3. Seguridad en el Ciclo de Vida del Software en las Fases de Codificación, Pruebas y Operacion

3.1. Introducción y objetivos

Para conseguir software confiable y minimizar al máximo los ataques en la capa de aplicación, es necesario seguir un proceso sistemático o disciplina que aborde la seguridad en todas las etapas del ciclo de vida de desarrollo del software que incluya una serie de buenas prácticas de seguridad (S-SDLC) como especificación, requisitos de seguridad, casos de abuso, análisis de riesgo, análisis de código, pruebas de penetración dinámicas, modelado de amenazas, operaciones de seguridad y revisiones externas, necesarias para asegurar la confianza y robustez del mismo.

Normalmente, el modelo más propicio para el desarrollo de software en las organizaciones suele ser una combinación de dos o más modelos de los ciclos de vida (cascada, espiral, etc.). Sin embargo, hay que tener en cuenta que lo importante no es el modelo seguido, sino la incorporación de buenas prácticas de seguridad en las diferentes fases de este y unos hitos o puntos de control donde se verifiquen los entregables de cada fase. Entre las razones principales para añadir prácticas de seguridad en el SDLC tenemos:

* Mayor probabilidad de capturar adecuadamente los requisitos y tomar decisiones de diseño correctas y no cometer errores involuntarios de codificación.
* Dificultad de los agentes maliciosos para explotar vulnerabilidades y debilidades del software, pues el resultante de un ciclo de vida S-SDLC será más robusto en su entorno de ejecución y en su interacción con entidades externas.

Los objetivos del presente tema son los siguientes:

* Conocimiento y comprensión de las buenas prácticas de seguridad a incluir en un S-SDLC en las fases de codificación, pruebas y operación.
* Profundizar en el estudio de la principal práctica de seguridad: revisión de código.
* Estudiar las prácticas de seguridad aplicables en estas fases del SDLC, como pruebas de penetración, operaciones de seguridad, revisión externa y pruebas de seguridad basadas en riesgo.

3.2. Pruebas de seguridad basadas en riesgo

Hasta hace unos pocos años, las pruebas de software se desarrollaban en base a la demostración de sus requisitos como forma de comprobar el correcto funcionamiento de sus funcionalidades y servicios de seguridad; sin embargo, no sirven para determinar cómo se comportará en condiciones anómalas y hostiles, ni si está libre de vulnerabilidades.

Identificando los riesgos del sistema y diseñando las pruebas en base a ellos bajo la perspectiva de un atacante, un probador de seguridad de software puede enfocar correctamente las áreas de código donde un ataque probablemente pudiera tener éxito.

A diagram of a diagram

Description automatically generated

Figura 1. Pruebas seguridad basadas en riesgo. Fuente: elaboración propia.

Aunque los componentes de seguridad, como la criptografía, la autenticación y el control de acceso, jueguen un papel crítico en la seguridad del software, la seguridad en sí misma es una característica de la totalidad del sistema; por lo tanto, no se refiere solamente a los mecanismos y elementos de seguridad. Un desbordamiento de *buffer*es un problema de seguridad independientemente de si existe un componente de seguridad, o en un interfaz hombre máquina no crítico. Por esta razón, es necesario que las pruebas de seguridad impliquen dos tipos de aproximaciones:

A screenshot of a computer program

Description automatically generated

Figura 2. Tipos de pruebas seguridad del software. Fuente: elaboración propia.

Los objetivos de las pruebas de seguridad basadas en el riesgo son los siguientes:

* Verificar la operación confiable del software bajo condiciones hostiles de ataque.
* Verificar la fiabilidad del software, en términos de comportamiento seguro y cambios de estado confiables.
* Verificar la falta de defectos y debilidades explotables.
* Verificar la capacidad de supervivencia del software ante la aparición de anomalías, errores, y el manejo de estas mediante excepciones, que minimicen el alcance e impacto de los daños que puedan resultar de los ataques.

Las pruebas de seguridad deberían comenzar a nivel de componente antes de la integración del sistema. Se deben de utilizar los modelos de ataque, casos de abuso y análisis de riesgos desarrollados al comienzo del ciclo de vida, para mejorar el plan de pruebas con casos diseñados desde el punto de vista de los atacantes basados en los escenarios de abuso. Esto implica tanto pruebas de caja negra como de caja blanca.

Técnicas de pruebas que son particularmente útiles para las pruebas basadas en riesgo incluyen las mostradas en el siguiente diagrama:

A screenshot of a computer screen

Description automatically generated

Figura 3. Tipos de pruebas seguridad basadas en riesgo. Fuente: elaboración propia.

El análisis de caja blanca implica el análisis y el entendimiento tanto de código fuente como del diseño. Esta clase de pruebas es muy eficaz en el hallazgo de errores de programación, bugs, y debilidades de diseño, *flaws.* Las exploraciones de código pueden automatizarse con un analizador estático. Una desventaja de esta clase de pruebas consiste en que se podría encontrar una vulnerabilidad potencial donde en realidad existe un falso positivo. Hay que revisar el resultado de las herramientas de análisis estático para identificarlos.

A screenshot of a computer program

Description automatically generated

Figura 4. Puntos fuertes y débiles pruebas caja blanca. Fuente: adaptado de López, S. (2012).

