Testing y buenas prácticas

Josep Vañó Chic

PID_00208417

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Para desarrollar aplicaciones seguras, hay que tener en cuenta la seguridad. Esto implica llevar a cabo una serie de tareas enfocadas de manera específica a la seguridad y que estén integradas en el ciclo de vida del desarrollo de software seguro. Además, es importante conocer las técnicas de código seguro y las buenas prácticas para que las aplicaciones tengan un nivel de seguridad óptimo.

En la fase de desarrollo, se debe incorporar la revisión de código seguro. Esta revisión permite encontrar y solucionar un gran número de problemas de seguridad antes de entrar en la fase de pruebas. Además, también debe llevarse a cabo cada vez que hay un cambio significativo.

En la fase de pruebas es necesario efectuar de manera específica pruebas de seguridad para validar las mitigaciones diseñadas en el modelado de amenazas. A diferencia de las pruebas funcionales, las pruebas de seguridad tratan de detectar si, por ejemplo, se puede suplantar la identidad de otro usuario, si se pueden manipular los datos, si es posible acceder a datos a los que no se ha de tener acceso, si se puede tener más privilegios debido al uso malicioso de la aplicación, si se producen desbordamientos, y un largo etcétera. Es decir, en esta fase se trata de pensar como un *hacker* para intentar encontrar vulnerabilidades por donde atacar la aplicación.

Las técnicas de código seguro y las buenas prácticas son un conjunto de consideraciones que deben tenerse en cuenta en la programación de código seguro, y que conviene aplicar y considerar para mitigar lo máximo posible las amenazas.

Objetivos

Al finalizar la lectura de este módulo, los estudiantes habrán conseguido las competencias siguientes:

1. Conocer las técnicas de código seguro básicas sobre criptografía, *SQL injection* y *cross site scripting*.

2. Conocer las tareas que se han de llevar a cabo en la revisión de código en una programación de código seguro.

3. Conocer las tareas que se tienen que efectuar en las pruebas de seguridad.

4. Conocer las buenas prácticas en la programación de código seguro.

1. Técnicas de código seguro

Los tipos de ataques que pueden sufrir las aplicaciones son muy distintos, y para mitigar las amenazas hay que utilizar técnicas de código seguro. Este apartado se centra en llevar a cabo una aproximación en tres ámbitos en concreto: la criptografía, la *SQL injection* y el *cross site scripting*.

1.1. Criptografía

En muchas aplicaciones, es necesario almacenar y recuperar información confidencial, como por ejemplo contraseñas, cadenas de conexión y otros datos confidenciales a los que no han de tener acceso otros usuarios o aplicaciones. Esta característica se puede conseguir por medio de soluciones criptográficas estándar.

Cada plataforma (MS Windows, Linux, Mac OS, etc.) proporciona alguna solución para resolver este problema, como por ejemplo:

> CryptoAPI, CAPICOM, DPAPI, WebCryptoAPI, Keychain Services API on Mac OS, GNOME-Keyring, KWallet, etc.

Se trata de librerías que pueden utilizar los desarrolladores en sus aplicaciones, y que permitirán a los usuarios crear e intercambiar documentos y otros datos en un entorno seguro, sobre todo en los medios de comunicación no seguros como Internet.

Hay que tener en cuenta que el uso de la criptografía se tiene que considerar como una capa dentro de la defensa en profundidad, es decir, conviene añadir más elementos de seguridad a los ficheros o datos con información sensible, como por ejemplo la ocultación y los privilegios de acceso.

En la fase de diseño, se tiene que definir en qué puntos es preciso utilizar la criptografía y qué tipo es la más adecuada en función de la clase de amenaza. A continuación se muestran algunas de las soluciones que se pueden utilizar en función del tipo de amenaza.

Amenaza	Técnica de cifrado	Ejemplos de algoritmos
Information disclosure	Criptografía simétrica	RC2, RC4, DES, 3DES, AES
Tampering	Funciones *hash*, firma digital	SHA-1, SHA-256, SHA-384,SHA-512, MD4, MD5, HMAC, RSA digital signatures, DSS digital signatures, XML Dsig
Spoofing	Autenticación de datos	Certificados de clave pública y firma digital

1.2. *SQL injection*

Una de las vulnerabilidades que ocupan los primeros lugares del *top* 10 es la *SQL injection*. Se trata de una vulnerabilidad fácil de explotar y, dado que su finalidad es el acceso a la base de datos, el impacto puede ser muy grave. Hay que tener en cuenta que para llevar a cabo este tipo de ataque no se necesita tener conocimientos de tecnología, ni de seguridad, ni herramientas especiales: basta con introducir una serie de caracteres en el navegador cuando se accede al formulario de una web.

Página web

Podéis ver el *top* 10 de las vulnerabilidades en: The Open Web Application SecurityProject (*OWASP*).

