ARTÍCULO MONOGRÁFICO

Prácticas seguras de programación para sistemas web[1]

[Safe practice of programming systems for web]

Gunnar Eyal Wolf-Iszaevich [2]
Recibo: 04.09.2009 - Aprobación: 12.11.2009

Resumen

El campo de la seguridad en cómputo muchas veces es visto como un trabajo investigativo (búsqueda de nuevas categorías de fallos), reactivo (corrección de fallos encontrados, o incluso buscarlos proactivamente) o, en el peor de los casos, un campo donde los personajes más destacados son quienes saben abusar de los sistemas ajenos. Nada más lejano de la realidad. En la seguridad en cómputo, el rol más importante debe ser el desarrollador de sistemas, una de las piezas más importantes de la sociedad actual. Este trabajo parte definiendo qué debe entenderse por seguridad en cómputo, y por qué este concepto debe ir más allá de definiciones duras y frías, para reflejar que antes que otra cosa, el cómputo es una disciplina con la misma flexibilidad que los estudios humanísticos.

La parte medular del trabajo se centra en ejemplificar, analizando tres categorías de vulnerabilidad informática de alto perfil hoy en día (inyecciones de SQL, Cross-Site Scripting y manejo de sesiones a través de galletas HTTP), de especial relevancia para los sistemas Web desarrollados hoy en día, poco abordadas en especifico por los principales textos con que se enseña la disciplina.

1 Modelo para citación:
WOLF ISZAEVICH, Gunnar Eyal. (2009). Prácticas seguras de programación para sistemas web. En: Ventana Informática. No. 21 (jul-dic., 2009). Manizales (Colombia): Universidad de Manizales. p. 91-111. ISSN: 0123-9678

2 B.Sc. Computer Science. Técnico Académico Asociado C de Tiempo Completo, Instituto de Investigaciones Económicas, Universidad Nacional Autónoma de México, México. Correos electrónicos: gwolf@iiec.unam.mx, gwolf@gwolf.org.

Palabras Clave: *Seguridad en cómputo, redefinición, inyecciones SQL, mapeo, Objeto-relacional, cross-site scripting, XSS, sesiones HTTP, Galletas HTTP, resumen criptográfico.*

Abstract

The Computes Security field is often seen as one where work is mostly devoted to research (finding new bug/vulnerability categories), reactive (fixing bugs as they are found, or even proactively looking for them), or in the worst case, a field where

the most renowned characters are those who can abuse other systems. Nothing farther from truth In Computer Security, the most important role must always be that of the systems developer, one of the key parts of today's society.

This work starts by defining what we want to understand about Computer Security, and why this concept must reach beyond the hard and cold definitions, to reflect that, before anything else, computing is a discipline with the same amount of flexibility tan humanistic studies.

The core of this work is centered on exemplifying, by analyzing three currently high-profile computer vulnerability categories which are especially relevant for Web systems developed today (SQL injections, cross-site scripting and session handling through HTTP cookies), which do not receive enough coverage by the main texts with which this discipline is learnt.

Keywords: *Computer Security, redefinition, SQL injection, object-relation, mapped, cross-site scripting, XSS, HTTP sessions, HTTP cookies, cryptographic digest*

Introducción

La evolución del rol que cumplen los sistemas en las organizaciones ha cambiado por completo -afortunadamente- el punto de vista que la mayor parte de los desarrolladores tiene con respecto a la seguridad.

Hace una o dos décadas el tema de la seguridad en cómputo era frecuentemente evitado. Y hasta cierto punto, justificable: ¿Intrusos? ¿Integridad? ¿Validaciones? Conceptos que hoy a todos parecen fundamentales eran vistos como distracciones teóricas que sólo entorpecían la usabilidad de los sistemas. En la década de los 80, había muy poco software diseñado para su operación en red, y mucho menos, para la idea de red que se

tiene hoy en día. Y si bien es cierto que la mayor parte de los ataques se originan -y siempre se han originado- dentro del parámetro de confianza de la organización, hace 20 años sencillamente había menos información sensible alojada en medios electrónicos, menos gente con el conocimiento necesario para manipularla, e incluso la manipulación tenía efectos más nocivos: Si bien hace años la infraestructura de cómputo era el soporte facilitador, la copia maestra estaba en papel - Hoy se transita hacia la situación opuesta, en que la versión electrónica es la primaria. Hoy, una intrusión en los sistemas puede poner en jaque la integridad de la información primaria.

Mantener la seguridad en los sistemas que se desarrollan implica un alto costo: Los programadores tienen que aprender a evitar errores comunes; tienen que concientizarse y acostumbrarse a dedicar recursos a implementar baterías de pruebas; tienen que entrar en juego validaciones y conversiones sobre los datos que se manipulan, con costos en tiempos de procesamiento de cada solicitud... Pero, afortunadamente, ha crecido también la conciencia de la importancia de destinar a la seguridad la atención y recursos que requiere.

El problema sigue siendo, coloquialmente... *¿con quién se come?.* La seguridad en cómputo sigue siendo un campo difícil de entender, con muchas aristas ocultas. En este articulo se abordan algunos temas fundamentales, que a pesar de ser bien conocidos, siguen siendo origen de un sinfín de nuevos sistemas creados con obvios agujeros.

1. ¿Qué es la seguridad en cómputo?

A riesgo de que parezca perogrullada, *un sistema seguro no es otra cosa que un sistema que responde como debe, un sistema que cubre las necesidades y requerimientos con que fue concebido*. Claro, esta pregunta hay que verla a la luz de varios criterios para que en verdad signifique algo. Un sistema seguro presenta:

- **Consistencia:** Ante las mismas circunstancias, debe presentar el mismo comportamiento. Ante un sistema seguro, el tradicional remedio de ¿ya intentaste reiniciarlo?, no surte efecto. Si una grandísima proporción de usuarios se ha acostumbrado a que un reinicio resuelve las cosas, es porque el ambiente de ejecución se ensucia con elementos que debería haber descartado. Por tanto, puede concluirse que los sistemas en cuestión son inherentemente inseguros.

