Auditoría y desarrollo seguro

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

PID_00191664

UOC
Universitat Oberta de Catalunya

Índice

1. Introducción

Este módulo describe algunas de las medidas que hay que adoptar para securizar la infraestructura ante posibles ataques. El problema es que no existe una solución perfecta. Hay distintos tipos de vulnerabilidad que no disponen de solución, y cada día aparecen nuevos tipos de ataque para los que no se ha pensado una defensa. Para empeorarlo todo, la securización es un proceso difícil y no estándar que depende de multitud de factores.

Sin embargo, existen una serie de procedimientos y de aspectos a considerar durante la implantación y mantenimiento que ayudan a hacer mucho más difícil cualquier ataque. Los ataques son imprevisibles, en cuándo y en cómo van a ocurrir: lo importante es estar preparado para cuando ocurra. Esta preparación puede evitar la gran mayoría de tentativas de ataques, incluyendo los automáticos que no discriminan a los objetivos, como los gusanos que infectan al máximo número posible de víctimas. Una securización adecuada puede incluso llegar a evitar tipos de ataques desconocidos hasta la fecha.

Otro aspecto importante a considerar es la implantación de políticas y procedimientos. Todo esto implica seguir una serie de políticas, crear un entorno adecuado para la base de datos, evitar cualquier funcionalidad innecesaria, etc. Esto se suele traducir en un acuerdo entre seguridad y funcionalidad.

Por último, estudiaremos herramientas que ayudan a determinar el estado de seguridad de la infraestructura administrada y a instaurar las medidas necesarias para evitar problemas, así como dónde conseguir recursos para conocer las mejores prácticas en este sentido. También se explican acciones para asegurar altos niveles de seguridad por parte de expertos mediante auditorías.

2. Auditorías

Una vez se han implantado todas las medidas de seguridad, toda la funcionalidad deseable está preparada, el personal formado adecuadamente, se han preparado planes de contingencia y políticas para todos los aspectos posibles, es hora de ver qué falla.

Por mucho que un administrador esté completamente convencido de la seguridad de su sistema, nunca puede estar seguro. Lo mejor es probar la misma por parte de un auditor externo experto en el tema y que nos ayude a detectar las posibles debilidades del sistema antes de que lo haga un atacante potencial. Por esta razón hacer auditorías de forma regular es muy recomendable.

Las **auditorías de caja negra** ponen al atacante en una situación en la que no tiene ninguna información del objetivo y debe partir de cero para averiguar todo lo posible e intentar hacerse con el control del mismo. Es la situación que encontraría un atacante que no tuviese conocimiento interno del objetivo.

En las **auditorías de caja blanca** el auditor dispone de todas las credenciales necesarias para acceder al sistema, así como ayuda por parte de los administradores. En esta situación, el auditor utiliza sus privilegios para estudiar el sistema a fondo desde dentro, comprobar que dispone de todas las medidas de seguridad adecuadas, así como de las políticas recomendadas.

2.1. OWASP

OWASP[1] es una asociación sin ánimo de lucro que vela por la seguridad en la web. Entre sus cometidos está la formación y concienciación de los usuarios y los programadores. También generan código para hacer aplicaciones más seguras o para el testeo de las aplicaciones web.

(1)*Open Web Application Security Project*

Otras aportaciones

Destacan las de las charlas periódicas que organizan por todo el mundo incitando a la mejora en la seguridad de las aplicaciones y proponiendo nuevas soluciones para problemas conocidos.

Destaca por su relevancia y fiabilidad el top ten de fallos de seguridad informática que publicaron en el 2007. En este top ten tenemos las técnicas más comunes y los errores más usuales a la hora de auditar la seguridad de una aplicación web. Se basan en los datos reales de los expertos en seguridad que colaboran con el OWASP:

1) *Cross site scripting* (XSS): Técnica consistente en introducir código Javascript dentro de la aplicación que visita un usuario.

2) Inyecciones de código (SQL Injection, XPath Injection y LDAP Injection): Modifica o extrae información desde un almacén de datos.

3) *Remote file inclusion*: Ejecuta un fichero externo al servidor como si se encontrase dentro del mismo.

4) Referencia directa a ficheros: o lo que es lo mismo, *local file inclusion* (LFI). Un usuario puede manipular la URL para acceder a un recurso que, inicialmente, el desarrollador no había pensado permitir.

5) *Cross site request forgery* (CSRF): Manipular el navegador del usuario mediante XSS para que realice acciones no deseadas entre dominios.

6) Fugas de información y errores no controlados: También conocido como *path disclosure*. Cualquier error no controlado puede dar pie a que un atacante conozca datos internos sobre la aplicación y su entorno.

7) Autenticación débil: Usar un sistema de cifrado débil para asegurar las comunicaciones puede dar lugar a que los atacantes logren generar un *token* de autenticación válido para un usuario cualquiera.

8) Almacenado inseguro de credenciales: Almacenar las contraseñas en texto plano o con algoritmos reversibles es un grave riesgo de seguridad debido a que un atacante podría extraer las contraseñas mediante SQL Injection.

9) Comunicaciones inseguras: El envío de información sin un cifrado suficientemente potente puede dar lugar a que un atacante intercepte estas comunicaciones y las descifre.

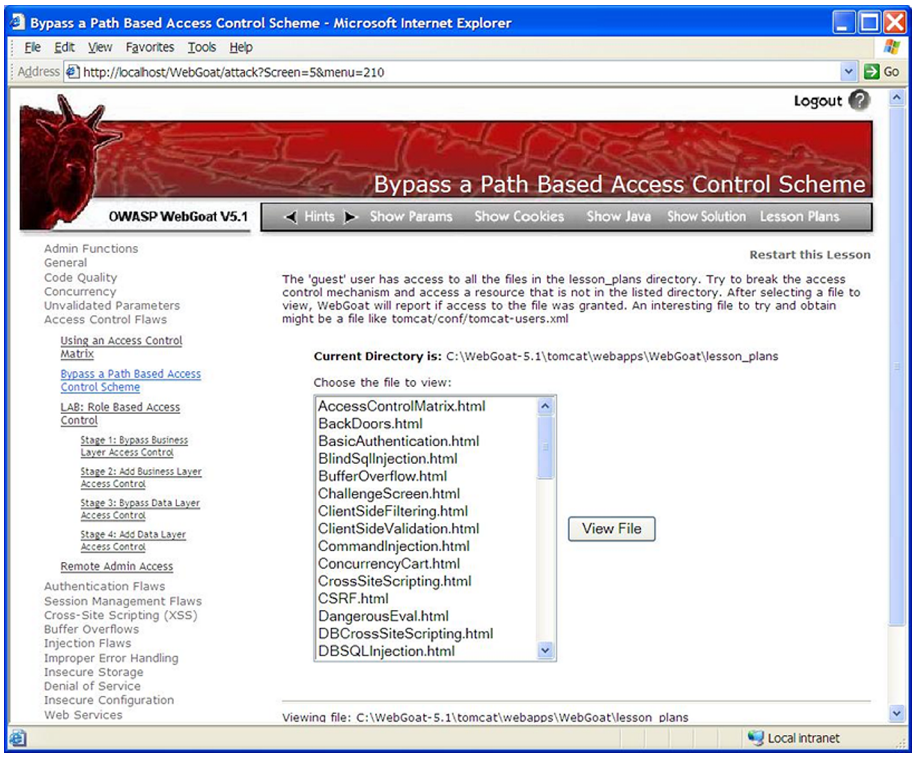
10) Fallos al restringir el acceso a URL: La ocultación de recursos en zonas sin control de acceso y confiando en que ningún usuario malintencionado intentará localizar estas zonas es el último de los fallos de seguridad expuestos en esta lista.

Como se puede ver, la lista cubre un gran abanico de opciones en lo que se refiere a fallos de seguridad. Es tan completo el compendio que algunas empresas y grupos de desarrollo lo han adoptado como decálogo para comprobar la seguridad de sus aplicaciones, como por ejemplo el CMS (*content management system*) Plone.

Otra de las actividades del OWASP es la generación de herramientas que ayuden al desarrollo seguro de aplicaciones y a la auditoría de seguridad de los servidores web. Entre los proyectos que promueven, están:

1) ***WebGoat***: Es un proyecto educativo. Consiste en una aplicación web JSP con distintos tipos de vulnerabilidades. Esto sirve como entrenamiento para aquellas personas interesadas en aprender seguridad web.

Figura 1. WebGoat en funcionamiento

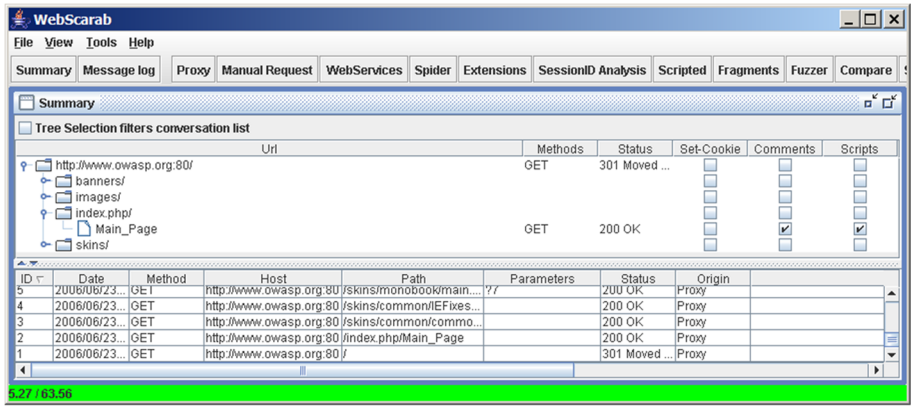


Los fallos que contiene la aplicación incluyen alguno de los siguientes puntos:

- *Cross site scripting*
- Manipulación de campos ocultos
- Debilidades en las *cookies* de sesión
- SQL Injection
- Blind SQL Injection
- Información sensible en comentarios
- Servicios web inseguros

2) ***WebScarab***: Seguramente la mejor manera de lograr modificar el comportamiento de una aplicación web es la de mandar datos que no se espera recibir. Para realizar esta acción nada mejor que un *proxy* que nos permita modificar en tiempo real las peticiones que realizamos. WebScarab es uno de ellos.

