FUNDAMENTOS DEL DISEÑO SEGURO

Estas notas de apoyo tienen como objetivo suplementar las transparencias y facilitar la comprensión de los contenidos vistos en clase. Son, por tanto, un complemento y no un sustituto de estas.

1 REDUCCIÓN DE LA SUPERFICIE DE ATAQUE

La superficie de ataque es un concepto muy útil para identificar, evaluar y mitigar los riesgos asociados a los sistemas de software actuales. Se podría definir como todos los puntos de entrada posibles que un atacante puede usar para atacar la aplicación o el sistema en cuestión; es el área dentro de la red o aplicación que es visible para un atacante y que puede ser atacada. Incluso una persona que sea susceptible a un ataque de ingeniería social se considera parte de la superficie de ataque.

Reducir la superficie de ataque del software es un buen enfoque para mejorar la seguridad de la información. Las estrategias básicas de reducción de la superficie de ataque incluyen reducir la cantidad de código en ejecución, reducir los puntos de entrada disponibles y desactivar los servicios o funcionalidad con una demanda escasa.

Al haber menos código disponible, habrá menos fallos y vulnerabilidades; y al desactivar la funcionalidad innecesaria, habrá menos riesgos de seguridad. Aunque la reducción de la superficie de ataque ayuda a prevenir problemas de seguridad, no mitiga el daño que un atacante podría infligir una vez que se encuentra una vulnerabilidad en la superficie de ataque existente.

Se detallan algunas de las transparencias a continuación:

[3] Ejemplo de la Superficie de Ataque

Un ladrón que quisiera robar la casa podría intentar entrar en un primer momento por la puerta principal, las ventanas frontales o incluso las laterales.

En un segundo paso, podría descubrir que, escalando el toldo del primer piso, tal vez pueda acceder a las ventanas del piso superior que, muy probablemente, estén menos protegidas que las de la primera planta.

En un análisis más concienzudo, el ladrón podría descubrir que otro punto de entrada posiblemente menos protegido todavía sea la chimenea.

Al igual que con una casa, la superficie de ataque del software puede parecer evidente, pero tras un análisis más detallado se suele descubrir que hay otros muchos puntos de entrada que no habían sido considerados en un principio. Identificar la superficie de ataque es una tarea relativamente sutil y que requiere de experiencia.

[4, 5 y 6] Análisis de la Superficie de Ataque

En un primer paso, se deben identificar y definir todos los puntos de entrada al sistema software que se está diseñando. Estos pueden consistir en entrada/salida de red o de ficheros, cualquier entrada que provenga del usuario, llamadas externas o de RPC, etc.

Una vez se han identificado todos los puntos de entrada, es necesario priorizarlos por su nivel de peligrosidad. No es lo mismo un punto de entrada con acceso anónimo (puede acceder cualquiera) que

otro que requiera acceso autentificado (es necesario tener credenciales de usuario válidas), puesto que esto limita la posibilidad de que ocurra un ataque asociado a dicho punto de entrada.

De igual forma, si un punto de entrada está limitado a un acceso a nivel de administrador es menos peligroso que si simplemente se necesita ser un usuario sin privilegio; un elemento accesible únicamente por red local o intranet está más limitado que uno accesible por internet externo; un acceso por TCP es más limitado que uno por UDP (al requerir TCP mantener la conexión); etc.

El proceso de análisis es un proceso iterativo, en el que cada punto de acceso o funcionalidad tiene elementos que deben analizarse y que a su vez tienen otros subelementos o subfuncionalidades a considerar.

Por ejemplo, si la aplicación lee ficheros, además es necesario analizar cada tipo o formato por separado puesto que pueden suponer problemas de seguridad distintos. En este caso se trata de formatos de imagen, pero hay ataques de desbordamiento de búfer asociados a cabeceras malformadas en GIF, ataques asociados al esquema de descompresión de JPEG o incluso la posibilidad de incluir código ejecutable dentro de los contenedores configurables de un fichero TIFF.