- **Protección y separación:** Los datos, instrucciones y espacio de memoria de un programa, componente o usuario no deben interferir ni permitir interferencia de otros. Las condiciones anormales ocurridas en uno de los componentes -sean accidentales o expresas- deben tener un impacto mínimo en el sistema como un conjunto.
- **Autenticación:** El sistema debe poseer los mecanismos necesarios para asegurarse que un usuario es realmente quien dice ser.
- **Control de acceso:** El sistema debe poder controlar con toda la granularidad necesaria, los permisos de acceso a sus datos, es decir, quién tiene acceso a que recursos, y qué tipo de acceso tiene.
- **Auditoría:** El sistema debe ser capaz de registrar, así como de notificar, a quien sea necesario, de cualquier anomalía o evento importante.

Los atributos mensionados deben ir matizados, priorizándolos al *nivel adecuado a las necesidades*. Ir demasiado lejos en uno de ellos puede ser perjudicial para los fines del sistema; por ejemplo, es de todos bien conocido que el tradicional esquema de autenticación basado en usuario y contraseña es fácil de engañar, basta adivinar (o conseguir) un pedazo de información, típicamente de muy débil complejidad, para estar autenticado como determinado usuario. En México, desde hace algunos años, los bancos exigen la identificación del cliente a través de dispositivos que presenten una mayor complejidad, generando cadenas de números que cambian periódicamente.

Pero, obviamente, poca gente requerirá un nivel de seguridad similar a éste, o basado en parámetros biométricos, para abrir su cuenta de correo. Si bien es aceptable demandar que un usuario bancario tenga acceso al dispositivo como una medida de seguridad (lo cual es una inconveniencia, dado que muchos usuarios prefieren no cargarlo consigo constantemente, precisamente pensando en la seguridad), requerir medidas similares para acceso al correo electrónico será de inmediato criticado por todos los usuarios como excesivo y como limitante a la usabilidad.

Y otra anotación: Es natural aspirar a la perfección, al 100%. Sin embargo, dice el refrán que *'lo mejor es enemigo de lo bueno'*. Es importante recordar que, en toda etapa de la planeación, desarrollo, implantación y tiempo de vida de un sistema, un 100% de seguridad es una utopía, un objetivo que sólo puede servir para guiar el trabajo diario.

Los programas son escritos por humanos, y son también humanos quienes administran los sistemas en que corren. Hay una gran cantidad de interacciones entre los elementos de un programa y el sistema, y un cambio en cualquiera de ellos puede tener consecuencias inesperadas si no se hace con cuidado y conciencia. Constantemente aparecen nuevas categorías de errores capaces de llevar a problemas de seguridad. Parte fundamental, de la actuación como profesionales en este campo, debe ser la constante actualización con los últimos desarrollos y las últimas amenazas.

2. El estado actual de la seguridad en cómputo

En un estudio, Sans y Mitre (2009) publicaron la lista de los 25 errores de seguridad más importantes en frecuencia y relevancia. Este listado incluye muchos temas fundamentales que deberían ser comprendidos por cualquier programador que se diga profesional. Vienen explicados con un buen nivel de detalle, detallando como evitar o mitigar sus efectos. Este estudio ha contado con amplio apoyo por parte de diversos sectores, tanto de la academia como de la industria.

Sin embargo, volviendo al tema, en realidad ¿el hecho de que ahora la seguridad en cómputo sea un tema que forma parte del inconsciente colectivo, realmente ha llevado al mejoramiento de la calidad general del código? Es muy posible estar ante un claro caso de falso sentido de seguridad.

Usuarios y programadores están al tanto del peligro que corren al utilizar un sistema informático. Obviamente, los programadores están (o por lo menos, deberían estar) más atentos, dado que se dedican profesionalmente al cómputo y que comprenden (o, nuevamente, deberían comprender) mucho mejor las interacciones que llevan a las principales vulnerabilidades. Sin embargo, una y otra vez se encuentran sistemas con las mismas clases básicas de errores, de tremendos agujeros de seguridad, a través de los cuales podrían pasar marchando regimientos completos.

¿Por qué sucede esto? En síntesis, podría resumirse en los siguientes factores:

- **Confianza ciega en la propaganda de los proveedores.** Prácticamente todos los proveedores de herramientas de desarrollo, de

modelado, de verificación aseguran ser la más segura. En el mundo del software propietario (en contraposición al Software Libre), muchos usuarios tienden a creer en la voz de los gurúes sin poderla verificar, algo que no pueden siquiera validar lo que estos anuncian.

- **Ceguera.** Si bien algunos modelos de programación actuales hasta protegen de las más graves fallas técnicas, muchos programadores se sentirán mágicamente protegidos por la tecnología que tenga la mezcla correcta de palabras de moda, sin verdaderamente averiguar que es lo que dichas palabras significan. *No hay peor ceguera que la de quien no quiere ver*, asegura la sabiduría popular, y *no hay peor profesional que el que cree en un acrónimo sin comprender su significado*.
- **Falta de actualización.** Al estudiar los peligros relativos a la seguridad, se sigue pensando en las mayores amenazas de seguridad que fueron estudiadas y descubiertas hace décadas.
- **Desbordamiento de pila** (Wheeler, 2003), desbordamiento de enteros Williams (2009). Sin embargo, pocos cursos relativos a la seguridad se han actualizado a los errores que plagan a la clase actual de aplicaciones, mayormente desarrolladas en marcos que inherentemente se protegen de estos graves errores de hace décadas.
- **El síndrome NIH** (*Not Invented Here* - no inventado aquí) (Katz y Allen, 1982), es un antipatrón de diseño definido como la tendencia a reinventar algo ya existente, basándose en la casi siempre falsa creencia de que un desarrollo llevado a cabo *en casa* resultara más adecuado, más seguro o más controlado que implementaciones ya existentes.
- **Procedimientos tediosos.** Muchos errores de diseño de sistemas podrán ser fácilmente paliados si los desarrolladores no tuvieran que llevar a cabo prácticas engorrosas o tediosas para evitarlos. Por ejemplo, casi la totalidad de los lenguajes desarrollados en las últimas décadas liberan a sus desarrolladores del manejo (asignación, liberación) explícito de bloques de memoria. Si bien esto reduce sensiblemente la rapidez del software desarrollado y resta un poco de flexibilidad (imposibilita trabajar sobre conjuntos de datos con simple aritmética de apuntadores), evita por completo los principales problemas que plagaban al desarrollo de software en los años 80 y 90, como se menciona dos párrafos atrás, los diversos tipos de desbordamiento, o errores típicamente no asociados con la seguridad, como las dobles liberaciones (Mitre, 2009) o las fugas de memoria (Erickson, 2002).

