4. Seguridad en Applicaciones Web. Introducción

4.1. Introducción y objetivos

Este tema estudia las medidas que son necesarias para implementar la seguridad de las aplicaciones web para que, una vez desplegadas *online*, se comporten de la forma esperada, tanto por parte de los propietarios como de los usuarios, y puedan mitigar cualquier ataque.

Se pretende sentar las bases de la seguridad de las aplicaciones *online* e introducir las actividades de seguridad que se realizan antes y una vez desplegada la aplicación *online*, como, por ejemplo, los distintos tipos de análisis y las operaciones de seguridad. Las actividades de seguridad se deben llevar a cabo de forma ordenada y procedimental por personal experto en esta materia.

El pilar fundamental en el que ha de basarse la seguridad de la información de toda organización, ya sea pública o privada, es la definición de la política de seguridad. Una política de seguridad bien definida e implantada en una organización gestionada por procesos es la base para implantar un sistema de gestión de seguridad de la información (SGSI) que regule y gobierne los procesos y procedimientos que han de implementarse. Además, todo el personal, cada uno a su nivel, debe seguir este SGSI de forma coordinada para conseguir el objetivo de seguridad requerido por la organización, para protección ante cualquier tipo de amenaza interna o externa.

A nivel de seguridad en las aplicaciones, y más concretamente en el apartado del desarrollo, el sistema de gestión de seguridad de la información debe implantar un ciclo de vida de desarrollo seguro de *software*(*secure software development life cycle,*SSDLC) que permita acometer el diseño e implementación de la seguridad desde las primeras fases del ciclo de vida del desarrollo de las aplicaciones.

El SSDLC incluye una serie de actividades de seguridad que implementar en cada una de sus fases, como pueden ser el análisis y gestión del riesgo de forma dinámica, la revisión de seguridad del código, las pruebas de seguridad o la monitorización continua en fase de despliegue.

Las aplicaciones web son, por excelencia, el principal tipo de aplicación *online* accesible desde Internet, las intranets de organizaciones y mediante un PC, una tableta o un móvil, entre otros, característica que también posibilita que sean accesibles por atacantes que quieren obtener algún beneficio económico, información, etc. Por tanto, en este tema es necesario introducir las características de las principales arquitecturas de aplicaciones web y de su protocolo HTTP empleado en la comunicación.

Los principales objetivos de este tema son los siguientes:

* Analizar las actividades que se desarrollan en cada fase del SSDLC.
* Conocer los principales tipos de arquitecturas de aplicaciones web.
* Estudiar el protocolo HTTP y sus métodos.

Los objetivos de seguridad por alcanzar en los sistemas de tecnología de la información y comunicaciones (TIC) pasan por:

* Identificar a las personas que acceden a la información manejada por un sistema o a los recursos de este.
* Autenticar a las personas que acceden a la información manejada por un sistema o a los recursos de este.
* Controlar el acceso a la información manejada por un sistema o a los recursos de este.
* Proporcionar confidencialidad a la información manejada por un sistema.
* Proporcionar integridad a la información manejada por un sistema o a los recursos de este.
* Mantener la disponibilidad de la información manejada por un sistema o de los recursos de este.
* No repudio: proporcionar la prueba de que una determinada transmisión o recepción ha sido realizada, sin que su receptor/transmisor pueda negar que se haya producido.
* Trazabilidad: proporcionar los controles que determinen que en todo momento se podrá determinar quién hizo qué y en qué momento.

4.2. Ciclo de vida de desarrollo seguro de software (SSDLC)

La seguridad de la aplicación tiene que tratarse obligatoriamente en todas las fases del SSDLC de aplicaciones (Opensamm, s. f.; OWASP CLASP Project/es, 2020). En cada una de las fases, se han de realizar prácticas que tienen que ver con el diseño, la implementación y pruebas de la seguridad de la aplicación,tratando y cubriendo todos los aspectos y principios de la seguridad.

El SSDLC también contempla las operaciones y actividades de seguridad *online* en fase de producción con la aplicación desplegada, la cual ya puede ser objetivo de ataques de cualquier naturaleza.

Se recomienda visualizar la clase magistral *SSDLC,* que introduce en qué consisten las actividades de seguridad que se llevan a cabo en sus distintas fases.