En el caso de protocolos, es posible que la aplicación soporte distintos protocolos de transporte seguro y que cada uno tenga ciertos tipos de ataques asociados (por ejemplo, las versiones de SSL antiguas soportan por compatibilidad cifradores antiguos que están rotos, mientras que las versiones modernas han eliminado esos cifradores).

La conectividad HTTP también implica una gran cantidad de subfuncionalidades con características especiales: no es lo mismo usar GET que POST, etc. Conviene filtrar la funcionalidad HTTP que no esté en uso. Otros puntos a considerar son la gestión de las sesiones o la información que se proporciona en los mensajes de error, entre otros muchos.

También se incluye SMTP (Send Mail Transfer Protocol, asociado al email) como ejemplo de un protocolo problemático, con una historia de vulnerabilidades extensa y una dificultad de configuración elevada.

Reducir la superficie de ataque no implica exclusivamente desactivar funcionalidad, en la tabla de la transparencia 6 se indican una serie de decisiones de diseño y alternativas preferibles con un riesgo más reducido.

[7] Ejemplos de Reducción de SA

En Windows originalmente se permitían las llamadas de procedimiento remoto (RPC, *Remote Procedure Call*) sin autentificar. Las RPC son una serie de funciones que se pueden llamar de forma remota y, por lo tanto, permiten la ejecución de código remoto. En las versiones modernas de Windows se reduce la posibilidad de que un atacante logre ejecutar código en nuestra máquina mediante una RPC exigiendo que dicha llamada provenga de un usuario autentificado.

Otro aspecto de seguridad que ha mejorado en Windows es que, antiguamente, el firewall venía desactivado por defecto provocando que un gran número de máquinas de usuarios no expertos quedaran sin firewall y vulnerables a muchos ataques. Hoy en día el firewall está activo por defecto al instalar Windows, reduciendo la superficie de ataque y mitigando los riesgos asociados a ataques de red.

Internet Information Services (IIS) es un servidor web desarrollado por Microsoft. En muchos casos se instalaba asociado a otras aplicaciones y quedaba activo por defecto sin conocimiento del usuario. Hoy en día se instala desactivado por defecto, reduciendo la posibilidad de ataques a este servidor. Además, se ejecuta como servicio de red por defecto, lo que implica una seguridad mayor (ejecución bajo

demanda y con un nivel de privilegio menor) y la configuración inicial sólo incluye soporte para páginas web estáticas, impidiendo la ejecución de scripts o código en el servidor.

SQL Server es un sistema de gestión de bases de datos de Microsoft. Al igual que IIS, se instala como complemento de muchos programas por lo que no resulta evidente que está instalado para usuarios menos expertos. Dentro de su funcionalidad, soporta la ejecución de código arbitrario a través de SQL mediante el procedimiento *xp_cmdshell*; en las versiones modernas está desactivado por defecto. También están desactivados por defecto el Common Language Runtime (CLR) que permite ejecutar código en C# y otros lenguajes compatibles con memoria gestionada, así como el sistema Component Object Model (COM), más antiguo, y que permite la ejecución de código nativo. Otro aspecto que ha sido mejorado con el tiempo es que por defecto SQL Server sólo soporta conexiones locales.

Visual Studio es un entorno de desarrollo de software de Microsoft. Antiguamente, Visual Studio instalaba tanto una versión de IIS como de SQL Server para habilitar cierta funcionalidad y sus configuraciones permitían el acceso remoto a ambos. Esto provocaba con mucha frecuencia que alguien que había estado trabajando en su máquina en un entorno local (su casa o la intranet de la empresa) expusiera involuntariamente datos del software desarrollado al conectarse a una red pública (cafetería, aeropuerto, etc.). Para evitar este problema, ahora se instalan con acceso limitado a *localhost*, obligando a modificar conscientemente la configuración en caso de requerir acceso remoto.

2 PRIVACIDAD

[8 y 9] Privacidad Básica

Si bien suelen confundirse e interrelacionarse, la privacidad y la seguridad son conceptos diferentes.