Figura 2. Pantalla principal de WebScarab



WebScarab tiene una serie de opciones que lo hacen útil para cualquier analista de seguridad. Algunas de estas características son:

- Extraer comentarios HTML y de *scripts* de las páginas que se visitan.
- Convierte campos ocultos en campos de texto normales para editarlos de una manera más cómoda.
- Realiza acciones de búsqueda de URL dentro de las páginas navegadas para detectar nuevos objetivos.
- Busca posibles fallos de XSS en las páginas visitadas.
- Permite el *scripting* para modificar automáticamente algún aspecto de un sitio web cada vez que se visite.

Como se puede ver es un producto muy completo y complejo, que permite al auditor olvidarse de ciertos aspectos y automatizarlos.

3) *AntiSamy*: Los fallos de XSS se comprobaron devastadores cuando un pequeño gusano[2] XSS tumbó la página web de MySpace.com. De aquí nace este proyecto, una librería que permite sanear la entrada de nuestro código web para evitar la inclusión de código XSS. Por defecto viene con cuatro tipos de reglas que van a ser más o menos restrictivas según el escenario que deseemos recrear:

(2)Este gusano se llamaba Samy, como su creador.

- **antisamy-slashdot.xml**: Se basa en las reglas del popular sitio de noticias. Solo se pueden introducir las etiquetas `<a>`, `<b>`, `<blockquote>`, `<i>` y `<u>`. Además no se permite ningún tipo de CSS.
- **antisamy-ebay.xml**: Aunque Ebay no ofrece una lista pública de las etiquetas HTML que permite, los desarrolladores han intentado imitar las funcionalidades que presta este popular sitio de compra-venta.
- **antisamy-myspace.xml**: Las reglas que dieron pie al proyecto. Se permiten gran número de etiquetas y CSS, pero ningún tipo de Javascript.
- **antisamy-nothinggoes.xml**: El filtro más restrictivo. No permite ningún tipo de etiqueta HTML, fichero CSS o Javascript. Es útil para usarlo como base para nuestros propios filtros.

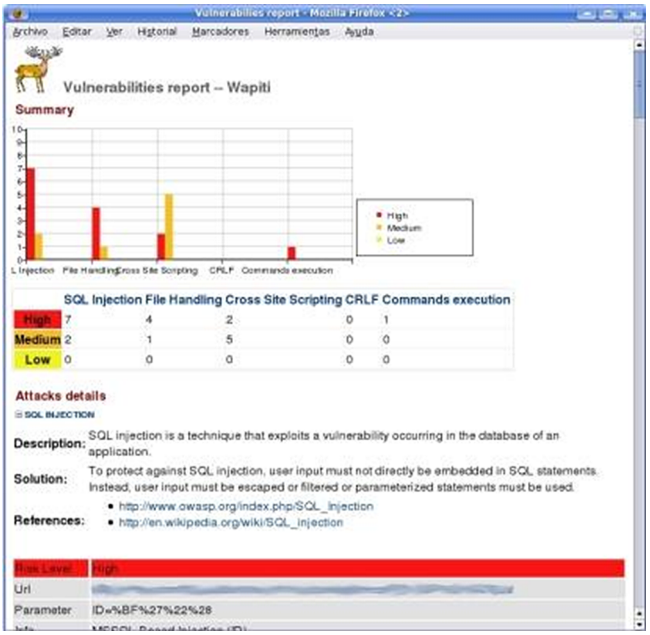
Existen versiones de esta librería para Java, .NET e incluso Python. Sin embargo, no se ha desarrollado para PHP, para el cual recomiendan el uso de la librería HTMLPurifier.

4) ***Watipi***: Un analizador de seguridad de sitios web. Programado en Python, y permite recorrer un sitio web en búsqueda de errores de tipo:

- Extracción de información de errores
- LDAP Injection
- XSS
- SQL Injection
- Ejecución de comandos

Además de esto permite generar un completo informe con gráficas y datos estadísticos sobre las vulnerabilidades encontradas.

Figura 3. Reporte de Wapiti



2.2. Escáner de vulnerabilidades de caja negra

Los llamados escáneres de vulnerabilidades de caja negra son aquellos que realizan un análisis en búsqueda de vulnerabilidades sin revisar el código fuente de la aplicación, esto es, solo comprobará la seguridad de los elementos disponibles para un usuario externo.

El 70% de las aplicaciones web son vulnerables a algún tipo de ataque que permiten el robo de información sensible o confidencial. Es posiblemente uno de los entornos donde más se debería priorizar la securización, ya que las páginas web están accesibles 24 horas al día y 7 días a la semana los 365 días del año. Son la vía más utilizada por los *hackers* a la hora de realizar una intrusión, localizando los puntos débiles de las aplicaciones en los formularios, sistemas de inicio de sesiones, contenidos dinámicos, comunicaciones contra *backends*, etc.

Dos conocidas aplicaciones utilizadas para las auditorías de aplicaciones web son Acunetix y W3af. Este tipo de herramientas automatizadas deben usarse como complemento de una auditoría de seguridad web, pero sin ignorar auditorías de tipo *pen-testing*[3].

(3)Auditorías no automatizadas.

2.2.1. Acunetix

Acunetix es una aplicación de tipo comercial enfocada en la seguridad web a nivel interno. Ofrecen su producto como complemento a las auditorías internas de seguridad que se realicen sobre el código. Realiza escaneos automatizados con posibilidad de ejecución tanto en interfaz gráfica como en línea de comandos. Solo está disponible para plataformas Windows.

A pesar de ser un programa comercial ofrece para la descarga una versión gratuita que se limita a la búsqueda de fallos de XSS en el dominio que especifiquemos. Además permite realizar un escaneo completo sobre tres páginas vulnerables, programadas con:

- PHP,
- ASP,
- y ASP.Net.

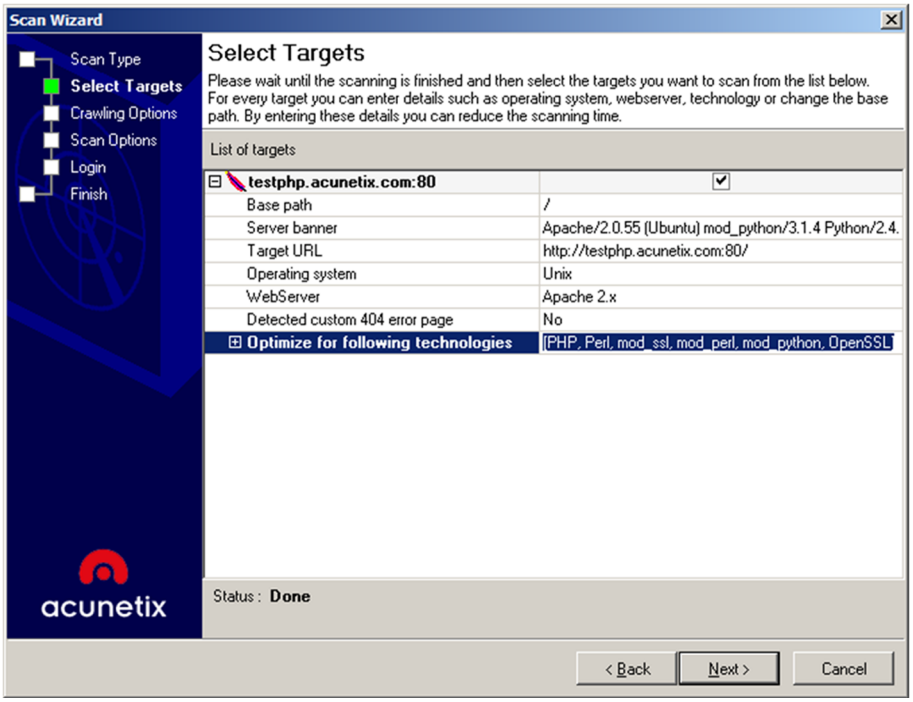
Las características principales de Acunetix son:

- Analizador Javascript, permitiendo auditar Ajax y aplicaciones con Web 2.0.
- Utiliza las más avanzadas técnicas de SQL Injection y Cross Site Scripting.
- Utilización de tecnología 'AcuSensor'.
- Analiza *websites* incluyendo contenido Flash, SOAP y AJAX.
- Realiza un escaneado de puertos contra el servidor web y busca vulnerabilidades en sus servicios.
- Detecta el lenguaje de programación de la aplicación.
- Proporciona extensos informes del estado de seguridad.

Para analizar las funcionalidades del programa vamos a realizar un escaneo sobre uno de los dominios vulnerables, donde se nos permite probar todas las características del programa. En este caso vamos a hacerlo sobre el dominio que ejecuta la aplicación PHP.

El primer paso es establecer el dominio a escáner. Con este dato Acunetix realiza un primer análisis del dominio, intentando detectar las características intrínsecas del servidor: tecnología, servidor HTTP, cabeceras, etc. Estos datos los usa para configurar automáticamente las reglas más óptimas para el análisis.

