Architettura dei processori Intel (For Totally Newbies)		
Data	by Spider	
ottobre 2001	***UIC's Home Page***	**Published by Quequero**
	Qualche mio eventuale commento sul tutorial :)))	
....	**Home page:** http://bigspider.cjb.net **E-mail:** spider_xx87@hotmail.com **Server IRC: irc.azzurra.it** **Canali: #crack-it #asm** **Nickname: ^Spider^**	....
Difficoltà	(X)**NewBies** ( )**Intermedio** ( )**Avanzato** ( )**Master**	

Prima di incominciare a crackare o a programmare è indispensabile avere una certa confidenza con l'architettura dei processori Intel e compatibili.

Architettura dei processori Intel

(For Totally Newbies)

Written by Spider

Introduzione

Prima di incominciare a crackare o a programmare è indispensabile avere una certa confidenza con l'architettura dei processori Intel e compatibili. (Sì, è uguale a quello di sopra!...)

Tools usati

- Un cervello funzionante!

- Possibilmente i manuali Intel. Li trovate qui: http://developer.intel.com

Essay

Ogni applicazione che viene eseguita, anzi più precisamente ogni task, ha a disposizione un set di registri che servono ai più disparati usi.

Ricordatevi queste uguaglianze:

1 byte = 8 bit
1 word = 2 bytes = 16 bit
1 dword = 2 words = 4 bytes = 32 bit

Per prima cosa vediamo i *General Purpose Registers*, cioè i registri di uso generale. Essi sono EAX, EBX, ECX, EDX. Questi registri sono a 32 bit, ma fino ai processori 286 erano a 16 bit. Il corrispondete a 16 bit di ogni registro si ottiene togliendo la 'E'. Ognuno di questi registri è così strutturato:

EAX		
	AX	
	AH	AL

EAX è un registro a 32 bit. AX è un registro a 16 bit, contenuto nella parte bassa di EAX, ovvero nei sedici bit inferiori. AH e AL, contenuti rispettivamente nella parte alta e nella parte bassa di AX, sono registri a 8 bit. Dato che AH ed AL sono contenuti in AX, è chiaro che modificando uno di questi di modifica anche il contenitore, e lo stesso vale per AX ed EAX.
EBC, ECX ed EDX sono strutturati come EAX, e quindi avremo EBX, BX, BH e BL; ECX, CX, CH e CL; EDX, DX, DH e DL.
Questi registri vengono di solito usati come variabili temporanee.

ESI ed EDI sono i registri indice, rispettivamente *Source Index* e *Destination Index*. Questi registri sono generalmente usati per lavorare con le stringhe, ma possono essere utilizzati per qualunque cosa, anche come registri di uso generale.
ESI ed EDI sono registri a 32 bits, ma hanno dei corrispondenti a 16 bits denominati SI e DI. Non contengono tuttavia registri a 8 bits.

Passiamo ai registri puntatori, cioè EBP ed ESP (*Base Pointer* e *Stack Pointer*). EBP punta generalmente alla base dello stack, ed ESP punta all'indirizzo dello stack correntemente in uso. Vedremo più avanti cos'è lo stack e come funziona.
EBP ed ESP sono registri a 32 bits, e come i registri indice (ESI ed EDI) hanno i rispettivi registri a 16 bits, denominato BP e SP.

EIP è anch'esso un registro puntatore, che punta all'istruzione da eseguire. Questo registro è diverso dagli altri perché non è possibile modificarlo direttamente. E' bene fare attenzione quando si maneggia questo registro, perché se si sbaglia un crash è assicurato.

Tra i registri analizzati finora abbiamo:

8 registri di un byte (AH, AL, BH, BL, CH, CL, DH, DL)

8 registri di una word (AX, BX, CX, DX, SI, DI, BP, SP), più l'IP che però è un registro a parte e che funziona in modo differente

8 registri di 1 dword (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), più l'EIP.

Adesso vediamo i *Segment Registers*. Essi sono:

CS - Code Segment (Segmento di codice)
DS - Data Segment (Segmento dati)
SS - Stack Segment (Segmento dello stack)
ES - Extra Segment (Segmento extra)

Poi ci sono FS e GS, i cui nomi non hanno un significato particolare. Questi sono stati aggiunti a partire dai processori 386.
I registri segmento vengono utilizzati assieme ad altri registri o comunque con altri valori per ottenere gli indirizzi di memoria.