Solemos asociar el concepto de privacidad con que no se conozcan los datos de una persona o usuario, pero, de modo más general, la privacidad implica que los usuarios sean los que controlen el uso, recolección y distribución de su información personal.

Por otra parte, cuando un sistema es seguro se garantizan la integridad, disponibilidad y confidencialidad de la información que gestiona dicho sistema.

El software robusto y de confianza se basa en una combinación de seguridad y privacidad de la información, pero es importante entender que se puede tener seguridad sin privacidad, pero no al revés. Por ejemplo, muchas redes sociales son tremendamente seguras, desarrolladas por empresas con grandes presupuestos de ciberseguridad y personal altamente entrenado para reducir al mínimo cualquier posibilidad de ataque o vulnerabilidad. Sin embargo, en muchos casos, su modelo de negocio choca frontalmente con la privacidad al financiarse mediante la venta de anuncios que se beneficia de la explotación de los datos privados de los propios usuarios de la red.

[10] Comportamientos y Controles

En muchos entornos de aplicación existe cierta legislación que limita lo que puede hacer o no el software respecto a la privacidad. Si bien los aspectos legales pertenecen al ámbito del derecho, es importante investigar y conocer la legislación aplicable al proyecto, porque puede determinar ciertos aspectos de diseño del software.

Por ejemplo, la legislación COPPA de EEUU define cómo se debe comportar el software que va dirigido a menores. Cuando se trata información sensible, es interesante conocer GLBA (asociada a banca y aseguradoras) o HIPAA (asociada a la información sanitaria). Tanto la UE como la FTC (entidad gubernamental de EEUU) tienen legislación acerca de cómo tratar con información personal, aunque ésta sea no sensible. Las aplicaciones que realicen modificaciones del sistema operativo pueden estar

sujetas al CFAA para operar dentro de la legalidad y evitar ser consideradas como *malware*. De igual forma, aquellas aplicaciones que realicen un monitoreo continuo del usuario o de ciertos aspectos del sistema o dispositivo pueden estar sujetas a la legislación anti-spyware y pueden llegar a ser bloqueadas por sistemas operativos o tiendas de aplicaciones. Otro aspecto que puede provocar el bloqueo de instalación es la transferencia no autorizada de información por parte de la aplicación, incluso si dicha información es anónima.

3 MODELADO DE AMENAZAS

[11 y 12] Modelado de Amenazas

El modelado de amenazas es un proceso que busca identificar, enumerar y entender las amenazas que puede sufrir un software determinado, así como priorizar los distintos mecanismos de mitigación de dichas amenazas.

Es importante comprender que una amenaza no es lo mismo que una vulnerabilidad. Una amenaza es un ataque potencial capaz de comprometer la seguridad de un sistema y por lo tanto extrínseco al sistema, mientras que una vulnerabilidad es una debilidad concreta del sistema (intrínseca) que permite que una amenaza sea efectiva.

El modelado de amenazas se realiza de forma iterativa, partiendo de la visión que se tiene del proyecto de software y su entorno, se establece un modelo de amenazas, se identifican, se mitigan con controles de seguridad adecuados y se validan dichos controles para proceder a refinar el modelo de amenazas, etc.

[13] Herramienta de Microsoft

Microsoft proporciona una herramienta gratuita para el modelado de amenazas. El enlace cambia continuamente (actualmente está aquí), pero se puede buscar también por "Microsoft Threat Modelling tool". Esta herramienta se escapa un poco del alcance de la asignatura, pero cabe destacar que Microsoft basa su modelado de amenazas en la metodología *STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service y Elevation of privilege).*

4 OTROS ASPECTOS DE DISEÑO SEGURO

[14 y 15] Defensa en profundidad

La defensa en profundidad es un concepto que se basa en la seguridad por capas. En muchos casos utilizamos una única capa de seguridad (por ejemplo, un firewall o SSL) que provoca una catástrofe en caso de fallar.