Figura 4. Asistente de Acunetix



Desde este asistente podremos especificar algunos parámetros para mejorar la eficiencia de nuestro análisis así como para hacerlo menos intrusivo. Algunas de las opciones que podemos configurar son:

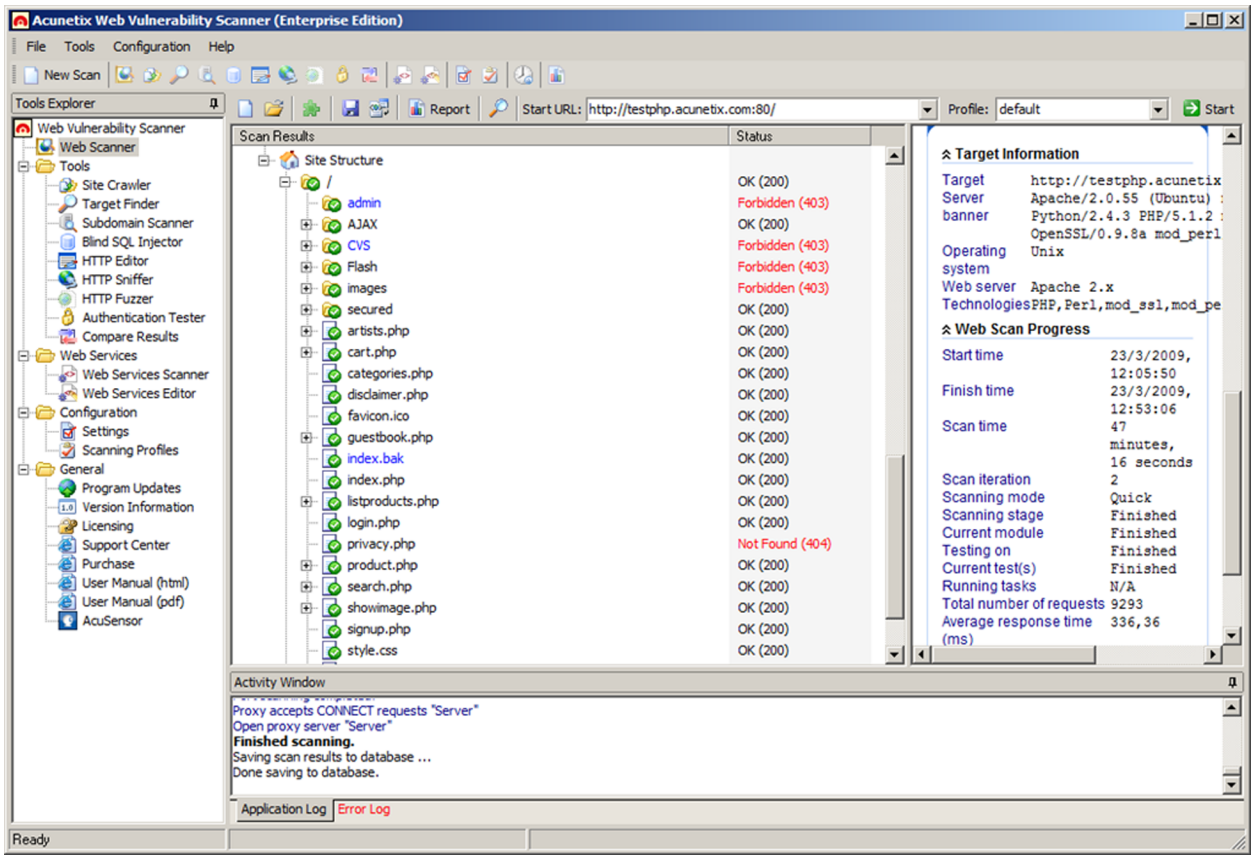
- Envío de datos mediante los formularios existentes en la página web.
- Intentar extraer listado de directorios.
- Ignorar mayúsculas en los nombres de los ficheros.
- Procesar los ficheros robots.txt y sitemap.xml.
- Manipular las cabeceras HTTP.
- Habilitar el escaneo de puertos.
- Establecer usuario y contraseña para la autenticación HTTP.
- Habilitar el análisis del código Javascript mediante la ejecución del mismo.

Como se puede ver obtendrá muchos detalles, que se obtendrán a través de muchas peticiones y solicitudes al servidor web, por lo que lanzar esta herramienta contra nuestro servidor debería limitarse a horas de bajo tráfico para no interferir con su normal funcionamiento.

Después de completar el asistente, se nos presentará una ventana de análisis donde iremos comprobando en tiempo real lo que va analizando y descubriendo el programa.

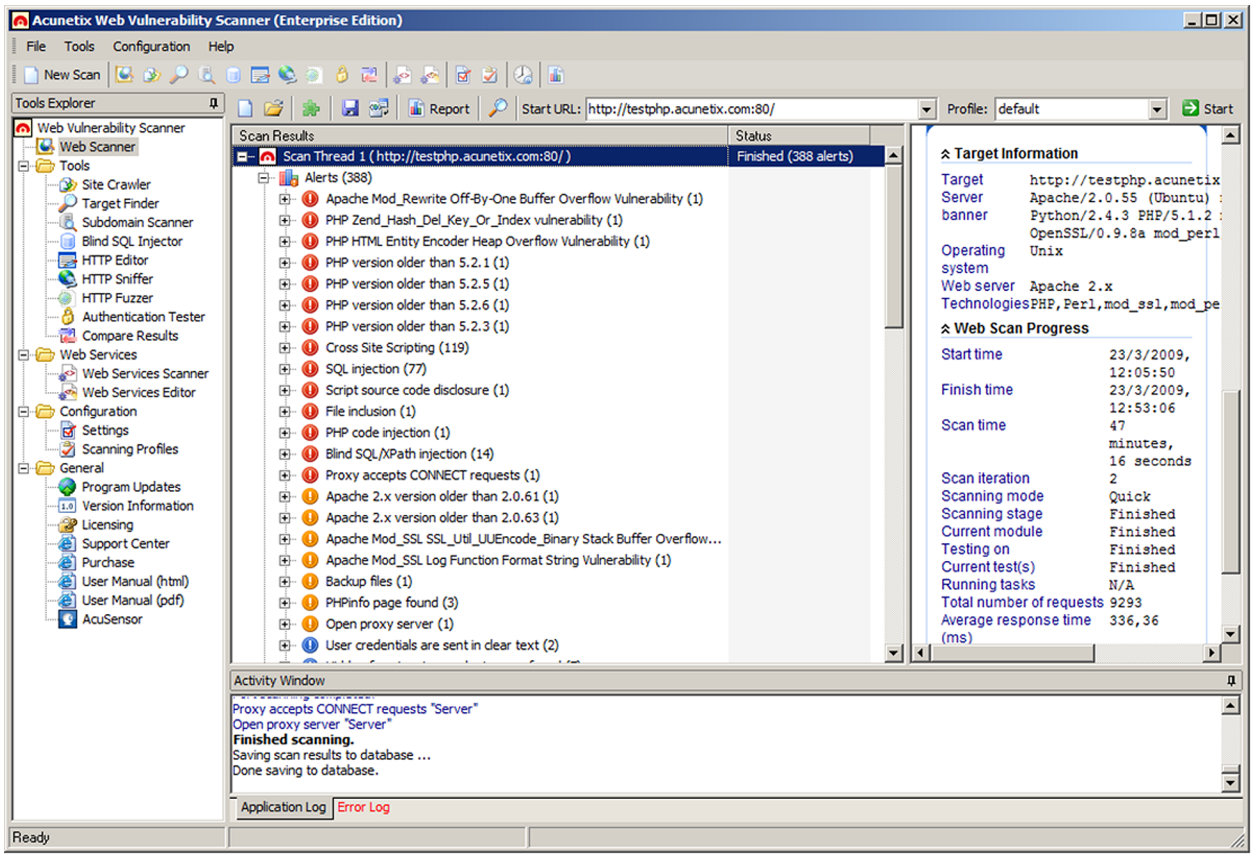
Una de las características más interesantes de Acunetix es la posibilidad de ver un árbol de los ficheros que existen en el servidor y que podremos recorrer como si un explorador de ficheros de nuestro sistema operativo se tratase.

Figura 5. Árbol de ficheros de un sitio mediante Acunetix



Sobre cada una de las páginas detectadas el programa realiza una serie de comprobaciones para determinar si la aplicación va a ser vulnerable a cualquiera de las técnicas mencionadas anteriormente. Estos análisis darán lugar a un reporte con información detallada de lo ocurrido.

Figura 6. Reporte final de Acunetix



Este análisis a pesar de ser completo y exhaustivo es muy intrusivo. Un atacante real nunca lanzaría una herramienta de este tipo para localizar las vulnerabilidades de nuestro sitio web, por lo que no podemos usarlo como caso real para auditar nuestras contramedidas frente a ataques reales.

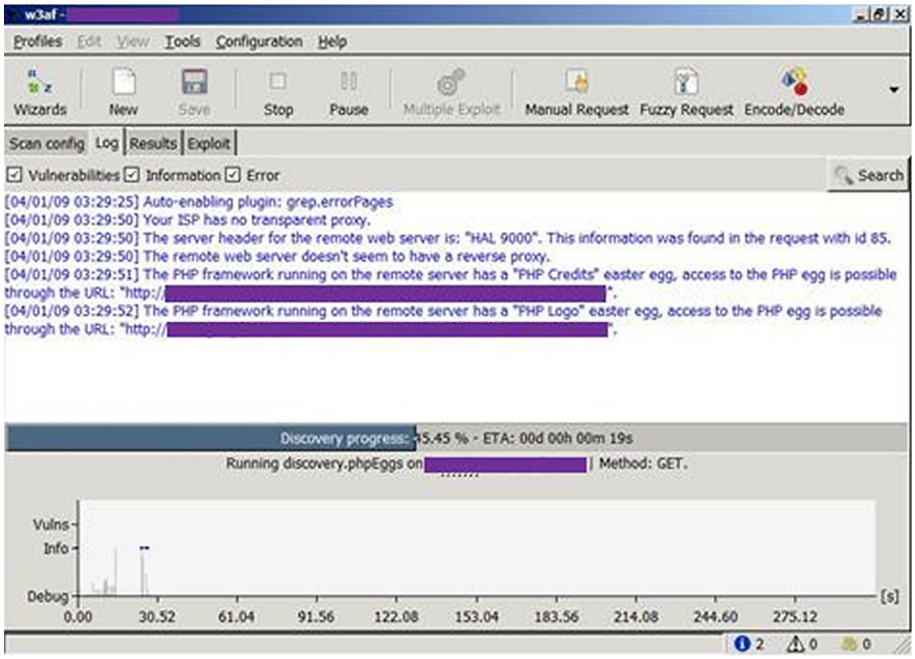
2.2.2. W3af

W3af es una herramienta de auditoría web *OpenSource* disponible tanto en Windows como Linux (nota al margen). Dispone de una interfaz gráfica de usuario y en línea de comandos. El objetivo de W3af es crear un *framework* para encontrar y ejecutar vulnerabilidades en aplicaciones web.