En un proceso SSDLC hay que llevar a cabo diversas actividades. En primer lugar, hay que derivar los requisitos de seguridad en función del resultado obtenido en la actividad de modelado de amenazas o derivación de casos de abuso de la aplicación y del análisis de riesgos de todos los activos de la organización.

Hay que diseñar la seguridad de la aplicación con base en los requisitos mencionados y teniendo en cuenta las dimensiones de seguridad: autenticación, autorización, control de accesos a recursos, cifrado de datos, seguridad en profundidad para asegurar el no repudio, confidencialidad e integridad de datos, etc. Además, se deben tener presentes los principios de seguridad contemplados en la política de seguridad.

En este paso, de diseñan los casos de tests funcionales de seguridad basados en el riesgo.

Luego, se implementa el código de la aplicación siguiendo buenas prácticas de desarrollo seguro, como son la validación de las entradas y salidas de la aplicación, gestión de errores, etc. Finalmente, se analiza la seguridad del código fuente mediante la ayuda de herramientas de análisis estático de código (caja blanca).

En la fase de despliegue o de pruebas, se ejecutan los casos de tests para prueba del diseño funcional de la seguridad en cada una de sus dimensiones. Estas pruebas pueden conducir a un ciclo volviendo a la primera fase para definir nuevos requisitos o redefinir los existentes para solucionar las vulnerabilidades encontradas. A continuación, se lleva a cabo un test de penetración a la aplicación con la ayuda de herramientas de análisis dinámico de caja negra y de caja blanca.

En la fase de producción, se ejecutan operaciones de seguridad, como monitorización, cambio de configuraciones, permisos, auditoría, gestión de *backups*, centro de respaldo, gestión de incidentes de seguridad, etc.

Como en cualquier otro modelo de SDLC, en un SSDLC se deben tener procesos definidos en todo lo que se refiere a desarrollo de *software*, gestión de configuración y gestión y aseguramiento de la calidad con sus correspondientes procedimientos y actividades, donde un proyecto de *software* tiene una dirección de proyecto, equipo de desarrollo y, además, un equipo de expertos en seguridad (que pueden pertenecer al área o unidad de seguridad) que apoyan para la realización de las actividades mencionadas.

De esta manera, se planifica y gestiona adecuadamente y se sigue un modelo de ciclo de vida adaptado a las características del proyecto. Partiendo de la política de seguridad y dentro del sistema de gestión de seguridad de la información, se debe contar con un equipo que permita implantar el SSDLC asesorando y realizando las actividades relacionadas con la seguridad durante el ciclo de vida.

Cualquier modelo de ciclo de vida SDLC (ciclo de vida en espiral, *extreme programming*, proceso unificado de *rational*) o metodología de tipo SCRUM se puede adaptar y convertir en un SSDLC. Si no se tuviera una estructura por procesos dentro de la organización, sería muy difícil pensar en aplicar diseño e implementación de la seguridad durante el ciclo de vida de *software* correctamente.

Una propuesta de SSDLC sería:

A diagram of a software development process

Description automatically generated

Figura 1. Modelo de SSDLC. Fuente: Chess y West, 2007.

La Figura 1 muestra un ejemplo de SSDLC donde se especifican las actividades y pruebas de seguridad que efectuar en cada fase de este. A continuación, se enumeran esas actividades, mejores prácticas o *touchpoints* en orden de eficacia e importancia según el autor McGraw (2006):

* Revisión de código.
* Análisis de riesgo arquitectónico.
* Pruebas de penetración.
* Pruebas de seguridad basadas en riesgo.
* Casos de abuso.
* Requisitos de seguridad.
* Operaciones de seguridad.
* Revisión externa.

Este orden descrito no se adecuará perfectamente a todas las organizaciones. De hecho, el orden refleja el trabajo desarrollado en años de experiencia de aplicar estas prácticas en empresas que tienen gran cantidad de *softwares*. Por esa razón, la revisión de código viene antes del análisis de riesgos arquitectónicos. Hay que resaltar que estas actividades hay que repetirlas a lo largo del ciclo de vida, lo que supone un ciclo continuo en la ejecución de las distintas actividades de seguridad:

* Según se van descubriendo nuevas amenazas, se tienen nuevos casos de abuso que implican nuevos riesgos, lo que implica rehacer el análisis.
* Si se introducen cambios o nuevos componentes de *software* o de *hardware* en el sistema, hay que rehacer el análisis de riesgos y revisar el código de nuevos componentes de *software*.
* Nuevos defectos de implementación de partes, que se modifican con arreglo a nuevas especificaciones o cambios, implican una nueva revisión de código y nuevas pruebas de seguridad en operación del sistema.