Es por esto que resulta tan importante enfatizar en explicar y enseñar respecto a las principales vulnerabilidades actuales.

Una grandísima proporción de los sistemas desarrollados hoy en día siguen el paradigma cliente-servidor. Y si bien hay muy diferentes maneras de implementar estos sistemas, indudablemente la más difundida y prevalente es la de los sistemas Web. Al mismo tiempo, dado el medio mismo a través del cual se transporta y por el ámbito global que han adquirido muchos de estos sistemas, es este modelo de desarrollo la categoría más expuesta directamente a elementos hostiles, por lo cual el presente artículo adopta un enfoque principalmente orientado a este tipo de sistemas.

Si bien hay una tremenda cantidad de categorías en las cuales puede ahondarse, los tres casos (Inyecciones de SQLF, *Cross-Site Scripting* (XSS) y Manejo de sesiones a través de galletas HTTP) presentados a continuación, son una buena muestra de las vulnerabilidades más comunes, peligrosas, y de las cuales es más importante estar consciente y atento al programar o al auditar sistemas existentes.

3. Ejemplo: Inyecciones de SQL

Se toma como ejemplo un URL típico generado por uno de los sistemas de administración de contenido (CMS) en boga hoy en día: *Joomla*. Por razones obvias, el nombre verdadero del sitio en cuestión ha sido reemplazado por *www.ejemplo.com* (http://www.ejemplo.com/content/view/825).

Para quien haya analizado URLs, resultará obvio que *825* corresponda al ID de la nota en la base de datos, y que los componentes *content* y *view* indiquen la operación que el sistema debe realizar ante una solicitud. Ahora bien, ¿a qué se refiere la expresión *cruzamos las barreras entre las capas*? ¿Y cuáles son las principales?

Enfoque en el ID: Al analizar el URL, el ID es un pedazo de texto (formalmente es una cadena que es recibida como parte del método GET, uno de los métodos definidos para el protocolo HTTP). El servidor Web que recibe la solicitud interpreta este método GET y encuentra -utilizando *mod_rewrite*, en caso de tratarse de un servidor Apache, como la mayoría de los sitios de la red, a través de configuración típicamente indicada en el archivo *.htaccess*- que el contenido, indicado por la ruta */content/view/*,* debe ser procesado por el archivo *index.php*, que a su vez (dada su terminación o demás reglas que pueden aplicarse), es

manejado por el lenguaje PHP. El archivo *index.php* es provisto por el sistema *Joomla*, que reconoce la ruta, convierte al ID en su representación numérica, y lo utiliza para pedir a la base de datos le entregue los datos relacionados con determinado artículo. Entonces, aquí puede reconocerse los siguientes puntos principales de manipulación de la solicitud:

Apache recibe una solicitud HTTP, y (vía *mod_rewrite*) la reescribe, indicando *content*, *view* y *825* como parámetros a index.php. PHP analiza, separa y estructura los parámetros recibidos para ser utilizados por *Joomla* solicita el artículo 825 a la base de datos

La variabilidad de los primeros pasos es en realidad menor, pero al solicitar a la base de datos el artículo *825* (y este es el caso base, el más sencillo de todos) deben pasar muchas cosas. Primero, que *825* es una cadena de caracteres. PHP es un lenguaje débilmente tipificado (los números se convierten en cadenas y viceversa automáticamente según sea requerido), pero una base de datos maneja tipos estrictamente.

Un atacante, una persona que quiere causar daño u obtener acceso mayor al cual tiene autorizado en el sistema, basará su acercamiento en ser creativo respecto a cómo engañarlo: Es muy poco frecuente que busque adivinar usuarios/contraseñas; más bien, intentará engañar al sistema para entregar resultados distintos de aquello que parece estar siendo solicitado. ¿Y cómo se engaña a un sistema? Pidiéndole algo que no se espere - Por ejemplo, *825aaa*. En este caso, el código PHP que invoca a la base de datos se verifica que el tipo de datos sea correcto: Hace una conversión a entero, y descarta lo que sobra. Sin embargo, en muchos sistemas desarrollados *en casa*, una solicitud similar lleva a un mensaje como:

```
Warning: pg_execute () [function.pg-execute]: Query failed: ERROR: Invalid input syntax for integer: "825aaa" in /home/(...)/index.php on line 192
```

Esto indica que uno de los parámetros fue pasado sin verificación de PHP al motor de base de datos, y fue este el que reconocía al error.

Ahora, esto no califica aún como inyección de SQL (dado que el motor de bases de datos supo reaccionar ante esta situación), pero se está prácticamente a las puertas. El código desarrollado por una determinada persona, en un lapso de tiempo dado tiende a repetir muchos patrones Es casi un hecho que si el desarrollador no valida la entrada en un punto, habrá muchos otros en que no lo haya hecho. Este error en particular

indica que el código construye la cadena de consulta SQL a través de una interpolación parecida a la siguiente:

```
$id_art = $_GET ['id'];
$sql = "SELECT * FROM articulo WHERE id = $id_art"
```

La vulnerabilidad aquí consiste en que el programador no toma en cuenta que *$id_art* puede contener cualquier cosa enviada por el usuario - Por el atacante en potencia. ¿Cómo puede aprovecharse esto?... No hay límites más que la imaginación.

A continuación, se presentan algunos ejemplos, evitando el enfoque en ningún lenguaje en específico, ya que lo importante es el proceso y el tratamiento que se da al SQL generado.

