

Scrittura di un eseguibile ELF – status di un i386 dopo la sua esecuzione

Pubblicato da *Ritz* il 20/05/2000

Livello *base*

Introduzione

Ecco in questo tute un esempio di asm prog sotto Linux con conseguente spiegazione su cosa succede una volta che esso è avviato.

Programmi usati

- *nasm*: compilatore (assembly sintassi intel)
- *ld*: linker

Iniziamo

E finalmente ecco che si inizia a programmare... ma prima degli appunti da tenere in considerazione.

Linux, come Win32, gira sui proc Intel x86 in modalità protetta (ring rulezz), e utilizza il modello di memoria flat, quindi abbiamo 4Gb di memoria allocabili come preferiamo (beh vabbè sono meno di 4 Gb., tra SO ecc viene mangiato molto spazio). Per ora non serve saper altro sulla gestione della memoria.

Una nota particolare meritano invece le convenzioni di chiamata: infatti in Linux i parametri di una call non devono essere pushati nello stack come d'altronde accade per gli stessi FreeBSD e BeOS, bensì devono essere messi sequenzialmente nei registri ebx, ecx, edx, esi ed edi, eax contiene il valore di ritorno, lo stack non è nemmeo sfiorato.

Le chiamate al kernel avvengono semplicemente tramite un int 0x80, il numero della chiamata va messo in eax.

La struttura specifica di un file elf potete trovarla nella sezione risorse, cmq anch'essi sono formati da varie sezioni, la .text deve essere sempre presente e rappresenta quella eseguibile, la .data contiene i dati, .bss per i dati non inizializzati.

Detto questo, ecco un sorgente Linux per NASM (sintassi Intel):

```
section .text ;tipo sezione
global _start ;questo serve per il linker
msg db 'Salve mondo',0xA ;messaggio + carattere invio
len equ $ - msg ;lunghezza messaggio
_start: ;entrypoint
mov edx,len ;edx = lunghezza
mov ecx,msg ;ecx = offset mex
mov ebx,1 ;standard output
mov eax,4 ;numero chiamata (sys_write)
int 0x80 ;chiamata al kernel
mov eax,1 ;numero chiamata
int 0x80 ;chiamata al kernel (sys_exit)
```

I sorgenti sono molto semplici da comprendere: vengono dichiarati mex e sua lunghezza, si dichiara l'entrypoint, ebx contiene il file descriptor, edx la lunghezza, ecx punta alla stringa (che deve terminare con 0x00), in eax viene messo il numero di chiamata del kernel che si desidera, in questo caso 4, ovvero sys_write, e poi la chiamata stessa viene eseguita.

Subito dopo in eax si mette il valore 1 e viene chiamato ancora il kernel, precisamente la funzione sys_exit, che esce dal prog. Fine.

Altre info per Linux su /sys/syscall.h (sys call numbers) o asm/unistd.h, kernel src su arch/i386/kernel/entry.S, include/asm-i386/unistd.h, include/linux/sys.h.

Per quanto riguarda FreeBSD, le regole sono un po' diverse da Linux; ecco lo stesso prg:

```
section .text
global _start
msg db "Hello, world!",0xa
len equ $ - msg
_syscall:
int 0x80 ;la nostra chiamata al kernel
ret
_start:
push dword len
push dword msg
```

```

push dword 1 ;convenzione della chiamata
mov eax,0x4
call _syscall
;si sarebbe potuto chiamare il kernel anche con
;push eax
;call 7:0
add esp,12 ;stack pulito
push dword 0
mov eax,0x1
call _syscall

```

La parte iniziale non cambia, ma la convenzione delle chiamate sì: infatti i parametri passati devono essere pushati nello stack, e le chiamate al kernel non devono mai essere fatte direttamente, bensì attraverso un'altra chiamata interna al prog che a sua volta chiama l'int 0x80. Dopo tale chiamata, il valore di ritorno è sempre in eax e lo stack deve essere pulito (in questo caso tramite add esp, 12). C'è cmq un'alternativa a tale chiamata, infatti è possibile, al posto dell'int 0x80, chiamare il gate 7:0 pushando prima eax, e ciò comporta un aumento di dimensioni dell'elf finale, anche se il risultato non cambia.