* Revisión de diseño. Las vulnerabilidades de nivel de diseño son la categoría de defectos más difícil de manejar, además son tanto frecuentes como críticos. Lamentablemente, averiguar si un programa tiene vulnerabilidades de nivel de diseño requiere conocimiento y experiencia. Ejemplos de problemas de nivel de diseño incluyen manejo de errores en sistemas orientados a objetos, objetos compartidos, relaciones de confianza, canales de datos sin protección, mecanismos de control de acceso incorrectos, carencia de auditorías, registro (logs) y errores en el ordenamiento de eventos.
* Revisión de código o análisis estático de código. Se considera una de las prácticas de seguridad más importantes, consiste básicamente en el análisis del código fuente para detectar errores, construcciones inseguras de codificación e indicadores de vulnerabilidades o debilidades.
* Inyección de fallos en código fuente. Una forma de análisis dinámico en el que el código fuente se ha «instrumentalizado» mediante la inyección de errores. A continuación, se deberá compilar y ejecutar el código instrumentado para observar los cambios en el estado y el comportamiento que surgen cuando las partes instrumentadas del código se ejecutan. De esta manera, se puede determinar y cuantificar cómo, incluso, el software reacciona cuando es forzado en estados anómalos, tales como los provocados por fallos intencionales. Esta técnica ha demostrado ser particularmente útil para detectar el uso incorrecto de punteros y matrices, y condiciones de carrera.

La Inyección de fallos es un complejo proceso de prueba y, por lo tanto, tiende a ser limitada al código que requiere garantía muy alta de seguridad. Para la realización de estas pruebas se utilizan motores inyección. Este instrumento usa el navegador para interceptar transacciones y permitir a un analista estudiarlas.

El análisis de caja negra se refiere al análisis de un programa en ejecución al que se le sondea con diferentes tipos de entradas. Por tanto, se centra en estructuras de datos, componentes, APIs, estado de programa, etcétera. No usa análisis de código fuente de ninguna clase. La visión obtenida durante el análisis de riesgo arquitectónico es muy útil en la planificación de este tipo de pruebas de seguridad.

A screenshot of a computer program

Description automatically generated

Figura 5. Puntos fuertes y débiles pruebas caja negra. Fuente: adaptado de López, S., 2012.

* Pruebas de penetración. Su propósito se centra en la determinación de vulnerabilidades internas a un componente o entre ellos, que estén expuestas al acceso externo y si pueden ser explotadas para comprometer el software.

La explotación de las vulnerabilidades encontradas se puede realizar de forma automática con las siguientes herramientas:

Herramientas de explotación de vulnerabilidades:

* [Metasploit Frameword](https://www.metasploit.com/)
  + Licencia: comercial / libre
  + S.O.: Windows, Linux
* [Core Impact](https://www.coresecurity.com/products/core-impact)
  + Licencia: comercial
  + S.O.: Windows, Linux
* [Canvas](http://www.immunitysec.com/products/canvas/index.html)
  + Licencia: comercial
  + S.O.: Windows, Linux
* Análisis dinámico (Dynamic Application Security Testing-DAST). Detectan debilidades y vulnerabilidades de seguridad en aplicaciones web mientras se están ejecutando. Emplea técnicas de inyección de fallos en una aplicación, por ejemplo, entrada de datos maliciosos, tal y como se representa en la figura siguiente, para identificar vulnerabilidades comunes, como inyección SQL, *cross-site scripting, open redirect, command injection, path traversal*, etc. También identifica problemas de configuración del servidor, autenticación, inicio de sesiones, etc. Lo más eficiente es la realización de pruebas conjuntas SAST y DAST, pues ayudan a minimizar la cantidad de falsos positivos.

A screenshot of a chat

Description automatically generated

Figura 6. Funcionamiento pruebas Análisis Dinámico (DSAT). Fuente: López, S. (2012).

Herramientas DAST:

* [Acunetix WVS](https://www.acunetix.com/)
  + Licencia: comercial / libre
  + S.O.: Windows
* [AppScan](https://www.ibm.com/ar-es)
  + Licencia: comercial
  + S.O.: Windows
* [Burp Suite](https://portswigger.net/burp)
  + Licencia: comercial / libre
  + S.O.: Windows/Iun
* [GamaScan](http://www.gamasec.com/Gamascan.aspx)
  + Licencia: comercial
  + S.O.: Windows
* [Grabber](http://rgaucher.info/beta/grabber/)
  + Licencia: *open source*
  + S.O.: Python
* [Grendel-Scan](https://sourceforge.net/p/grendel/code/ci/c59780bfd41bdf34cc13b27bc3ce694fd3cb7456/tree/)
  + Licencia: *open source*
  + S.O.: Windows, Linux y Macintosh
* [N-Stealth](http://www.nstalker.com/alliance/)
  + Licencia: comercial
  + S.O.: Windows
* [Inviciti](https://www.invicti.com/)
  + Licencia: comercial
  + S.O.: Windows
* [Nikto](https://cirt.net/nikto2)
  + Licencia: *open source*
  + S.O.: Unix/Linux
* [NTOSpider](https://www.rapid7.com/products/appspider/)
  + Licencia: comercial
  + S.O.: Windows
* [ParosPro](https://sourceforge.net/projects/paros/)
  + Licencia: comercial
  + S.O.: Windows
* [QualysGuard](https://www.qualys.com/apps/web-app-scanning/)
  + Licencia: comercial
  + S.O.: N/A
* [Retina](https://www.beyondtrust.com/vulnerability-management)
  + Licencia: comercial
  + S.O.: Windows
* [SecPoint Penetrator](https://www.secpoint.com/penetrator.html)
  + Licencia: comercial
  + S.O.: Windows, Linux y Macintosh
* [Trustkeeper Scanner](https://www.trustwave.com/en-us/)
  + Licencia: comercial
  + S.O.: SaaS
* [Vega](https://subgraph.com/vega/)
  + Licencia: *open source*
  + S.O.: Windows, Linux, Macintosh
* [Wapiti](https://wapiti-scanner.github.io/)
  + Licencia: *open source*
  + S.O.: Windows, Linux y Macintosh
* [WebApp360](https://www.tripwire.com/)
  + Licencia: comercial
  + S.O.: Windows
* [WebInspect](https://www.microfocus.com/en-us/cyberres/application-security/webinspect)
  + Licencia: comercial
  + S.O.: Windows
* [WebKing](https://www.parasoft.com/products/parasoft-soatest/api-testing/)
  + Licencia: comercial
  + S.O.: Windows, Linux y Solaris
* [Websecurify](https://websecurify.com/)
  + Licencia: comercial / libre
  + S.O.: Windows, Linux y Macintosh
* [Wikto](https://sectools.org/tool/wikto/)
  + Licencia: *open source*
  + S.O.: Windows
* [Zap](https://code.google.com/archive/p/zaproxy/)
  + Licencia: *open source*
  + S.O.: Windows, Linux, Macintosh
* Análisis código binario. Es comparable a la exploración del código fuente, pero se dirige al código ensamblador o ejecutable compilado binario del software antes de ser instalado y ejecutado. No hay escáneres de código binario específico, se utilizan herramientas de ingeniería inversa y análisis de ejecutables binario; por ejemplo, descompiladores, desensambladores y escáneres de código binario como los utilizados por Veracode para analizar el código máquina y modelar una representación independiente del idioma de los comportamientos del programa, el control y los flujos de datos, árboles y las llamadas externas de función. Ejemplos de este tipo de herramientas son IDAPRO, WinDbg, etc.
* Inyección de fallos en binarios. La inyección de fallos de seguridad induce tensiones en el software, crea problemas de interoperabilidad entre los componentes y simula fallos en el entorno de ejecución. Simula tipos de fallos y anomalías que pudieran resultar de patrones de ataque o la ejecución de lógica maliciosa que hacen elsoftware vulnerable. En tiempo de ejecución, el probador modifica los datos que se pasan por el entorno del software. Sin embargo, los fallos inyectados no deben limitarse solo a aquellos que simulan ataques. Para obtener una comprensión más completa de todos los posibles comportamientos del software y sus estados, el probador también debe inyectar fallos que simulan condiciones muy poco probables, incluso los «imposibles».
* Fuzz testing. Al igual que en la inyección de fallos binario, consiste en la introducción de datos no válidos (por lo general producido por la modificación de una entrada válida) al software a través de su entorno o a través de otro componente de este. Las pruebas se llevan a cabo mediante un *fuzzer*,programa o *script* que realiza, modifica o combina entradas del software, para revelar cómo se comporta. Suelen ser específicos de un tipo particular de entrada, como HTTP, y se escriben para ser utilizados para probar un programa específico, por lo que no son fácilmente reutilizables. Sin embargo, su valor radica en su especificidad, ya que a menudo puede revelar vulnerabilidades de seguridad que las herramientas genéricas de evaluación no detectan. Para que sean eficaces se requiere que el probador tenga una comprensión completa del software que está probando y la forma en que interactúa con entidades externas, cuyos datos simulará el *fuzzer.* En la siguiente tabla se presenta herramientas de este tipo.

Herramientas para la auditoría *fuzzing:*

* [Mangle:](https://github.com/sughodke/fsfuzzer) *fuzzer* que genera etiquetas HTML y se autolanzará dentro de un navegador.
* [Spike:](https://resources.infosecinstitute.com/topic/intro-to-fuzzing/) colección de *fuzzers.*
* [Wfuzz:](http://www.edge-security.com/wfuzz.php) *fuzzer* para web que aplica fuerza bruta sobre el protocolo HTTP. Realiza pruebas de *path discovery* (recursivas) o de fuerza bruta sobre variables (pasadas por GET o por POST). Emplea diccionarios (en el caso de las inyecciones SQL el diccionario está preparado con los patrones SQL más empleados).
* [Netcat:](https://sourceforge.net/projects/netcat/) Utilidad Unix que lee y escribe datos a través de conexiones de red usando los protocolos TCP o UDP. Está diseñada para ser una utilidad de tipo *back-end* que pueda ser usada directa o fácilmente manejada por otros programas y scripts.
* [Radamsa:](https://github.com/aoh/radamsa) *fuzzer*de caja negra basado en mutaciones.
* [Blab:](https://code.google.com/archive/p/ouspg/wikis/Blab.wiki) *fuzzer* basado en gramática.
* [American Fuzzy Lop:](https://code.google.com/archive/p/american-fuzzy-lop/) *fuzzer* basado en mutaciones.
* [Zzuf:](http://caca.zoy.org/wiki/zzuf)CERT *basic fuzzing framework.*Búsqueda de *bugs.*
* [Sulley:](https://github.com/OpenRCE/sulley)extras para gestionar *fuzzing,*como parte de las pruebas de penetración.
* [Hping2:](http://www.hping.org/) herramienta de red capaz de enviar paquetes ICM/UDP/TCP hechos a medida, y de monitorizar las respuestas del host desino. Maneja fragmentación y tamaños arbitrarios de paquetes.
* [Sniffit:](http://sniffit.sourceforge.net/) herramienta de monitorización y *packet*de *sniffer* para paquetes de TCP/UDP/ICMP capaz de dar información técnica muy detallada acerca de estos paquetes.
* [Firewalk:](http://packetfactory.openwall.net/projects/firewalk/) emplea técnicas del estilo *traceroute*para determinar las reglas de filtrado que se están usando en un dispositivo de transporte de paquetes.
* Escaneo de vulnerabilidades. Este tipo de pruebas analizan los sistemas en busca de vulnerabilidades conocidas. Disponen de información sobre vulnerabilidades existentes en los sistemas operativos y aplicaciones mediante bases de datos actualizadas, que utiliza para la detección de estas.