La característica principal de W3af es que su sistema de auditoría está basado completamente en *plugins*[4] escritos en Python, por lo que consigue crear un *framework* fácilmente escalable y una comunidad de usuarios que contribuyen con la programación de nuevos *plugins* ante los fallos de seguridad web que puedan ir apareciendo.

(4)Todos ellos están documentados en la web oficial de W3af.

Figura 7. W3af mostrando el visor de *logs* durante una auditoría web.

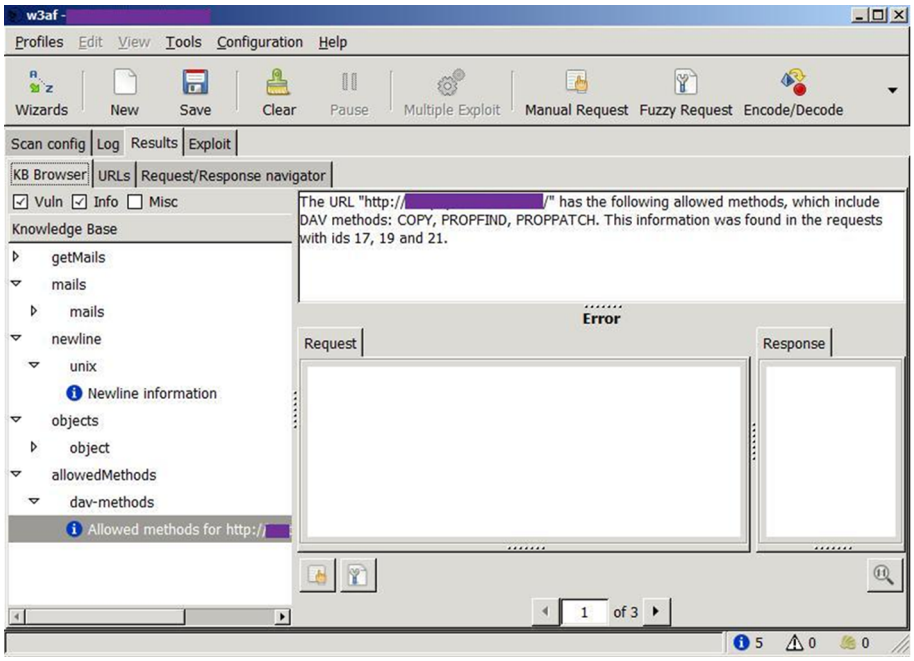


Entre las vulnerabilidades que detectan y explotan los *plugins* disponibles se encuentran:

- CSRF
- XPath Injection
- WebDAV
- *Buffer overflows*
- Extensiones de FrontPage
- SQL Injection
- XSS
- LDAP Injection
- Remote File Inclusion

Se puede observar cómo, a pesar de ser una herramienta libre, es tan completa como cualquier otra.

Figura 8. Informe generador por W3af



A pesar de haber comentado únicamente dos de los escáneres de vulnerabilidades webs más conocidos, existen muchos otros, cada uno con sus características propias. Sin embargo, el uso indiscriminado de estas herramientas puede dar lugar a problemas relacionados con la automatización de las tareas. Siempre será preferible que una persona ejecute estas pruebas debido a que sus acciones serán menos previsibles, más exactas y mejor adaptadas a la realidad de una intrusión.

3. Fortificación: servicios, permisos y contraseñas

Tanto un aplicativo web como una base de datos son piezas de software de una complejidad elevada que proporcionan una serie de servicios. Los **servicios** pueden ser:

- **Públicos** y accesibles directamente por otro software.

- **Privados** para efectuar determinadas tareas internas, o que interactúen únicamente con el sistema operativo que la hospeda.

En cualquier caso, al instalar cualquier software suelen activarse una serie de servicios por defecto que se encargan de proporcionar la mayor parte de la funcionalidad necesaria. Sin embargo, esto puede suponer un problema para la seguridad. Uno de los principios a aplicar es el de economía, esto es, utilizar el mínimo número de recursos para ofrecer el servicio estrictamente necesario. Por lo tanto, esto se traduce en utilizar únicamente el mínimo de servicios que proporcionen la funcionalidad deseada. Esto se debe a que cada servicio adicional es susceptible de sufrir algún tipo de vulnerabilidad: cuanto más servicios se ofrecen, más crece la ventana de ataque. Sin embargo, no siempre es sencillo determinar qué servicios son los necesarios.

Además de la habilitación y deshabilitación de servicios, también hay que securizar debidamente los que se decide dejar activos. Algunos servicios han sido célebres por presentar vulnerabilidades que han permitido el acceso a la base de datos; otros incorporan contraseñas por defecto (o simplemente no incorporan ninguna contraseña), por lo que se debe realizar un trabajo adecuado de configuración antes de publicar el servicio en entornos de producción.

A continuación se muestran algunas de las recomendaciones de seguridad en lo referente a las **contraseñas de los servicios**:

- Contraseñas por defecto: deben cambiarse todas estas contraseñas o deshabilitarse los servicios que las usan.

- Servicios sin contraseña: hay que evitar la existencia de este tipo de servicios o restringirlos adecuadamente mediante herramientas adicionales.

- Servicios poco usados: es importante discriminar qué servicios se usan realmente para deshabilitar los que no son necesarios.

Una vez se ha encontrado el conjunto de servicios que proporciona la funcionalidad necesaria para el entorno de trabajo, hay que concentrarse en los permisos que tienen estos servicios. La relación de la base de datos con el sistema

operativo se basa en un usuario que ejecuta los servicios a través de cierto número de programas. El usuario que ejecuta dichos programas dispone de una serie de privilegios en función del grupo al que pertenezca y a los privilegios adicionales de los que pueda disponer.

En caso de que un atacante logre el control del que se encarga de la ejecución de un servicio, nunca logrará realizar ninguna acción que no pueda hacer el usuario con el que se ejecuta el mismo. Por ello, hay que intentar mantener estos permisos al mínimo para evitar que, en caso de que un atacante tenga éxito, logre el control total del equipo. Hay un par de salvedades:

- En algunos casos es necesario ejecutar el servicio con el usuario con máximos privilegios en el sistema (sobre todo en entornos Windows).

- Aunque un atacante se haga con el control de un servicio corriendo con un usuario que no tenga todos los privilegios, siempre existe la posibilidad de que logre una escalada de privilegios, aunque esto añade dificultad al ataque.

Una vez definidos los permisos con los que interactúa la base de datos o el aplicativo web con el sistema operativo a través de los servicios, es necesario centrarse en los permisos que tendrán los distintos usuarios dentro de la propia base de datos o aplicativo. Los principios básicos que se han visto en el apartado anterior son también aplicables en este, como se explica a continuación.

Los distintos usuarios tienen distintos niveles de permisos en función de los roles asignados. Es algo parecido a los usuarios de un sistema operativo, en los que los permisos vienen dados por los grupos a los que pertenecen. Aparte del rol asignado, normalmente se pueden otorgar y revocar permisos individuales a los usuarios, por lo que los privilegios de un usuario no tienen por qué coincidir exactamente con los de los grupos a los que pertenece.

Por lo tanto, configurar los **permisos** para los usuarios es parecido a cómo se realiza el mismo proceso en un sistema operativo: intentar restringir a cada usuario para que únicamente tenga el mínimo conjunto necesario de permisos para realizar todas las tareas habituales, pero nada más.

El proceso consiste en dos pasos:

- **Autenticación**: el usuario se identifica en el sistema mediante el mecanismo correspondiente (habitualmente, par usuario-contraseña).

- **Autorización**: en función del usuario, se otorgan una serie de permisos asociados a su perfil sobre los objetos existentes en la aplicación o base de datos y las acciones que puede realizar sobre ellos.

Es recomendable no reaprovechar usuarios nunca, aunque ejecuten servicios que necesiten el mismo grupo de privilegios, ya que es importante disponer de usuarios individuales para, por ejemplo, seguir el rastro de un posible problema o intrusión mediante el registro de las acciones en los *logs* del sistema. También es una buena práctica emplear un sistema de nomenclatura de usuarios estándar que ayude a facilitar la tarea de administración, dando nombres descriptivos para los usuarios de servicios y empleando un mismo patrón para los usuarios "particulares".

Uno de los errores más típicos es el de usar un usuario con un gran conjunto de privilegios para la interacción entre la base de datos y un aplicativo web, en ocasiones el usuario administrador. Normalmente, los aplicativos necesitan de funcionalidad avanzada y con un nivel alto de permisos, por lo que se puede dar el caso de proporcionar un usuario administrador para simplificar la creación de un rol adecuado. Este tipo de decisiones son las que pueden comprometer toda la base de datos en caso de una intrusión, incluso el sistema operativo en el caso de que este usuario pueda ejecutar procedimientos que interactúen con el mismo.

Las escaladas de privilegios hacen referencia directa a estos roles y usuarios administradores. En caso de que un atacante (o usuario legítimo) disponga de un usuario con un conjunto de permisos limitados y quiera realizar ciertas acciones que no le son permitidas, puede intentar escalar privilegios en el sistema. En caso de conseguirlo, accede a otro conjunto superior de permisos o, en el mejor de los casos, a un usuario con permisos de administración, logrando un control potencial de todo el sistema.