Altre info per FreeBSD su sys/syscall.h (contiene anche i vari call numbers), i386/i386/exception.s, i386/i386/trap.c.

Per quanto riguarda BeOS, le chiamate sono *identiche* a FreeBSD (lo stesso sorgente non cambia), a parte il fatto che l'int chiamato è 0x25 e i system call number (quelli da mettere in eax) sono diversi (il sys_write è 3 e sys_exit è 3F). Piccola nota: da BeOS i src devono essere compilati col NASM inserendo la riga #include "nasm.h" in float.h e #include <stdio.h> in nasm.h. Per ulteriori info sul BeOS, consultare il file os_beos.inc delle asmutils.

Per tutti e 3 i sistemi, compilate con \$ nasm -f elf ciao.asm, che produrrà l'obj file, e quindi linkate con \$ ld -s -o ciao ciao.asm per creare l'eseguibile.

Lanciate dopo aver dato i giusti permessi eventualmente.

E adesso ecco un piccolo approfondimento su cosa accade a grandi linee in un sistema i386 che usi eseguibili ELF (Linux, FreeBSD, BeOS, ecc.) quando un prg viene lanciato (tali cose possono essere facilmente scoperte osservando i file binfmt_rlf.c e sched.h).

Una volta che dalla shell si preme Invio dopo aver scritto il nome di un prg, viene eseguita la call sys_execve(). Ogni prg in memoria ciuccia minimo 2 pagine: una per la sezione .data e una per lo stack, argomenti ed esecuzione. Ecco un classico layout del segmento creato.

code	sezione .text
data	sezione .data
bss	sezione .bss
.....	spazio vuoto
stack	stack
argoments	argomenti del prg
environment	environment
program name	nome del file elf
dw null	dword finale

Ed ecco cosa accade durante l'esecuzione.

Funzione	Kernel File	Commento
shell	il loggato ;) scrive il nome del prg e preme invio
execve()	la shel chiama la funzione libc
sys_execve()	la libc chiama il kernel
sys_execve()	arch/i386/kernel/process.c	zona kernel
do_execve()	fs/exec.c	apre il file e prepara un po' di robette
search_binary_handler()	fs/exec.c	osserva il tipo di eseguibile
load_elf_binary()	fs/binfmt.c	carica l'elf e le relative librerie e crea il segmento apposito
start_thread()	include/asm-i386/processor.h	passa il controllo al codice del nostro prg

Questo invece è lo stato dello stack quando l'elf viene lanciato.

argc	dw (argumenti counter, integer)
argv(0)	dw (nome del prg, puntatore)
argv(1) argv(argc-1)	dw (argomenti del programma, puntatori)

NULL	dw (fine argomenti, integer)
env(0) env(1) env(n)	dw (puntatori alle variabili)
NULL	dw

Ecco quindi che per prendere gli argomenti basta popparli uno per uno dallo stack. Per finire, ecco lo status dei registri come viene organizzato una volta lanciato il prg, quali sono modificati e quali no (i valori, a parte 0x0, sono cmq solo un esempio, dipendono da caso a caso).

Registri	Kernel 2.0.x	Da kernel 2.2.10 in poi
EAX	0x0	0x0
EBX	non modificato	0x0
ECX	non modificato	0x0
EDX	0x0	0x0
ESI	non modificato	0x0
EDI	non modificato	0x0
EBP	non modificato	0x0
ESP	0xBFFFFFFE14	0xBFFFFFFB40
EFLAGS	0x282	0x292
CS	0x23	0x23
DS	0x2B	0x2B
ES	0x2B	0x2B
FS	0x2B	0x0
GS	0x2B	0x0
SS	0x2B	0x2B

Anche per oggi è tutto, buon divertimento con NASM o GAS;) !!

Conclusioni

Spero di esservi stato utile, per dubbi chiarimenti proposte ecc. scrivetemi pure a ritz@freemail.it.
Ciauz a tutti, Ritz.