Por supuesto, siempre hay que incidir en la prevención y formación en seguridad, documentando y realizando cuantificación y análisis de métricas de seguridad que se puedan emplear en futuros proyectos para mejorar.

### **4.3. Arquitecturas de las aplicaciones web**

El extraordinario **éxito del modelo web** puede atribuirse a una característica fundamental: es un modelo más débilmente acoplado que los modelos de programación distribuida tradicionales como RPC, RMI, DCOM y CORBA. Las interacciones entre clientes y servidores web son sencillas: intercambian mensajes que transportan datos de tipo multipurpose internet mail extensions (MIME) y la semántica de un mensaje puede modificarse utilizando cabeceras. El destino de un mensaje se especifica indirectamente utilizando una URL, y este nivel de indirección puede aprovecharse para implementar balance de carga, seguimiento de sesiones y otras funcionalidades.

Las **arquitecturas monolíticas** son las que se utilizan normalmente cuando se desarrolla la aplicación mediante un lenguaje de script como ColdFusion o PHP alojada en un servidor web.

Cuando se necesita una aplicación **escalable** y que ofrezca mayor **rendimiento y seguridad,** se ha de recurrir a una de las opciones de desarrollo empresarial: .Net o J2EE (Padilla y Pérez, 2010). Estas especificaciones proporcionarán mayores posibilidades debido a las distintas opciones de **configuración e implementación de la seguridad** que ofrecen, como puede ser la utilización de lenguajes más seguros potencialmente como Java o C#.

Una arquitectura de aplicación escalable normalmente se divide en niveles y, si se utilizan **patrones de diseño,** muchas veces se dividen en **porciones reutilizables,** usando diferentes lineamientos específicos para reforzar su modularidad, los requerimientos de interfaz y la reutilización de objetos. Además, dividir la aplicación en niveles permite que se pueda distribuir entre varios servidores, mejorando, por lo tanto, la escalabilidad de la aplicación a expensas de mayor complejidad con **controles de acceso** y **detección avanzados.** Existe la separación de tareas y responsabilidades.

### **Arquitectura de aplicación web clásica**

Las **capas** de las que se compone una aplicación web en una arquitectura clásica son:

* **Capa cliente:** navegador web (Internet Explorer, Chrome, etc.).
* **Capa de aplicación:** servidor de aplicación (WebLogic, Tomcat, WebSphere, Struts, .NET, ColdFusion, etc.). Contiene la lógica de negocio y de presentación. Puede incluir un servidor web (Apache, Niginx, etc.) para alojar contenido estático o aplicación desarrollada con lenguajes interpretados (PHP, Python, etc.).
* **Capa de persistencia:** servidor gestor de base de datos (Oracle, MS SQL Server, MySQL, PostgreSQL, etc.).
* **Comunicación:** se lleva a cabo mediante el protocolo HTTP/1.1/2.0. El navegador realiza peticiones y el servidor envía las respuestas correspondientes. Cada petición y su respuesta contienen una parte de cabeceras y otra correspondiente al cuerpo de la petición o respuesta.

A diagram of a process

Description automatically generated

Figura 2. Arquitectura de tres capas de aplicaciones web. Fuente: elaboración propia.

### **Arquitectura de aplicaciones RIA**

Las tecnologías que se usan en las arquitecturas de **aplicaciones ricas de internet** (RIA) se incluyen en el término **AJAX** (Asynchronous JavaScript and XML). AJAX es un conjunto de tecnologías que incluyen JavaScript asíncrono junto con XML, XSLT, XHTML y Document Object Model (DOM), todas ellas tecnologías ampliamente difundidas y utilizadas (binitamayekar, 2019).

El objetivo principal de las aplicaciones AJAX es eliminar la naturaleza start-stop-start-stop de la **interacción entre el cliente y el servidor** de aplicaciones web introduciendo un **intermediario:** un **motor AJAX/JS** que intercambia información con el servidor en el backend sin que el usuario lo perciba (como ocurre en la arquitectura web tradicional síncrona):

A diagram of data transmission

Description automatically generated

Figura 3. Modelo de comunicación síncrono. Fuente: binitamayekar, 2019.