Para estos ejemplos, se cambia un poco el caso de uso, nuevamente, recordando que errores derivados del estilo de codificación detectados en un punto, muy probablemente se repitan a lo largo del programa. En vez de ubicar recursos, se habla acerca de una de las operaciones más comunes: La identificación de un usuario con *login* y contraseña. Se supone que el mismo sistema del código recién mencionado utilizala siguiente función para validar a sus usuarios:

```
$data = $db->fetch ("SELECT id FROM usuarios WHERE login
= '$login' AND passwd = '$passwd'");
If ($data) {
$uid = $data [0];
} else {
print "<h1>Usuario invalido!</h1>";}
```

Aquí puede apreciarse la práctica muy cómoda y común de *interpolar* variables dentro de una cadena. Muchos lenguajes permiten construir cadenas donde se expande el contenido de determinadas variables. En caso de que el lenguaje favorito no maneje esta característica, concatenar las sub-cadenas y las variables lleva al mismo efecto. Por ejemplo, en Visual BASIC se obtendría la misma vulnerabilidad construyendo la cadena así:

```
Dim sql as String = "SELECT id FROM usuarios WHERE login
= '" & login & "' AND passwd = '" & passwd & "
```

Sin embargo... ¿Que pasaría aquí si el usuario jugara un pequeño truco? Si solicitara, por ejemplo, entrar al sistema utilizando como *login* a *fulano';--,* esto llevaría al sistema a ignorar lo que se diera por contraseña: se estaría ejecutando la siguiente solicitud:

```
SELECT id FROM usuarios WHERE login = 'fulano';--' AND
PASSWD = ''
```

La clave de este ataque es confundir a la base de datos para aceptar comandos generados por el usuario. El ataque completo se limita a cuatro caracteres (';--). Al cerrar la comilla e indicar (con el punto y coma) que termina el comando, la base de datos entiende que la solicitud se da por terminada, y cualquier cosa que siga es otro comando. Se podría enviarle más de un comando consecutivo que concluyera de forma coherente, pero lo más sencillo es utilizar el doble guión indicando que inicia un comentario. De este modo, se logra vulnerar la seguridad del sistema, entrando como un usuario cuyo *login* se conoce, aún desconociendo su contraseña.

Pero puede irse más allá. Siguiendo con este ejemplo, típicamente el ID del administrador de un sistema es el más bajo. Si se imagina el resultado de los siguientes nombres de usuario falsos: ninguno' OR id = 1;-- *ninguno* no es un usuario válido del sistema, pero esto resultará en una sesión con privilegios de administrador.

```
INSERT INTO usuarios (login, passwd) VALUES ('fulano', 'de
tal'); --
```

Esto no otorgará en un primer momento una sesión válida, pero creará una nueva cuenta con los valores especificados. Obviamente, es posible que fuera necesario averiguar -a prueba y error- qué otros valores son necesarios agregar para que esto otorgue un usuario válido con suficientes privilegios.

DROP TABLE usuarios: Un atacante puede darse por satisfecho con destruir la información. En este caso, el atacante destruirá de un plumazo la tabla de usuarios. Es imposible dejar de sugerir la visita al ya famoso *Bobby Tables* (Munroe, 2007).

3.1 Evitando las inyecciones de SQL

¿Y qué puede hacerse? Protegerse de inyección de SQL es sencillo, pero hay que hacerlo en prácticamente todas la consultas, y convertir la manera natural de escribir código en una segura.

La regla de oro es nunca cruzar fronteras incorporando datos no confiables, y esto no sólo es muy sencillo, sino que muchas veces (específicamente cuando se itera sobre un conjunto de valores, efectuando la misma consulta para cada uno de ellos) hará los tiempos de respuesta del sistema sensiblemente mejores. La respuesta es separar *preparación* y *ejecución* de las consultas. Al preparar una consulta, el

motor de bases de datos la compila y prepara las estructuras necesarias para recibir los parámetros a través de *placeholders*, marcadores que serán substituidos por los valores que se indiquen en una solicitud posterior.

Volviendo al ejemplo del *login/contraseña*, en este caso, se presenta como será construido desde el lenguaje Perl:

```
$query = $db->prepare ('SELECT id FROM usuarios WHERE login =? AND passwd =?'); $data = $query->execute ($login, $passwd);
```

Los símbolos de interrogación son enviados como literales a la base de datos, que sabe ya qué se le pedirá y prepara los índices para responder. Puede enviarse contenido arbitrario como *login* y *password*, ya sin preocupación de si el motor lo intentará interpretar.

Algunos lenguajes y bibliotecas de acceso a bases de datos (notablemente, la popular combinación PHP+MySQL) no implementan la funcionalidad necesaria para separar los pasos de preparación y ejecución. La respuesta en dichos lenguajes es utilizar funciones que *escapen* explícitamente los caracteres que puedan ser nocivos. En el caso mencionado de PHP+MySQL, esta función será mysql_real_escape_string (Achour, 1997-2009):

```
$login = mysql_real_escape_string ($_GET['login']);
$passwd = mysql_real_escape_string ($_GET['passwd']);
$query = "SELECT id FROM usuarios WHERE login = '$login' AND passwd = ''";
$data = $db->fetch ($query);
```

Una gran desventaja que esto conlleva es la cantidad de pasos que el programador debe efectuar manualmente, especialmente cuando entran en juego mayores cantidades de datos, y hay que verificar cada uno de ellos. Cabe mencionar que las bibliotecas que implementan conectividad a otras bases de datos para PHP (por ejemplo, PostgreSQL), ofrecen las facilidades necesarias para la preparación y ejecución como dos pasos separados, con una semántica muy similar a la anteriormente descrita.

Revisar todas las cadenas enviadas a la base de datos puede parecer una tarea tediosa, pero ante la facilidad de encontrar y explotar este tipo de vulnerabilidades, bien vale la pena. En Mavituna (2007), Microsoft (2009) y Friedl (2007) existe información mucho más completa y detallada acerca de la anatomía de las inyecciones SQL, y

diversas maneras de explotarlas, incluso cuando existe cierto grado de validación.