Infine ci sono alcuni registri speciali:

DR0, DR1, DR2, DR3, DR6 e DR7 sono i Debug Registers.

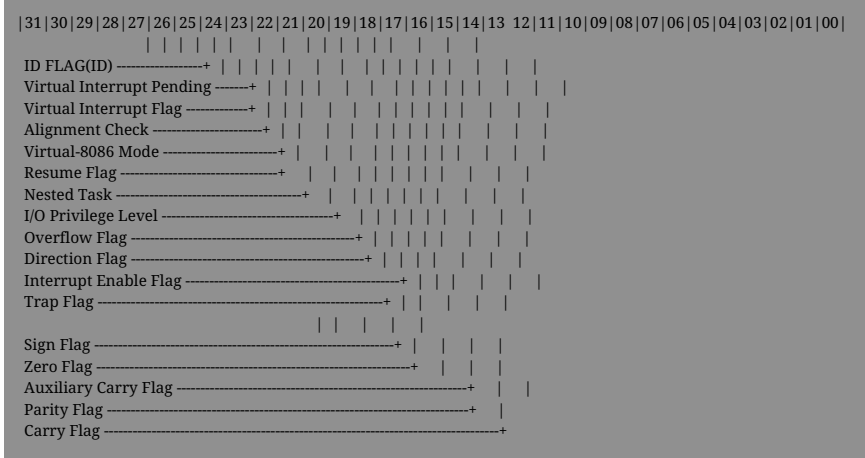
CR0, CR2, CR3, CR4 sono i Control Registers.

TR4, TR5, TR6, TR7 sono i Test Registers.

Questi registri sono tuttavia utilizzati solo nella programmazione avanzata, dai sistemi operativi, dai debuggers, etc. Addirittura i Test Registers sono sostanzialmente inutili per i programmatori.

In ultimo vediamo EFLAGS. Questo è un registro diverso dagli altri, perché ogni bit va considerato separatamente dagli altri.

Ecco uno schema:



Questi Flags si dividono in due gruppi: *System Flags* e *Status Flags.* Quelli del primo gruppo sono, appunto, Flag di sistema, e qui non li analizzeremo. Diamo invece un'occhiata al secondo gruppo, cioè quello degli Status Flags. Questi Flags indicano il risultato di una operazione aritmetica o logica. Analizziamoli singolarmente:

Carry Flag (CF) - Flag di riporto. Viene settato (cioè viene messo uguale ad uno) quando c'è un riporto o un prestito dal bit più significativo di una operazione... E' un po' difficile da spiegare =)

Parity Flag (PF) - Flag di parità. Viene settato quando il risultato di un'operazione contiene un numero pari di 1. Viene generalmente usato nei sistemi di trasmissione dati come sistema di controllo.

Adjust Flag o Auxiliary Carry Flag (AF) - Viene settato quando c'è un riporto o un prestito dal terzo bit di una operazione. E' azzerato in caso contrario.

Zero Flag (CF) - Viene settato se il risultato di una operazione è zero, altrimenti viene azzerato.

Sign Flag (CF) - Flag di segno. Viene settato Se dopo un'operazione aritmetica (o logica) il bit più significativo (ovvero quello che rappresenta il segno) è 1. In caso contrario viene azzerato.

Overflow Flag (CF) - Viene settato settato quando il risultato di una operazione è troppo grande o troppo piccolo per essere contenuto nel registro di destinazione; è azzerato altrimenti.