Este **elemento arquitectónico adicional del lado cliente** es lo más característico de AJAX y tendrá algún inconveniente de seguridad. El navegador cargará el motor AJAX, usualmente escondido en un frame oculto, y este será responsable del procesamiento tanto de la interfaz que el usuario ve como de la comunicación con el servidor en nombre del usuario.

Este modelo permitirá la **interacción del usuario con la aplicación** de forma asincrónica independientemente de la comunicación con el servidor:

A diagram of a computer process

Description automatically generated

Figura 4. Modelo de comunicación asíncrono (AJAX). Fuente: binitamayekar, 2019.

De esta forma, el usuario no queda esperando la respuesta del servidor, que tiene la percepción de que no hay tiempos de recarga. Los toolkits AJAX han permitido a los desarrolladores web construir aplicaciones web 2.0 de forma rápida y relativamente con poco esfuerzo, generando un **nuevo nivel de interactividad con los usuarios,** como el que otorga, por ejemplo, Google Maps, Google Docs, Flickr, etc.

En cuanto a la arquitectura en sí, se observa en la Figura 5. Cada acción del usuario que normalmente generaría una petición HTTP se convierte en una **llamada JS al motor AJAX** en su lugar. Para cualquier respuesta a una acción del usuario que no requiera un viaje de vuelta al servidor, tal como la validación de datos simple, edición de datos en la memoria e incluso un poco de navegación, el motor se encarga por sí solo de servir el código HTML, CSS y JS. Si el motor necesita algo del servidor con el fin de responder al cliente web, se encarga de realizar esas peticiones de forma asíncrona (HTTP request), normalmente, usando XML, de modo transparente y sin penalizar, a priori, la experiencia del usuario del navegador (Asadullah et al. 2013).

A diagram of a software development

Description automatically generated

Figura 5. Modelo de aplicación AJAX web. Fuente: binitamayekar, 2019.

### **Patrón de diseño lógico MVC**

Uno de los patrones de diseño lógico de las aplicaciones web más comunes es **modelo-vista-controlador**(MVC) (Modelo-Vista-Controlador, 2021). El MVC se puede implementar en muy diversos frameworks, como, por ejemplo:

* Las arquitecturas de aplicaciones J2EE, como Apache Foundation Jakarta Struts.
* PHP (Laravel, 2020).
* .NET, Rubí y Phyton.
* Meteor, Tesla.js y Node.js.

MVC articula el código de una aplicación web en **tres capas** diferenciadas: modelo, vista y controlador.

**Modelo**

El modelo es la capa de la aplicación que se ocupa de los **datos que necesita y los accesos a ellos.** En una aplicación J2EE, los componentes de la lógica de negocio necesitan acceder a los datos de la aplicación (sistema gestor de base de datos [SGBD], repositorio XML) y, para ello, solicitan el servicio a un JavaBean o EJB, que se encarga de la llamada al SGBD para extraer, insertar, actualizar o borrar información.

**Vista**

Es la capa que se ocupa de generar la **presentación al usuario** en aplicaciones web. La vista se compone de páginas HTML que componen la **interfaz** del usuario. Como se señaló anteriormente, pueden incluir el código JavaScript que interpretará el navegador. Los usuarios, a través de la interfaz de la aplicación web, pueden rellenar un formulario y enviarlo al servidor de aplicación, que en tecnología J2EE dispone de un controlador (servlet) que se encarga de seleccionar la lógica de aplicación que se ocupará de servir la petición del usuario.

En cuanto a cuestiones de seguridad, hay que tener en cuenta la **información de autenticación** que se puede enviar dentro de las peticiones o cualquier otra información que pueda comprometer la seguridad de la aplicación con**ataques de** sniffing**y repetición, fuerza bruta basada en diccionarios de descubrimiento de contraseñas** y todos los aspectos que conciernen a la interpretación de lenguajes de script por parte del navegador. Si se permite, hay que validar por qué constituyen una **potencial fuente de ataque** (Scambray et al. 2010; Sullivan et al. 2012; Moeller, 2016). Lo más deseable para implementar los servicios de autenticación es utilizar los proporcionados por el framework de desarrollo.

Los **mecanismos de autenticación** HTTP basic y HTTP digest (The Internet Society, 1999a) tienen **vulnerabilidades de seguridad.**

A continuación, se representa un esquema del patrón MVC para J2EE:

A diagram of a control system

Description automatically generated

Figura 6. Patrón MVC para J2EE. Fuente: Programacion.net, s. f.