Supongamos que en algún punto de un formulario de una web se pide el código de cliente para acceder a los datos del mismo. Una vez introducido el código de cliente, la aplicación para obtener los datos en la base de datos lleva a cabo una sentencia SQL como la siguiente:

```
String sql = "SELECT * FROM clientes WHERE CodigoCliente = ' " + pCodigoCliente + "'"
```

Donde la variable *pCodigoCliente* contiene el valor del código que se ha introducido en el formulario. Si el valor *pCodigoCliente = A1234*, la sentencia SQL quedaría de la manera siguiente:

```
SELECT * FROM clientes WHERE CodigoCliente = 'A1234'
```

Esta sentencia retornaría concretamente los datos del cliente A1234.

Sin embargo, si el usuario introduce:

```
Codigo Cliente: A1234' OR 1=1 - -
```

En este caso, la sentencia quedaría construida de la manera siguiente:

```
SELECT * FROM clientes WHERE CodigoCliente = 'A1234' OR 1=1 - -  '
```

Con esta sentencia, la base de datos retornaría los datos de todos los clientes, puesto que siempre se cumpliría la condición. Hay que tener en cuenta que, por ejemplo, en las bases de datos Microsoft SQL y Oracle Database, con los dos guiones " - - " se interpreta que todo lo que hay a continuación de la misma línea es un comentario y, por lo tanto, no se ejecuta.

Como se puede observar, debido a la vulnerabilidad en la entrada de información se puede provocar la alteración de la funcionalidad del formulario y obtener información de manera malintencionada. Supongamos otro ejemplo en el que se piden un usuario y una contraseña para acceder a la aplicación. La consulta SQL para verificar los datos podría ser de este modo:

```
"SELECT count(*) FROM usuarios WHERE user = '" + pUser + "' AND password = '" + pPassword + "'"
```

Una entrada normal, en la que por ejemplo el usuario sea *admin* y la contraseña fuera *mykey*, construiría la consulta siguiente:

```
SELECT count(*) FROM usuarios WHERE user = 'admin' AND password = 'mykey'
```

En este caso, si en la entrada de usuario se introduce la sentencia, la consulta se construiría de esta manera:

```
admin'  - -
```

Como se puede observar, la consulta no verificaría la contraseña, ya que la base de datos interpretaría la parte `' AND password = ''` como un comentario y, en consecuencia, se podría acceder como usuario *admin* sin conocer la contraseña y así suplantar su identidad, con todo el peligro que esto supone.

Un caso similar sería introducir:

```
' OR 1=1  - -
```

En este caso, también se podría eludir la autenticación, puesto que la sentencia construida sería de este modo:

```
SELECT count(*) FROM usuarios WHERE user = '' OR 1=1  - - ' AND password = ''
```

En el siguiente ejemplo, se accedería a la aplicación suplantando la identidad de algún usuario:

```
' OR 1=1 AND user  > 'A'   - -
```

```
SELECT count(*) FROM usuarios WHERE user = '' OR 1=1  AND user > 'A'  - - ' AND password = ''
```

Con la *SQL injection* no solo se puede obtener información o eludir la autenticación, sino que también es posible insertar información, puesto que si se introduce:

```
' OR 1=1 ; INSERT  INTO usuarios (user, password)  VALUES ('josep' , 'mykey')   - -
```

La sentencia construida sería de este modo:

```
SELECT count(*) FROM usuarios WHERE user = '' OR 1=1 ;
INSERT  INTO usuarios (user, password)  VALUES ('josep' , 'mykey')   - - ' AND password = ''
```

El hecho de haber introducido un punto y coma hace que lo que sigue a continuación se interprete como una nueva sentencia que también se ejecutaría. En este ejemplo, en la segunda sentencia el atacante insertaría a un usuario en la base de datos. Esto implica que el atacante podría acceder a la aplicación con el usuario que ha creado.

Como se ha podido observar en este ejemplo, por medio de la inyección de código se pueden alterar las sentencias SQL, con todo lo que esto supone, como por ejemplo, extraer, insertar y modificar datos o incluso ejecutar órdenes en el servidor de la base de datos.

Para evitar que la aplicación sea atacada con *SQL injection*, hay varias opciones, a pesar de que no todas ofrecen un grado de seguridad óptimo. A continuación se describen las posibles soluciones:

1) Filtrar la entrada de datos

Hay dos formas de filtrado: listas blancas (*white lists*), en las que se bloquea todo salvo lo que de manera explícita se permite en la lista; y las listas negras (*black lists*), que permiten dejar pasar todo excepto un conjunto de caracteres definidos que pueden causar problemas. Las listas pueden ser complicadas de definir y mantener, y tampoco aseguran del todo que sean efectivas (por ejemplo, el apóstrofe se utiliza en la gramática de varios idiomas y también en algunos nombres, como por ejemplo *O'Connor*). Aun así, conviene utilizar este método de listas blancas o negras junto con otros mecanismos de defensa para que forme parte de la defensa en profundidad y evitar la *SQL injection*.

Otro problema que presenta el sistema de filtrado en esta vulnerabilidad es que filtrar o convertir los apóstrofes no soluciona la *SQL injection* en los parámetros numéricos.