Es importante revisar la configuración por defecto de la base de datos. En ocasiones existen una serie de roles predefinidos con una serie de permisos que no tienen por qué ser los que deseamos usar para nuestros usuarios.

Una vez definidos los roles y asignados los usuarios para cada uno de ellos, es importante definir una política[5]. Por ejemplo, puede aplicarse una política en cuanto a actualizaciones del sistema o en cuanto a emergencias. Una política no tiene por qué restringirse únicamente a las acciones a realizar de modo automático por la base de datos, sino que es mejor entenderlas desde un contexto más amplio. De hecho, puede ser un conjunto de acciones a realizar por parte del personal de administración en caso de una emergencia y estar escrita en un folio de papel. Sin embargo es importante definirlas, porque implica pensar sobre un problema y establecen un protocolo de actuación. En situaciones en las que es difícil pensar con claridad, siempre ayuda una guía hecha a conciencia y con tranquilidad.

(5)Una política es una serie de acciones y restricciones que se definen para la base de datos y que se puede aplicar en distintos niveles.

En una política se pueden añadir restricciones y controles que afecten en buena medida a los usuarios, uno de los ejemplos más habituales es el de las contraseñas. Se trata de la puerta de entrada al sistema de forma legítima, por lo que una política adecuada puede evitar muchos problemas. En general, no disponer de una política de contraseñas adecuada supone depender de los usuarios en cuanto a la complejidad de las mismas, lo que propicia que ataques de fuerza bruta puedan tener éxito y entrar de modo legítimo.

Finalmente, cabe hacer referencia al almacenamiento de estas contraseñas. Varía mucho en función de la base de datos que se utilice, de la versión y del método de autenticación dentro de la propia base de datos. Como se ha visto en las arquitecturas, algunas bases de datos permiten sistemas de autenticación mixtos que interactúan con el sistema operativo, otros son totalmente nativos, por lo que el modo de almacenamiento depende del mismo.

En algunas de las primeras versiones de MySQL, la contraseña se almacenaba totalmente en claro, por lo que si alguien conseguía el acceso a la tabla en la que se almacenan los usuarios, podía consultar tranquilamente su contenido. En otros casos, se utilizan sistemas de cifrado más o menos seguros. En caso de sistema de autenticación mixto, el almacenamiento de la contraseña suele correr por parte del sistema operativo. Por ejemplo, en versiones antiguas de SQL Server se almacena en una entrada de registro. En las primeras versiones se almacenaba totalmente en claro, posteriormente se optó por un método de cifrado, aunque era fácilmente reversible, por lo que tampoco era muy seguro.

Es importante conocer bien el sistema de autenticación para evitar dejar puntos débiles en el proceso, así como establecer todas las medidas y políticas necesarias, y dar a los usuarios únicamente los permisos estrictamente necesarios.

4. Desarrollo seguro

Como hemos podido ver, el principal punto de ataque para un software que comunique con la base de datos son los parámetros de entrada, todos ellos. Por lo que es importante filtrarlos sin excepción.

Pasemos a conocer las técnicas básicas existentes para evitar los problemas asociados con este aspecto:

1) Listas negras (o filtra lo que sabes que es malo)

Esta técnica se basa en filtrar todas las entradas que se corresponden con un patrón conocido de ataque o vulnerabilidad. De este modo, se suele tener una lista de patrones que puede consistir en literales o en expresiones regulares reconocidas como maliciosas, y cuando se detecta en cualquier entrada se realiza el filtrado.

Existen como mínimo dos problemas asociados con esta técnica, lo que la hacen poco recomendable:

- El primero es que una vulnerabilidad puede ser explotada de diversos modos, de modo que, a no ser que la lista negra sea muy exhaustiva, es muy posible que se pueda realizar un ataque exitoso a pesar de esta lista.

- El segundo problema está relacionado con la constante evolución de los ataques y de las diversas técnicas que surgen para explotarlos, de modo que las listas pueden quedar rápidamente obsoletas. Esto es un problema grave debido a la cantidad de dificultades que supone tener que gestionar listas de este tipo, especialmente si se encuentran en el código fuente del aplicativo.

2) Listas blancas (o dejar pasar lo que no es peligroso)

Este caso es el opuesto al anterior. Se dejan pasar las entradas que coinciden con una lista blanca, ya sea una lista de literales, expresiones regulares, o requisitos que debe cumplir la entrada, como que la entrada tenga un número de caracteres máximo o que esté compuesta únicamente por caracteres numéricos.

Cuando es posible aplicar esta técnica, se comprueba que posiblemente es la más efectiva debido a que los desarrolladores únicamente permiten el tratamiento de entradas que se sabe que no tiene ningún problema. Sin embargo, el problema estriba en que debido a lo restrictivo de esta técnica hay muchas ocasiones en las que no se puede aplicar, como entradas de texto libre o nom-

bres en los que haya que aceptar comillas simples, por ejemplo, que son caracteres usados frecuentemente para inyecciones SQL. De este modo, a pesar de ser una técnica efectiva, normalmente no es aplicable o es de difícil configuración debido a la complejidad de encontrar listas blancas adecuadas para toda la casuística de entrada.

3) Limpieza de parámetros

Esta aproximación considera que no es posible realizar una tipificación completa de la entrada para aplicar una lista blanca, y que, por lo tanto, la entrada no es confiable. Sin embargo, se realiza un tratamiento a estos parámetros para evitar que sean peligrosos, a pesar de no ser confiables, y para evitar el filtrado de los mismos.

En este caso, se trata de evitar caracteres o literales considerados como peligrosos, como por ejemplo las comillas simples para evitar casos de inyección de SQL, o literales como puede ser "`<script>`" para evitar ataques de *cross-site scripting*. Lo mismo sería aplicable con literales que se utilizan en sentencias SQL, como "`unión`" o "`select`". La eliminación de estos literales también puede cambiarse por sustitución.

Por ejemplo, la siguiente entrada:

```
<scri<script>pt>
```

quedaría como:

```
<script>
```

en caso de un tratamiento de eliminación no iterativo, de modo que este tipo de técnicas también tiene problemas si no se realiza adecuadamente. Aparte de la iteratividad, hay que tener en cuenta que las expresiones regulares deben ser flexibles para evitar tratamientos inadecuados en casos como entradas con mayúsculas, minúsculas y caracteres en blanco intercalados, por ejemplo. Es algo que puede resultar trivial, pero no lo es, y en muchas ocasiones hay filtrados aparentemente buenos y que fallan ante algún caso concreto, resultando inútiles en la práctica.

4) Tratamiento de excepciones adecuado

Existen muchos errores provocados por un tratamiento inadecuado de parámetros de entrada que son inesperados por la lógica del aplicativo. Durante mucho tiempo, este tipo de problemas en el tratamiento incorrecto de cadenas largas de caracteres ha provocado los errores de tipo desbordamiento de *buffer* en programas compilados, aunque en la actualidad ya no es algo tan frecuente.

Sin embargo, esto pone de relevancia la importancia de considerar todas las opciones, y en caso de producirse una condición inesperada, tener un tratamiento adecuado de excepciones que evite que el aplicativo se corrompa o que se muestre un mensaje de error que proporcione pistas a un atacante o pueda ser aprovechado para obtener datos internos de la propia base de datos.

Finalmente, y respecto a todo el tema del filtrado de parámetros de entrada, hay que considerar que existen técnicas de ofuscación para evitar ser detectados por este tipo de soluciones. Hay que tener en cuenta que existen distintas posibilidades para evitar reconocer entradas maliciosas, como puede ser el uso de diferentes tipos de codificaciones que reconoce el servidor de aplicativos o la base de datos, el caso de UTF8 o URL *encoding*, pero que pueden no tenerse en cuenta a la hora de realizar el filtrado.

Por otra parte, se pueden utilizar las funciones de la propia base de datos para convertir los caracteres a partir del código ASCII en literales que entiende e interpreta la base de datos.

Existen varias formas de intentar engañar a los filtros de entrada. A continuación se muestran varios ejemplos, cada uno de ellos explota una técnica distinta para evitar ser filtrados:

- Evitar caracteres bloqueados: aunque el aplicativo bloquee ciertos caracteres, siempre es posible intentar trabajar sin ellos. Por ejemplo, el filtrado de las comillas simples no afecta al caso de campos numéricos. En caso de filtrado de puntos y comas para la separación de sentencias cuando se intenta realizar una inyección de varias de ellas no es un problema, ya que la base de datos interpreta correctamente todas las sentencias si son correctas sintácticamente, a pesar de no estar separadas por punto y coma. Finalmente, en el caso de los símbolos de comentarios, pueden no ser necesarios si se usa una sentencia equivalente que no rompa la estructura sintáctica de la sentencia original.

- En el caso de las codificaciones, algunos filtros no las tienen en cuenta, por lo que en lugar de intentar la inyección con el literal SELECT, se puede realizar con la cadena URL-encodeada %53%45%4c%45%43%54.

- Inserción de comentarios dentro de las sentencias inyectadas. Al hacerlo, un parser incorrecto no detectará las palabras clave.

```
Select/*xx*/password/*xx*/from/*xx*/users;
```

En algunos casos, la base de datos permite insertar los comentarios incluso dentro de las propias palabras clave, de modo que hace aún más difícil crear un parser adecuado.

Uso de funciones de la propia base de datos para ofuscar sentencias. Funciones que crean sentencias a partir de codificaciones, como código ASCII, dificultan aún más la tarea de parsear adecuadamente la entrada. Por ejemplo, se podría usar `chr(97) ||chr(100) || chr(109) || chr(105) || chr(110)` en lugar del literal "`admin`" en una sentencia, lo que interpretaría la base de datos perfectamente.