La forma correcta de diseñar la seguridad de la aplicación es asumir que los sistemas de seguridad pueden y van a fallar en algún momento, colocando diversas capas o mecanismos de seguridad para evitar que el fallo de un único sistema suponga el fallo total de la aplicación.

En el ejemplo, se ve[1] que el atacante puede llegar al servidor cuando logra que falle el firewall[2], pero esto no ocurre si hay otras capas de seguridad (los bloques rojos) que rechacen el ataque.

[1] Mejor con animaciones, la verdad.
[2] El muro de ladrillos en llamas (literalmente).

[16 y 17] Minimización del privilegio

Al asumir que todas las aplicaciones pueden ser y serán atacadas de forma satisfactoria, entendemos la importancia de la minimización de privilegio.

Consiste en intentar minimizar el daño causado por un potencial ataque al reducir la capacidad de acción del atacante. Para ello se analiza la funcionalidad de la aplicación y se determina el nivel de acceso o privilegio mínimo para realizar dichas funciones, siguiendo una estrategia de privilegio dinámico: se mantiene el nivel mínimo posible y sólo se eleva para realizar las acciones que lo requieran, regresando al nivel mínimo en cuanto sea posible.

En el ejemplo se puede ver al atacante que ha comprometido una aplicación en un servidor remoto. Si dicha aplicación corría con privilegios de administrador, el atacante será capaz de realizar prácticamente cualquier cosa; mientras que, si la aplicación corría con privilegios mínimos, el atacante tiene una capacidad muy limitada de provocar daños en el sistema.

[18 y 19] Valores por defecto seguros

Es importante a la hora de entregar o instalar una aplicación que la configuración por defecto sea lo más segura y/o restrictiva posible. De esta forma obligamos a que sea el usuario quien, de forma consciente e informada, active funcionalidad o reduzca la seguridad o privacidad si es necesario.

Esto contrasta con la mentalidad antigua de instalar con una configuración por defecto con funcionalidad máxima que, si bien facilitaba que usuarios sin experiencia dispusieran de un sistema complejo de forma sencilla, también maximizaba las posibilidades de éxito de los atacantes.

En el ejemplo se detallan algunos aspectos o elementos de diseño seguro y cuál sería su configuración por defecto más segura. Cabe destacar que se describen (por simplificar) como alternativas para el almacenaje de contraseñas las posibilidades de en claro o con hashes de contraseñas, pero nosotros sabemos por lo visto en el tema 2c que la mejor forma es utilizar un *PBKDF* como *Argon2* y una sal aleatoria.

5 ANÁLISIS DE CÓDIGO

[21] Introducción

El análisis de código consiste en emplear software para analizar de manera automática el código de la aplicación o sistema y detectar errores de programación, comprobar que se sigan las prácticas recomendadas, etc.

En principio, como se trata de analizar el código, podemos destacar dos clases diferenciadas en función de si analiza el código fuente (análisis estático) o el código ejecutable (análisis dinámico o binario). Ambos tipos de análisis tienen ventajas e inconvenientes.

Cabe destacar que estas herramientas no son infalibles a la hora de detectar problemas. Sin embargo, pueden resultar muy útiles ya que, el ser automatizadas, escalan adecuadamente con la complejidad del proyecto software y, por lo tanto, ayudan a reducir los costes de ingeniería.

[22, 23 y 24] Análisis estático vs binario

Aunque todos deberíamos saber la diferencia entre código fuente y código binario o ejecutable, sí que merece la pena destacar las diferencias clave entre los dos enfoques de análisis de código.

En el **análisis estático** se analiza el código fuente. Esto implica que se dispone de información útil como el nombre de las variables, de las funciones, comentarios, etc. que facilitan la comprensión de lo que está haciendo el software.

Por otra parte, un analizador de código fuente es dependiente del lenguaje de programación y, en algunos casos, de una implementación concreta de un lenguaje de programación. Esto hace necesario tener varios analizadores distintos si en el proyecto coexisten distintos lenguajes de programación, lo que es muy frecuente en la actualidad.