Supongamos la consulta SQL siguiente:

```
SELECT nif, nombre FROM clientes WHERE codigo = 43000001
```

El usuario podría introducir esto:

```
430000001 AND 1=0 UNION SELECT  user, password FROM usuarios
```

La construcción de la consulta resultaría de este modo:

```
SELECT nif, nombre FROM clientes WHERE codigo = 43000001 AND 1=0 UNION SELECT  user,
password FROM usuarios
```

El resultado de esta consulta sería la lista de usuarios y sus contraseñas.

2) Construcción de sentencias SQL seguras

El modo más seguro de evitar la *SQL injection* son las sentencias parametrizadas (*parameterized commands*). A pesar de que la forma de indicar el parámetro puede variar en función de la base de datos, un ejemplo podría ser este:

```
SELECT count(*) FROM usuarios WHERE user = ?  AND password = ?
```

En este caso, la construcción de la sentencia no se lleva a cabo por código en la aplicación, sino que la hace el motor de la base de datos. En función del lenguaje de programación, las instrucciones y la forma de pasar los parámetros pueden variar, pero en cualquier caso no se utiliza el código de programación a partir de concatenar cadenas y valores, sino que se pasan los valores de los parámetros a la base de datos y es la base de datos la que construye la sentencia.

1.3. *Cross site scripting* (*XSS*)

Esta también es una de las vulnerabilidades más extendidas. En este caso el atacante inyecta código que el navegador puede interpretar, ya sea con marcas HTML, Javascript, etc.

Hay dos tipos de vulnerabilidades XSS:

- Persistente: en este caso el atacante consigue grabar datos en el servidor, de modo que cuando un usuario accede a la web, es víctima del XSS.

- No persistente: este tipo se presenta cuando un usuario accede a una web con una URL especialmente modificada en la que se encuentran los parámetros del ataque.

Un ejemplo de cómo se puede llevar a cabo este ataque podría ser el siguiente.

> Supongamos una web en la que se puede escribir texto, por ejemplo, un libro de visitas o un foro. Si en el texto escribimos `<script> alert ('hello world'); </script>`, el navegador lo interpretará como una orden y la ejecutará haciendo aparecer un mensaje, en este caso inofensivo. El problema radica en que en lugar de la orden `alert` se pueden inyectar otras órdenes con fines maliciosos.

Protegerse del XSS es realmente complicado, puesto que no hay una solución universal. Uno de los documentos interesantes para prevenir el XSS es la guía *XSS Filter Evasion Cheat Sheet*, publicada por *The Open Web Application Security Project* (*OWASP*).

Como mínimo, para prevenir este ataque, en el diseño de la aplicación se puede hacer lo siguiente:

- Validar todas las entradas en los formularios: verificar que el tipo de datos y la longitud de cada campo se corresponde con lo que se espera y filtrar caracteres especiales que puedan resultar peligrosos.

- Filtrar determinadas palabras como SCRIPT, OBJECT, APPLET, EMBED, FORM, etc.

Una buena práctica en el filtrado consiste en aplicar el filtrado basado en listas blancas, de modo que se deniega todo excepto lo que de manera explícita se quiere permitir. Lo que no se tiene que hacer es filtrar a partir de los elementos

de peligrosidad que conocemos, puesto que resulta muy difícil que los conozcamos todos, y con sus variantes. Además, es posible que aparezcan elementos y vectores de ataque nuevos, de modo que el filtrado no sería eficiente.

2. Revisión de código seguro

La revisión de código se efectúa en la fase de desarrollo y permite encontrar y solucionar un gran número de problemas de seguridad antes de entrar en la fase de pruebas.

La revisión del código se tiene que hacer cada vez que se produzca un cambio significativo, en lugar de esperar hasta el final del proyecto y llevar a cabo la revisión de una sola vez. De este modo, se puede uno centraren lo que se ha cambiado en lugar de encontrar todos los problemas de seguridad al mismo tiempo.

Hay herramientas que se pueden utilizar para llevar a cabo esta tarea, pero siempre es necesario complementarla con una revisión efectuada por un técnico. Las herramientas no entienden el contexto, lo que es importante para una revisión de código seguro. Aun así, las herramientas son útiles para la evaluación de grandes cantidades de código y para detectar posibles vulnerabilidades, pero posteriormente será una persona quien deba verificar los resultados, determinar si se trata de un problema real y si es explotable y valorar el riesgo que supone para la organización. Además, quizá las herramientas no detecten todos los problemas de seguridad y, por lo tanto, es necesario igualmente una revisión humana.

El equipo de revisión

Encontramos varios criterios sobre cómo confeccionar un equipo de revisión. Pese a todo, la mayoría de los criterios coinciden en que lo idóneo es que se trate de un equipo reducido.

Una de las modalidades es la que se denomina *peer review*, en la que dos desarrolladores revisan por separado el código que han efectuado.

Una modalidad similar es la formada por el desarrollador y uno o dos revisores destinados a esta tarea. El desarrollador hace una primera revisión, y después son los revisores quienes la hacen. En esta etapa, el desarrollador debe colaborar explicando qué hace el código, sobre todo en las secciones donde este es más complejo.

