ARCHITETTURA DELL' x86

written by AndreaGeddon

Esaminiamo l'architettura generale della famiglia di processori x86.
Prima di tutto alcuni chiarimenti:
1 byte = 8 bit
1 byte = un valore da 00 a FF in esadecimale
1 byte = 1 carattere della tabella ASCII (che va da 0 a 255 decimale = 00 -> FF hex)

Registri generali (General Purpose Registers):
EAX, accumulatore
EBX, base
ECX, contatore
EDX, dati

questi registri nei moderni processori sono tutti a 32 bit, ma prima erano a 16 e prima ancora ad 8 bit. Ecco la loro struttura:

```
bit         00000111100001111000011110000 1111
                                |---AH----|---AL---|          AH AL  = 8 bit
                                |---------AX--------|         AX     = 16 bit
                |-------------------EAX-------------------|   EAX      = 32 bit
```

però nel cracking il registro verrà considerato non in binario, ma in esadecimale, quindi avremo:

```
hex             1 2 A 6 B E 9 C
                        | ah  | al  |           ah al = 1 byte
                        |     ax    |           ax   = 2 byte (1 word)
                  |         eax         |       eax =   4 byte (1 dword)
```

anche gli altri registri EBX, ECX e EDX sono strutturati in questo modo. Vengono generalmente usati come variabili. E' importante tenre d'occhio quali registri vengono modificati dopo le call alle API per vedere i valori restituiti: di solito ci si trovano cose interessanti.

Poi ci sono i regsitri puntatori ed indici:
ESP, puntatore allo stack
EBP, puntatore alla base dello stack
EDI, indice destinazione
ESI, indice sorgente
Questi sono registri a 32 bit (che prima erano a 16 bit). ESP punta allo stack correntemente in uso, e quindi ci troverete gli ultimi dati salvati. Per capire meglio lo stack vedi sotto. EBP punta alla base dello stack, e quindi la parte di memoria tra EBP ed ESP identifica tutto lo stack correntemente in uso. EDI ed EDI servono come puntatori da sorgente a destinazione per le operazioni tra stringhe: per esempio se voglio controllare due stringhe di testo e vedere se sono uguali, il codice in assembler sara:

```
PUSH  EDI          ---->    salva il puntatore alla stringa sorgente
PUSH  ESI          ----->   salva il puntatore alla stringa di destinazione
REPZ  CMPSB        ---->    esegue un compare byte per byte delle stringhe puntate da edi e da esi
JNZ errore         ----->   se le due stringhe sono diverse salta ad un messaggio di errore
```

NOTA: i registri fin qui descritti sono stati progettati per svolgere il compito che vi ho descritto, ma ciò non esclude che possano essere usati (e questo nella maggior parte delle volte) per qualsiasi scopo, praticamente sono usati come variabili.

Infine c'è anche **EIP**, che è il puntatore alla prossima istruzione da eseguire. Questo registro è SEMPRE usato per puntare l'esecuzione della prossima istruzione, e conviene non toccarlo, a meno che non si è sicuri di quello che si sta facendo.

I registri segmenti:
CS, Segmento codice
DS, Segmento dati
SS, Segmento stack
ES, Segmento extra
FS, Da 386 in poi

GS, Da 386 in poi
Questi registri vengono combinati con altri registri per formare gli indirizzi di memoria. Generalmente si può fare a meno di conoscerli, ma è sempre meglio sapere che cosa sono. Per capire perchè vengono usati per formare indirizzi di memoria, leggete il testo relativo al problema dell'indirizzamento dell' x86.

I registri speciali:
Ci sono anche questi registri:
CR0, CR2, CR3
Sono i Control Registers.
TR4, TR5, TR6, TR7
Sono i Test Regsters.
DR0, DR1, DR2, DR3, DR6, DR7
Sono i Debug Registers.
Questi ultimi non sono utili ai fini del cacking, quindi non serve conoscerli.

Il registro FLAG:
Il FLAG register è un registro a 16 bit nel quale ogni bit è considerato separatamente. Ogni bit rappresenta un Flag, ovvero una condizione specificata dalle istruzioni del programma. Ad esempio, se eseguo un compare di due registri, e il cmpare vede che sono uguali, mi setta il flag ZERO su 1. Se la linea successiva contiene l'istruzione JZ (salta se zero flag è attivo), il codice salta alla parte di codice da eseguire nel caso che i registri fossero uguali.
Ecco il suo significato bit-per-bit:

| 11 | 10 | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
flag register
dove:
0 ----> CF Carry Flag (flg di riporto)
viene posto ad uno quando c'è stato un riporto o un prestito dal bit di ordine alto del risultato a 8 0 16 bit.
1 ----> 1
2 ----> PF Parity Flag (flag di parità)
Se è ad uno vuol dire che il risultato dell'operazione eseguita ha un numero pari di 1. Viene spesso usato per il controllo degli errori nella trasmissione dei dati.
3 ----> 0
4 ----> AF Auxiliary Flag (flag ausiliaria)
Se è ad uno allora c'è stato un riporto del nibble inferiore a quello superiore o un prestito dal nibble superiore a quello inferiore.
5 ----> 0
6 ----> ZF Zero Flag
Viene messo a zero se il risultato di un'operazione è zero.
7 ----> SF Sign Flag (flag di segno)
Viene messo ad uno quando il bit superiore del risultato di una operazione è un bit di segno. Se SF è ad 1, il primo bit del risultato sarà: 0 - positivo e 1 - negativo.
8 ----> TF Trap Flag (flag trappola)
9 ----> IF Interrupt Flag
A ----> DF Direction Flag (flag di direzione)
B ----> OF Overflow Flag (flag di overflow)
C ----> IOPL I/O Privilege Level (livello di privilegio I/O)
D ----> IOPL I/O Privilege Level (livello di privilegio I/O)
E ----> NT Nested Task Flag (flag di task annidato)
F ----> 0
10 ----> RF Resume Flag (flag di ripresa)
11 ----> VM Virtual Mode Flag (flag del modo virtuale)

Alcune parole sul funzionamento dello STACK:
Lo stack (dall'inglese "catasta") è una porzione di memoria usata per memorizzare i dati dal programma in esecuzione. Tecnicamente fa parte della categoria delle "pile", ed ha una struttura LIFO (last in first out) che vuol dire che l'ultimo dato immesso sarà poi il primo ad esser preso. E' importante capire il meccanismo dello stack, altrimenti si potrebbero commettere errori gravi. Il solito esempio che si fa è quello di considerare lo stack come una pila di piatti: l'ultimo piatto che poggeremo sulla cima sarà il primo che poi prenderemo. Solo che nel caso dello stack, la pila di piatti è capovolta. Facciamo un esempio:

condizione dello stack all'inizio:

contenuto					WORD 5	WORD 4	WORD 3	WORD 2	WORD 1
indirizzo	0004	0003	0004	0005	0006	0007	0008	0009	0010

Consideriamo il contenuto e gli indirizzi di memoria. Nel contenuto supponiamo che siano state salvate 5 WORD (cioè 5 valori da 16 bit). Se noi salviamo una nuova word (WORD 6), la nuova condizione dello stack sarà:

contenuto				WORD 6	WORD 5	WORD 4	WORD 3	WORD 2	WORD 1
indirizzo	0004	0003	0004	0005	0006	0007	0008	0009	0010

Ora se noi eseguiamo l'operazione POP per richiamare un dato dalla memoria, richiameremo WORD 6. Se vogliamo richiamare direttamente la WORD 2 non possiamo (dovremmo definire un puntatore a questa locazione). Dopo aver inserito la WORD 6 di fatto noi abbiamo decrementato lo stack, che prima aveva la sommità all'indirizzo 0006, adesso ce l'ha all'indirizzo 0005.
Se adesso richiamiamo due valori, con ad esempio POP EAX e POP EBX, avremo:

contenuto						WORD 4	WORD 3	WORD 2	WORD 1
indirizzo	0004	0003	0004	0005	0006	0007	0008	0009	0010

lo stack si è incrementato, e in EAX viene messo il valore WORD 6, in EBX il valore WORD 5.
Non c'è nulla di difficile, basta sempre ricordarsi di LIFO (last in first out).
Un errore che capita spesso è quello di scrivere:

```
PUSH  EAX
PUSH  EBX
```

e poi richiamare nei registri il loro rispettivo valore così:

```
POP  EAX
POP  EBX
```

in questo caso verrà messo in EAX il valore prima contenuto in EBX, poi verrà messo in EBX il valore prima contenuto in EAX. Il tutto è stato invertito.
La giusta sequenza sarebbe dovuta essere questa:

```
PUSH EAX
PUSH EBX
...
POP EBX
POP EAX
```

in questo modo nei registri viene ripristinato il loro valore di partenza.

Gli INTERRUPT
Gli interrupt sono delle funzioni che il sistema mette a disposizione del programma. Ogni interrupt ha una sua funzione specifica, con uno specifico risultato. Sono un pò delle API in miniatura. Ad esempio, per scrivere una stringa sul testo possiamo utilizzare il seguente codice:

```
Stringa         DB    'Stringa da stampare su schermo$'
...
mov      dx, offset Stringa
mov      ah, 9
int       21h
```

Qui usiamo l'interrupt 21 con funzione 9 (in ah), che prende il testo puntato da dx e lo stampa su schermo. Ci sono tanti interrupt per tantissime funzioni che permettono di controllare il pc davvero a bassissimo livello. Per conoscere tutti gli interrupt o vi scaricate la famosa lista di Ralph Brown, o vi prendete un manuale. Insomma, conoscerli è essenziale.

E con questo si conclude la mia piccola guida all'architettura interna degli x86.
AndreaGeddon