# LINGUAGGIO ASSEMBLY

**Cos’è?**

Il linguaggio Assembly è il livello più basso di linguaggio leggibile dall'uomo. È anche il livello più alto di linguaggio in cui un binario può essere decompilato in modo affidabile. Nel contesto dell'apprendimento dell'ingegneria inversa del malware, conoscere le basi del linguaggio Assembly è essenziale. Questo perché, quando otteniamo un campione di malware da analizzare, è molto probabile che sia un binario compilato. Non possiamo visualizzare il codice in linguaggio C/C++ o in altri linguaggi perché non è disponibile per noi. Quello che possiamo fare, tuttavia, è decompilare il codice utilizzando un decompiler o un disassembler. Il problema con la decompilazione è che molte informazioni nel codice scritto vengono rimosse durante la compilazione in un binario; quindi, non vedremo nomi di variabili, nomi di funzioni, ecc., come facciamo durante la scrittura del codice. Quindi, il codice più affidabile che abbiamo per un binario compilato è il suo codice assembly.

**A cosa serve?**

Il codice di un programma, come scritto sul disco e compreso dalla CPU, è in formato binario. Questo significa che il codice effettivo è una sequenza di 1 e 0. Per renderlo comprensibile, spesso raggruppiamo una serie di 8 bit (chiamati byte) in un singolo numero in esadecimale. Quindi le istruzioni che un computer sta eseguendo saranno solo una sequenza di numeri casuali in esadecimale per un essere umano. Tra questi numeri casuali ci sono codici operativi e operandi. I codici operativi indicano l'esadecimale per le operazioni effettive, e gli operandi sono i registri o le posizioni di memoria su cui vengono eseguite le operazioni.

**Codici Operativi**

I codici operativi sono numeri che corrispondono alle istruzioni eseguite dalla CPU. Quando usiamo un disassemblatore per disassemblare un programma, legge i codici operativi. Li traduce in istruzioni di assembly per renderli leggibili all'uomo. Ad esempio, l'istruzione per spostare **0x5F** nel registro **eax** è:

**mov eax, 0x5f**

Quando lo guardiamo in un disassemblatore, vedremo:

**040000: b8 5f 00 00 00 mov eax, 0x5f**

Qui, **il 040000**: corrisponde all'indirizzo dove si trova l'istruzione. b8 si riferisce al codice operativo dell'istruzione **mov eax, e 5F 00 00 00 indica l'altro operando 0x5f.** Si noti che a causa della endianness, l'operando 0x5f è scritto come 5f 00 00 00, che in realtà è 00 00 00 5f ma in notazione little-endian. Allo stesso modo, c'è un codice operativo per ogni istruzione nel linguaggio assembly. Ci sono riferimenti per convertire i codici operativi in istruzioni di assembly. Tuttavia, a meno che non stiamo scrivendo un disassemblatore, non ne avremo bisogno, poiché un disassemblatore svolge quel lavoro piuttosto bene. Tuttavia, è utile capire cosa succede sotto il cofano per avere un'immagine più completa.

Abbiamo visto che nell'operazione sopra abbiamo avuto tre parti, **un'istruzione, mov,** **e due operandi, eax e 0x5f.** In questa particolare istruzione, il valore 0x5f viene spostato in eax; tuttavia, possiamo anche avere altri tipi di operandi nel linguaggio assembly.

**Tipi di Operandi**

In generale, ci sono tre tipi di operandi nel linguaggio assembly.

Gli operandi immediati possono anche essere considerati costanti. Questi sono valori fissi come abbiamo avuto il 0x5f nell'esempio sopra.

I registri possono anche essere operandi. L'esempio precedente mostra eax come registro dove viene memorizzato l'operando immediato.

Gli operandi di memoria sono indicati da parentesi quadre e fanno riferimento a posizioni di memoria. Ad esempio, se vediamo [eax] come operando, significherà che il valore in eax è la posizione di memoria su cui deve essere eseguita l'operazione.