Según la dimensión y el tipo del software, el equipo estará formado por los roles siguientes:

- **Analista de negocio:** describe los objetivos de seguridad desde el punto de vista de negocio.

- **Arquitecto de aplicaciones:** describe los objetivos de seguridad desde el punto de vista de la arquitectura del software y del sistema.

- **Desarrollador:** describe los detalles del código, lo revisa y localiza los *bugs* o agujeros de seguridad.

- **Revisor:** revisa el código y localiza los *bugs*.

La modalidad que se adoptará irá en función del tipo y la magnitud del software, del tipo de organización o de la dimensión del equipo del departamento, pero en lo que sí coinciden las distintas modalidades es en que la revisión de código la tienen que realizar equipos reducidos, con un máximo de cuatro personas y, si la dimensión del proyecto lo requiere, es necesario crear varios equipos reducidos de revisión. Además, la formación de estos equipos en el proceso de revisión de código permite al equipo de desarrollo compartir experiencias y buenas prácticas de seguridad que pueden prevenir problemas de seguridad en el futuro.

2.1. Fases en la revisión de código seguro

- Identificar los objetivos de la revisión.
- Análisis preliminar.
- Revisión de código seguro.

2.1.1. Identificar los objetivos de la revisión

Para saber qué se tiene que revisar, es preciso establecer los objetivos y las limitaciones para la revisión de código, puesto que para llevarla a cabo se tiene que saber qué se está buscando. Si se empieza la revisión sin el establecimiento de objetivos, aumentan las posibilidades de quedar abrumados por el código y disminuyen las posibilidades de encontrar problemas de seguridad.

Hay que tener en cuenta que en el modelado de amenazas se han identificado y documentado las amenazas y, al mismo tiempo, se ha analizado y efectuado un diagrama de flujo de datos. Por lo tanto, si se ha hecho el modelado de amenazas, ya tenemos el punto de partida. Hay que recordar que el modelado es un proceso iterativo y que durante la revisión de código quizá se encontrarán nuevas amenazas. Así pues, es muy importante que estas nuevas amenazas encontradas sean incorporadas en el modelado de amenazas para tener el modelo actualizado y hacer los controles posteriores, como por ejemplo en la fase de pruebas, de una manera correcta.

En la determinación de los objetivos de la revisión, consideraremos lo siguiente:

- ¿Cuáles de las amenazas identificadas en el modelado afectan al código que se está revisando? En este punto se pueden separar las amenazas que han sido mitigadas y las que no.

- ¿Qué errores comunes de codificación pueden estar presentes en el código que se está revisando? Para la revisión, es muy útil disponer de una lista de errores comunes que ayude a concentrarse en puntos en concreto y, al mismo tiempo, a encontrar errores de manera más fácil. Esta lista, obviamente, se tiene que ir actualizando con los nuevos errores que se encuentren y que sean reiterativos o habituales.

2.1.2. Análisis preliminar

En este punto se lleva a cabo un análisis del código para encontrar posibles problemas y/o puntos débiles de seguridad. Para hacer este análisis preliminar podemos utilizar dos tipos de análisis, teniendo en cuenta que uno no excluye al otro, de modo que es posible combinar ambos:

- Revisión automática.
- Revisión manual.

1) Revisión automática

Una revisión automática efectuada con una herramienta de análisis estático puede dar como resultado un conjunto de falsas alarmas de seguridad y, al mismo tiempo, no detectar todos los problemas reales. Aun así, también permite detectar problemas de seguridad que quizá no se encontrarían en una revisión manual.

2) Revisión manual

Teniendo en cuenta que posteriormente se efectuará una revisión de código en materia de seguridad de una manera más exhaustiva, en el análisis preliminar se trata de efectuar una revisión que responda, por ejemplo, al tipo de preguntas siguiente:

- Validación de entrada: ¿se hace validación de entrada?, ¿dónde se hace, en el cliente, en el servidor o en ambos lugares?

- Autenticación y autorización de los usuarios: ¿qué mecanismo se utiliza en la autenticación y la autorización?

- Tratamiento de excepciones: ¿hay áreas de código con un tratamiento de excepciones demasiado denso o escaso?

- Código complejo: ¿hay áreas de código complejo?

2.1.3. Revisión de código seguro

La frontera entre una revisión manual en un análisis preliminar y la revisión de código seguro no es una línea marcada de una manera categórica. De hecho, se podría considerar como una revisión en espiral en la que se empieza por una visión general y se va profundizando en la revisión.

Hay que tener en cuenta que en la revisión participa personal adicional, además del autor del código. Incluso se puede dar la circunstancia de que el autor del código no estuviera.

Lo primero que debe hacer el equipo de revisión es conocer el flujo básico de la aplicación y sus funcionalidades. Una vez se conoce cómo funciona la aplicación, hay que hacer referencia al modelado de amenazas que se ha creado previamente y al diagrama de flujo de datos. Esto hará que la revisión se pueda enfocar hacia las partes de la aplicación donde la seguridad es más crítica.

