la cantidad de información a proporcionar en los mensajes de error.

Objetivo: reducir al mínimo la cantidad de tiempo que el componente de un software defectuoso o fallido sigue siendo incapaz de protegerse de los ataques.

La seguridad por oscuridad: error

Una de las asunciones que se debe realizar a la hora de diseñar una aplicación es que el atacante obtendrá, con el tiempo, acceso a todos los diseños y todo el código fuente de esta. La seguridad por oscuridad es un mecanismo de defensa que consiste en ocultar información sobre la aplicación de forma que sea difícil de obtener, pero que en poder de un atacante puede proporcionarle un medio para comprometerla.

El principio de diseño abierto*(open design)* establece que la implementación de salvaguardas de seguridad debe ser independiente del diseño, de modo que la revisión de este no comprometa la protección que ofrecen las salvaguardas. Esto es particularmente aplicable en la criptografía, donde los mecanismos de protección están desacoplados de las claves que se utilizan para las operaciones criptográficas, y los algoritmos utilizados para el cifrado y descifrado están abiertos y disponibles para que cualquier persona pueda revisarlos.

No es aconsejable depender de la seguridad por oscuridad, solamente indican que es válido usarla como una pequeña parte de una estrategia general de defensa en profundidad, para confundir y dificultar las actividades de ataque de un intruso. A continuación, se muestran algunos ejemplos de seguridad por oscuridad que pueden constituir un error:

* Guardar información en archivos binarios.
* Contraseñas en código fuente.
* Capetas ocultas en un servidor web.
* Falsa sensación de seguridad.
* Criptografía privada.
* Cambio del nombre del usuario «administrador».

Objetivo: concienciarse de que la seguridad por oscuridad es un mecanismo de defensa que puede proporcionar a un agente malicioso información para comprometer la seguridad de una aplicación.

Seguridad por defecto

Los sistemas deben diseñarse y configurarse de forma que garanticen la seguridad por defecto (ENS, 2010):

* El sistema proporcionará la mínima funcionalidad requerida para que la organización solo alcance sus objetivos y no alcance ninguna otra funcionalidad adicional.
* Las funciones de operación, administración y registro de actividad serán las mínimas necesarias, y se asegurará que solo sean accesibles por las personas, o desde emplazamientos o equipos autorizados, de manera que se puedan exigir, en su caso, restricciones de horario y puntos de acceso facultados.
* En un sistema de explotación se eliminarán o desactivarán, mediante el control de la configuración, las funciones que no sean de interés e, incluso, aquellas que sean inadecuadas al fin que se persigue.
* El uso ordinario del sistema ha de ser sencillo y seguro, de forma que una utilización insegura requiera de un acto consciente por parte del usuario.

Una cadena es tan fuerte como sus eslabones más débiles. Este principio de seguridad establece que la resistencia de su software contra los ataques dependerá, en gran medida, de la protección de sus componentes más débiles, se trate del código, un servicio o una interfaz. Una ruptura en el eslabón más débil resultará en una brecha de seguridad.

El principal objetivo del principio de seguridad, por defecto, es el de minimizar la superficie de ataque de cualquier aplicación o sistema TIC, deshabilitando aquellos servicios y elementos no necesarios y activando solo los necesarios.

Usualmente, cuando se instala o pone en producción una aplicación o servicio se activan o instalan servicios que no se usan con normalidad y que pueden suponer un punto potencial de entrada para los atacantes. Se deben elegir los componentes que van a ser utilizados de forma explícita por los usuarios e instalarlos y configurarlos de forma segura. Hay que minimizar los puntos vulnerables de la aplicación o sistemas, pues amenazan su seguridad global, por muy «securizado» que se tenga el resto; en concreto, los siguientes:

* Entradas y salidas de red.
* Número de puertos abiertos.
* Puntos de entrada a la aplicación, autenticados o no autenticados, locales o remotos y privilegios administrativos necesarios.
* Número de servicios. Desactivar los instalados por defectos no usados.
* Número de servicios con privilegios elevados.
* Número de cuentas de usuario administrador.
* Estado de las listas de control de acceso a directorios y ficheros.
* Control de cuentas de usuario y claves de acceso por defecto a servicios.
* Contenido dinámico de páginas web.

Varias organizaciones, como el National Institute of Standards and Technology (NIST), la Agencia de Seguridad Nacional (NSA) y el Centro Criptológico Nacional (CCN) han publicado guías de seguridad de configuración y *scripts*para productos COTS populares y de software libre que incluye la eliminación o restricción de servicios, usuarios, permisos y software innecesario.

El bastionado de sistemas es un proceso necesario en el marco de cualquier proyecto que contemple la aplicación de controles de seguridad sobre los sistemas TIC. Tiene como principio implementar todas las medidas de seguridad a nivel técnico, posibles para proteger un sistema sin que pierda la funcionalidad para la que fue destinado. Habrá casos en los que surjan conflictos que no se puedan superar, estos deben ser cuidadosamente documentados para proporcionar una justificación de la renuncia a los requisitos de configuración de seguridad que impidan el correcto funcionamiento del software.

Objetivo: reducir la superficie de ataque de una aplicación o sistema.