Lo Stack

Un'altra fondamentale caratteristica è lo Stack. Per Stack (dall'inglese "catasta, pila") si intende un'area di memoria utilizzata dai programmi per memorizzare dati e informazioni temporanee. Lo Stack ha una struttura definita LIFO, ovvero Last In First Out, nel senso che l'ultimo elemento immesso sarà il primo ad essere prelevato. E possibile immaginare lo Stack come una pila di piatti, in cui l'ultimo piatto messo su sarà il primo ad essere preso.
Per lavorare con lo Stack occorrono fondamentalmente 2 cose: il Segmento dello Stack, puntato dal registro SS, e il puntatore allo Stack, formato appunto dal registro ESP. Ogni volta che immettiamo un valore nello Stack (generalmente tramite un'istruzione chiamata PUSH), ESP viene decrementato di 4 (nel caso di processori a 32 bit. Se il processore dovesse essere o dovesse lavorare a 16 bit invece si decrementa di 2) e il valore voluto viene copiato nell'area di memoria puntata da ESP. Curiosamente ad ogni immissione nello stack il registro ESP viene decrementato, mentra ad ogni prelievo (con un'istruzione chiamata POP)

viene incrementato. Chiariamo con uno schema:

Supponiamo che sia questa la situazione dello Stack:

Indirizzo:	*Valore in memoria:*	
0063FE40		
0063FE3C		
0063FE38		
0063FE34	22334455	<--- Valore puntato da ESP
0063FE30	11223344	

Supponiamo adesso di voler salvare in memoria il valore contenuto in EAX. Faremo dunque un:

PUSH EAX

Dopo quest'istruzione, la situazione dello Stack sarà la seguente:

Indirizzo:	*Valore in memoria:*	
0063FE40		
0063FE3C		
0063FE38	*<EAX>*	<--- Valore puntato da ESP
0063FE34	22334455	
0063FE30	11223344	

Con *<EAX>* ovviamente intendo il valore contenuto in EAX al momento del push.
Quando successivamente vorremo ripristinare il valore di EAX, basterà fare:

POP EAX

Dopo questa istruzione lo Stack tornerà allo stadio iniziale, ed EAX avrà il valore precedentemente salvato.

Molto spesso accadrà di trovare (o scrivere) codice simile a questo:

```
push eax
push ebx
push ecx

[. . .]

pop ecx
pop ebx
pop eax
```

Questo codice non fa altro che salvare e ripristinare i valori iniziali dei rispettivi registri, che potrebbero eventualmente essere modificati, compromettendo in alcuni casi l'esecuzione del programma stesso. In questo modo, invece, non toccando i registri, siamo sicuri di non fare danni :-) Notate come i registri vengono poppati nell'ordine inverso rispetto a come vengono pushati. Un errore molto comune è quello di popparli con lo stesso ordine, e ciò è sbagliatissimo. Ricordatevi sempre della struttura LIFO! :)

Con questo si chiude la parte sull'architettura fondamentale dei processori Intel. Tuttavia, i processori moderni hanno altre nuove caratteristiche, come la FPU e la più recente MMX Technology, che vale la pena di menzionare.

Floating-Point Unit

La FPU è un argomento abbastanza vasto, e tra l'altro non lo conosco molto bene, quindi faremo solo un riassunto molto breve.

La FPU è stata inserita, credo, a partire dal processore 486. Essa è in grado di eseguire operazioni a 32, 64 e 80 bits, in intero e in virgola mobile.
Essa è dotata di 8 registri da 80 bits (ovvero da 1 tenbyte), chiamati ST(i), dove i è un identificativo diverso per ogni registro. Questi registri non sono tuttavia accessibili direttamente. Essi, infatti, hanno una struttura molto simile a quella dello Stack. L'equivalente del registro ESP, per la FPU è il TOS, ovvero Top Of Stack, che tuttavia non è neanch'esso accessibile. Intanto vediamo la loro struttura con uno schema:

Registro:	*Valore:*	
ST	13355345	
ST(1)	2366af76	
ST(2)	9816a7fe	
ST(3)	be578d9	<------ Top Of Stack
ST(4)	46456856	
ST(5)	f7388762	
ST(6)	9998723a	

ST(7)	FFE467DB

Prima di iniziare ad utilizzare la FPU, è bene fare un FINIT. Infatti noi non sappiamo in che stato si trovano i suoi registri, e quindi ci viene in aiuto FINIT, che azzera tutto, compreso il TOS. Dopo un FINIT lo stato che troveremo sarà il seguente:

Registro:	*Valore:*	
ST	0	<------ Top Of Stack
ST(1)	0	
ST(2)	0	
ST(3)	0	
ST(4)	0	
ST(5)	0	
ST(6)	0	
ST(7)	0	