3.2 Alternativa específica: el uso de ORMS

Otra alternativa interesante que todo programador debe conocer es el uso de marcos de abstracción, como los mapeos objeto-relacionales, ORMs (Ulloa, Salazar y Solar, 2006). Estos marcos se encargan de crear todo el *pegamento* necesario para hermanar a dos mundos diferentes (el de las bases de datos relacionales, basados en registros almacenados en tablas, y el de la programación orientada a objetos, basado en objetos instanciados de clases), mundos que comparten algunas características (estructura homogénea a todos los elementos del mismo tipo), pero no otras (manejo de relaciones complejas como composición o agregación). Sin embargo, en el ámbito aquí discutido, la principal característica de los ORMs es que se encargan de cubrir los detalles relativos a la integración de dos lenguajes y dos formas de ver al mundo muy distintas. Uno de los grandes atractivos de los ORMs es que, entre otras muchas ventajas, su uso puede liberar por completo (o en una gran medida) de escribir directamente código SQL en la totalidad del proyecto.

Una solicitud similar a la anterior a través del ORM Active Record en el lenguaje Ruby sería sencillamente:

```
usuario = Usuario.find_by_login (login, :conditions => 'pas-
swd = ?', passwd)
```

En este caso, será directamente la biblioteca la que convierte una llamada *find* a la clase Usuario (que hereda de *Active Record*: Base) en una consulta SQL, escapa/limpia las entradas y envía la solicitud a la base de datos. Esto proporciona la ventaja adicional de que todo el código relativo al paso de parámetros a la base de datos está concentrado en un solo punto (y en un punto ampliamente utilizado y escrutado por profesionales de todo el mundo); toda omisión o error que vaya siendo detectado, llevará a que este sea corregido en un solo punto, y la aplicación recibirá, en todas las consultas que pasen a través del ORM, el beneficio de esta mejora. Además, estando desarrollada verdaderamente en un solo lenguaje, la aplicación se mantiene más limpia y resulta más fácil de comprender y depurar.

Active Record es solo uno de muchos ORMs existentes; está implementado para *Ruby* (Heinemeier, 2004-2009) y para .NET (Verissimo, 2003-2009), además de implementaciones muy cercanas a este patrón en varios otros lenguajes.

4. Ejemplo 2: Cross-Site Scripting (XSS)

Como ya se mencionó, las inyecciones se presentan cuando se *cruzan fronteras*. Las inyecciones SQL no son, ni mucho menos, el único tipo de inyección de código malicioso; cualquier lugar en que un mismo dato puede significar cosas diferentes, dependiendo del contexto en que es evaluado, es susceptible a ser vulnerable a inyección. Las siguientes categorías las menciono únicamente para invitar al lector a empaparse en la problemática que conllevan.

Del mismo modo que deben *sanitizarse*[3] los datos, que se llevan hacia abajo a través de las capas del diseño del sistema, debe hacerse lo mismo al subir. ¿Qué significa esto? Que no sólo deben protegerse las capas privadas de la aplicación, a las cuales un posible atacante no debería tener acceso directo, sino también proteger a las capas superiores, aquellas que están incluso fuera de control, como el navegador de los demás usuarios (Mitre, 2008-2009b).

En un sistema Web (especialmente si se quiere participar en la llamada Web 2.0) muchas veces se despliega a un usuario información provista por otros usuarios. Para ofrecer toda la funcionalidad y respuesta ágil de un sitio moderno, los navegadores están típicamente configurados para ejecutar todo código *Java Script* que reciben, confiando en quien lo origina. Ahora, ¿qué pasará si un usuario malicioso deja en un blog el siguiente comentario?:

```
<script language=”Java Script”>
window.location=”http://www.hackme.org/1234”; </script>
```

En efecto, inmediatamente al cargar la página en cuestión, el usuario será redirigido a un sitio diferente (y con justa razón: para su navegador, esta indicación viene del sitio, no puede saber que viene de un elemento hostil).

Este sitio podrá hacerse pasar por el sitio víctima, sin que el usuario se diera cuenta, y podrá llevar al usuario a diferentes escenarios donde se le extrajera información confidencial (por ejemplo, se le puede pedir que se vuelva a autenticar). La mayor parte de los usuarios caerán en el engaño, con lo que puede volverse un potente mecanismo para suplantación de identidad.

Por si no bastara, cada usuario, obviamente, tiene un perfil diferente y normalmente, nivel o credenciales de acceso también diferentes. ¿Que

3 Entendido como limpiar o escapar.

pasaría si el código *Java Script* fuera muy ligeramente más malicioso? Por ejemplo (y nuevamente, meramente como ejemplo ficticio), podrá hacer que un administrador le diera acceso completo al sitio. Si el atacante recibía el usuario número 152, podrá enviar un mensaje privado al administrador del sitio que incluyera:

```
<script language="Java Script">
  window.location="/admin/user/152/edit?set_admin=1";
</Script>
```

Claro está, asignar una nueva dirección a *window.location* es probablemente la manera más burda y notoria para llevar a cabo estos ataques; hay muchas formas más sigilosas, que pueden pasar completamente desapercibidas por un usuario casual.

Este tipo de ataques son conocidos genéricamente bajo el nombre de XSS o *Cross-Site Scripting* (Mitre, 2008-2009c). La clave para evitarlos es nuevamente *sanitizar* toda la información, pero en este caso, toda la información a enviar al cliente. En el caso de HTML, prácticamente basta con escapar ciertas *entidades* (caracteres que pueden tener significados especiales). Por ejemplo, reemplazando todos los caracteres « » por su representación « » y todos los caracteres «>» por «g» (como primer acercamiento), este código será desplegado de una manera limpia. Este es típicamente un proceso aún más engorroso que *sanitizar* la entrada, por la cantidad de puntos donde hay que repetir la validación. Dependiendo del caso, muchos desarrolladores optan por limpiar a la entrada todo lo que será eventualmente desplegado, pero esto puede llevar al usuario a algunas condiciones en que el navegador no lo *desanitiza*, resultando sencillamente en una salida aparentemente llena de basura.

Como nota adicional, es posible (y altamente deseable) *sanitizar* toda la información a la entrada, desechando lo que no sea claramente aceptable. Sin embargo, siempre hay casos en que se requiere guardar la información completa y sin manipular.

Si bien evitar XSS es más difícil que evitar inyecciones de SQL, a pesar de que el impacto de XSS, a primera vista, es menos severo que el de una inyección de SQL. Este puede tardar mucho más tiempo en ser corregido, puede estar presente en más puntos del código, y por sobre todo, es más tedioso de arreglar, por lo que es fundamental tener la costumbre de verificar, a tiempo, toda la información que se despliegue.

