How to write Shellcodes

por Luiz Fernando Camargo

1. Introdução

Shellcode é um grupo de instruções assembler em formato de opcode para realizar diversas funções como chamar uma shell, ou escutar em uma porta.
Geralmente, um shellcode é utilizado para se explorar determinada vulnerabilidade,
ganhando-se controle sobre a aplicação vulneravel e podendo-se executar qualquer instrução
desejada.
O objetivo deste artigo, não é de explicar todas as possibilidades do uso de uma shellcode, mas analisar e entender a sua essência.

2. Registradores

Antes de analisarmos o codigo assembler e o binario, é necessário darmos uma olhada nos registradores do CPU para entendermos a importância da linguagem assembly. A arquitetura aqui apresentada é Intel-X86. Todos os registradores da plataforma intel, suportam 32 bits dos quais podem ser divididos em subseções de 16 e 8 bits.

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

EAX, AX, AH, AL - Estes registradores são chamados de acumuladores e podem ser usados para operações aritméticas e de entrada e saida ou para executar chamadas de interrupções. Veremos como é indispensavel o uso deles quando temos que realizar chamadas dos sistema(system calls).
EBX, BX, BH, BL - Estes registradores são a base dos registradores e são usados como
ponteiros base para o acesso a memória. Nós iremos usa-los para passar argumentos
de chamada do sistema. Eles também são usados para guardar o valor de retorno de uma
interrupção.
ECX, CX, CH, CL - Estes registradores são contadores.
EDX, DX, DH, DL - Estes são os registradores de dados, eles podem ser usados para operações
aritméticas, chamadas de interrupções e algumas operações de input/output.

3. Introdução a linguagem Assembly
 A linguagem assembly que vamos nos dirigir é chamada de "Inline Assembly" e isso
 adota a sintaxe do AT&T. O nome dos registradores são precedidos pelo simbolo "%",
 assim quando tivermos que usar o registrador eax, precisaremos digitar "%eax".
 Se falarmos em constantes numéricas, esse valor deve ser precedido pelo simbolo "$".
 Vamos agora a uma breve analise das instruções mais usadas no assembly.

 MOV - Esta instrução nos permite mover um valor em um registrador.
 mov %0x8, %al   ; move 0x8 para al
 mov %eax, %ebx   ; move o que eax possui em ebx

 PUSH - Empura dados sobre a Stack.
 push $0x0

 POP - Ao contrario de push, pop retira dados da stack
 pop %edx

 INT - Chamada de interrupção.
 int $0x80   ; isso da o controle ao kernel.

4. A fase de codificação

 O algoritimo que iremos implementar no assembly e no binario, é para escrever na tela "WWW.FIREWALLS.COM.BR". Em C, seria o mesmo que:

```
 int main()
 {
 write(0,"WWW.FIREWALLS.COM.BR", 20);
 exit(0);
 }
```

 Para realizarmos o write() e o exit() precisaremos executar suas chamadas de sistema. É possivel achar no Linux a library "unistd.h", onde esta guardado todas as chamadas de sistemas que podem ser usadas.
 [luiz@firewalls ~]$ cat /usr/include/asm-i386/unistd.h |more

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                 1 <- Este é nosso exit()
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4 <- Este é nosso write()
#define __NR_open                 5
```


Voltando ao nosso código.
O primeiro argumento "0" é o padrão output(video) onde temos que escrever a string que aparece como segundo argumento. O ultimo argumento "20" é o numero de caracteres da string(www.firewalls.com.br). Vamos tentar implementar essas instruções em assembly.

```
xor %eax, %eax   ; Limpa o registrador %eax
xor %ebx, %ebx
xor %edx, %edx
push %eax        ; Insere NULL na stack, fechando a string.
push $0x52422e4d       ; push RB.M na stack
push $0x4f432e53       ; push OC.S na stack
push $0x4c4c4157       ; push LLAW na stack
push $0x45524946       ; push ERIF na stack
push $0x2e575757       ; push .WWW na stack
```

Os cincos push acima inserem na stack a string "WWW.FIREWALLS.COM.BR" em hexadecimal. Como pode ser percerbido, precisamos inserir strings *SEMPRE* ao contrario na stack. Pois é dessa forma que a stack trabalha(LIFO). O output padrão é
associado com o registrador %ebx o qual contém no momento valor 0, então não tivemos que indicar nada mais.