Uso de ejecución dinámica. Algunas bases de datos permiten la ejecución de sentencias pasadas como parámetro directamente, interpretando la sentencia parámetro de diversas formas, como trozos de texto que posteriormente reensambla o hexencodea. Un ejemplo:

```
exec ('sel' + 'ect * f' + 'rom users');
```

Finalmente, tener en cuenta que todos estos filtros deben realizarse siempre en la parte del servidor, y nunca en la del propio cliente, que es quien puede realizar el intento de intrusión. Aunque filtrar los parámetros de entrada en el propio aplicativo del cliente (o en el navegador, en caso de ser un aplicativo web) pueda parecer una buena idea en cuanto a eficiencia (el filtrado se realiza en el propio cliente y se puede ahorrar un procesamiento innecesario en el servidor), esta solución es totalmente inefectiva. Cualquier filtrado en el cliente puede ser evitado de varias formas, sirva como ejemplo la más evidente: el tráfico de salida de su máquina, después de cualquier filtrado que pueda realizar cualquier aplicativo, puede ser interceptado por el propio usuario y modificado a su antojo, haciendo totalmente inútil el filtrado realizado por nuestro aplicativo.

La única forma de poder realizar este proceso sería mediante el uso de canales seguros de comunicación que encriptaran adecuadamente el tráfico de salida y establecieran una relación de confianza entre cliente y servidor. Aun así, hay que tener en cuenta que por muy segura que parezca una solución de este estilo es siempre susceptible de ser "hackeada" (por ejemplo, se puede realizar ingeniería inversa en el aplicativo y evitar cualquier filtrado), por lo que el filtrado en el servidor siempre es necesario.

Para terminar, comentaremos algunos de los problemas más importantes que tiene el desarrollo seguro. Existe una gran cantidad de software que se desarrolla para pequeñas aplicaciones de forma rápida y descuidada, formando la semilla de la desgracia. Las prisas, los presupuestos ajustados que no tienen en cuenta la seguridad, el uso de *frameworks* de desarrollo aparentemente milagrosos, pero que no tienen en cuenta aspectos básicos para evitar vulnerabilidades típicas, la convivencia de software heredado que no sabe nadie cómo está hecho ni tampoco se atreven a tocarlo... Todos estos escenarios son muy frecuentes en la actualidad, y suponen posiblemente el peor problema en cuanto a desarrollo inseguro.

4.1. Auditoría de código fuente

Esta técnica consiste, en su versión más básica, en realizar búsquedas dentro del mismo código para localizar patrones de fragmentos de código que sean potencialmente vulnerables a problemas conocidos. Lo recomendable es que la auditoría no la realice la misma persona que ha escrito el código, sino un equipo distinto, para asegurar la imparcialidad.

Podemos pensar que realizar una auditoría de este estilo con un código de tamaño grande puede ser una odisea, sin embargo, existen herramientas para ayudar en esta tarea. En este punto, hay que destacar que auditar código no es solo realizar una búsqueda de texto, aunque algunas herramientas se limitan a ello, sino que se trata de algo más complejo. Es necesario seguir la ejecución del programa y la contextualización del código en ciertos puntos potencialmente vulnerables, como es el filtrado de parámetros de entrada y la interacción con la base de datos. La solución más adoptada consiste en una mezcla entre herramientas automáticas y validación manual, como acostumbra a ocurrir en estos casos.

Destacar que una programación ordenada y modular ayuda en gran medida a la limpieza del código y la estructuración del mismo, por lo tanto, hace mucho más sencilla la auditoría. Es por ello por lo que el seguir unas buenas prácticas de programación repercute en la seguridad del sistema. Un código que solo entiende el programador que lo ha escrito es difícil de auditar por parte de terceros.

Veamos un ejemplo de lo que se podría buscar en el código fuente para intentar evitar una inyección de código SQL. Se trata de buscar los puntos potencialmente vulnerables a este problema y asegurar que están debidamente securizados; como por ejemplo, un filtrado adecuado de los parámetros de entrada. Una de las construcciones que pueden dar lugar a este problema son las sentencias creadas dinámicamente junto con parámetros de entrada del usuario. Por ejemplo:

```
StringBuilder consulta = newStringBuilder ("Select id, nombre, descripcion, precio
From libros Where2 + condicion);
```

```
StringBuilder consulta.Append ("id=");
consulta.Append (Request.QueryString("ID").toString());
```

En este caso, la construcción sería vulnerable a una inyección de código SQL en caso de que no se haga un filtrado apropiado del parámetro ID en ningún punto del aplicativo. Una forma sencilla de buscar partes de código que tengan este problema potencial es buscar cadenas que forman parte de la construc-

ción de sentencias SQL, como "Select", "Insert", "Delete", etc. También habría que tener en cuenta el caso en el que estos literales se asignaran a constantes, de modo que la misma búsqueda se debería aplicar a los estos.

Más que la búsqueda de problemas concretos, muchas veces se trata de buscar fragmentos de código potencialmente peligrosos debido a su función. Básicamente, se trata de intentar buscar la interacción con el sistema de ficheros, con el sistema operativo y con la base de datos. Nos centraremos en este último.

La búsqueda de los puntos de interacción varía en función de la tecnología usada. Por ejemplo, en Java se utilizan comúnmente las siguientes API para ejecutar consultas contra la base de datos, por lo que sería interesante buscar los puntos en los que se utilizan para estudiar que no haya problemas:

```
java.sql.Connection.createStatement
java.sql.Statement.execute
java.sql.Statement.executeQuery
```

Existe una alternativa que representa una API más robusta a problemas de seguridad, como las inyecciones:

```
java.sql.Connection.prepareStatement
java.sql.PreparedStatement.setString
java.sql.PreparedStatement.execute
java.sql.PreparedStatement.executeQuery
```

En el siguiente ejemplo, se puede ver que en caso de usarse debidamente, se evita una inyección SQL, mientras que con la primera API el código sería vulnerable:

```
String usuario = "admin' or 1=1-";
String pass = "contraseña";
Statement s = con.prepareStatement(&#8220;Select * from usuarios where usuario =? and password = ?");
s.setString(1,usuario);
s.setString(2.pass);
s.executeQuery();
```

Quedando el siguiente código que sería el que se ejecutase contra la base de datos:

```
Select * from usuarios where usuario = 'admin'' or 1=1-- and password = 'contraseña';
```

En este caso, no se lograría el efecto deseado por parte del atacante, siendo mucho mejor la API en cuanto a seguridad. Sin embargo, un buen filtrado previo de parámetros, tal y como se ha explicado previamente, sería muy recomendable.

Siguiendo con el repaso a tecnologías, en ASP.NET las siguientes son las API más comunes para interaccionar con la base de datos:

```
System.Data.SqlClient.SqlCommand
System.Data.SqlClient.SqlDataAdapter
System.Data.Oledb.OleDbCommand
System.Data.Odbc.OdbcCommand
System.Data.SqlServerCe.SqlCeCommand
```

Igual que en el caso anterior, es necesario trabajar con los métodos recomendados en dichas API para evitar inyecciones. Para lograr un efecto igual al ejemplo anterior en Java, sería necesario utilizar los métodos para añadir parámetros a sentencias disponibles en todas las API anteriores. Por ejemplo:

```
OdbcCommand c = new OdbcCommand ("Select * from users where usuario = @user and password
= @pass", connection);
c.Parameters.Add (new OdbcParameter('@user',OdbcType.Text).Value=user);
c.Parameters.Add (new OdbcParameter('@pass',OdbcType.Text).Value=pass);
c.ExecuteNonQuery();
```

Finalmente, y para acabar el repaso a algunos de los lenguajes más comunes, veamos las API más comunes para la comunicación con la base de datos en PHP:

```
mysql_query
mssql_query
pg_query
```

Siendo los siguientes métodos los más recomendables para evitar las inyecciones, y que permiten la inserción de parámetros de manera controlada, como se ha visto en los ejemplos anteriores:

```
mysqli->prepare
stmt->prepare
stmt->bind_param
stmt->execute
odbc_prepare
```

Por ejemplo, igual que en los casos anteriores, un código que sería equivalente a los casos no vulnerables a inyección:

```
$sql = $db_connection->prepare ("Select * from users where user = ? and password = ?");
$sql->bind_param("ss",$user,$pass);
$sql->execute();
```

En PHP hay una directiva relacionada con la seguridad en cuanto a filtrado de parámetros que se define en el entorno del sistema. Se trata de `magic_quotes_gpc`, que en caso de estar activada, cualquier comilla simple, doble, antibarra (`\`) y carácter nulo, se "escapan" utilizando una antibarra. Existe una directiva similar llamada *`magic_quotes_sybase`*, en la que el comportamiento es el mismo con la diferencia de que el "escape" se realiza mediante una comilla simple. Aunque esta funcionalidad no evita las inyecciones totalmente (por ejemplo, con los campos de tipo numérico), suponen una dificultad añadida a la explotación del sistema por parte de un atacante. Hay que considerar que el comportamiento añadido por esta directiva puede no ser el deseado para el aplicativo que lo usa. Además, el hecho de confiar en una directiva del entorno en lugar de asegurar el filtrado en el propio aplicativo no es lo más recomendable. Hay que intentar realizar el filtrado en todos y cada uno de los módulos que configuran el sistema, por lo que cuanto más medidas, mejor, pero nunca confiar en un filtrado para descuidar otro.

La funcionalidad de *`magic_quotes`* ha sido una de las más extendidas y utilizadas en PHP 5, existiendo en la actualidad en millones de desarrollos, y por este motivo se comenta en este curso. Sin embargo, en la nueva versión de PHP ya no está presente.

4.2. Herramientas de filtrado: Web Application Firewalls

Existen herramientas diseñadas para eliminar algunos de los problemas explicados anteriormente, como el filtrado de parámetros de entrada. El uso de estas herramientas NO sustituye el crear un código seguro en ningún caso. Sin embargo, pueden servir para añadir una capa extra de seguridad en caso de que no podamos realizar un desarrollo propio o utilicemos software de terceros.