Cuando se tienen clasificadas las partes de código en función del riesgo, hay que priorizar y aplicar un esfuerzo proporcional al nivel de riesgo. Por ejemplo, en un área de código de alto riesgo, se llevará a cabo una revisión detallada, línea por línea; en un módulo de menos riesgo se puede hacer una revisión menos detallada; y en una zona de poco riesgo, se podrían examinar solo las llamadas a funciones que supongan algún tipo de riesgo.

Para hacer la revisión, se tienen que combinar las técnicas siguientes:

- **Análisis del control del flujo:** en este análisis se revisan paso a paso las condiciones lógicas, los condicionales y los bucles iterativos en el código, teniendo en cuenta las diferentes ramas que pueden existir.

- **Análisis del flujo de datos:** en este análisis se rastrean los datos desde los puntos de entrada hasta los puntos de salida. Partiendo desde los puntos de inicio, hay que hacer el recorrido hasta el final del proceso. Si un punto de partida tiene demasiadas ramificaciones, conviene crear una nueva ramificación como punto de partida pero teniendo en cuenta dónde está el punto de partida inicial. En el análisis del flujo de datos es necesario prestar especial atención a las zonas donde se analizan datos, a los puntos donde se pueden enviar a varios puntos de salida y también a si se tratan con el nivel de confianza adecuado.

Hay que evitar el código excesivamente complicado. Cuanto más complicado sea, más posibilidades hay de que tenga errores de seguridad. Al mismo tiempo, corregir código excesivamente complicado implica una alta probabilidad de que se introduzcan nuevos errores en los cambios efectuados.

En general, hay ciertos errores que la mayoría de los programadores cometen e, incluso, es posible encontrar patrones de errores para cada programador en concreto.

Un aspecto que también se tiene que considerar en la revisión del código son las llamadas a funciones no seguras de las API. Por ejemplo, funciones del lenguaje C como strcpy y strcat, no son seguras puesto que si no se filtran correctamente los datos, pueden provocar un *buffer overrun*. Así pues, en la revisión del código se tendrían que cambiar por sus funciones equivalentes y seguras. En general, se debe evitar que se produzca cualquier tipo de desbordamiento, puesto que los *exploits* pueden aprovecharlo.

3. Pruebas de seguridad

Las pruebas de seguridad se hacen para validar las mitigaciones diseñadas en el modelado de amenazas. A diferencia de las pruebas funcionales, las pruebas de seguridad tratan de detectar si, por ejemplo, se puede suplantar la identidad de otro usuario, si es posible manipular los datos, si se puede acceder a datos a los que no se ha de tener acceso, si se tienen más privilegios debido al uso malicioso de la aplicación, si se producen desbordamientos, etc. Es decir, en esta fase se trata de pensar como un *hacker* para intentar encontrar vulnerabilidades por donde atacar la aplicación.

Las pruebas de seguridad se tienen que hacer de manera posterior a las pruebas funcionales. Un error funcional también representa un potencial problema de seguridad. Por lo tanto, primero es necesario resolver los posibles problemas funcionales, y después, los de seguridad.

Esta fase se tiene que prever desde el inicio del proyecto, y es necesario involucrar al equipo de pruebas desde la fase del diseño y el modelado de amenazas, así como revisar las especificaciones de problemas de seguridad.

Para hacer las pruebas de seguridad, se llevará a cabo el proceso siguiente:

- Descomponer la aplicación en componentes.
- Identificar las interfaces de los componentes.
- Clasificar las interfaces por su potencial vulnerabilidad.
- Determinar las estructuras de datos utilizados por cada interfaz.
- Encontrar problemas de seguridad mediante la inyección de datos transformados.

1) Descomponer la aplicación en componentes

La lista de componentes en el sistema, los tipos de amenazas para cada componente (STRIDE) y el riesgo de amenazas (DREAD) son elementos que previamente se han elaborado en el modelado de amenazas.

2) Identificar las interfaces de los componentes

Se trata de determinar las interfaces expuestas por cada componente, que pueden o no estar expuestas en el modelado de amenazas. Aun así, las interfaces expuestas por cada componente se tendrían que encontrar en las especificaciones funcionales. En caso contrario, deberán obtenerse con la lectura del có-

digo o preguntándolo al equipo de desarrollo. En el caso de que las interfaces no se hubieran documentado anteriormente, se aprovechará este momento para hacerlo.

A continuación se muestran algunos ejemplos de interfaces y tecnologías de transporte.

- *TCP and UDP sockets*.
- *Wireless data*.
- *NetBIOS*.
- *Mailslots*.
- *Dynamic data exchange (DDE)*.
- *Named pipes*.
- *Shared memory*.
- *The clipboard*.
- *Local procedure call (LPC)* y remote procedure call (RPC) interfaces.
- *COM* (métodos, propiedades y acontecimientos).
- *ActiveX controls* y applets.
- *EXE* y DLL functions.
- *HTTP requests & responses*.
- *Simple object access protocol (SOAP) requests*.
- *Console input*.
- *Command line arguments*.
- *Dialog boxes*.
- Ficheros.
- *Database access technologies* (incluidos OLE DB y ODBC).
- *Database stored procedures*.
- *Environment variables*.
- *LDAP, active directory*.
- *Hardware devices*.

