Defensa

Detectando backdoors silenciosos

Carlos García Prado

Grado de dificultad

Normalmente, un backdoor (puerta trasera) se define como un mecanismo utilizado por un individuo para asegurarse el acceso posterior a una intrusión. Pero no es éste el único escenario, ya que estos pueden haber sido instalados por antiguos empleados descontentos, por ejemplo.

Es decir, cualquier persona que haya tenido acceso administrativo en cualquier momento al equipo, puede haber colocado uno, así que la preocupación por la existencia de uno de estos mecanismos en nuestros sistemas está justificada.

Antiguamente *los backdoors* eran bastante primitivos ya que se limitaban a técnicas como escuchar en un puerto alto, etc. Hoy día, cualquier administrador consciente mantiene regularmente un listado de puertos abiertos que compara con un patrón normal de operación de la red, por lo que estos métodos rudimentarios no harán más que llamar la atención del administrador de la red.

La acción mas común que realiza *un backdoor* es la obtención de *un shell* de una cuenta de administración, pero puede hacer en principio cualquier otra cosa; enviar por correo un copia de seguridad de una base de datos y luego purgarla, encriptar archivos importantes con una clave que enviemos, de modo que no sean accesibles por su legitimo dueño, etc.

Sin embargo, existe una característica que comparten todos *los backdoors* (por lo menos los mas útiles) y es que debe producirse una acción (*trigger*) por parte del individuo, que a partir de ahora llamaremos *Eva*, que desencadene la acción de éste.

Como en nuestro caso de estudio queremos interaccionar remotamente con la maquina comprometida, el método que dispare *el backdoor* debe ser el envío de “algo” por la red. En nuestro ejemplo es el envió de un paquete *TCP/IP* especialmente preparado con la ayuda de la valiosísima herramienta *Hping2*.

En este artículo aprenderás...

- Cómo inspeccionar procesos corriendo en tu sistema,
- A obtener información sobre las librerías usadas por estos,
- A determinar si existen backdoors entre ellos.

Lo que deberías saber...

- Cómo funcionan los backdoors silenciosos,
- Estar familiarizado con la línea de comandos de Linux.
- Conocimientos básicos de compilación bajo Linux

Ahora bien, ¿Que ocurre con el firewall?. ¿No se supone que nos protege? Un firewall denegará conexiones entrantes no permitidas, pero permitirá la entrada de todo paquete con el flag *SYN* activado y destinado a un puerto permitido. Por ejemplo, todo paquete *SYN* destinado al puerto *80* será aceptado, ya que en principio su intención es iniciar una conexión *HTTP* con el servidor Web.

El firewall no inspecciona *el payload* de los paquetes a no ser que se configure para trabajar mas allá de las capas de red y transporte, que es su zona normal de operación. Aunque así fuese, como en cualquier búsqueda, necesitaríamos saber lo que estamos buscando y cual sería *el payload* que actuaría como disparador.

Los métodos para detectar si un proceso es, o contiene *un backdoor silencioso*, dependen de varios factores. Algunos de estos factores son si ha sido compilado estática o dinámicamente, si podemos reiniciar el proceso, etc. Vamos a ir viendo los distintos métodos, desde el caso más favorable al menos.

Podemos parar el proceso

Así pues, antes de nada debemos elegir el proceso que parece sospechoso. Recordemos que estos pueden engañar al comando `ps` y adoptar identidades falsas.

Un buen método para encontrar el ejecutable que da lugar a esa entrada falsa en `ps` es usar el comando `lsof`, que lista ficheros abiertos por procesos. Dependiendo del proceso, esta lista puede ser muy larga, así que utilizaremos el comando *grep*.

Una de las columnas de la lista indica el tipo de descriptor de fichero. A nosotros nos interesa el descriptor del ejecutable abierto que se representa con la abreviatura *txt*. Así, la salida del comando `lsof -p PID | grep txt` nos proporciona la ruta completa al ejecutable, aunque un atacante astuto buscaría un proceso no instalado y reproduciría tanto el nombre como la ruta de éste para no despertar sospechas.

Supongamos que el proceso no se está haciendo pasar por uno vital y podemos pararlo. Esta es sin duda la mejor opción para detectar *un backdoor*.

