



S10/L3

Fondamenti linguaggio Assembly

TRACCIA

Nella lezione teorica del mattino, abbiamo visto i fondamenti del linguaggio Assembly. Dato il codice in Assembly per la CPU x86 allegato qui di seguito, identificare lo scopo di ogni istruzione, inserendo una descrizione per ogni riga di codice. Ricordate che i numeri nel formato 0xYY sono numeri esadecimali. Per convertirli in numeri decimali utilizzate pure un convertitore online, oppure la calcolatrice del vostro computer (per programmatori).

```
0x00001141 <+8>:  mov  EAX,0x20
0x00001148 <+15>:  mov  EDX,0x38
0x00001155 <+28>:  add  EAX,EDX
0x00001157 <+30>:  mov  EBP,EAX
0x0000115a <+33>:  cmp  EBP,0xa
0x0000115e <+37>:  jge  0x1176 <main+61>
0x0000116a <+49>:  mov  eax,0x0
0x0000116f <+54>:  call 0x1030 <printf@plt>
```



Assembly - definizione

L'assembly è un linguaggio di programmazione di **basso livello**, simile al linguaggio macchina, che traduce direttamente le istruzioni della CPU. Le istruzioni dell'assembly sono costituite da un **codice mnemonico**, ovvero una parola che identifica l'**istruzione da eseguire**, e uno o più operandi che identificano **variabili, valori o indirizzi di memoria**. È specifico per l'architettura del processore e permette il controllo dettagliato dell'hardware. È usato per scrivere codice ad alte prestazioni o per l'interazione diretta con l'hardware del computer. Per lo scopo del corso ne abbiamo bisogno per capire l'**analisi dei malware**, nello specifico l'assembly che riguarda l'architettura **x86**.

Istruzioni e significato

La traccia dell'esercizio ci fornisce un codice composto da istruzioni Assembly per la CPU x86. Ogni riga del codice include un indirizzo di memoria **univoco** (a sinistra) che mostra la posizione **esatta** della specifica **istruzione all'interno del programma**, seguito dall'**offset**, che indica il punto di **esecuzione relativo**, e specifica la distanza tra due indirizzi di memoria. Da come mi è parso di capire, è come una misura della distanza tra due punti: ad esempio, quanto è lontano un elemento in un array rispetto all'inizio dell'array stesso. Gli offset sono utili per trovare elementi specifici dentro array o strutture dati.

L'istruzione stessa invece, a destra, specifica l'**operazione da eseguire**, come caricare valori nei registri, **sommare** numeri o **confrontare** valori.

Riguardo indirizzo dell'istruzione e offset ci viene chiesto di ignorarli in quanto non sono stati ancora affrontati nella lezione teorica. Comunque sia, da curioso quale sono, ho sbirciato dietro le quinte di un semplice programmino scritto in C **sulla mia Debian**:

Attraverso l'inferno a testa alta

Objdump è uno strumento da riga di comando che disassembla i file eseguibili. Su Linux, questi file sono di tipo ELF (Executable and Linkable Format), diversamente, su Windows, sono di tipo PE (Portable Executable), come abbiamo potuto constatare nella prima lezione dell'analisi di malware. Ho provato a lanciarlo sul programmino scritto in C, ed è uscito l'inferno. Però è una vista interessante, da approfondire e studiare.

```
Terminale - flavio@debian: ~
GNU nano 7.2 test.c *
#include <stdio.h>
int main() {
    printf("Ciao Flavio");
    return 0;
}
```

```
flavio@debian:~$ nano test.c
flavio@debian:~$ gcc -o ciao test.c
flavio@debian:~$
```

```
flavio@debian:~$ objdump -d ciao
ciao:      formato del file elf64-x86-64

Disassemblamento della sezione .init:
0000000000001000 <_init>:
1000:  48 83 ec 08      sub    $0x8,%rsp
1004:  48 8b 05 c5 2f 00 00  mov    0x2fc5(%rip),%rax        # 3fd0 <__gmon_start__@Base>
100b:  48 85 c0          test   %rax,%rax
100e:  74 02            je     1012 <_init+0x12>
1010:  ff d0            call   *%rax
1012:  48 83 c4 08      add    $0x8,%rsp
1016:  c3              ret

Disassemblamento della sezione .plt:
0000000000001020 <printf@plt-0x10>:
1020:  ff 35 ca 2f 00 00  push   0x2fca(%rip)             # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
1026:  ff 25 cc 2f 00 00  jmp     *0x2fcc(%rip)          # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
102c:  0f 1f 40 00      nopl   0x0(%rax)

0000000000001030 <printf@plt>:
1030:  ff 25 ca 2f 00 00  jmp     *0x2fca(%rip)          # 4000 <printf@GLIBC_2.2.5>
1036:  68 00 00 00 00 00  push   $0x0
103b:  e9 e0 ff ff ff  jmp     1020 <_init+0x20>

Disassemblamento della sezione .plt.got:
0000000000001040 <__cxa_finalize@plt>:
1040:  ff 25 9a 2f 00 00  jmp     *0x2f9a(%rip)          # 3fe0 <__cxa_finalize@GLIBC_2.2.5>
1046:  66 90            xchg   %ax,%ax

Disassemblamento della sezione .text:
0000000000001050 <_start>:
1050:  31 ed            xor     %ebp,%ebp
1052:  49 89 d1          mov     %rdx,%r9
1055:  5e              pop     %rsi
1056:  48 89 e2          mov     %rsp,%rdx
1059:  48 83 e4 f0      and     $0xfffffffffffffff0,%rsp
105d:  50              push    %rax
105e:  54              push    %rsp
```