Además, si bien el impacto es menos inmediato, es típicamente más sigiloso. Si un atacante obtiene la información de acceso de una gran

cantidad de usuarios, para propósitos prácticos no puede volverse a confiar en ninguno de ellos, ya que todos pueden estar potencialmente controlados por el atacante.

5. Ejemplo 3: Manejo de sesiones a través de galletas HTTP

La conjunción de un protocolo verdaderamente simple para la distribución de contenido (HTTP) con un esquema de marcado suficientemente simple, pero suficientemente rico para presentar una interfaz de usuario con la mayor parte de las funciones requeridas por los usuarios (HTML), llevó a la creación del entorno ideal para el despliegue de aplicaciones distribuidas.

Desde sus principios, el estándar de HTTP menciona cuatro *verbos* por medio de los cuales se puede acceder a la información: - GET (solicitud de información sin requerir cambio de estado), - POST (interacción por medio de la cual el cliente manda información compleja y que determinará la naturaleza de la respuesta, así como posibles cambios de estado del lado del servidor), - PUT (creación de un nuevo objeto en el servidor) y - DELETE (destrucción de un determinado objeto en el servidor). Sin embargo, por muchos años, los verbos fueron mayormente ignorados. La mayor parte de los sistemas hace caso omiso a través de qué verbo llega una solicitud determinada; muchos navegadores no implementan siquiera PUT y DELETE, dado su bajísimo nivel de uso, aunque con la popularización del paradigma REST (Costello, s.f.), principalmente orientado a servicios Web (interfaces expuestas va HTTP, pero orientadas a ser consumidas/empleadas por otros programas, no por humanos), esto probablemente está por cambiar.

El protocolo HTTP, sin embargo, junto con su gran simplicidad aporta un gran peligro, no una vulnerabilidad inherente a los sistemas Web, sino que un peligro derivado de que muchos programadores no presten atención a un aspecto fundamental de los sistemas Web: Cómo manejar la interacción repetida sobre de un protocolo que delega el mantener el estado o *sesión* a una capa superior. Esto por que, para un servidor HTTP, toda solicitud es única. En especial, un criterio de diseño debe ser que toda solicitud GET sea *idempotente*[4].

4 Esto significa que un GET no debe alterar de manera significativa el estado de los datos, es aceptable que a través de un GET, por ejemplo, se aumente el contador de visitas, (que haya un cambio no substantivo), pero muchos desarrolladores han sufrido por enlazar a través de

¿Cuál es el peligro? Consiste en que diversas aplicaciones, desde los robots indexadores de buscadores como Google y hasta aceleradores de descargas ingenuos que buscan hacer más ágil la navegación de un usuario (siguiendo de modo preventivo todas las ligas GET del sistema para que el usuario no tenga que esperar en el momento de seleccionar alguna de las acciones en la página) van a disparar estos eventos de manera inesperada e indiscriminada.

HTTP fue concebido, segúnBerners-Lee (1996), como un protocolo a través del cual se solicitará información estática. Al implementar las primeras aplicaciones sobre HTTP, se encuentra que cada solicitud debía incluir la totalidad del estado. En términos de redes, TCP implementa exclusivamente la capa 4 del modelo OSI, y si bien mantiene varios rasgos que permiten hablar también de *sesiones* a nivel conexión, estas son sencillamente descartadas. Las capas 5 y superiores deben ser implementadas a nivel aplicación. HTTP es un protocolo puramente capa 6 (omite toda información relacionada con la sesión y sirve únicamente para presentar ya sea la aplicación o el contenido estático). Es por esto que los muchos sistemas Web hacen un uso extensivo de los campos ocultos (*hidden*) en todos sus formularios y ligas internas, transportando los valores de interacciones previas que forman parte conceptualmente de una sola interacción distribuida a lo largo de varios formularios, o *números mágicos*, que permiten al servidor recordar desde quien es el usuario en cuestión, hasta todo tipo de preferencias que ha manifestado a lo largo de su interacción.

Sin embargo, este mecanismo resulta no solo muy engorroso, sino muy frágil: Un usuario malicioso o curioso puede verse tentado a modificar estos valores; es fácil capturar y alterar los campos de una solicitud HTTP a través de herramientas muy útiles para la depuración. E incluso sin estas herramientas, el protocolo HTTP es muy simple, y puede *codificarse* a mano, sin más armas que un *telnet* abierto al puerto donde escucha nuestro sistema. Cada uno de los campos y sus valores se indican en texto plano, y modificar el campo *user_id* es tan fácil como decirlo.

En 1994, *Netscape* introdujo un mecanismo denominado *galletas* (*cookies*), que permite al sistema almacenar valores arbitrarios en el cliente. Este mecanismo indica que todas las *galletas* que defina un servidor a determinado cliente serán enviadas en los encabezados de cada so-

un GET (como todo navegador responde a una liga HTML estándar), por ejemplo, el botón para eliminar cierto objeto.

licitud que este le realice, por lo que se recomienda mantenerla corta. Atendiendo a esta recomendación, varias implementaciones de *galletas* no soportan más de 4KB. Un año más tarde, Microsoft lo incluye en su *Internet Explorer*; el mecanismo fue estandarizado en 1997 y extendido en el 2000 (Kristol y Montulli (1997) y Kristol y Montulli (2000)). El uso de las *galletas* libera al desarrollador del engorro antes mencionado, y le permite implementar fácilmente un esquema verdadero de manejo de sesiones, pero, ante programadores poco cuidadosos, abre muchas nuevas maneras de adivinaron cometer errores.

Dentro del cliente (típicamente un navegador) las *galletas* están guardadas bajo una estructura de doble diccionario: en primer término, toda *galleta* pertenece a un determinado servidor (esto es, al servidor que la envía). La mayor parte de los usuarios tienen configurados a sus navegadores, por privacidad y por seguridad, para entregar el valor de una *galleta* únicamente a su dominio origen (de modo que al entrar a un determinado sitio hostil este no pueda robar su sesión en el banco); sin embargo, todo sistema puede solicitar *galletas* arbitrarias guardadas en el cliente. Para cada servidor, pueden almacenarse varias *galletas*, cada una con una diferente llave, un nombre que la identifica dentro del espacio del servidor. Además de estos datos, cada *galleta* guarda la ruta a la que esta pertenece, si requiere seguridad en la conexión (permitiendo solo su envío a través de conexiones cifradas), y su periodo de validez, pasado el cual serán consideradas *rancias* y ya no se enviarán. El periodo de validez se mide según el reloj del cliente.