3) Clasificar las interfaces por su potencial vulnerabilidad

La clasificación inicial de los riesgos tiene que provenir del modelo de amenazas. En caso de que sea necesario, el siguiente paso consiste en añadir más granularidad y precisión de cara a la ejecución de las pruebas. La clasificación estará ordenada según el nivel de riesgo. De este modo, se podrá tener una aproximación de la cantidad de interfaces vulnerables que deberán probarse más a fondo.

4) Determinar las estructuras de datos utilizadas por cada interfaz

El siguiente paso es determinar los datos que acceden a cada interfaz. Se trata de los datos que se modificarán para exponer los errores de seguridad. A continuación se muestra un ejemplo de fuentes de datos según la interfaz.

Interfaz	Datos
Sockets, RPC, named pipes, NetBIOS	Datos que llegan a través de la Red.
Ficheros	El contenido de los ficheros.
Active directory	Nodos en el directorio.
HTTP data	Cabeceras HTTP, entradas de formulario, *query strings*, etc.

Para llevar a cabo las pruebas de seguridad, se deberán construir los casos de pruebas. El enfoque STRIDE sobre los tipos de amenazas en el modelado de amenazas ayuda a determinar los tipos de pruebas en cada caso.

A continuación, se muestran algunos ejemplos de técnicas de pruebas para verificar las técnicas de mitigación.

Tipo de amenaza	Técnicas de pruebas
Spoofing	• Intentar forzar el uso de la aplicación sin autenticación. • ¿Se pueden ver las credenciales de usuario en el tráfico de la red o en un almacenamiento persistente? • ¿Pueden los *tokens* de seguridad –por ejemplo, mediante una cookie– ser reproducidos para omitir la fase de autenticación?
Tampering	• Intentar eludir la autorización. • ¿Es posible manipular los datos y volver a crear la función resumen (*rehash*) de los mismos? • Crear funciones resumen (*hash*) inválidas y firmas digitales y posteriormente verificar la autenticidad.
Repudiation	• ¿Hay condiciones que evitan la creación de un registro (*log*) o auditoría? • ¿Puede hacerse de algún modo que los datos del registro (*log*) se graben de manera incorrecta? Por ejemplo, añadiendo caracteres de salto de línea, retorno o de final de fichero en una petición válida. • ¿Hay algún tipo de acción que omita algún control de seguridad?
Information disclosure	• Intentar acceder a datos a los que solo se tendría que acceder con privilegios más elevados, tanto por medio de datos persistentes en ficheros y en bases de datos como a través del tráfico de la red. Los *sniffers* de red son una buena herramienta para esta tarea. • Inspeccionar si se han grabado datos sensibles en el disco, por ejemplo en ficheros temporales, *cookies*, etc.
Denial of service (DOS)	• Inundar de peticiones al servidor para intentar que quede colapsado y no pueda dar servicio. • ¿Enviar datos con dimensiones y/o formatos incorrectos provoca que se pare el servicio? • ¿Hasta qué punto afectan factores como, por ejemplo, poco espacio disponible en el disco, capacidad de memoria, sobrecarga del procesador o limitación de recursos en general, y provocan que falle la aplicación?
Elevation of privilege	• ¿Se ejecutan aplicaciones o servicios del sistema con privilegios más elevados de los necesarios? • ¿Puede un proceso forzar, por ejemplo, la carga de Command Shell y ejecutar procesos con privilegios más elevados?

En el modelado de amenazas, cada amenaza debe tener su plan de pruebas y prever la ejecución de una o varias pruebas.

En cada test hay que prever los parámetros que determinen el éxito o no del mismo, así como verificar si la función testeada ha fallado o no.

5) Encontrar problemas de seguridad mediante la inyección de datos transformados

En este caso se trata de introducir en cada vía de entrada caracteres, formatos de datos y datos no previstos, datos transformados, datos fuera de rango, etc. para intentar que la aplicación altere su comportamiento o dé un error.

Los datos transformados son utilizados con frecuencia por los atacantes para provocar cualquier tipo de desbordamiento y aprovecharlo para introducir un *exploit* a través de esta vulnerabilidad. Además de los ataques que se pueden hacer provocando un desbordamiento, también es posible recibir otros ataques a través de las vías de entrada de datos y formatos no previstos, como por ejemplo la *SQL injection* y el *cross site scripting*.

Para hacer este test será necesario preparar una batería de datos, que se introducirán por las vías de entrada para comprobar si la aplicación lleva a cabo correctamente la validación de datos introducidos. Esta batería de datos deberá ser tan amplia como resulte posible para verificar que un atacante no aproveche estos puntos de entrada.