Web Application Firewalls son las aplicaciones que se instalan en servidor para controlar las peticiones malintencionadas que podrían dar lugar a un posible ataque. Esto se hace a través de unas reglas que se configuran y que permiten detectar posibles ataques incluso antes de que se produzcan. Además, estos actúan en conjunción con los servidores web, permitiendo el bloqueo de la petición maliciosa y evitando que el ataque llegue a producirse.

Los *web application firewall* (o WAF) son un tipo de sistemas de detección de intrusos[6] a nivel de aplicación. Existen otros tipos de sistemas de detección de intrusos, como por ejemplo a nivel de red, que analizan los paquetes en busca de patrones conocidos. Sin embargo, aunque estos puedan ser más efectivos contra ataques generales, se muestran menos eficientes contra ataques más selectivos como son, por ejemplo, los ataques web. Además, al ser específicos para una aplicación, actuarán mejor y serán más eficientes.

(6) IDS son sus siglas en inglés.

En la actualidad existen múltiples soluciones para proteger a nuestros servidores pero aquí vamos a ver dos de ellas:

- *Mod_security*, un módulo para Apache que nos permitirá detener peticiones maliciosas y falsificar algunos de los datos que genere nuestro servidor para engañar a un posible atacante.

- *Request Filtering*, incluido con Internet Information Services 7 y que nos ofrece una manera gráfica de crear reglas de filtrado.

4.2.1. Mod_Security

Mod_Security es un módulo para Apache. Apache permite la expansión de sus funcionalidades a través de módulos. Estos módulos suelen estar programados en C o C++ y se pueden encontrar fácilmente en Internet.

Para la configuración del servidor necesitaremos descargar el módulo apropiado para nuestro sistema operativo o directamente compilar el código fuente. En cualquier caso deberemos obtener un fichero `mod_security2.so` que colocaremos en una carpeta `mod_security2` dentro de la carpeta de módulos del Apache.

Con esta acción no tendremos aún el módulo funcionando, sino que debemos realizar una pequeña configuración. Esta configuración pasa por establecer una serie de líneas dentro del fichero `httpd.conf` de la carpeta `conf`.

Las líneas que tenemos que añadir son:

```
LoadModule unique_id_module modules/mod_unique_id.so
LoadModule security2_module modules/mod_security2/mod_security2.so
Include conf/rules/*.conf
```

La última línea, la que hace referencia a los ficheros con extensión `conf` dentro de la carpeta `rules`, es la que se encarga de cargar todas las reglas de configuración que se descargan junto a `mod_security`. Estas reglas contienen la configuración básica del módulo así como diversas reglas específicas para distintos tipos de ataques.

La configuración básica pasa por editar el fichero `modsecurity_crs_10_config.conf`, que contiene la estructura base para empezar a funcionar con este módulo. Deberemos comentar la siguiente línea para que el módulo empiece a "logear" las peticiones que detectamos como posibles ataques:

```
SecDefaultAction "phase:2,log,deny,status:403,t:lowercase,t:replaceNulls, t:compressWhitespace"
```

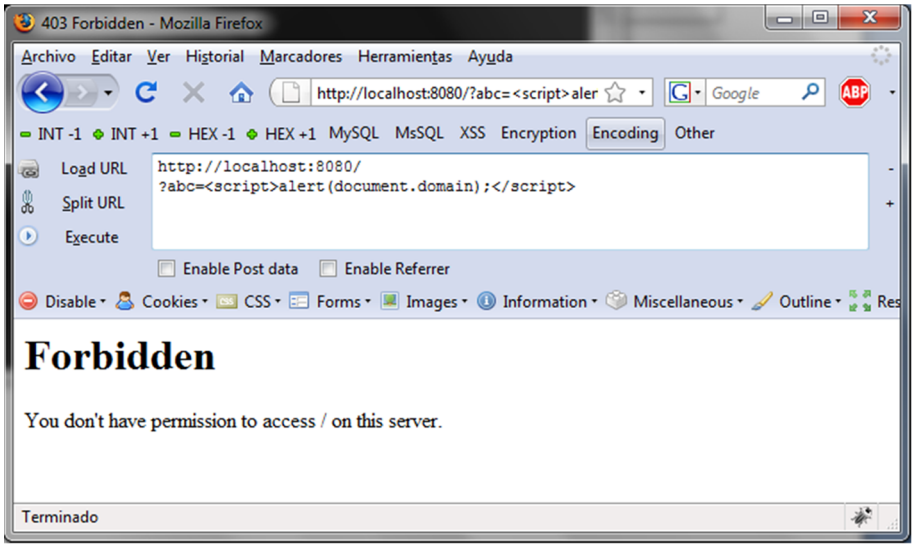
Esta línea lo que hace básicamente es activar el `mod_security` en modo `log` y deniega (*deny*) todas las peticiones que cumplan una regla de las activadas. Además, devolverá un código 403 de recurso prohibido.

Si hemos seguido los pasos correctamente y hemos reiniciado el servidor Apache (es necesario hacer esto para que el fichero de configuración se lea de nuevo y se cargue en memoria), podremos intentar producir un ataque controlado. Aunque actualmente nuestro servidor no dispone de ninguna aplicación web, podemos generar una petición que contiene un ataque Cross Site Scripting (XSS).

```
http://localhost/?abc=<script>alert(document.domain);</script>
```

Esta simple petición web nos devolverá un error 403 y generará una entrada en el fichero de `log`, por defecto el fichero `log/mod_security2.log`

Figura 9. Página de error producida por *mod_security*



En la figura anterior podemos ver una petición con XSS bloqueada. La regla que ha saltado en el ejemplo anterior para bloquear la petición que contiene la cadena de XSS se encuentra en el fichero `modsecurity_crs_40_generic_attacks.conf`. La regla se compone de expresiones regulares que permiten localizar un posible ataque de XSS.

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?
|ecma|vb)|application\b\W*?\bx-(?:java|vb))script|c(?:opyparentfolder|reatetextrange)|get
(?:special|parent)folder|iframe\b.{0, 100}?\bsrc)\b|on(?:(?:mo(?:use(?:o(?:ver|ut)|down
|move|up)|ve)|key(?:press|down|up)|c(?:hange|lick)|s(?:elec|ubmi)t|(?:un)?load|dragdrop
|resize|focus|blur)\b\W*?=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:(?:java|vb)script|shell|http)
|ivescript)|(?:href|url)\b\W*?\b(?:(?:java|vb)script|shell)|background-image|mocha):|s
(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)\(|rc\b\W*?\b(?:(?:java|vb)script|shell
|http):)|a(?:ctivexobject\b|lert\b\W*?\(|sfunction:))<(?:(?:body\b.*?\b(?:backgroun|onloa)d
```