Guardar la información de estado del lado del cliente es riesgoso, especialmente si se hace sobre un protocolo tan simple como HTTP. No es difícil para un atacante modificar la información enviada al servidor, y si bien en un principio los desarrolladores guardaban en las *galletas* la información de formas parciales, se llega a una regla de oro: *Nunca guardar información real en ellas*. En vez de esto, es recomendado guardar un *token* (literalmente, ficha o símbolo) que *apunte* a la información. Esto es, en vez de guardar el ID de un usuario, debe enviarse una cadena *criptográficamente fuerte* (Martínez y Vázquez, 2008) que apunte a un registro en la base de datos del servidor. ¿A qué se refiere esto?... a que tampoco es recomendable grabar directamente el ID de la sesión (dado que siendo sencillamente un número, será para un atacante trivial probar con diferentes valores hasta *aterrizar* en una sesión interesante), sino una cadena aparentemente aleatoria, creada con un algoritmo que garantice una muy baja posibilidad de colisión y

un espacio de búsqueda demasiado grande como para que un atacante lo encuentre a través de la fuerza bruta.

Los algoritmos más comunes para este tipo de uso son los llamados funciones de resumen o *digest* (Bellare and Kohno, 2003). Estos generan una cadena de longitud fija; dependiendo del algoritmo, hoy en día van de los 128 a los 512 bits. Las funciones de resumen más comunes son las variaciones del algoritmo SHA desarrollado por el NIST y publicado en 1994; usar las bibliotecas que los implementan es verdaderamente trivial. Por ejemplo, usando Perl:

```
use Digest::SHA1;
print Digest::SHA1->sha1_hex("Esta es mi llave");
entrega la cadena:
c3b6603b8f841444bca1740b4ffc585aef7bc5fa
```

Pero, ¿qué valor usar para enviar como llave? Definitivamente no servirá enviar, por ejemplo, el ID de la sesión, ya que dejará una situación igual de riesgosa que incluir el ID del usuario. Un atacante puede fácilmente crear un diccionario del resultado de aplicar SHA1 a la conversión de los diferentes números en cadenas (mecanismo conocido como *rainbow tables* o tablas arcoíris; hay varios proyectos que han construido tablas para diversas aplicaciones, como la recuperación de contraseñas en sistemas Windows). La representación hexadecimal del SHA1 de '1' siempre será d688d9b3d3ba401b25095389262a3ecd2ad5ad68, y del de 100 siempre será daaa8121aa28fca0edb4b3e1f7b7c23d6152eed; el identificador de la sesión debe contener elementos que varíen según algún dato no adivinable por el atacante (como la hora exacta del día, con precisión a centésimas de segundo) o, mejor aún, con datos aleatorios.

Este mecanismo lleva a asociar una cadena suficientemente aleatoria, como para asumir que las sesiones de los usuarios no serán fácilmente *secuestradas*[5], y así *poder dormir tranquilos,* sabiendo que el sistema de manejo de sesiones en el sistema es prácticamente inmune al ataque por fuerza bruta.

Y ya como último punto, las *galletas* son muchas veces vistas como un peligro por los activistas de la privacidad y el anonimato, dado que permiten crear un perfil de las páginas que va visitando un usuario (especialmente en el caso de empresas como *Google* o *DoubleClick*, que han sido especialmente exitosas en ofrecer herramientas de anuncios

5 Esto es, que un atacante no le atinará al ID de la sesión de otro usuario.

o de monitoreo/estadísticas a diversos administradores de sitios en todo el mundo).

Es importante recordar que algunas personas han elegido desactivar el uso de *galletas* en su navegación diaria, a excepción de los sitios que expresamente autoricen. Es interesante tener en cuenta que una *galleta* puede no haber sido guardada en el navegador cliente, y esto desembocará en una experiencia de navegación interrumpida y errática para dichos usuarios. Es importante detectar si en el momento de establecer una galleta, esta no fue aceptada, para dar la información pertinente al usuario, para saber qué hacer y no estar frente a un sistema inoperativo más.

6. Conclusiones

- A lo largo de este artículo se revisan tres de las principales categorías de vulnerabilidades/errores de programación que, a juicio del autor, más prevalentes y peligrosas resultan hoy en día. Ahora bien, no es casualidad que el encabezado de cada una de las secciones correspondientes fuera Ejemplo. Este artículo no implico que baste con estar conciente y alerta ante estas amenazas para dejar de lado las demás.
- La seguridad en cómputo, si bien es un campo fascinante, es un campo que requiere constante actualización. Además, si bien este artículo se enfoca a las vulnerabilidades más comunes en servicios Web no es, ni busca ser, comprehensivos. Hay amplísimas categorías a las cuales no hubo acercamiento, siquiera, y eso no debe entenderse como que carezcan de importancia Las tres categorías analizadas son meramente ejemplos de lo que hay que considerar.
- Ser desarrollador de sistemas es una profesión demandante. Para cumplirla a cabalidad, requiere estar al día en lo tocante a seguridad. No estarlo significa una irresponsabilidad ante usuarios o clientes del desarrollo.
- La sociedad depende más que nunca de la sistematización de los procesos y de la capacidad de sustentarlos a distancia. Los programadores, y en especial los programadores de sistemas en red, se han vuelto piezas fundamentales del tejido social. Es fundamental reconocer y aceptar el nuevo rol responsablemente, que quienes se sientan desarrolladores de sistemas, comprendan la importancia de este trabajo para la sociedad toda, y actúen en consecuencia.