```
mov %esp, %ecx   ; move %esp em %ecx
```

Agora o endereço da string está no registrador %esp e nós a colocamos no registrador %ecx, assim o CPU estara capacitado para achar a posição correta da string na stack.

```
mov %0x14,%dl          ; 20 bytes
```

Como na linguagem C, nós indicamos o tamanho da string, no caso 20 bytes (write(0,string,20).

```
mov %0x4,%al           ; chamada do sistema para write()
```

Nós colocamos no registrador eax(na menor parte) o numero da rotina de write().

```
int $0x80        ; executa a chamada do sistema
```

Agora o kernel ira tomar conta da aplicação e ira executar nossa rotina de write(). A implementação de exit(0) é muito mais facil.
exit(0):

```
xor %eax, %eax
xor %ebx, %ebx
```

Ebx e Eax limpados.

```
mov $0x1, %al          ; chamada do sistema para exit()
int $0x80        ; executa a chamada do sistema
```

5. Compilando e executando

O último passo agora é a codificação do binario. Para conseguirmos alcançar nosso objetivo, uasremos o gnu debugger (gdb). Mas antes, com o nosso código na mão, vamos abir um novo arquivo e acrescentar as seguintes linhas:

```
// firewalls.c - asm code
// por Luiz Fernando
 #include <stdio.h>
 #include <stdlib.h>
 int main()
 {
 __asm__("
 xor %eax, %eax
 xor %ebx, %ebx
 xor %edx, %edx
 push %eax
 push $0x52422e4d
 push $0x4f432e53
 push $0x4c4c4157
 push $0x45524946
 push $0x2e575757
 mov %esp, %ecx
 mov $0x14, %dl
 mov $0x4, %al
 int $0x80
 xor %eax, %eax
 xor %ebx, %ebx
 mov $0x1, %al
 int $0x80 ");
 }
```

Usamos o nosso código assembly em C para evitarmos possíveis problemas na identificação do arquivo. Saia e salve o arquivo como firewalls.c e em seguida compile-o,

```
 [luiz@firewalls ~]$ gcc -o fws firewalls.c
```

Agora vamos debugar o nosso programa com o gbd.

```
[luiz@firewalls ~]$ gdb fws

(gdb) disas main
Dump of assembler code for function main:
0x804830c <main>:       push   %ebp
0x804830d <main+1>:     mov    %esp,%ebp
0x804830f <main+3>:     sub    $0x8,%esp
0x8048312 <main+6>:     and    $0xfffffff0,%esp
0x8048315 <main+9>:     mov    $0x0,%eax
0x804831a <main+14>:    sub    %eax,%esp
0x804831c <main+16>:    xor    %eax,%eax
0x804831e <main+18>:    xor    %ebx,%ebx
0x8048320 <main+20>:    xor    %edx,%edx
0x8048322 <main+22>:    push   %eax
0x8048323 <main+23>:    push   $0x52422e4d
0x8048328 <main+28>:    push   $0x4f432e53
0x804832d <main+33>:    push   $0x4c4c4157
0x8048332 <main+38>:    push   $0x45524946
0x8048337 <main+43>:    push   $0x2e575757
0x804833c <main+48>:    mov    %esp,%ecx
0x804833e <main+50>:    mov    $0x14,%dl
0x8048340 <main+52>:    mov    $0x4,%al
0x8048342 <main+54>:    int    $0x80
0x8048344 <main+56>:    xor    %eax,%eax
0x8048346 <main+58>:    xor    %ebx,%ebx
0x8048348 <main+60>:    mov    $0x1,%al
0x804834a <main+62>:    int    $0x80
```

Nosso codigo começa a partir de main+16 e termina em main+63.
Então, vamos atrás das OPCODE.