**Controlador**

Recoge las **peticiones de los usuarios** (servlet) para seleccionar el código de aplicación o lógica de negocio encargadode**servir a ellas.** En tecnología J2EE, son los Java Server Pages (JSP), que integran código HTML y código Java, los encargados de llevar a cabo este cometido de ejecutar la lógica de negocio correspondiente a la petición.

En cuanto al apartado de seguridad, las vulnerabilidades en el **servicio de autorización** a los recursos de la aplicación y sistema operativo pueden dar lugar a ataques de **fuga de información.** Es más conveniente, desde el punto de vista de la seguridad, implementar la lógica de autorización en el SGBD a través del uso de procedimientos almacenados e implementar los servicios de autenticación utilizando los proporcionados por el framework de desarrollo.

En el apartado de seguridad puede haber vulnerabilidades en el aseguramiento del **control** **de acceso, integridad y confidencialidad de la información de los datos alojados en la aplicación y base de datos.**Para ello, es necesario diseñar mecanismos que aseguren la integridad y confidencialidad mediante servicios PKI de firma digital y cifrado de datos. Asimismo, hay que prestar atención al servicio de logging(los mensajes que deben aparecer deben ser los que realmente aportan información útil) y manejo de errores y excepciones, de forma que no aporten más información de la necesaria en cada caso que pudiera ser útil a los atacantes para sacar provecho de ella.

En resumen, las vulnerabilidades de seguridad debidas a la arquitectura de diseño física y lógica se deben a defectos en los servicios de autenticación, autorización, sesiones, integridad y confidencialidad de datos, trazabilidad, gestión de errores y excepciones y comunicación entre capas.

### **4.4. Protocolo HTTP**

El protocolo HTTP no fue diseñado para aplicaciones y, probablemente, tampoco para usos seguros. HTTP crea **oportunidades para tener vulnerabilidades** **de seguridad,** de la misma manera que en las funciones de string de la biblioteca estándar de C crea oportunidades para el desbordamiento de buffer. Los programadores tienen que aprender a desarrollar de forma segura.

El protocolo HTTP implementa una serie de métodos especificados en **HTTP 1.1 rfc 2616** (The Internet Society, 1999b) que son los siguientes:

* **HEAD:** pide una respuesta idéntica a la que correspondería a una petición GET, pero sin el cuerpo de la respuesta. Esto es útil para la recuperación de metainformación escrita en los encabezados de respuesta, sin tener que transportar todo el contenido.
* **GET:** pide una representación del recurso especificado. Por seguridad, no debería ser usado por aplicaciones que causen efectos, ya que transmite información a través de la URI agregando parámetros a la URL.

GET

GET /images/logo.png HTTP/1.1 obtiene un recurso llamado «logo.png».

Ejemplo con parámetros:

GET /index.php?page=main&amp;lang=es

* **POST:** envía los datos para que sean procesados por el recurso identificado (URL). Dichos datos se incluirán en el cuerpo de la petición. Esto puede resultar en la creación de un nuevo recurso, en las actualizaciones de los recursos existentes o en ambas cosas.
* **PUT:** sube, carga o realiza un upload de un recurso especificado (archivo). Es el camino más eficiente para subir archivos a un servidor, ya que en POST utiliza un mensaje multiparte que es decodificado por este. En contraste, el método PUT permite escribir un archivo en una conexión socket establecida con el servidor. La desventaja del método PUT es que los servidores de hosting compartido no lo tienen habilitado.

PUT

PUT /path/filename.html HTTP/1.1.

* **DELETE:** borra el recurso especificado.
* **TRACE:** solicita al servidor que envíe de vuelta en un mensaje de respuesta. Se utiliza con fines de comprobación y diagnóstico. El método TRACE da lugar a la vulnerabilidad cross-site tracing (XST), debido a que copia las cabeceras de la petición en el cuerpo de la respuesta haciendo inútil la protección mediante el parámetro HTTPOnly, que impide acceso a la cookie que protege. XST se puede combinar con un ataque cross-site scripting (XSS) para el robo de un identificador de sesión contenido en una cookie.

A screenshot of a computer

Description automatically generated

Figura 7. XST. Fuente: Grossman, 2003.