4. Buenas prácticas

Hay técnicas eficaces que los desarrolladores pueden utilizar para prevenir los ataques. Algunas de estas técnicas tendrían que ser utilizadas cuando se desarrolla cualquier aplicación en cualquier plataforma. Ciertos ataques se pueden mitigar mediante métodos específicos, entre ellos los siguientes:

- Ejecutar con privilegios mínimos.
- Reducir la superficie de ataque.
- Instalar las mínimas características por defecto.
- Validar la entrada del usuario.
- Defensa en profundidad.
- Considerar los sistemas externos como inseguros.
- No confiar en la seguridad y ocultar datos.
- No revelar información de los errores.
- No hacer decidir a los usuarios.
- Aprender de los errores.
- Solucionar los problemas de seguridad de manera correcta.

1) Ejecutar con privilegios mínimos

Ejecutar de manera estricta, justo con los privilegios necesarios para llevar a cabo la tarea o funcionalidad en cuestión.

Por ejemplo, si se produce un error por desbordamiento, como en el caso de un *buffer overrun*, y se está ejecutando en un contexto de administrador, un *hacker* puede sobrescribir la dirección de retorno y ejecutar su código malicioso con privilegios de administrador.

Ciertamente, lo más fácil para los desarrolladores es crear aplicaciones con permisos de administrador, porque así no hay errores de permisos, pero esta es una mala práctica en la que no se ha de caer. También lo es dar permisos de administrador de manera temporal, con la intención de cambiar posteriormente los privilegios por los que realmente correspondan. Hacer esto supone un alto riesgo, puesto que por falta de tiempo, por olvido o por cualquier otra circunstancia quizá no se haga el cambio y se deje la función con privilegios de administrador.

2) Reducir la superficie de ataque

Con frecuencia, los atacantes explotan las debilidades de los distintos servicios de la plataforma, como por ejemplo el *index service* del *Internet information server* (*IIS*), así como las interfaces que exponen las aplicaciones.

En este caso, una buena práctica consiste en deshabilitar todos los servicios que no se utilicen y limitar el número de interfaces expuestas por la aplicación.

3) Instalar las mínimas características por defecto

Minimizar la superficie de ataque también significa definir una instalación segura de la aplicación. Cuantas más características estén activadas por defecto, más se aumenta la posibilidad de una violación de seguridad. Por lo tanto, resulta aconsejable activar el mínimo de características posibles y asegurarse de que estas son seguras.

Se trata, por tanto, de activar las características mínimas y adecuadas para la mayoría de los usuarios, y de dejar desactivadas las funciones que se utilicen con menos frecuencia o por un grupo reducido de usuarios, para reducir la exposición de funcionalidades. Si una función no se está ejecutando, no puede ser vulnerable al ataque.

4) Validar la entrada del usuario

Nunca se tiene que confiar en los datos introducidos por el usuario, y hay que considerar que toda entrada es sospechosa hasta que se demuestre lo contrario. Por lo tanto, la aplicación tiene que validar los datos introducidos por el usuario antes de utilizarlos.

También hay que considerar la posibilidad de que un *hacker* pudiese eludir la validación junto al cliente de una aplicación. Por lo tanto, la validación de datos no solo se tiene que hacer en el punto de entrada de datos sino que también debe llevarse a cabo en otros puntos, por ejemplo, junto al servidor, antes de grabarlos en la base de datos, etc.

Hay varios métodos para validar los datos y que al mismo tiempo son compatibles entre sí, es decir, que se puede utilizar más de uno para validar una misma entrada.

- Aceptar valores válidos y denegar el resto en lugar de denegar valores inválidos y aceptar el resto.
- Invalidar entradas que contienen caracteres o palabras clave que pueden ser peligrosos (\, <, >, `script`, `object`, `insert`, ...).
- Invalidar entradas que excedan una cierta longitud.
- Restringir las entradas utilizando controles de validación y expresiones regulares.

5) Defensa en profundidad

No se tiene que confiar en una sola capa de defensa. Se puede aumentar la seguridad de un sistema con la construcción de múltiples capas de defensa.

Muchas aplicaciones están diseñadas y escritas de modo que confían totalmente en el cortafuego, pero esto no es suficiente. El hardware puede fallar o desconfigurarse, haberse configurado de manera incorrecta, etc. Por lo tanto, si se ha confiado en una sola capa de defensa, el riesgo de ser atacados con éxito es muy elevado.

A pesar de que se apliquen varias capas de seguridad en un ámbito de hardware, tampoco se tiene que dejar la responsabilidad de la seguridad en los demás; es decir, la última capa de seguridad tiene que estar en la propia aplicación, incluso en cada capa de la aplicación.

6) Considerar los sistemas externos como inseguros

No se tiene el control completo de los datos que se reciben desde otro sistema; podrían ser inseguros y una fuente de ataque. Hasta que se pueda demostrar lo contrario, todas las entradas externas son un ataque potencial. Los datos de entrada no solo pueden provenir de un usuario, también pueden hacerlo de un servidor externo que quizá sea un atacante.

En una arquitectura cliente-servidor, los clientes pueden ser redirigidos de varios modos y de manera malintencionada hacia otro servidor. Así pues, al escribir código junto al cliente (*client-side code*) no se debe suponer que este se está comunicando con el servidor correcto.