La herramienta más utilizada es Nessus, inicialmente de código abierto y versión gratuita, y actualmente en dos versiones, una profesional comercial y otra de prueba gratuita ([www.nessus.org](http://www.nessus.org)). Otras herramientas de extendido uso son OpenVas, Nexpose de Rapid 7, ISS Real Secure, GFI LANguard, QualysGuard, Nmap y Retina.

A continuación, se enumeran y se suministra información de varias herramientas con diversas funcionalidades que existen actualmente:

* [Nessus](https://www.tenable.com/products/nessus)
  + Licencia: libre
  + S.O.: Linux/Unix, Windows
* [Nexpose](https://www.rapid7.com/products/nexpose/)
  + Licencia: comercial
  + S.O.: Linux/Unix, Windows
* [AppDetective](https://www.trustwave.com/en-us/)
  + Licencia: comercial
  + S.O.: Windows
* [Open Vas](https://www.openvas.org/)
  + Licencia: Libre
  + S.O.: linux
* [GFI LANguard](https://www.gfi.com/)
  + Licencia: comercial
  + S.O.: Windows
* [MBSA](https://www.microsoft.com/es-ar/)
  + Licencia: freeware
  + S.O.: Windows
* [Nmap](https://insecure.org/)
  + Licencia: GPL
  + S.O.: Linux/Unix, Windows, Mac OS X
* [Retina Network Security Scanner](https://www.beyondtrust.com/vulnerability-management)
  + Licencia: comercial
  + S.O.: Windows
* [SATAN](http://www.porcupine.org/satan/)
  + Licencia: comercial
  + S.O.: Linux/Unix
* [SDS (Shadow Database Scanner)](http://www.safety-lab.com/en/)
  + Licencia: comercial
  + S.O.: Windows
* [SolarWinds Engineer’s Toolset](https://www.solarwinds.com/homepage-3c?adobe_mc_sdid=SDID%3D247B9333ED024EDE-503C7927709DF165%7CMCORGID%3D8D6867C25245AEFB0A490D4C%40AdobeOrg%7CTS%3D1647546234)
  + Licencia: comercial
  + S.O.: Windows
* [Symantec Enterprise Security Manager](https://www.broadcom.com/products/cyber-security)
  + Licencia: comercial
  + S.O.: Windows

El análisis de caja gris es una combinación de las dos anteriores. Se trata de un tipo de análisis de seguridad, que combina pruebas de caja blanca y negra, que tiene en cuenta la interfaz del aplicativo y su código fuente.

* Análisis hibrido o Interactive Application Security Testing (IAST). Se trata de otra de las pruebas enfocadas al análisis de aplicaciones, tanto web como de otro tipo, que implica el uso del código fuente y del binario ejecutable, compilado a partir de dicho código. Para la realización del análisis, se ejecuta el programa compilado con un agente dentro del mismo, al tiempo que se «alimentan» las interfaces externas de la muestra. El revisor sigue y analiza los datos que el programa produce como resultado de su ejecución, de forma que cualquier vulnerabilidad o anomalía que surja en dicha interfaz se traza simultáneamente con el código fuente que la genera con mayor eficacia. En realidad, en este tipo de revisión se realizan tres tipos de análisis:
* Análisis de cobertura. Pone de manifiesto las interacciones entre las diferentes partes del programa.
* Análisis de frecuencia de espectro. Revela las dependencias entre los componentes del programa.
* Análisis de patrones. Permite buscar patrones específicos en la ejecución del programa, tales como excepciones no capturadas, omisiones, errores de memoria dinámica y problemas de seguridad.

Dado que el agente IAST está trabajando dentro de la aplicación, su análisis se aplica a toda la aplicación incluido su código. Se comprueba el control de los tiempos de ejecución, flujos de datos, configuración, peticiones y respuestas HTTP, bibliotecas, *frameworks* y otros componentes. El acceso a toda esa información permite al motor IAST producir resultados más precisos y verificar una gama más amplia de problemas de seguridad que SAST o DAST.

Herramientas de análisis híbrido:

* [Valgrind:](https://valgrind.org/) para todo tipo de aplicaciones.
* [INSURE++:](https://www.parasoft.com/products/parasoft-insure/) herramienta para análisis de errores para todo tipo de aplicaciones escrita en C, C++.
* [SCA Fortify](https://www.hp.com/us-en/services/managed-print-services/print-solutions.html) más Web Inspect de HP: disponen de un agente para integrar el resultado de las dos herramientas.
* [IBM Appscan](https://www.ibm.com/products?types%5b0%5d=software), con el agente GLASS BOX.
* [Acunetix](https://www.acunetix.com/vulnerability-scanner/) más Acusensor.
* [SEEKER.](https://www.synopsys.com/software-integrity/security-testing/interactive-application-security-testing.html)

Las pruebas de seguridad basadas en el riesgo (pruebas de seguridad desde el punto de vista del atacante), según lo anteriormente expuesto, se deberían estructurar en las siguientes fases:

* Descomponer el sistema en sus componentes fundamentales.
* Identificar las interfaces de los componentes.
* Clasificar las interfaces de los componentes por su riesgo potencial.
* Averiguar las estructuras de datos usadas por cada interfaz.
* Encontrar problemas de seguridad inyectando datos maliciosos, apoyándose en el estado del riesgo ante las posibles amenazas.

