©Rosiello Security

http://www.rosiello.org

The Basics of Shellcoding

by Angelo Rosiello

Version: 1.2
Date: 21/05/2004

Contents

1 Introduction

A shellcode is a group of instructions which can be executed while another program is running.

Nowadays, lots of examples show us how a shellcode can be executed while an application is running. A classical way to compute instructions run-time is used by exploits.

In order to get advantage from a vulnerability a shellcode should be injected to get the control of the weak running application.

The goal of this article is not to explain all the possibilities of injecting a shellcode developed during last years, but to analyze and understand its essence.

2 Registers

Before analyzing the assembly code and then the binary one, it is necessary to give an overview of the registers of the CPU to understand their meaning in the assembly language.

The architecture that is going to be described is Intel-x86.

All the registers of the Intel platform support 32 bits which can be divided in subsections of 16 and 8 bits, just to let an heuristic use of the memory.

32 bits	16 bits	8 bits (high)	8 bits (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

EAX, AX, AH, AL These registers are said accumulators and can be used for arithmetical and input/output operations or to execute interrupt calls. We will see how it's important to use them when we have to realize system calls.

EBX, BX, BH, BL These registers are the base registers and they are used as base pointers to access in the memory. We will use these registers to pass the system call arguments. Now and then, they are also used to store the return value of an interrupt. (e.g. when we call an open(), the descriptor value of the file is stored in the register EBX.)

ECX, CX, CH, CL These registers are said counter registers.

EDX, DX, DH, DL These registers are the data registers and they can be used for arithmetical operations, interrupt calls

and some input/output operations.

3 Introducing the Assembly language

The assembly language we are going to approach is said "Inline Assembly" and it adopts the syntax of AT&T.

The name of the registers is preceded by the symbol "%", thus if we have to use the register eax we must type "%eax".

If we are going to refer to numerical constants, its value must be preceded by the symbol "$".

In the following scheme, one can observe the most used instructions in the assembly language.

MOV - This instruction let us to move a value in a register.
mov $0x4, %al - moves 0x4 into al
mov %eax, %ebx - moves what is in eax into ebx

PUSH - Put a value in the stack.

POP - Get a value from the stack and store it in a register or in a variable.

INT - interrupt call.
int $0x80 - it gives the control to the kernel.

4 Codification phase

The algorithm we're going to implement in assembly language and then in binary code(as hexadecimal version) it will print on the video the string "WWW.ROSIELLO.ORG".

The solution of the problem in C language is the following piece of code:

```
int main()
{
write(0, "WWW.ROSIELLO.ORG", 16);
exit(0);
}
```

To realize the write() and the exit() we have to execute their system calls.

It's possible to find in Linux the library "unistd.h" where all the system calls, that can be used, are stored.

```
angelo@rosiello.org$ cat /usr/include/asm-i386/unistd.h

/*
```

* This file contains the system call numbers.
*/

#define _NR_exit 1 <– This is our exit()
#define _NR_fork 2
#define _NR_read 3
#define _NR_write 4 <– This is our write()
#define _NR_open 5

write(0, "WWW.ROSIELLO.ORG", 16);

.....
.....

The first argument "0" is the standard output(video) where we have to print the string which appears as second argument. The last argument, "16", indicates the length of the string.

Let's try to implement this instruction in assembly.

xor %eax, %eax <– It cleans the register %eax

xor %ebx, %ebx

xor %edx, %edx

push %eax <– It inserts NULL into the stack closing the string, thus, no garbage characters will appear.

push $0x47524f2e #push GRO. into the stack

push $0x4f4c4c45 #push OLLE into the stack

push $0x49534f52 #push ISOR into the stack

push $0x2e575757 #push .WWW into the stack

The above four push, insert into the stack the string "WWW.ROSIELLO.ORG" in its hexadecimal codification.

As one can notice the string must be pushed into the stack overturned because of the stack working strategy.