Probamos a *matarlo* mediante el comando `kill -9 pid`, esperamos unos segundos y volvemos a ejecutar `ps` para ver si definitivamente ha *muerto*. ¿A que viene esto? Esto viene a que una técnica común utilizada cuando se implantan este tipo de procesos es la de hacer que se ejecuten como demonios, de manera que puedan ser utilizados infinitas veces utilizando el mecanismo que proporciona *respawn*. Es decir, aunque logremos *matar* el proceso, el sistema operativo vuelve a lanzar de inmediato otra instancia con un *PID* diferente.

En sistemas Debian, esto se consigue añadiendo la ruta completa del ejecutable al fichero */etc/inittab*. La sintaxis de las entradas en este fichero es:

```
id:runlevels:respawn:/ruta/del/archivo
```

Listado 1. *Esquema del código de un backdoor silencioso*

```
[...]
 setuid(0);
 setgid(0);

 if (pcap_lookupnet(NULL, &net, &mask, errbuf) == -1) exit(0);
 if (!(session = pcap_open_live(NULL, 1024, 0, 0, errbuf)))
 {
  exit(0);
 }

 pcap_compile(session, &filter, filter_string, 0, net);
 pcap_setfilter(session, &filter);
 pcap_loop(session, 0, pkt_handler, NULL);
[...]
 void pkt_handler(u_char *ptrnull, const struct pcap_pkthdr *pkt_info, const
                        u_char *packet)
 {
  // This code processes the packets.
 }
```

Listado 2. *Señales claros del inicio de una captura pcap*

```
root@/home/carlos# ltrace ./sb_dynamic
__libc_start_main(1, 0x7fadd874, 0x7fadd87c, 0x7fadd908, 0x3000c5a0
                        <unfinished ...>
setuid(0)                                          = 0
setgid(0)                                          = 0
pcap_lookupnet(NULL, 0x7fadd600, 0x7fadd604, 0x7fadd4ec) = 0
pcap_open_live(NULL, 1024, 0, 0, 0x7fadd4ec)       = 0x10013008
pcap_compile(0x10013008, 0x7fadd5f8, "dst port 80", 0, NULL) = 0
pcap_setfilter(0x10013008, 0x7fadd5f8, 1, 0, 0x10013e70) = 0
pcap_loop(0x10013008, 0, 0x10001960, 0, 0x100137a8 <unfinished ...>
--- SIGINT (Interrupt) ---
+++ killed by SIGINT +++
```

Listado 3. *Rastro a bajo nivel de una conexión*

```
root@/home/carlos# ltrace -p PID
--- SIGSTOP (Stopped (signal)) ---
--- SIGSTOP (Stopped (signal)) ---
strncpy(0x7ff6e40b, "SECRET", 6)                   = 0x7ff6e40b
strncmp("SECRET\366\344\020\177", "SECRET", 6)     = 0
htons(8080, 0x7ff6e409, 6, 84, 84)                 = 8080
socket(2, 1, 0)                                    = 4
connect(4, 0x7ff6e3bc, 16, 84, 84)                 = -1
exit(0 <unfinished ...>
+++ exited (status 0) +++
```

De modo que si se produce este comportamiento, es decir, el proceso vuelve a aparecer con distintos *PIDs* debemos echar un vistazo a */etc/inittab* y buscar algo sospechoso. Pero cuidado, este comportamiento es totalmente legítimo para multitud de aplicaciones, como el famoso *getty* que controla el proceso de login al sistema.

Al margen de esto, si podemos parar el proceso, la identificación es muy fácil, ya que podemos observar la inicialización de este, que muestra signos muy claros de su naturaleza de *backdoor*.

Podemos observar como el *backdoor* se delata solo al llamar a las funciones de la librería *pcap* que preparan la sesión de *sniffing*. Una explicación del propósito de cada función se muestra en la siguiente tabla. Nótese que la instrucción `pcap_compile` revela el filtro utilizado por nuestro *backdoor* (consultar tabla con la explicación de las principales funciones de *libpcap*).

Éste escucha paquetes con destino el puerto *80/TCP* (*HTTP*).

Si nos enfrentamos a un ejecutable compilado estáticamente no podemos utilizar *ltrace*, puesto que el ejecutable no tiene una sección de linkado dinámico. Debemos intentar entonces descifrar el inicio de la sesión de captura a través de las llamadas al sistema. Esto no es tan claro, pero podemos suponer la existencia de un filtro de paquetes si encontramos la siguiente llamada, donde se aplica el mecanismo de filtrado de sockets de *Linux LSF* (*Linux Socket Filtering*). *LSF* permite aplicar un filtro a cualquier *socket* y es utilizado a bajo nivel por programas como *tcpdump* o *wireshark*.