En la siguiente figura se muestran las diferentes pruebas a realizar en cada una de las fases del S-SDLC:

A table with blue text

Description automatically generated

Figura 7. Distribución pruebas de seguridad a lo largo de las fases del S-SDLC. Fuente: elaboración propia.

3.3. Revisión de código

El análisis estático de código fuente, tal y como comenta McGraw (2005), se considera la actividad más importante de entre las mejores prácticas de seguridad que se han de realizar en el curso del desarrollo de una aplicación. En los siguientes apartados se van a analizar los tipos y categorías de herramientas disponibles, tanto comerciales como de *open source,* para qué lenguajes están disponibles, cómo y cuándo se tienen que utilizar y cómo se enmarca el uso de estas herramientas en el proceso de revisión del código.

A diagram of a diagram of a company

Description automatically generated

Figura 8. Revisión de código. Fuente: elaboración propia.

Los problemas de seguridad de una aplicación pueden ser resultado de dos tipos de errores principales:

* Errores simples que comete el programador por confusión, un *lapsus*momentáneo, etc.
* Carencias de conocimientos del programador.

Con esto en mente, el análisis estático de código fuente es adecuado para identificar problemas de seguridad por ciertas razones:

* Las herramientas de análisis estático comprueban el código a fondo y coherentemente, sin ninguna tendencia. Un análisis valioso debe ser lo más imparcial posible.
* Examinando el código en sí mismo, las herramientas de análisis estático a menudo pueden indicar la causa de origen de un problema de seguridad, no solamente uno de sus síntomas.
* El análisis estático puede encontrar errores tempranamente en el desarrollo, aún antes de que el programa sea ejecutado por primera vez.
* Cuando un investigador de seguridad descubre una nueva variedad de ataque, las herramientas de análisis estático ayudan a comprobar de nuevo una gran cantidad de código.

La queja más común y contrastada contra las herramientas de análisis estático de seguridad es que producen demasiado ruido, es decir, falsos positivos: un problema descubierto en un programa cuando ningún problema existe en realidad. Un gran número de falsos positivos puede causar verdaderas dificultades.

Los falsos positivos son, seguramente, indeseables, pero desde una perspectiva de seguridad, los falsos negativos son mucho peores. Con un falso negativo, un problema existe en el programa, pero la herramienta no lo detecta. La penitencia por un falso positivo es la cantidad de tiempo gastado repasando el resultado. La penitencia por un falso negativo es mucho mayor, pues no solo se paga el precio asociado por tener una vulnerabilidad en el código, se vive con un sentido falso de seguridad que se deriva del hecho de que la herramienta hizo parecer que todo era correcto.

Todas las herramientas de análisis estático de código producen algunos falsos positivos y falsos negativos, su balance es normalmente indicativo del propósito de la herramienta de esta.

* Objetivo descubrir *bugs.* El coste de omitir un *bug*dentro de la primera categoría es relativamente pequeño hablando en términos de seguridad.
* Objetivo descubrir defectos relevantes de seguridad. La penitencia por*bugs* de seguridad pasados por alto es elevada, por lo que este tipo de herramientas, en general, producen más falsos positivos para reducir al mínimo falsos negativos.

Herramientas de revisión de código

Las herramientas de análisis de seguridad estáticas usan muchas de las mismas técnicas encontradas en los compiladores, pero su objetivo está más enfocado a identificar problemas de seguridad, por lo que aplican estas técnicas de manera diferente.

Para una herramienta de comprobación de propiedades, al buscar potenciales vulnerabilidades de desbordamiento de *buffer* habría que comprobar la propiedad «el programa no tiene acceso a una dirección fuera de los límites de memoria asignada».

Fortify Software e IBM fabrican herramientas de análisis estático de código que caen dentro de esta categoría.

Consulta más información sobre estas herramientas en: <https://www.microfocus.com/es-es/products/static-code-analysis-sast/overview>

En la práctica, lo importante es que estas herramientas suministran importantes resultados. El hecho de que sean imperfectas no les impide tener un valor significativo. Los principales factores prácticos que determinan la utilidad de una herramienta de análisis estático son:

* La capacidad de la herramienta para comprender el programa que se analiza.
* El equilibrio que la herramienta hace entre la precisión, la profundidad y la escalabilidad.
* Porcentaje de falsos positivos y falsos negativos de la herramienta.
* El conjunto de errores que la herramienta comprueba.
* Las características de la herramienta para hacerla fácil de usar.

Arquitectura de las herramientas de análisis estático de código

Independientemente de las técnicas de análisis usadas, todas las herramientas de análisis estático funcionan, aproximadamente, del mismo modo, tal y como se muestra en la siguiente figura. Todas aceptan el código, construyen un modelo (abstracción del código, modelo de autómata de estado finito, etc.) que representa el programa, analizan dicho modelo en combinación con una base de conocimiento de seguridad y terminan presentando sus resultados al usuario.

A diagram of a software process

Description automatically generated

Figura 11. Diagrama de bloques para una herramienta genérica de análisis de código. Fuente: elaboración propia.