Si lo analizamos con detenimiento, un analizador de código estático es muy similar a un compilador o un intérprete. Por ello, es una tecnología relativamente madura ya que podemos reutilizar gran parte del conocimiento adquirido a lo largo de los años en el desarrollo de compiladores y traductores de código para realizar el análisis de código estático. Aunque pueda parecer algo menor, la existencia de buenas herramientas para trabajar con la gramática de un lenguaje predispone a que existan muchos y buenos analizadores estáticos para dicho lenguaje. Este es el caso de Go, que incluye un analizador léxico y sintáctico del propio lenguaje en la librería estándar (paquetes en la categoría "go").

Estos analizadores estáticos pueden estar incorporados en el entorno de desarrollo activándose al grabar o editar código (esto ocurre como parte del plugin de Go en *VSCode*), como parte del proceso de compilación (los *warnings* de seguridad que proporciona Visual Studio al compilar C/C++ serían un ejemplo) o como una herramienta externa que se debe incorporar como parte del proceso de ingeniería del software. Idealmente, el análisis estático se realiza de forma automática e interactiva, permitiendo evitar errores de forma temprana en el desarrollo y evitando, en la medida de lo posible, que se incorpore código vulnerable al repositorio del proyecto.

En el **análisis binario o dinámico** se analiza el código ejecutable. Realmente, este análisis se puede realizar sobre el código objeto en un fichero o en ejecución en memoria (de ahí el nombre de dinámico). Si bien en el código binario se pierde gran parte de la información como nombres de variables, funciones u otros elementos que simplifiquen su comprensión, tiene la ventaja de analizar el código final optimizado que corre en el procesador.

Esto es importante ya que algunas vulnerabilidades se introducen precisamente en el proceso de compilación y optimización del código y no son detectables en el código fuente. Este aspecto resulta más relevante incluso en el análisis del código en ejecución (dinámico) ya que existe código que varía en memoria respecto a lo almacenado en disco; ejemplos de esto serían programas con partes de código comprimido en disco que se descomprimen en memoria o código polimórfico que varía en ejecución. Estas técnicas suelen ir asociadas a malware que intenta evitar la detección por antivirus y otro software similar, pero también se usan en software comercial para evitar la manipulación de código o implementar sistemas de protección de copia contra la piratería.

En muchos casos, los analizadores de código binario son muy similares a depuradores o intérpretes convencionales, pero con funcionalidad adicional para detectar patrones de error comunes y otros problemas de seguridad. Aunque pueda parecer que el hecho de analizar código binario los hace completamente independientes del lenguaje de programación, la realidad es que distintos lenguajes de programación crean ejecutables con estructuras muy diferentes, por lo que pueden estar especializados en código compilado que provenga de un lenguaje de programación específico. Además, también hay grandes diferencias entre distintas arquitecturas de procesador; por ejemplo, se podría utilizar un único analizador estático para un proyecto en Go que se compilará para arquitecturas ARM y x64 pero haría falta un analizador binario o dinámico específico para analizar los ejecutables de cada una de estas arquitecturas (aunque provengan del mismo código fuente).

[25] Pros y contras

Las ventajas del análisis de código son evidentes. Se puede automatizar por lo que permite reducir la complejidad asociada a realizar un análisis de código manual. Además, permite establecer una serie de políticas específicas de desarrollo y comprobar que el código cumple dichas políticas de forma automática: formato, llamadas a funciones o librerías prohibidas u obsoletas, errores en la gestión de memoria, etc. En definitiva, es prácticamente imperativo incorporar analizadores de código estático y binario en el desarrollo de software, especialmente si estamos enfocándonos en el desarrollo de software seguro.

Como contrapartida, a veces estas herramientas cometen falsos positivos (detectan errores que en realidad no son tales) o falsos negativos (no detectan problemas existentes en el código). Hay que tener en cuenta que son dependientes del lenguaje de programación y/o de la arquitectura del procesador por lo que podemos encontrarnos con que no tenemos una herramienta adecuada para cierto lenguaje o familia de CPU. Y, si bien son muy útiles en muchos casos, es importante entender que no detectan todos los fallos posibles de programación y que no son efectivos frente a errores de diseño o de concepto.