La llamada al sistema tiene la siguiente sintaxis:

```
setsockopt(3, SOL_SOCKET, SO_ATTACH_
FILTER, "\0'\205\370\20\0017\250", 8)
```

Oh, no! No podemos parar el proceso...

Estas técnicas de trazado de las llamadas a funciones de librerías linkadas dinámicamente o de llamadas al sistema son aplicables por supuesto a nuestro proceso aún después de la inicialización.

El problema que nos encontramos al usar este método es que las rutinas de descubrimiento de la interfaz de red, inicio de captura, aplicación del filtro, etc. ya se han realizado. Estas señales pueden revelar fácilmente una sesión de captura de paquetes, de modo que podemos deducir la existencia de *un backdoor* si el programa, *en teoría*, no debería usar ninguna de estas funciones. Por ejemplo, ¿si el programa dice ser *klogd*, un *kernel logger*, que hace capturando paquetes?

Así pues, si intentamos utilizar los mismos métodos con un proceso que ya se está ejecutando, los resultados no nos permitirán concluir si dicho proceso es malicioso. Veamos algunos posibles métodos para el reconocimiento de *los backdoors* en este caso.

Probando con Ltrace (ejecutable dinámico)

Si probamos con *Ltrace*, éste mostrará las llamadas de nuestro proceso a funciones de librerías linkadas dinámicamente. El problema es que *el backdoor* ya ha iniciado la captura, aplicado el filtro y se encuentra en modo de *escucha*. El proceso no realizará ninguna acción hasta que no capturemos un paquete que haya pasado el filtro, por lo que no veremos actividad hasta que no se produzca una coincidencia.

Listado 4. *Ineficiente script para automatizar la busqueda*

```
#!/bin/bash
for x in `ps -ef | awk '{print $2}' | tr -d 'PID'`; do
name=`lsof -p $x | grep txt | awk '{print $9}'`
   if [ -z $name ]; then
      continue
   fi
   echo "$x :: $name : " | tr -d '\n'
   hits=`objdump -t $name | grep pcap | wc -l`
   if [ $hits -gt 0 ]; then
      echo "Uh oh! Uh oh!"
   else
      echo "OK"
   fi
done
```

Tabla 1. *Principales funciones de libpcap y explicación*

Funcion	Descripcion
pcap_lookupnet()	Obtiene la IP y mascara asociados con la interfaz de red que especifiquemos como parametro.
pcap_open_live()	Devuelve un descriptor de captura para la interfaz especificada. Podemos definir tambien parametros como el tamaño de captura, etc.
pcap_compile()	Convierte la cadena de filtrado de un formato compatible con tcpdump a una estructura que puede manejar pcap.
pcap_setfilter()	Aplica la estructura anterior a nuestra sesion de captura.
pcap_loop()	Se encarga de leer paquetes que atraviesan la interfaz y llamar a la funcion de callback pasando como parametro los datos del paquete.

Si quieres formar parte de nuestro equipo y crear la revista hakin9 con nosotros como:

Autor

Nos gustaría que hakin9 fuera una revista realizada por y para los profesionales de la seguridad informática. Por ello, estamos buscando personas con un elevado conocimientos en la materia, expertos en seguridad informática y que les encante escribir. El autor será siempre quién elija el tema.

Corrector

Si la seguridad informática es tu pasión, conoces en profundidad la gramática y ortografía española y lees el Diccionario de la Real Academia todas las noches antes de dormir, posees un perfil ideal para ser nuestro corrector y corregir los textos antes de que sean publicados.

Betatester

Los betatesters son los que leen los artículos y después opinan sobre ellos antes de que salga la revista. Gracias a esto, sabemos cuales son los temas más interesantes para nuestros lectores. Si eres uno de nuestros betatesters tu nombre será publicado en la revista. Cuánto más nos ayudes, más puedes esperar de nosotros. ¡Nuestros betatesters son muy importantes para nosotros! Si quieres saber mas, entra en: http://www.hakin9.org/es/haking/beta.html

Recuerda: Todo depende de tu voluntad, nos ayudas cuando tienes tiempo, ¡y ganas!.

no lo dudes un instante, escribe ahora mismo a:

es@hakin9.org

```
root@/home/carlos# ltrace -p PID
--- SIGSTOP (Stopped (signal)) ---
--- SIGSTOP (Stopped (signal)) ---
```

Aunque no conocemos la expresión del filtro, no está de más hacer un intento a ciegas que podría tener gran probabilidad de éxito. Es obvio que para disparar *el backdoor*, el paquete que *Eva* envió debe estar dirigido a un puerto abierto por nuestro cortafuegos, de lo contrario será descartado antes de que *pcap* pueda analizarlo. Esto nos da ventaja, ya que reduce nuestro espacio de búsqueda.