```
(gdb) x /bx main+16
0x804831c <main+16>:    0x31
(gdb) x /bx main+17
0x804831d <main+17>:    0xc0
(gdb) x /bx main+18
0x804831e <main+18>:    0x31
(gdb) x /bx main+19
0x804831f <main+19>:    0xdb
...
(gdb) x /bx main+62
0x804834a <main+62>:    0xcd
```

Existe um metódo alternativo usando o 'objdump' que na minha opinião eu considero mais prático.

Para usarmos este meio, vamos ter que fazer o seguinte:
Abra um novo arquivo texto, insira nele o nosso código em assembler,

```
xor %eax, %eax
xor %ebx, %ebx
xor %edx, %edx
push %eax
push $0x52422e4d
push $0x4f432e53
push $0x4c4c4157
push $0x45524946
push $0x2e575757
mov %esp, %ecx
mov $0x14, %dl
mov $0x4, %al
int $0x80
xor %eax, %eax
xor %ebx, %ebx
mov $0x1, %al
int $0x80
```

Saia e salve-o como asmcode.o, agora vamos torna-lo legivel para a máquina,

```
[luiz@firewalls ~]$ as asmcode.o -o code.s
```

Pronto, agora vamos pegar as opcode com o objdump,

```
[luiz@firewalls ~]$  objdump -d code.s

code.s:     file format elf32-i386

Disassembly of section .text:

00000000 <.text>:

   0:   31 c0                   xor    %eax,%eax
   2:   31 db                   xor    %ebx,%ebx
   4:   31 d2                   xor    %edx,%edx
   6:   50                      push   %eax
   7:   68 4d 2e 42 52          push   $0x52422e4d
   c:   68 53 2e 43 4f          push   $0x4f432e53
  11:   68 57 41 4c 4c          push   $0x4c4c4157
  16:   68 46 49 52 45          push   $0x45524946
  1b:   68 57 57 57 2e          push   $0x2e575757
  20:   89 e1                   mov    %esp,%ecx
  22:   b2 14                   mov    $0x14,%dl
  24:   b0 04                   mov    $0x4,%al
  26:   cd 80                   int    $0x80
  28:   31 c0                   xor    %eax,%eax
  2a:   31 db                   xor    %ebx,%ebx
  2c:   b0 01                   mov    $0x1,%al
  2e:   cd 80                   int    $0x80
```

Após pegarmos todos os opcode, vamos montar o shellcode:

```
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x4d\x2e"
"\x42\x52\x68\x53\x2e\x43\x4f\x68\x57\x41"
"\x4c\x4c\x68\x46\x49\x52\x45\x68\x57\x57"
"\x57\x2e\x89\xe1\xb2\x14\xb0\x04\xcd\x80"
"\x31\xc0\x31\xdb\xb0\x01\xcd\x80"
```

Agora vamos utilizar o nosso shellcode, vamos abrir um novo arquivo texto e colocarmos o seguinte conteúdo:

```
// write() & exit();
// 43 bytes shellcode por Luiz Fernando ;)
#include <stdio.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xd2\x50\x68\x4d\x2e"
"\x42\x52\x68\x53\x2e\x43\x4f\x68\x57\x41"
"\x4c\x4c\x68\x46\x49\x52\x45\x68\x57\x57"
"\x57\x2e\x89\xe1\xb2\x14\xb0\x04\xcd\x80"
"\x31\xc0\x31\xdb\xb0\x01\xcd\x80";
main()
{
void (*trash) ();
(long) trash = &shellcode;
printf("Tamanho: %d bytes\n",sizeof(shellcode));
trash();
}
```

Saia e salve como scode.c e compile-o.

```
[luiz@firewalls ~]$ gcc -o scode scode.c
[luiz@firewalls ~]$ ./scode
Tamanho: 49 bytes
WWW.FIREWALLS.COM.BR[luiz@firewalls ~]$
```

Bom, isto é claramente apenas uma introdução o que são e como fazer shellcodes.
Escrito por Luiz Fernando Camargo.
Gostaria de agradecer a duas pessoas muito importantes para a conclusão deste trabalho que são, Adriano Lima e Rodrigo Rubira Branco, valeu mesmo :) !
Dúvidas/Contato: luiz.f@firewalls.com.br