Linguaggio macchina

Assembly

Indirizzo di memoria dell'istruzione

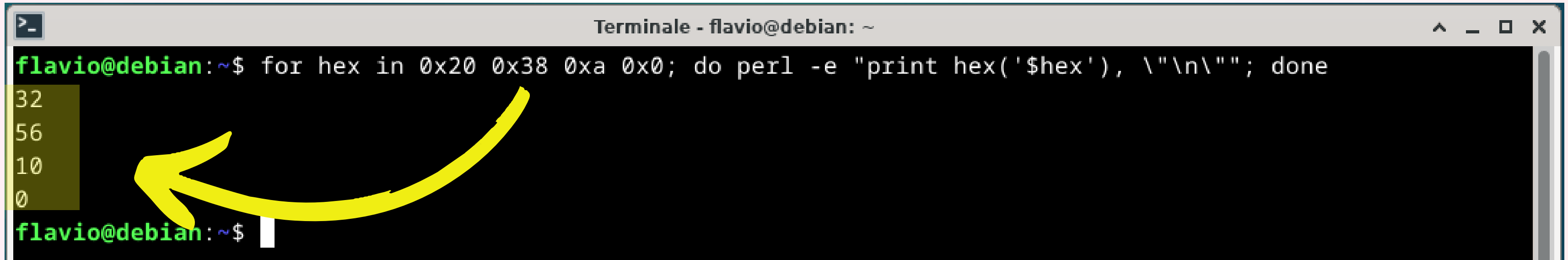
Istruzioni e significato

Tornando a noi, la traccia dell'esercizio ci fornisce un codice composto da istruzioni Assembly per la CPU x86. Ogni riga del codice include un indirizzo di memoria (a sinistra) che mostra la posizione **esatta** dell'istruzione nel programma, seguito dall'**offset**, che indica il punto di **esecuzione relativo**. Ci viene chiesto di ignorare questi ultimi due in quanto non sono stati ancora affrontati nella lezione teorica. L'istruzione stessa, a destra, specifica l'**operazione da eseguire**, come caricare valori nei registri, **sommare** numeri o **confrontare** valori.

Riguardo indirizzo dell'istruzione e offset Ci viene chiesto di ignorare questi ultimi due in quanto non sono stati ancora affrontati nella lezione teorica, ma a me piace ossessionarmi con la pratica per provare a capire ciò che ho davanti, per cui ho sperimentato una semplice situazione sulla mia Debian (architettura x86):

Valori esadecimali

I valori nel codice sono giustamente riportati in **esadecimale**, perchè generalmente, come appreso dalla teoria, non li troviamo in formato decimale. Giusto per completezza, con una semplice calcolatrice o come nel mio caso con un semplice ciclo in bash che ingloba al suo interno una funzione perl apposita per la conversione (hex), possiamo convertire, per semplicità e comprensione, i valori esadecimali della traccia nel loro corrispettivo valore decimale:



```
Terminale - flavio@debian: ~
flavio@debian:~$ for hex in 0x20 0x38 0xa 0x0; do perl -e "print hex('$hex'), \"\n\""; done
32
56
10
0
flavio@debian:~$
```


Registri e codici utilizzati nell'esercizio

EAX

Registro di uso generale. Utilizzato solitamente per operazioni aritmetiche e per restituire valori da funzioni.

EDX

Registro di uso generale. Utilizzato per operazioni aritmetiche e per memorizzare dati temporanei.

EBP

Registro di base del frame. Utilizzato per mantenere il puntatore alla base del frame della funzione corrente, utile per accedere alle variabili locali e agli argomenti della funzione.

MOV

Muove (o carica) un valore in un registro o in una posizione di memoria. Ad esempio, `mov EAX, 0x20` carica il valore 32 nel registro EAX.

ADD

Aggiunge il valore di un registro a un altro. Ad esempio, `add EAX, EDX` somma il valore di EDX a quello di EAX.

CMP

Confronta due valori e imposta i flag di stato per indicare se il primo valore è maggiore, minore o uguale al secondo. Ad esempio, `cmp EBP, 0xA` confronta il valore di EBP con 10.

JGE

Salta a un'altra istruzione se il confronto precedente ha indicato che il primo valore è maggiore o uguale al secondo. Ad esempio, `jge 0x1176` salta all'indirizzo 0x1176 se il valore di EBP è maggiore o uguale a 10.

CALL

Chiama una funzione all'indirizzo specificato. Ad esempio, `call 0x1030` esegue la funzione `printf` all'indirizzo 0x1030.

Analisi codice

mov EAX,0x20 - Mette il valore 32 (0x20 in esadecimale) nel registro EAX.

mov EDX,0x3 - Mette il valore 56 (0x38 in esadecimale) nel registro EDX.

add EAX,EDX - Somma il valore di EDX al valore di EAX. Di conseguenza EAX conterrà 88 (32 + 56).

mov EBP,EAX - Copia il valore di EAX (88) nel registro EBP.

cmp EBP,0xa - Confronta il valore di EBP (88) con 10 (0xa in esadecimale). Destinazione (EBP) > sorgente (0xa), i flag ZF e CF vengono settati a 0.

jge 0x1176 - Di conseguenza questa istruzione salta all'indirizzo di memoria specificato (0x1176). **SE** il valore di EBP è maggiore o uguale a 10. In questo caso, EBP è 88, quindi il salto viene effettuato.

mov eax,0x0 - Imposta EAX a 0. Solo se il salto non è stato effettuato (ovvero EBP era minore di 10).

call 0x1030 - Chiama la funzione printf (all'indirizzo specificato). In questo caso, printf verrà chiamata con EAX (0) come argomento, se il salto non è stato effettuato. La funzione **printf** l'abbiamo conosciuta grazie al linguaggio C nelle lezioni precedenti.



GRAZIE

Flavio Scognamiglio