Hay algunas técnicas importantes y estructuras de datos que comparten compiladores y las herramientas de análisis estático, como son los componentes que tienen la mayoría de los compiladores:

* Analizador léxico. Constituye la primera fase de la herramienta en recibir como entrada el código fuente de un programa (secuencia de caracteres) y produce una salida compuesta de símbolos (componentes léxicos). Estos símbolos sirven para una posterior etapa del proceso de traducción y son la entrada para el analizador sintáctico. Realiza, además, funciones como eliminar espacios en blanco, saltos de línea, tabuladores, ignorar comentarios, detección y recuperación de errores.
* Analizador sintáctico. Convierte la entrada de componentes léxico de la etapa anterior en una estructura del árbol, más útil para el posterior análisis. Capturan la jerarquía implícita de la entrada.
* Analizador semántico. Utiliza la entrada el árbol sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas.
* Estructuras de datos como las tablas de símbolos y de tipos. Es una estructura de datos donde a cada símbolo en el código fuente de un programa se le asocia información como la ubicación, tipo de datos y ámbito de cada variable, constante o procedimiento.

Las herramientas de análisis estático realizan varios tipos de análisis:

* Análisis estructural. Realiza comprobaciones de los detalles de la gramática y la sintaxis del texto de programa creando una estructura de datos denominada árbol de sintaxis abstracto (AST). Con la tabla de símbolos realiza la comprobación de tipos *typechecking.*
* Análisis de flujo de control. Exploración de los diferentes caminos de ejecución que se pueden seguir cuando una función se ejecuta.
* Análisis de flujo de datos. Examinan los caminos de movimiento de datos a través de un programa; por lo general, atraviesan el gráfico de flujo de control de una función y anotan dónde se generan los valores de datos y dónde se usan.
* *Taint Propagation.* Análisis de flujo de datos para determinar qué es lo que un atacante puede controlar desde las diversas entradas a la aplicación. Se requiere saber por dónde entra la información en el programa y cómo se mueve a través de él. Mediante esta técnica se encuentran muchos defectos de validación de entrada.
* *Pointer Aliasing.* Los alias de los punteros son otra clase de problema del flujo de datos. La finalidad del análisis de alias es determinar qué punteros apuntan a la misma posición de memoria.
* Análisis local. Analiza una función individualmente en cuanto a posibles caminos de ejecución, lo que elimina los caminos falsos, que son caminos por los cuales el código nunca puede ejecutarse porque son lógicamente incoherentes.
* Análisis global. Este análisis requiere hacer comprobaciones de las interacciones entre las funciones.
* Interpretación abstracta. Es una técnica general formalizada para descartar los aspectos del programa que no son relevantes. Modela el programa como una máquina abstracta consistente en una caracterización matemática que recopila información sobre el flujo de control y de los datos de este, simulando su comportamiento.
* *Model checking.* Para propiedades temporales de seguridad como «la memoria solo debería ser liberada una vez» y «solo los punteros no nulos deberían ser referenciados» es fácil representarlas como pequeños autómatas de estado finito.
* SAT Solvers. El denominado problema *boolean satisfacibility* consiste en averiguar si, dada una expresión, hay alguna combinación en los valores de las variables de la expresión que haga la expresión TRUE.

A continuación, se van a enumerar algunas de estas herramientas, tanto comerciales como libres *(open source)*, para lenguajes como C/C++ y Java, aunque las herramientas comerciales soportan varios lenguajes más. Las herramientas libres son, en la mayoría de los casos, proyectos de investigación de universidades. Este apartado se limita a enumerar algunas de ellas e indicar su tipo de licencia y los lenguajes que es capaz de revisar.

Herramientas de análisis estático de código fuente

* [SCA](https://www.hp.com/es-es/services/managed-print-services/print-solutions.html) de Fortify Software de HP
  + Licencia: comercial.
  + Lenguaje: C, C++, Java y otros
* [AppScam](https://www.ibm.com/products)de IBM.
  + Licencia: comercial
  + Lenguaje: C, C++, Java y otros
* [Prevent](https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html)de Coverity
  + Licencia: comercial
  + Lenguaje: C, Java y otros
* [K8 Inshight](https://www.perforce.com/support/consulting/klocwork-consulting) de Klocwork,
  + Licencia: comercial
  + Lenguaje: C, Java y otros
* [VERACODE](https://www.veracode.com/products/binary-static-analysis-sast)
  + Licencia: SaaS
  + Lenguaje: C, Java y otros
* [CXSUITE (CHECKMARX)](https://checkmarx.com/product/cxsast-source-code-scanning/)
  + Licencia: comercial
  + Lenguaje: C, Java y otros
* [CodeSonar](https://www.grammatech.com/codesonar-cc) de Grammatech,
  + Licencia: comercial
  + Lenguaje: C, C++
* [SATURN](http://saturn.stanford.edu/index.html) de Stanford University.
  + Licencia: libre
  + Lenguaje: C
* [BOOP (C)](http://boop.sourceforge.net/) de Graz University of technology.
  + Licencia: libre
  + Lenguaje: C
* [Magic](http://www.cs.cmu.edu/~chaki/magic/) de Carnegie Mellon University.
  + Licencia: libre
  + Lenguaje: C
* [Jtest](https://www.parasoft.com/) de Parasoft
  + Licencia: comercial
  + Lenguaje: Java
* [FindBugs](http://findbugs.sourceforge.net/) de Maryland k,
  + Licencia: libre
  + Lenguaje: Java

* [Java PathFinder](http://javapathfinder.sourceforge.net/)
  + Licencia: libre
  + Lenguaje: Java
* [Cppcheck](https://cppcheck.sourceforge.io/)
  + Licencia: libre
  + Lenguaje: C,C++
* [Polyspace](https://www.mathworks.com/products/polyspace.html) de Mathworks.
  + Licencia: comercial
  + Lenguaje: C, ada
* [Pc-lint](https://gimpel.com/) de Gimpel Sotfware
  + Licencia: comercial
  + Lenguaje: C, C++

3.4. Pruebas de penetración

Una vez terminada la fase de desarrollo se despliega el sistema. Se deben llevar a cabo muchas de las operaciones o actividades de seguridad relacionadas con la puesta en marcha de la aplicación para su posterior paso a producción y explotación por parte de los usuarios. Las actividades de seguridad a realizar en esta fase comprenden la implementación y comprobación de la eficacia de las salvaguardas, tanto del tipo de software (por ejemplo, autenticación) como de los dispositivos hardware (por ejemplo, *firewall*) que se derivaron de la actividad de análisis de riesgos realizada en la fase de diseño.