Bibliografía

ACHOUR, Mehdi; et al. (1997-2009). PHP Manual [en línea]. s.l.: PHP Documentation Group. <http://www.php.net/manual/en/> <http://www.php.net/mysql_real_escape_string> [Consulta: 08/2009]

BELLARE, M. and KOHNO, T. (2003). A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications [en línea] Eurocrypt 2003 (4-8/05/2003), Warsaw (Poland): International Association for Cryptologic Research, IACR. *Proc. EUROCRYPT 2003*, Lecture Notes in Computer Science No. 2656, p. 491-506, Springer-Verlag. <www.iacr.org/archive/eurocrypt2003/26560491/26560491.ps> [Consulta: 08/2009]

BERNERS-LEE, Tim. (1996). The Original HTTP as defined in 1991 [en línea]. s.l.: W3C. <http://www.w3.org/Protocols/HTTP/AsImplemented.html> [Consulta: 08/2009]

COSTELLO, Roger (s.f.) Building Web Services the REST Way [en línea]. s.l.: xFront. <http://www.xfront.com/REST-Web-Services.html> [Consulta: 08/2009]

ERICKSON, Cal (2002). Memory Leak Detection in Embedded Systems. En: Linux Journal, No. 101 (sep 2002) Beltdown Media, ISSN 1075-3583 <http://www.linuxjournal.com/article/6059> [Consulta: 08/2009]

FRIEDL, Steve (2007) SQL Injection Attacks by Example [en línea]. Yorba Linda (USA): Unixwiz. <http://www.unixwiz.net/techtips/sql-injection.html> [Consulta: 08/2009]

HALFOND, W. G.; ORSO, A. and MANOLIOS, P. (2006). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. [en línea]. In: Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Portland, Oregon, USA, November 05 - 11, 2006). SIGSOFT '06/FSE-14. ACM, New York, NY, p. 175-185. <http://portal.acm.org/citation.cfm?id=1181797> [consulta: 08/2009]

HEINEMEIER, David (2004-2009) ActiveRecord para Ruby [en línea]. <http://ar.rubyonrails.org/> [Consulta: 08-2009]

KATZ, Ralph and ALLEN, Thomas J. (1982). Investigating the Not Invented Here (NIH) syndrome: A look at the performance, tenure, and communication patterns of 50 R & D Project Groups [en línea]. En: R&D Management Volume 12, Issue 1, p. 7 – 20 (Published Online: 05.05. 2007). Hoboken (USA): Basil Blackwell. <http://www3.interscience.wiley.com/journal/120045051/abstract?CRETRY=1&SRETRY=0> [consulta: 08/2009]

KRISTOL, David M. and MONTULLI, Lou (2000). RFC 2965: HTTP State Management Mechanism [en línea]. <http://www.ietf.org/rfc/rfc2965.txt> [Consulta: 08/2009]

KRISTOL, David M. and MONTULLI, Lou. (1997). RFC 2109: HTTP State Management Mechanism [en línea]. <http://www.ietf.org/rfc/rfc2965.txt> [Consulta: 08/2009]

MARTÍNEZ, I. y VÁZQUEZ, M. R. (2008) Esquemas de Autenticación de Mensajes Utilizando Funciones Hash. En: V Congreso Internacional de Telemática y Telecomunicaciones CITTEL´08 (1-5/12/2008), La Habana (Cuba): Departamento de Telecomunicaciones y Telemática del Instituto Superior Politécnico José Antonio Echeverría, CUJAE. Cittel 2008: Congreso Internacional de Telemática y Telecomunicaciones. <http://www.cujae.edu.cu/eventos/convencion/cittel/Trabajos/CIT060.pdf> [consulta: 08/2009]

MAVITUNA, Ferruh (2007). SQL Injection Cheat Sheet. [en línea]. <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/> [Consulta: 08/2009]

MICROSOFT (2009). SQL Server 2008 Product Documentation - SQL Server 2008 Books Online - Threat and Vulnerability Mitigation – SQL Injection [en línea]. <http://msdn.microsoft.com/en-us/library/ms161953.aspx> [Consulta: 08/2009]

MITRE (2008-2009a). Common Weakness Enumeration (CWE) 415: Double Free [en línea]. <http://cwe.mitre.org/data/definitions/415.html> [Consulta: 08/2009]

MITRE (2008-2009b). Common Weakness Enumeration (CWE) 116: Improper Encoding or Escaping of Output [en línea]. <http://cwe.mitre.org/data/definitions/116.html> [Consulta: 08/2009]

MITRE (2008-2009c). Common Weakness Enumeration (CWE) 79: Failure to Preserve Web Page Structure ('Cross-site Scripting') [en línea]. <http://cwe.mitre.org/data/definitions/79.html> [Consulta: 08/2009]

MUNROE, Randall (2007). Webcomic xcdr [en línea]. <http://es.xkcd.com/xkcd-es/strips/exploits-de-una-madre/> [Consulta: 08/2009]

SANS/MITRE (2009). Top 25 Most Dangerous Programming Errors [en línea]. <http://www.sans.org/top25errors/> [Consulta: 08/2009]

SHMOO GROUP (2005-2006). Rainbow Tables [en línea]. <http://rainbowtables.shmoo.com/> [Consulta: 08/2009]

ULLOA, C.; SALAZAR J. P. y SOLAR, M. (2006). Metodología para evaluar el rendimiento de sistemas de almacenamiento y recuperación de documentos xml en bases datos nativas para xml y no nativas habilitadas para XML. [en línea]. En: Síntesis Tecnológica, Vol. 3 No. 1 (may, 2006) p. 1-13. Valdivia (Chile): Facultad de Ciencias de la Ingeniería, Universidad Austral de Chile. ISSN 0718-025X <http://mingaonline.uach.cl/scielo.php?script=sci_arttext&pid=S0718-025X2006000100001&lng=es&nrm=iso>. [consulta: 08/2009]

VERISSIMO, Hamilton, et. al. (2003-2009). Castle ActiveRecord [en línea]. <http://www.castleproject.org/activerecord/index.html> [Consulta: 08-2009]

WHEELER, David A. (1999-2003) Secure Programming for Linux and Unix HOWTO; capítulo 6: Avoid Buffer Overflow [en línea]. <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/buffer-overflow.html> [Consulta: 08/2009]

WILLIAMS, Jeff, et. al. (2006-2009) Integer overflow [en línea]. OWASP: the free and open application security community <http://www.owasp.org/index.php/Integer_overflow> [Consulta: 08/2009]