Desde el punto de vista del servidor, tampoco se tiene que suponer que los datos o las peticiones se reciben desde la interfaz de un cliente web. Los atacantes pueden enviar datos maliciosos y eludir la interfaz del cliente.

7) No confiar en la seguridad y ocultar datos

En ocasiones debemos guardar claves de cifrado u otro tipo de datos de alto riesgo, y esto se hace guardando esta información de manera oculta. El problema radica en que en el momento en que se guarda información en el disco, aunque sea oculta, un *hacker* la puede encontrar; de hecho, los *hackers* saben que se utiliza la técnica de ocultación de datos, por lo que buscarán precisamente estos datos porque seguro que son importantes.

Ocultar datos es una buena defensa, pero no tiene que ser la única que se ha de aplicar. La ocultación solo se tendría que utilizar como una pequeña parte de una defensa integral en la estrategia de la defensa en profundidad.

8) No revelar información de los errores

Si la aplicación da un error, no se tiene que proporcionar información de este a los usuarios finales. Se ha de evitar dar información (por ejemplo, de qué instrucción, línea del código, tipo de error, etc.), es decir, cualquier información técnica. Como información también se entiende dar explicaciones del

porqué del error, el motivo o cómo se ha producido. Esta información puede ser aprovechada por un atacante para provocar el error y atacar el sistema. Para capturar información de los errores, se puede guardar esta información en un fichero de registro de acontecimientos (*log*). El manejo de errores debe hacerse de manera estructurada, de tal modo que los detalles del error no se propaguen de vuelta al usuario.

9) No hacer decidir a los usuarios

Hay muchas aplicaciones en las que el usuario tiene que tomar decisiones de seguridad. Debemos tener en cuenta que la mayoría de los usuarios no entienden sobre seguridad ni sobre cuestiones técnicas del hardware ni del software y, además, tampoco desean conocer sobre el asunto. Lo que quieren los usuarios es que sus datos y los ordenadores estén protegidos sin tener que tomar decisiones complejas. También hay que recordar que si se les da varias opciones para elegir, la mayoría seleccionarán el botón por defecto, aunque no sepan muy bien lo que se les pregunta.

Aun así, si se prefiere que el usuario tome alguna decisión, hay que tener en cuenta que no es ningún técnico en informática y, por lo tanto, la redacción de la pregunta debe ser simple, fácil de entender y sin tecnicismos. Además, no hay que saturar el cuadro de diálogo con explicaciones innecesarias y, si es posible, debemos añadir un botón de ayuda mediante el cual se puedan dar explicaciones más detalladas en un vocabulario comprensible para el usuario.

10) Aprender de los errores

Si no se aprende de los errores, es muy probable que se vuelvan a cometer en un futuro.

En ocasiones, un mismo problema de seguridad sucede varias veces. Si ya se ha tenido la experiencia de un problema en concreto, ¿por qué vuelve a repetirse?

Aprender de los errores es un ejercicio: hay que tener el hábito de documentar y actualizar el modelado de amenazas, de crear una base de datos con información sobre las amenazas, de concienciar a todo el equipo y de formularse preguntas como las siguientes:

- ¿Cómo se produjo el error de seguridad?
- ¿Se repite el error en otras áreas del código?
- ¿Cómo se podría haber evitado el error?
- ¿Qué se puede hacer para asegurarse de que este tipo de error no vuelva a suceder en un futuro?
- ¿Es necesario actualizar algún tipo de herramienta o de análisis?

11) Solucionar los problemas de seguridad de manera correcta

Si se encuentra una vulnerabilidad de seguridad en el código o en el diseño, se tiene que arreglar el problema y buscar problemas similares en la aplicación. Es muy probable que se encuentren otros semejantes y que el mismo problema esté también en otros puntos de la aplicación.

De manera similar, si se encuentra un conjunto de defectos que siguen un mismo patrón, hay que llevar a cabo los pasos necesarios para crear mecanismos de defensa que reduzcan o solucionen este tipo de problemas, en lugar de limitarse a resolver los problemas por partes.

Para empezar, si se encuentra un problema de seguridad, se tiene que arreglar en el lugar más cercano posible de la ubicación de la vulnerabilidad. Por ejemplo, si el error está en una función denominada *DataControl*, y la solución se puede aplicar en varios puntos del programa (filtrando los datos a la entrada del usuario, en la lógica del programa o dentro de la función), el primer lugar donde se debe hacer es en la función: si un atacante elude una parte del programa y accede directamente a la función, el sistema continúa siendo vulnerable a un ataque. Posteriormente, también se tendrían que arreglar las otras partes del programa para que no lleguen datos incorrectos a la función.

Bibliografía

Howard, M; LeBlanc, D. (2002). *Writing Secure Code, Second Edition*. Redmond Washington: Microsoft Press.

Microsoft (2013). *Testing forSecurability* [en línea]. http://msdn.microsoft.com

OWASP. The Open Web Application Security Project (2013). http://www.owasp.org/

OWASP. The Open Web Application Security Project (2013). *Security Code Review in the SDL* [en línea]. https://www.owasp.org/index.php/security_code_review_in_the_sdlc