The standard output descriptor is associated with the %ebx register wich contains, at the moment, the value 0 then we have not to indicate anything else. (write(0,..)).

mov %esp, %ecx # it moves %esp into %ecx

Now the string address is in the register %esp (remember that esp is increased/decreased only by pop/push) and we put it in the register %ecx, thus the CPU will be able to find the accurate position of the string in the stack (write(0, string, ..)).

mov $0x10,%dl #size 16 bytes

Exactly as in C language, we say that the string size is 16 bytes (write(0, string, 16)).

mov $0x4,%al #syscall for write()
We put into the register eax (into the low part: al) the number of the write() routine.

int $0x80 #execute the syscall
Now the kernel will gain the control of the application and it will execute our write() routine.

The implementation of the exit(0) is even easier.

exit(0):

xor %eax, %eax
xor %ebx, %ebx
eax and ebx registers are clean.

mov $0x1, %al #syscall for exit()
Let's insert the value of the exit into al.

int $0x80 #execute the syscall
Let's give the control to our kernel.

5 Compile and Execute

The last step to be done is the codification in binary code. In order to reach our purpose we will use the gnu debugger (gdb).

angelo@rosiello.org:\~shellcode$ gdb rosiello

(gdb) disas main
Dump of assembler code for function main:
0x80482f4 <main>: push %ebp
0x80482f5 <main+1>: mov %esp,%ebp
0x80482f7 <main+3>: sub $0x8,%esp
0x80482fa <main+6>: and $0xfffffff0,%esp
0x80482fd <main+9>: mov $0x0,%eax
0x8048302 <main+14>: sub %eax,%esp
0x8048304 <main+16>: xor %eax,%eax
0x8048306 <main+18>: xor %ebx,%ebx
0x8048308 <main+20>: xor %edx,%edx
0x804830a <main+22>: push %eax
0x804830b <main+23>: push $0x47524f2e

0x8048310 <main+28>: push $0x4f4c4c45
0x8048315 <main+33>: push $0x49534f52
0x804831a <main+38>: push $0x2e575757
0x804831f <main+43>: mov %esp,%ecx
0x8048321 <main+45>: mov $0x10,%dl
0x8048323 <main+47>: mov $0x4,%al
0x8048325 <main+49>: int $0x80
0x8048327 <main+51>: xor %eax,%eax
0x8048329 <main+53>: xor %ebx,%ebx
0x804832b <main+55>: mov $0x1,%al
0x804832d <main+57>: int $0x80
End of assembler dump.

Our code begins at the <main+16> instruction and terminates at <main+57>.

To gain the opcode you should adopt the following way.

(gdb) x/bx main+16
0x8048304 <main+16>: 0x31 <– OPCODE
(gdb)
0x8048305 <main+17>: 0xc0 <– OPCODE
(gdb)
0x8048306 <main+18>: 0x31 <– OPCODE
....
and so on till <main+57>.

Now anything must be put in the following way "\x31\xc0\x31..".

```
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x2e\x4f"
"\x52\x47\x68\x45\x4c\x4c\x4f\x68\x52\x4f"
"\x53\x49\x68\x57\x57\x57\x2e\x89\xe1\xb2"
"\x10\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0"
"\x01\xcd\x80"
```

To compile and execute the shellcode you can organize it in a C program in the following schema.

angelo@rosiello.org:\~shellcode$ cat shellcode.c

```
#include <stdio.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x2e\x4f"
"\x52\x47\x68\x45\x4c\x4c\x4f\x68\x52\x4f"
"\x53\x49\x68\x57\x57\x57\x2e\x89\xe1\xb2"
"\x10\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0"
"\x01\xcd\x80";
```

```
main()
{
void (*routine) ();
(long) routine = &shellcode;
printf("Size: %d bytes\n", sizeof(shellcode));
routine();
}
```

```
angelo@rosiello.org:\~shellcode$ gcc shellcode.c -o shellcode
angelo@rosiello.org:\~shellcode$ ./shellcode
Size: 44 bytes.
WWW.ROSIELLO.ORG
```

6 Conclusions

Making a shellcode isn't difficult, but you will need patience and practice to become skilled in doing it.

Shellcoding is very important mainly in the low level applications. For example, if you want to write an exploit you will need to write shellcode to have the exploited program execute the code you want.

Personally, I think that anyone interested in security of computer science should know these basic concepts and theories which support research of new bugs and exploiting ways.

http://www.rosiello.org
contact: angelo@rosiello.org