Podemos enviar un paquete *SYN*, utilizando, por supuesto, `Hping2` a cada puerto abierto en nuestra configuración del firewall y comprobando si se produce alguna respuesta en la salida de *Ltrace*.

```
$ hping2 -S -c 1 -p 80 -e HOLA
ip.de.la.victima
```

El modificador *-e* indica *el payload* del paquete IP, es decir, la secuencia de bytes que se encuentra justo después de la sección de opciones IP.

En nuestro caso, el filtro empleado por *el backdoor* es muy simple y va a filtrar simplemente atendiendo al puerto de destino (*HTTP*) así que si enviamos un paquete al puerto *80*, obtendremos algo como esto:

```
strncpy(0x7ff6e40b, "HOLA", 6)
                        = 0x7ff6e40b
strncmp("HOLA", "SECRET", 6)
                        = -11
```

Vemos que el proceso ha copiado parte del *payload* en una dirección de memoria (en una variable) y compara posteriormente la cadena de texto con la cadena *SECRET*. Suponemos que no ocurre nada más porque no hay coincidencia entre las cadenas. SOSPECHOSO, MUY SOSPECHOSO. El siguiente paso lógico es enviar *el payload* que el programa parece esperar.

```
$ hping2 -S -c 1 -p 80 -e SECRET
                        ip.de.la.victima
```

El resultado de enviar esto es bastante revelador (Listado 4).

Vemos que esta vez la condición se cumple y por lo tanto se desencadena una acción sospechosa. Es el típico código de una conexión inversa, vemos que se abre un `socket()` al puerto *8080* seguido de un `connect()`. Es decir, el ejecutable se conectará a ese puerto de la maquina remota y presumiblemente abrirá una *shell*.

Estas podrían ser pruebas suficientes para matar el ejecutable, aunque no olvidemos que esto es un modelo de juguete con un comportamiento muy simple. En la vida real esto puede no funcionar por diversos motivos:

- El filtro es muy complejo y no conseguimos reproducir el paquete que hace de disparador.
- La acción no es una shell inversa, sino algo más complejo y la salida de *Ltrace* no es concluyente.
- El ejecutable ha sido compilado estáticamente, por lo tanto no podemos aplicar *Ltrace*.

Si estas perdido, mira el mapa

Aunque las llamadas al sistema no sean significativas porque el proceso esta en espera todavía podemos obtener información muy valiosa sobre este.

La mayoría de los sistemas tienen montado el sistema */proc*, que es una forma de comunicación entre el espacio de usuario y el área de kernel. En este sistema de ficheros existe un directorio que representa cada proceso identificado por su *PID*. Dentro de cada directorio hay un montón de ficheros que contienen información detallada proporcionada directamente por el kernel. No hay que dejarse engañar por el tamaño de 0 bytes que lista el *ls*, realizaremos un *cat* del archivo y podremos observar que contienen mucho más de lo que parece a simple vista.

Para nuestro propósito concreto, estamos interesados en el fichero *maps* que contiene el mapa de memoria del proceso en ejecución. Por ejemplo, un fragmento de la salida del comando

```
cat /proc/PID:
0ff9d000-0ffa0000 rwxp 0014e000 03:05
688737     /lib/tls/libc-2.3.6.so
0ffee000-0ffef000 rwxp 0002b000 03:05
180774     /usr/lib/libpcap.so.0.9.5
30000000-30019000 r-xp 00000000 03:05
688179     /lib/ld-2.3.6.so [...]
```

A partir de este ejemplo, podemos observar como el proceso no puede ocultar al kernel la organización de su memoria. La primera columna de la salida anterior muestra precisamente detalles sobre las direcciones de memoria utilizadas por las distintas partes del programa. De este modo podemos determinar que el proceso esta haciendo uso de esta librería, seguramente para nada bueno.