**Istruzioni generali**

Le istruzioni dicono alla CPU quale operazione eseguire. Spesso utilizzano operandi da registri, memoria o operandi immediati per eseguire operazioni e quindi memorizzare i risultati sia nei registri che nella memoria. Queste istruzioni eseguono operazioni semplici, come spostare un valore da un tipo di memoria all'altro.

**L'istruzione MOV**

L'istruzione mov sposta un valore da una posizione all'altra. La sua sintassi è la seguente:

mov destinazione, origine

L'istruzione mov può spostare un valore fisso in un registro, un registro in un altro registro e un valore in una posizione di memoria in un registro.

Per esempio, l'istruzione seguente copia un valore fisso in un registro. In questa particolare istruzione, 0x5f viene spostato in eax:

**mov eax, 0x5f**

In questo caso particolare, il valore memorizzato in eax viene spostato in ebx:

**mov ebx, eax**

L'istruzione seguente copia il valore memorizzato in una posizione di memoria in un registro:

**mov eax, [0x5fc53e]**

Come visto sopra, utilizziamo le parentesi quadre quando ci riferiamo alla memoria. Allo stesso modo, se vediamo un registro tra parentesi quadre, ciò significherà che il valore in quel registro verrà trattato come una posizione di memoria, e il valore in quella posizione di memoria verrà spostato nella destinazione. Questo significa che l'esempio **mov eax, [0x5fc3e]** e l'esempio seguente avranno lo stesso risultato.

**mov ebx, 0x5fc53e**

**mov eax, [ebx]**

Possiamo utilizzare l'istruzione mov per eseguire calcoli aritmetici quando ci riferiamo agli indirizzi di memoria. Ad esempio, l'istruzione seguente calcola ebp+4 (aggiungendo un offset di 4 byte all'indirizzo di memoria) e sposta il valore nell'indirizzo di memoria risultante in eax:

**mov eax, [ebp+4]**

**L'istruzione LEA**

L'istruzione lea sta per "carica l'indirizzo effettivo". Il formato di questa istruzione è il seguente:

lea destinazione, origine

Mentre l'istruzione mov sposta i dati dall'indirizzo di origine alla destinazione, l'istruzione lea sposta l'indirizzo dell'origine nella destinazione. Nell'esempio seguente, il valore di ebp sarà aumentato di quattro e spostato in eax. Tuttavia, se avessimo usato un'istruzione mov qui invece di lea, avrebbe spostato il valore nella posizione di memoria ebp+4.

**L'istruzione NOP**

L'istruzione nop sta per no operation. Questa istruzione scambia il valore in eax con sé stesso, senza cambiare nulla. Le istruzioni nop vengono utilizzate per consumare cicli CPU mentre si attende un'operazione o altri scopi. Ha la seguente sintassi:

nop

**Le istruzioni nop vengono utilizzate dagli autori di malware quando si reindirizza l'esecuzione al loro codice**. La posizione esatta in cui si reindirizzerà l'esecuzione è spesso sconosciuta; quindi, l'autore del malware utilizza una serie di istruzioni nop per garantire che l'esecuzione del codice non inizi dal mezzo. Questo padding di istruzioni nop è chiamato una scivolata nop.

**Istruzioni di Shift**

La CPU utilizza le istruzioni di shift per spostare ciascun bit del registro al bit adiacente. Ci sono due istruzioni di shift per spostare a destra o a sinistra. Le istruzioni di shift hanno la seguente sintassi:

shr destinazione, conteggio

shl destinazione, conteggio

Qui l'istruzione shr è per l'operazione di shift a destra e il shl è per l'operazione di shift a sinistra. Questa istruzione sposta i bit nell'operando destinazione. L'operando conteggio decide il numero di bit da spostare. I bit che vengono spostati dalla loro posizione vengono riempiti con zeri. **Ad esempio, se abbiamo 00000010 in eax e lo spostiamo a sinistra, diventerà 00000100**. Il flag di carry (CF) viene utilizzato per aumentare la destinazione, poiché è riempito dall'ultimo bit che trabocca la destinazione.