La comprobación de la eficacia de las salvaguardas implementadas se realiza principalmente mediante las pruebas de penetración, que tiene como principal misión verificar cómo el software se comporta y resiste ante diferentes tipos de ataque.

A diagram of a diagram

Description automatically generated

Figura 10. Test de penetración en el SDLC. Fuente: elaboración propia.

Las pruebas de penetración deben centrarse en aspectos de comportamiento del software, sus interacciones y vulnerabilidades que no puedan ser detectadas mediante otras pruebas realizadas fuera del entorno de producción. Deben tratar de encontrar problemas de seguridad que puedan originarse en su arquitectura y diseño (frente a errores de codificación que se manifiestan como vulnerabilidades), ya que este tipo de problema tiende a ser pasado por alto por otras técnicas de prueba.

El plan de pruebas de penetración debe incluir los «peores» escenarios en los que se puedan reproducir vectores de ataques e intrusiones que se consideran altamente perjudiciales, como los escenarios de amenazas internas. El plan de pruebas debe capturar:

* La política de seguridad del sistema se supone que debe respetar o hacer respetar.
* Amenazas previstas.
* Riesgos de seguridad (conducido por casos de abuso, riesgos arquitectónicos y modelos de ataque).
* Secuencias de ataques probables que se puedan producir.

Las pruebas de penetración de una aplicación deben centrarse en aspectos de comportamiento del software, sus interacciones y vulnerabilidades que no puedan ser detectadas mediante otras pruebas realizadas. En este vídeo *(Pruebas de Penetración en desarrollo de aplicaciones)* se estudia la buena práctica de seguridad «Pruebas de Penetración» en el contexto del desarrollo y pruebas de seguridad de una aplicación.

Consideraciones sobre las pruebas de penetración

Una buena parte de los defectos y vulnerabilidades del software no está relacionada directamente con la funcionalidad de seguridad. Muchas de las cuestiones asociadas con la seguridad implican un mal uso de una aplicación, inesperado y descubierto por un atacante, normalmente. Las pruebas de penetración tienen que verificar los aspectos negativos del sistema; es decir, se debe probar la seguridad de la aplicación en base a los riesgos (conducido por casos de abuso, riesgos arquitectónicos y modelos de ataque) para determinar cómo se comporta ante los ataques.

Las pruebas de penetración son pruebas de caja negra. En cualquier caso, la prueba de aspectos negativos es un desafío mucho mayor que la verificación de un aspecto positivo. Es realmente fácil probar si un componente funciona adecuadamente o no, pero es muy difícil demostrar si realmente un sistema es seguro a un ataque malévolo.

Si la realización de pruebas para detección de aspectos negativos no revela ningún defecto, estas únicamente demuestran que ningún defecto ocurre en las condiciones particulares de esas pruebas, en ningún caso demuestran que ningún defecto existe.

Las pruebas de penetración son, de forma general, las más aplicadas de todas las mejores prácticas de seguridad del software, y suelen ser parte del proceso de aceptación de final. A causa de restricciones de tiempo, la mayor parte de este tipo de evaluaciones son realizadas de forma apresurada como un punto de la lista de comprobación de seguridad al final del ciclo de vida.

Una limitación principal de este acercamiento es que casi siempre representa una tentativa demasiado corta y muy tardía para abordar el problema de la seguridad al final del ciclo de desarrollo.

El verdadero valor de las pruebas de penetración viene de sondar un sistema en su ambiente final de operación. El entendimiento del ambiente de ejecución y de los problemas de configuración es el mejor resultado de cualquier prueba de penetración. Esto es así, sobre todo, porque estos problemas pueden ser solucionados en realidad tarde en el ciclo de vida. Las pruebas de penetración llegan al corazón del entorno de ejecución y de los aspectos de configuración rápidamente.

El éxito de una prueba de penetración depende de muchos factores, pocos de los cuales se prestan a la métrica y la estandarización. La primera variable y más obvia es la habilidad, el conocimiento, y la experiencia del probador. El resultado de estas depende de ello.

Todos estos problemas palidecen en comparación con el problema de que las pruebas de penetración a menudo son usadas como una excusa para declarar la victoria de seguridad «y ya está todo hecho». Lamentablemente, las pruebas de penetración realizadas sin basarse en el análisis de riesgos de seguridad conducen a una falsa sensación de seguridad.

Herramientas para test de penetración

Uso de herramientas para realizar las pruebas de penetración. Normalmente, la realización de este tipo de pruebas de caja negra engloba la realización de otros tipos de pruebas:

* Escaneo de vulnerabilidades.
* Explotación de vulnerabilidades.
* Fuzz testing.
* Análisis dinámico (DAST) para aplicaciones web.