* **OPTIONS:** devuelve los métodos HTTP que el servidor soporta para una URL específica. Este método puede ser utilizado para comprobar la funcionalidad de un servidor web.
* **CONNECT:** se utiliza para saber si un proxy nos da acceso a un host. No necesariamente la petición llega al servidor.

En la Figura 8 se observa cómo se puede especificar un método HTTP:

A diagram of a computer code

Description automatically generated with medium confidence

Figura 8. Métodos HTTP. Fuente: elaboración propia.

En una petición GET, los parámetros que se incluyen en la cadena de la URL pueden ser rutinariamente escritos a LOG caché y almacenados en el historial del navegador guardando información sensible como contraseñas, por ejemplo. Es más sencillo prevenir XSS deshabilitando GET en las aplicaciones. Es más fácil para un atacante que envía correos con una URL con parámetros maliciosos:

A close-up of a computer screen

Description automatically generated

Figura 9. Método HTTP GET. Fuente: elaboración propia.

Sin embargo, con el método POST, los parámetros se especifican en el cuerpo de la petición.

Para que sea más seguro, se debe de utilizar POST en lugar de GET:

A close-up of a computer code

Description automatically generated

Figura 10. Método HTTP POST. Fuente: elaboración propia.

La procedencia de peticiones surge si la aplicación usa un **identificador de sesión** (cookie) con un **formulario** (administración) para **crear un usuario:**

A computer code with black text

Description automatically generated

Figura 11. Procedencia de peticiones cross-site request forguery (CSRF). Fuente: Chess y West, 2007.

Un atacante podría tener un sitio web con:

A computer screen shot of a computer code

Description automatically generated

Figura 12. Procedencia de peticiones CSRF. Fuente: Chess y West, 2007.

Si el administrador visita el sitio controlado por el atacante, se puede dar un **ataque** cross-site request forguery (CSRF).

Para prevenirlo, se puede incluir algún **campo que la aplicación pueda validar:**

A screenshot of a computer program

Description automatically generated

Figura 13. Procedencia de peticiones. Protección CSRF. Fuente: Chess y West, 2007.

Otra vulnerabilidad que se puede presentar en el protocolo HTTP es la vulnerabilidad HTTP response splitting, que consiste en incluir caracteres \r\n de retorno de carro y de línea para generar una **respuesta doble** (Acunetix, 2020), que se ejecutará en el navegador del usuario permitiendo **lanzar otros ataques,**como XSS:

A screenshot of a computer program

Description automatically generated

Figura 14. HTTP response splitting. Fuente: Chess y West, 2007.

Si el atacante suministra una cadena como Wiley Hacker\r\n\r\nHTTP/1.1 200 OK\r\n…, la respuesta se convierte en **doble,** lo que permite **inyectar código HTML:**

A white background with black text

Description automatically generated

Figura 15. HTTP response splitting. Fuente: Chess y West, 2007.

El problema del protocolo HTTP es más agudo cuando se maneja el estado de sesión que es necesario en la mayor parte de aplicaciones porque el protocolo en sí mismo es sin estado. Como HTTP es sin estado, construir casi cualquier tipo de aplicación sofisticada requiere un **identificador de sesión** que sirva para ambos sentidos de la comunicación para **asociar las peticiones previas de un usuario** con las siguientes.

Los identificadores de sesión pueden ser pasados hacia adelante y hacia atrás como parámetros URL, pero, hoy, la mayor parte de las aplicaciones manejan cookies, que son un mecanismo que el protocolo incorporó posteriormente para almacenar datos en la memoria del navegador con un espacio de 4096 bytes. La razón más común de usar un identificador de sesión es permitir a un usuario autenticarse solo una vez para continuar con una serie de interacciones con la aplicación.

Se puede utilizar una cookie, además de otros métodos, para almacenar el identificador de sesión. Esto quiere decir que la seguridad de la aplicación depende de ello y debe ser muy difícil para un atacante averiguar u obtener el identificador de sesión para un usuario autenticado. Una buena gestión de sesión HTTP escoge **identificadores de sesión fuertes** y se asegura de que son **publicados y revocados en puntos apropiados** en el programa.

Se recomienda visualizar la clase magistral Proyectos seguridad web, que repasa las principales organizaciones que implementan proyectos que ayudan a diseñar e implementar la seguridad en las aplicaciones web.