Binarios compilados estáticamente

Ya hemos visto cual es la alternativa a *Ltrace* para binarios compilados estáticamente. Aunque un proceso no llame a funciones de librerías compartidas, en última instancia debe llamar al kernel y éstas llamadas al sistema son también *trazables*. El problema es que al ser de más bajo nivel, no siempre es tan claro interpretar lo que esta haciendo el proceso, como hemos visto anteriormente.

Además, el análisis de la información que el kernel almacena sobre su mapa de memoria no aclara nada, ya que no se encuentra desglosada. Solo podemos ver un gran bloque de memoria donde se encuentran todas las partes del ejecutable. Pero aun así, no debemos desesperarnos, saquemos el arsenal pesado, las herramientas de desensamblado, en concreto *objdump*.

Es bien sabido que a los profesionales no especializados en tareas de programación les tiemblan las piernas cuando oyen hablar del desensamblado, pero en caso concreto realizaremos tareas simples sólamente.

Tanto en la compilación estática

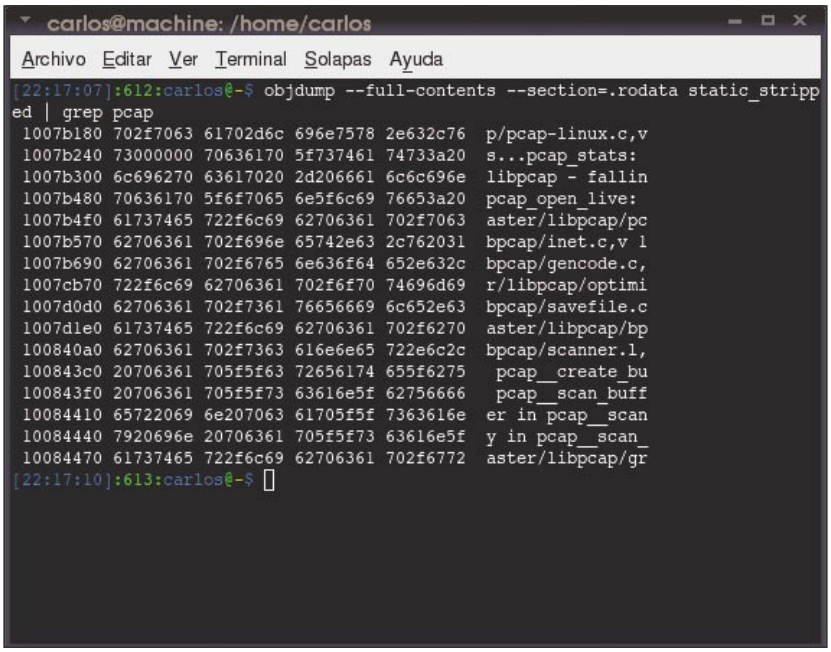

Figura 1. *Buscamos entonces en la sección de datos de sólo lectura de un ejecutable ELF*

como en la dinámica, el paso final es el linkado de los objetos creados durante la compilación (es un abuso del lenguaje llamar compilación al proceso que genera un ejecutable a partir del código fuente, estrictamente debería llamarse *compilación* y *linkado*).

El comando *objdump* esta pensado para mostrar información interna de ficheros objeto, pero si especificamos un ejecutable mostrará información de todos los objetos que lo componen.

La *tabla de símbolos* contiene información sobre los elementos a los que llama un programa y por lo tanto, debe conocer su posición en memoria. Es por ello que independientemente del tipo de linkado, en la tabla de símbolos, encontraremos las referencias a cada elemento, que es justo lo que queríamos.