Dato che noi vogliamo solo capire il funzionamento e non spiegare tutte le funzionalità della CPU, chiariamo il tutto con un semplice esempio. Supponiamo che vogliamo sommare Valore1 e Valore2, entrambe, in questo caso, dwords (ovvero sono operandi di 32 bits):

```
.data
Valore1 dword 1
Valore2 dword 2
Risultato dword 0

.code

[. . .]

finit
fld Valore1
fadd Valore2
fstp Risultato
```

Analizziamo passo passo la situazione. Dopo finit la FPU sarà così:

Registro:	*Valore:*	
ST	0	<------ Top Of Stack
ST(1)	0	
ST(2)	0	
ST(3)	0	
ST(4)	0	
ST(5)	0	
ST(6)	0	
ST(7)	0	

Dopo il "fld Valore1" sarà così:

Registro:	*Valore:*	
ST	Valore1	
ST(1)	0	<------ Top Of Stack
ST(2)	0	
ST(3)	0	
ST(4)	0	
ST(5)	0	
ST(6)	0	
ST(7)	0	

Dopo il "fadd Valore2",avremo questa situazione:

Registro:	*Valore:*	
ST	Valore1 + Valore2	
ST(1)	0	<------ Top Of Stack
ST(2)	0	
ST(3)	0	
ST(4)	0	

ST(5)	0
ST(6)	0
ST(7)	0

Dopo il "fstp Risultato":

Registro:	*Valore:*	
ST	0	<------ Top Of Stack
ST(1)	0	
ST(2)	0	
ST(3)	0	
ST(4)	0	
ST(5)	0	
ST(6)	0	
ST(7)	0	

e la variabile *Risultato* conterrà il valore appena trovato.

Ci sarebbero moltissime istruzioni da vedere sulla FPU, ma questo tute riguarda l'architettura, quindi vediamo di non andare fuori argomento! :)

Ora due parole sulla MMX Technology.

MMX Technology

Come è ampiamente documentato nei manuali Intel, le estensioni MMX sono state introdotte nei processori Intel Pentium II e nei Processori Intel Pentium con Tecnologia MMX. Esse sono destinate principalmente a migliorare le prestazioni delle applicazioni multimediali o comunque di tutte quelle che fanno un largo uso di dati in modo ripetitivo.

La Tecnologia MMX fornisce 8 registri da 64 bits, denominati da MM0 a MM7, tramite i quali è possibile lavorare con i cosiddetti "packed data types". Ciò significa che ogni registro lavora con più bytes, words o dwords contemporaneamente. Più precisamente, queste istruzioni possono lavorare con 8 bytes, con 4 words o con 2 dwords nello stesso momento, con chiari vantaggi nel caso di operazioni ripetitive su array di memoria, ovvero dove si deve ad esempio ripetere la stessa operazione per ogni byte di un determinato array, e invece di lavorare con un byte per volta, possiamo lavorare con 8 bytes per iterazione :)

La MMX Technology consiste di 47 istruzioni:

- Trasferimento di dati
- Aritmetica
- Confronti tra operandi
- Conversioni
- Unpacking, ovvero separazione dei vari componenti di un packed data
- Operazioni logiche e di shift

Non analizziamo tutte le istruzioni... questo è solo un tute sull'architettura. Se siete interessati fate riferimento ai manuali Intel.

Con questo abbiamo finito.Spero di non avervi troppo annoiato con tutta questa teoria, ma è proprio essenziale! Io agli inizi tendevo a saltarla, e molto probabilmente anche voi farete lo stesso. Beh, vi dico solo di tenere sempre presente della buona documentazione teorica, e di fare mooooooolta pratica, perchè è con la pratica che si impara la teoria.

Ciauz!

Spider

Note finali

Ringrazio AndreaGeddon perchè mi sono ispirato al suo tutorial :)

Saluti a Quequero, di nuovo AndreaGeddon, Yado, kill3xx, +Malattia, case, albe, TheMR, bubbo, deimos, dades, Blackdeath, phobos, DsE, Cieli Sereni, [cHr], Quake2, True-love... e basta credo. Spero di non aver dimenticato nessuno! Se l'ho fatto, non l'ho fatto apposta! =)

Disclaimer

Home