Con las herramientas de escaneo de vulnerabilidades se encuentran vulnerabilidades conocidas con poco esfuerzo, que aprovechan las herramientas de explotación de vulnerabilidades. Las herramientas de análisis dinámico y de *fuzzing* pueden observar cómo se ejecuta un sistema, pueden someter los datos mal formados, malévolos y arbitrarios en los puntos de entrada del sistema en una tentativa de destapar fallos. Los defectos se reportan al personal de prueba para su análisis. Cuando es posible, el empleo de estas herramientas se debe dirigir por los resultados de análisis de riesgo y los modelos de ataque.

3.5. Operaciones de seguridad

El proceso final para llevar a cabo, previo al paso a producción de la aplicación segura, se compone de las actividades centrales de:

* Distribución.
* Despliegue.
* Operaciones.

A diagram of a diagram

Description automatically generated

Figura 14. Operaciones seguridad. Fuente: elaboración propia.

Distribución

El objetivo de la distribución segura es reducir al mínimo las posibilidades de acceso y manipulación del software durante la transmisión de un proveedor a su consumidor (envío a través de medios físicos o descarga de red) por los agentes maliciosos.

A la hora de realizar la distribución y despliegue del software desarrollado, es recomendable el realizar las siguientes buenas prácticas:

* Cambio de los valores de configuración predeterminados durante el desarrollo.
* Utilizar mecanismos de distribución estándar, como los utilizados en los derechos de propiedad intelectual.
* Distribuir el software con una configuración por defecto segura y lo más restrictiva posible.
* Realizar una guía de configuración de seguridad.
* Proporcionar una herramienta de instalación automática.
* Establecer un medio de autenticación para la persona que va a ejecutar la instalación y configuración.
* Los interfaces de configuración proporcionados por la herramienta o el*script* de instalación deben ser seguros.
* Revisión y limpieza de todo el código fuente por el visible usuario (por ejemplo, código del cliente de aplicaciones web).

Despliegue

Una configuración cuidadosa y la personalización del entorno de despliegue de cualquier aplicación de software pueden mejorar enormemente su seguridad. El diseño de un entorno de despliegue adaptado para una aplicación requiere de un proceso:

* Comienza en el nivel de componente de red.
* Continúa por el sistema operativo y otro software de base como puede ser un gesto de base de datos, etc.
* Termina con la propia configuración de seguridad de la aplicación y el sistema.

El software puede haber sido diseñado y desarrollado para ser extremadamente seguro, pero no lo será si sus parámetros de configuración no se establecen como el diseñador lo diseñó. De manera similar, los parámetros de configuración de su entorno de ejecución deben ajustarse de modo que el software no sea innecesariamente expuesto a amenazas potenciales, a esta tarea se le denomina «bastionado».

A diagram of a diagram with text

Description automatically generated

Figura 13. Capas del sistema a proteger. Fuente: elaboración propia.

El bastionado del software comprende los procesos y herramientas necesarias para realizar el control y corrección de los defectos y debilidades de las configuraciones del software sobre la base de un proceso de control de configuración. En España, el Centro Criptológico Nacional (CCN) y, en Estados Unidos, el departamento de Defensa, y otros departamentos, elaboran directrices de configuración seguras y listas de control para productos de software comercial. Podemos encontrar algunas en:

* [Centro Criptológico Nacional (CCN)](https://www.ccn-cert.cni.es/)
* [NIST Listas de comprobación de configuración de productos TIC](https://ncp.nist.gov/repository)

Operaciones

Muchos desarrolladores de software argumentan que la distribución, el despliegue y las operaciones no son parte del proceso de desarrollo de software. Hay que ajustar los controles de acceso a la red y niveles del sistema operativo, así como diseñar un sistema de registro de eventos, de monitorización, de *backup* y recuperación de los sistemas de ficheros, que será lo más eficaz durante operaciones de respuesta ante incidentes. Los ataques llegarán, y se debe estar preparado para defenderse de ellos y para deshacer el daño ocasionado después de que un ataque ha tenido lugar.

3.6. Revisión externa

Un análisis externo por personal ajeno al equipo de diseño es bastante eficaz y fundamental aportando otra visión de la seguridad del sistema y del riesgo y contribuyendo a una mejora de la seguridad, porque seguramente este análisis va a descubrir alguna amenaza y riesgo residual existente. Después, terminado el análisis externo habrá que revisar el análisis de riesgos y, si hay nuevas amenazas y riesgos, gestionarlos para que sean mitigados; si es necesario, también se deberán realizar cambios en la arquitectura hardware y software*,* realizar cambios en el código y, por lo tanto, repetir la revisión de este, los *tests* de seguridad basados en el riesgo que ha cambiado y volver, después del despliegue de la aplicación, a pasar los *tests* de penetración.

A diagram of a diagram of a company

Description automatically generated

Figura 16. Revisión externa. Fuente: elaboración propia.

Esto conforma un esquema de seguridad en el ciclo de vida de carácter cíclico en el que es posible que se tengan que realizar varias veces el mismo tipo de actividades, consecuencia de la naturaleza cambiante y de continua evolución de las aplicaciones y también del carácter cíclico de los distintos ciclos de vida de desarrollo de aplicaciones donde, en muchos casos, se van refinando prototipos que evolucionan hasta conseguir el sistema final.

Una práctica común en la adquisición es aceptar un software que satisface la funcionalidad con poca o ninguna consideración con respecto a la especificación y aseguramiento de las propiedades de seguridad. Esto aumenta la exposición y riesgos de seguridad de la organización. Es este vídeo *(Seguridad en las adquisiciones de código)* se estudian las mejores prácticas que permiten verificar la seguridad del software adquirido.