```
|input\b.*?\btype\b\W*?\bimage)\b| ?(?:(?:script|meta)\b|iframe)|!\[cdata\[)|(?:\.(?:
(?:execscrip|addimpor)t|(?:fromcharcod|cooki)e|innerhtml)|\@import)\b)" \"phase:2,capture,
t:none,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,ctl:auditLogParts=+E,log,auditlog,
msg:'Cross-site Scripting (XSS) Attack',id:'950004',tag:'WEB_ATTACK/XSS',logdata:'%{TX.0}',
severity:'2'"
```

Al igual que esta regla para detectar ataques XSS existen muchas otras para, de una manera genérica, detectar ataques de SQL Injection, LDAP Injection, Local File Inclusion, etc.

4.2.2. Request Filtering

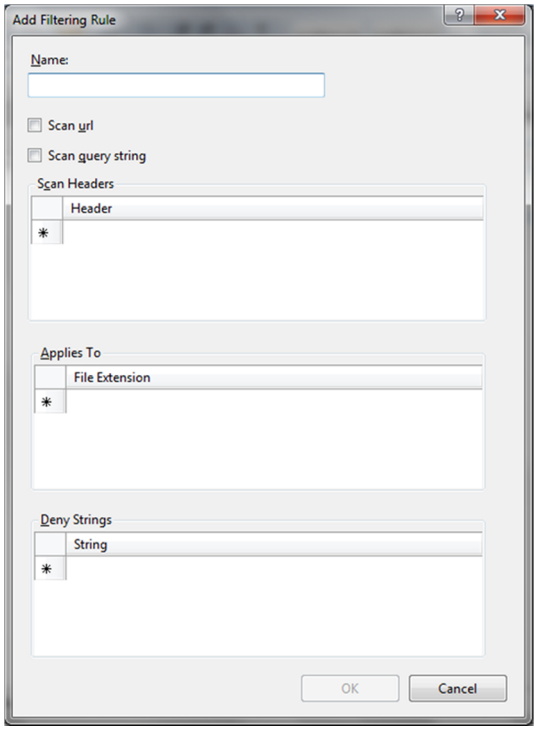
El servidor web de Microsoft, Internet Information Services, en su versión 7 incluye un módulo especializado en el bloqueo de peticiones web que pudieran ser maliciosas. Este módulo se puede activar junto con la instalación del servidor web y no requiere de configuración inicial.

Una vez instalado y abierta la consola de configuración, nos encontraremos frente a una interfaz gráfica que nos va a permitir establecer restricciones en distintos ámbitos de las peticiones web. Estos son:

1) Extensiones de ficheros: A veces no queremos que los usuarios puedan acceder a tipos de ficheros concretos, como por ejemplo los que tienen extensión `.bak` o `.old`. Estas extensiones son típicas de cuando realizamos acciones de actualización de nuestra aplicación web y suelen suponer un grave riesgo de seguridad. Con este filtro podemos bloquear extensiones que sabemos que nunca usará nuestra aplicación. Si hemos instalado el soporte para ASP.Net, se nos añadirán automáticamente como denegadas distintos tipos de extensiones que se usan durante el desarrollo de una aplicación web. Esto se hace para evitar un error de un programador que suba un fichero no deseado a nuestro servidor.

2) Reglas personalizadas: Quizás la zona más parecida a `mod_security`. Desde aquí y mediante algunas opciones podemos generar reglas de bloqueo. Permite un mayor control sobre las reglas.

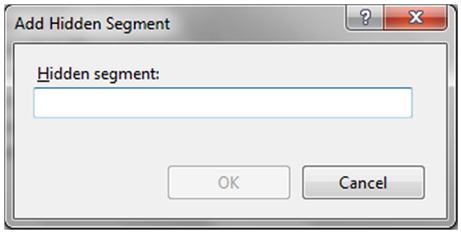
Figura 10. Diálogo de generación de reglas



3) Zonas ocultas: En toda aplicación web existen unas zonas privadas donde los programadores han almacenado ficheros de configuración. Estos directorios no deberían ser accesibles por nadie desde su navegador. Desde esta zona estableceremos los directorios que queremos bloquear. Por defecto, y si hemos instalado soporte para ASP.Net, tendremos los siguientes elementos:

- `web.config`
- `bin`
- `App_code`
- `App_GlobalResources`
- `App_LocalResources`
- `App_WebReferences`
- `App_Data`
- `App_Browsers`

Figura 11. Bloqueo de segmentos

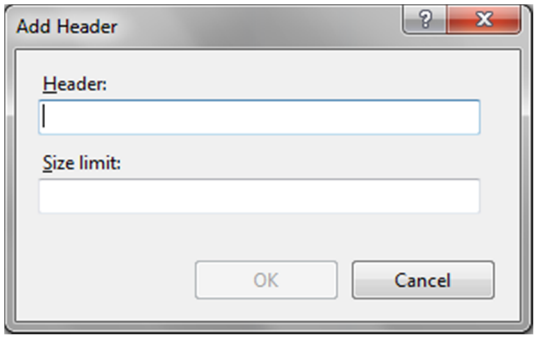


4) URL: El apartado URL es quizás la zona más general. Desde aquí podemos prohibir el acceso a URL completas. Esto será útil para, por ejemplo, bloquear un fichero concreto dentro de un directorio.

5) Verbos HTTP: Muchos ataques web se producen al usar el atacante verbos HTTP incorrectos. Esto puede provocar que la aplicación no sepa cómo responder o que responda de una manera no deseada. Desde este apartado podemos bloquear o permitir ciertos verbos, como `GET`, `POST` o `HEAD`.

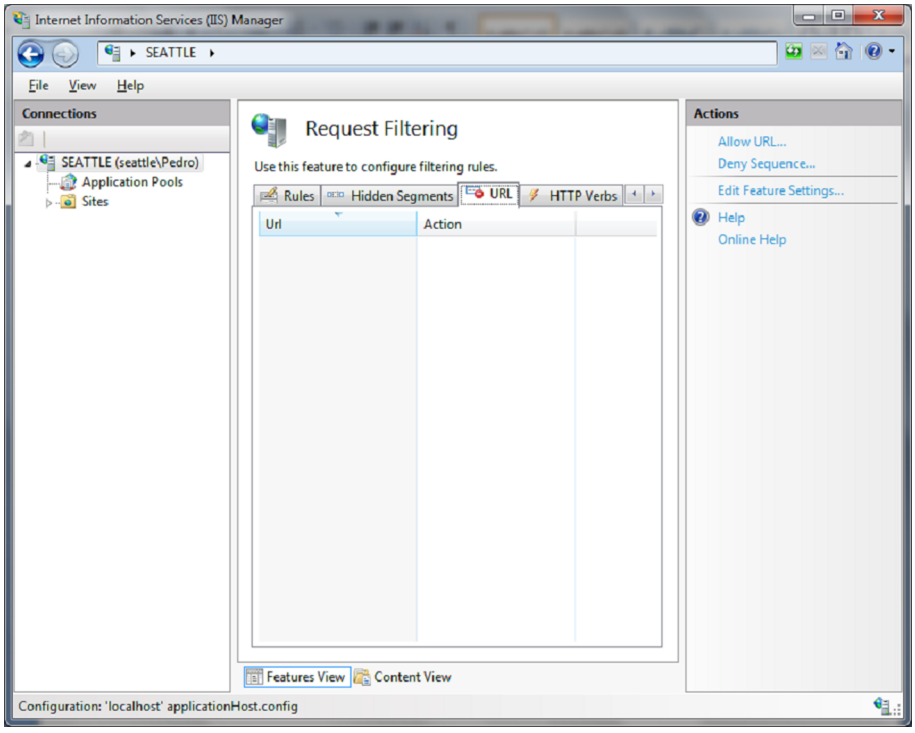
6) Cabeceras: Los valores enviados en las cabeceras HTTP a veces son interpretados por las aplicaciones para generar estadísticas o recopilar datos necesarios de los clientes. A veces los programadores olvidan sanear las entradas recibidas mediante cabeceras por considerarlas imposibles de modificar. Desde este apartado podremos bloquear cabeceras específicas.

Figura 12. Bloqueo de cabeceras



7) Parámetros: El mayor número de ataques viene producido por el uso de parámetros incorrectos o malformados. Desde esta última opción podremos establecer filtros frente a cadenas que puedan ocasionar un riesgo para nuestro servidor.

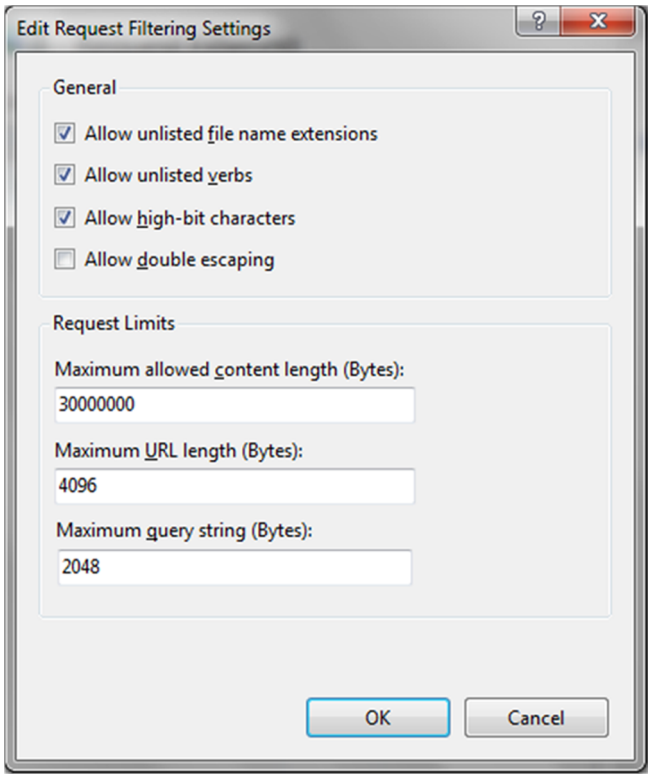
Figura 13. Pantalla general del módulo Request Filtering



Además de todas estas opciones específicas, para elementos concretos de las peticiones HTTP podemos encontrar aún un último apartado de configuración. La configuración general del módulo que nos permite establecer una serie de límites en cuanto al tamaño de las peticiones y a algunas características de seguridad permitidas se refiere:

- Permitir extensiones de ficheros no listadas.
- Permitir verbos no listados.
- Permitir caracteres de código ASCII extendido (como la ñ).
- Permitir doble escapado de caracteres.
- Tamaño máximo de la petición.
- Tamaño máximo de la URL.
- Tamaño máximo de los parámetros.

Figura 14. Configuración general del módulo



Estas opciones son muy útiles. Los tamaños especificados por defecto como máximos son lo suficientemente grandes como para que ninguna página deje de funcionar. Estos tamaños deberemos ajustarlos a los límites reales de nuestro servidor después de realizar un análisis del uso de la aplicación.

El objetivo será dejar a nuestros usuarios interactuar con nuestro servidor durante un periodo de tiempo prudencial, permitiendo que realicen todas las tareas posibles en la aplicación. Después de este tiempo debemos analizar el *log* del servidor e inferir cuáles son los límites reales de nuestra aplicación. Por ejemplo, si hemos detectado que ninguna URL excede de los 68 caracteres y

que los parámetros enviados no sobrepasan los 54, podríamos establecer unos límites de quizás serían de 80 caracteres para la URL y 70 para los parámetros, dejando un margen de seguridad para no bloquear futuras peticiones.

Esto es muy útil frente a ataques de SQL Injection donde los atacantes pueden llegar a generar URL de más de 500 caracteres, debido a las múltiples condiciones que han de establecer para extraer en cada momento el contenido deseado de la base de datos.

Bibliografía

Hope, H and Walther, W. (2009). *Web Security Testing CookbookSystematic Techniques to Find Problems Fast*. O'Reilly Media.

Mcdonald, J. (2006). *Art of Software Security Assessment*. Pearson Professional Education.

Ristic, I. (2005). *Apache Security*. Ed. O'Reilly.

Schaefer, K.; Cochran, J.; Forsyth, S.; Baugh, R.; Everest, M. and Glendenning, D. (2008). *Professional IIS 7*. Ed. Wrox.

Takanen, A.; Demott, J.D. and Miller; C. (2007). *Fuzzing for Sotfware Security Testing and Quality Assurance*. Ed. Artech House.

Webs de interés

http://www.petefinnigan.com/orasec.htm

http://www.codeproject.com/KB/database/SqlInjectionAttacks.aspx

http://www.techsupportalert.com/search/p1528.pdf

http://www.microsoft.com/sql/prodinfo/previousversions/securingsqlserver.mspx

http://www.databasesecurity.com/oracle-forensics.htm

http://iase.disa.mil/stigs/checklist/

http://www.codeproject.com/KB/database/SqlInjectionAttacks.aspx

http://msdn.microsoft.com/

http://www.ngssoftware.com/research/papers/cursor-snarfing.pdf

http://www.sommarskog.se/dynamic_sql.html