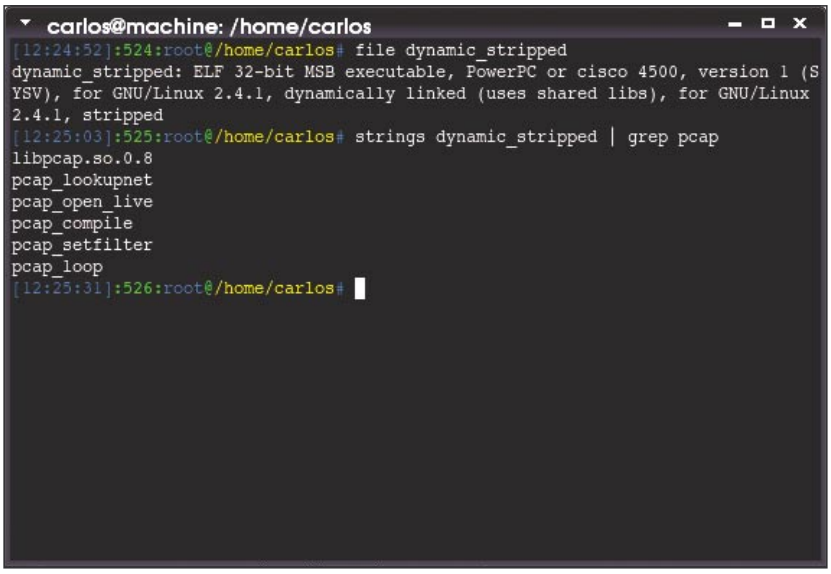

Figura 2. *Observamos las cadenas de texto plano del archivo ejecutable*

Una pequeño arreglo con la shell

Podemos escribir un pequeño script en *bash* para realizar una comprobación de nuestro sistema basándonos en *objdump*.

Stripped? No gracias!

Pero podemos toparnos con una oponente todavía mas hábil, una *Eva* que conoce el comando `strip`.

El resultado de aplicar `strip` a un ejecutable (compilado de forma dinámica o estática) es la eliminación de la tabla de símbolos, de las referencias a las zonas de memoria donde se encuentran los distintos componentes del programa. Por lo tanto no podemos utilizar la técnica anterior ya que la tabla de símbolos esta vacía.

Para descubrir el engaño, debemos profundizar más en la estructura del ejecutable. Por ejemplo, mirando la sección de datos de solo lectura. Esta sección del ejecutable no puede eliminarse ya que contiene cadenas de texto necesarias para la ejecución del código, como por ejemplo, mensajes que aparecen en caso de error. Es obvio que esta sección será mucho mayor en un binario compilado estáticamente pues contiene todo el código necesario para su ejecución, mientras que para uno que haya sido compilado dinámicamente, la mayoría de estas cadenas se encontrarán dentro de la propia librería compartida.

En el caso de un binario compilado estáticamente podemos entonces buscar la cadena `pcap` en su sección de datos estáticos mediante este comando:

```
objdump -full-contents -section=.rodata
executable | grep pcap
```

Podemos observar la salida de este comando en la Figura 1.

¿Pero que ocurre si tenemos un binario compilado dinámicamente (que es lo más probable) al que se le ha aplicado el comando `strip`? ¡Este es un buen ejemplo de mala suerte! Lo más probable es que la inspección de la sección de datos de sólo lectura no nos proporcione ninguna información, debido a que las cadenas de texto se encontrarán en la sección de la propia *libpcap*.

Pero afortunadamente el paquete *binutils*, que podemos encontrar en los sistemas GNU/Linux, contiene un comando que puede salvarnos la vida. Se trata de *strings* y su función es la de mostrar las cadenas de texto plano contenidas dentro de un ejecutable. *Strings* Es básicamente un *wrapper* para *hexdump* que filtra cadenas de más de cuatro caracteres imprimibles seguidas de un carácter no imprimible. A diferencia del método anterior, *strings* busca en todo el ejecutable, por lo que encuentra muchos resultados aunque no todos ellos nos aportan información útil.

Podemos ver una salida de este comando en la Figura 2.

Conclusión

Hemos visto como, debido a la escurridiza naturaleza de este tipo de software, la única manera de detectarlo es buscar trazas de la librería *pcap* en el propio ejecutable. A pesar de que ninguna de las pruebas realizadas se puede considerar como definitiva de la existencia de *un backdoor* silencioso, los resultados de todas ellas en conjunto nos proporcionarían una seguridad casi total de que nos encontramos ante una amenaza.

Aun así, analizar con frecuencia todos los procesos de una equipo mediante el uso de métodos manuales en una tarea bastante tediosa. Es necesaria una herramienta que analice automáticamente y en profundidad el archivo ejecutable, buscando huellas de la existencia de las funciones de *pcap* de un modo similar a como Opera un antivirus en modo de heurística, es decir, buscando huellas de funciones compiladas que no deberían encontrarse en el ejecutable analizado.

Los lectores interesados en este tema pueden consultar la dirección Web *http://sbfinder.sourceforge.net*, donde se hospeda un proyecto de software libre para desarrollar una herramienta de este tipo. ●

Sobre el Autor

Carlos García es licenciado en Física de Partículas por la Universidad de Santiago de Compostela. Posee la certificación CCNA de Cisco Systems y está interesado en múltiples aspectos del networking y la seguridad. Actualmente trabaja en el desarrollo de una herramienta para automatizar la detección de este tipo de backdoors.