6 FUZZ TESTING

[27] Introducción

El *fuzz testing* consiste en introducir datos no esperados o con una estructura incorrecta (malformados) en las entradas de una aplicación para comprobar si ésta responde dentro de unos parámetros aceptables. Esta es una herramienta muy potente porque si la aplicación falla con una entrada inesperada podría suponer una vulnerabilidad muy importante (desbordamiento de búfer, denegación de servicio, etc.)

En la figura[3] de la transparencia podemos ver cómo una aplicación que espera una entrada numérica falla al recibir una entrada alfanumérica.

[28] Ventajas y desventajas

Como se ha comentado, cuando se detecta un problema mediante *fuzz testing*, suele suponer un problema muy severo en la aplicación (excepción de memoria, terminación de proceso, etc.). Esto lo convierte en una herramienta muy útil. Además, se puede automatizar de forma muy sencilla, lo que facilita su incorporación al proceso normal de ingeniería del software; por ejemplo, se puede incorporar *fuzz testing* como parte del sistema de pruebas que se realice.

Por el contrario, sólo es capaz de detectar problemas que causen una excepción. Esto excluye a todos los demás tipos de errores: pérdida de información, errores en la criptografía, problemas en el diseño, etc.

Como todas las herramientas automáticas, no es una solución infalible, pero es muy útil en el desarrollo de software seguro.

[29 y 30] Tipos y Realización

Los datos que se utilizan como entrada para el *fuzz testing* pueden ser puramente *aleatorios* o incorporar aspectos de cierta *inteligencia*: depender de la salida de cierta función, incorporar experiencia previa o aspectos asociados al comportamiento esperado, etc.

---

[3] Como curiosidad, en estos diagramas (provienen de la documentación de SDL gratuita de Microsoft) podemos identificar a los atacantes porque nunca llevan corbata, mientras que los usuarios válidos siempre la llevan. *No sabemos a ciencia cierta si esto es extrapolable a la vida real*.

Cabe destacar que ambas clases de datos de entrada pueden coexistir al realizar *fuzz testing* de una aplicación, bien de forma alternativa en función del tipo de entrada o de forma dinámica al dispararse cierto evento o condición.

Si bien la lista de acciones a realizar en un fuzz test (transparencia 30) no requiere una explicación adicional, cabe destacar que las acciones de determinar los puntos de entrada y ordenarlos por riesgo (privilegio y accesibilidad) están íntimamente relacionadas con el análisis de la superficie de ataque del software. Es importante entender que los puntos de entrada pueden ser diversos: formularios, red, ficheros, llamadas remotas, resultados de bases de datos, etc.

Las acciones restantes (introducir datos malformados, análisis de resultados y reparación) se realizarían con cada ciclo de pruebas de la aplicación.

7 AMPLIACIÓN

En vuestro proyecto de prácticas:

- Ya estáis incorporando un analizador de código estático en la mayoría de los casos al utilizar el plugin para Go de editores de código como VSCode o Atom. Tal vez tenga sentido investigar la configuración de dicho plugin para ver qué herramientas está utilizando y qué alternativas existen.
- Existen paquetes para realizar fuzz testing en Go, por ejemplo go-fuzz. Recientemente, con la versión 1.18, Go ha incorporado fuzz testing como parte del lenguaje.
- Otras herramientas interesantes son gosec o Delve.

Otros recursos externos:

Uno de los analizadores de código binario más populares es IDA pro (comercial).

OWASP publica una lista de herramientas útiles para el análisis de código fuente y otros recursos similares.

Existen muchos materiales en O'Reilly Safari (acceso gratuito con la cuenta de la UA) para ampliar, entre otros:

- Libro "Threat Modeling, Designing for Security"
- Libro "Secure and Resilient Software Development"