**Istruzioni di Rotazione**

Le istruzioni di rotazione sono simili alle istruzioni di shift. L'unica differenza è che i bit vengono ruotati verso l'altro estremo del registro anziché spostare il bit in overflow nel flag di carry o aggiungere zeri anziché bit spostati. L'istruzione di rotazione ha la seguente sintassi:

ror destinazione, conteggio

rol destinazione, conteggio

Qui, l'istruzione ror ruota la destinazione verso destra e rol ruota la destinazione verso sinistra. Il resto della sintassi rimane lo stesso. Ad esempio, se abbiamo 10101010 in al, e lo ruotiamo a destra di 1 bit, il risultato sarà 01010101. Analogamente, ruotare questo risultato a sinistra di 1 bit produrrà nuovamente 10101010.

**Flags**

Nel linguaggio assembly x86, la CPU dispone di diversi flag che indicano l'esito di determinate operazioni o condizioni. Questi flag sono bit in un registro speciale noto come registro dei flag o registro EFLAGS. Ogni flag rappresenta una condizione specifica o il risultato dell'ultima operazione aritmetica o logica. Ecco una tabella con i flag più comuni nell'assembly x86 e le loro spiegazioni:

|  |  |  |
| --- | --- | --- |
| **Flag** | **Abbreviazione** | **Spiegazione** |
| Carry | CF | Impostato quando è necessario un carry-out o un prestito dal bit più significativo in un'operazione aritmetica. Utilizzato anche per operazioni di spostamento bit-wise. |
| Parità | PF | Impostato se il byte meno significativo del risultato contiene un numero pari di bit a 1. |
| Ausiliario | AF | Impostato se è necessario un carry-out o un prestito dal bit 3 al bit 4 in un'operazione aritmetica (aritmetica BCD). |
| Zero | ZF | Impostato se il risultato dell'operazione è zero. |
| Segno | SF | Impostato se il risultato dell'operazione è negativo (cioè il bit più significativo è 1). |
| Overflow | OF | Impostato se c'è un overflow aritmetico con segno (ad esempio, aggiungendo due numeri positivi e ottenendo un risultato negativo o viceversa). |
| Direzione | DF | Determina la direzione per le istruzioni di elaborazione delle stringhe. Se DF=0, la stringa viene elaborata in avanti; se DF=1, la stringa viene elaborata all'indietro. |
| Abilitazione Interruzione | IF | Se impostato (1), abilita le interruzioni hardware mascherabili. Se cancellato (0), le interruzioni sono disabilitate. |

I flag possono essere utilizzati in salti condizionali e sono cruciali per implementare il branching condizionale nel codice assembly. Ad esempio, potresti saltare solo a un indirizzo specifico se un determinato flag è impostato o cancellato.

**Aritmetica e Istruzioni logiche**

Le Operazioni Aritmetiche vengono eseguite da una CPU utilizzando istruzioni aritmetiche.

**Istruzioni di Addizione e Sottrazione**

La sintassi per l'istruzione di addizione è la seguente. Il valore viene aggiunto alla destinazione, e il risultato viene memorizzato nella destinazione.

**add** destinazione, valore

L'istruzione di sottrazione segue una sintassi simile. Nella sintassi seguente, il valore viene sottratto dalla destinazione, e il risultato viene memorizzato nella destinazione.

**sub** destinazione, valore

Nei precedenti esempi, il valore può essere un valore costante fisso o un registro. Per l'operazione di sottrazione, il Flag Zero (ZF) viene impostato se il risultato della sottrazione è zero. Se la destinazione è più piccola del valore sottratto, allora il Carry Flag (CF) viene impostato.

**Istruzioni di Moltiplicazione e Divisione**

Le operazioni di moltiplicazione e divisione utilizzano i registri eax e edx. Pertanto, dovremo guardare all'ultima istruzione che ha manipolato questi registri per ogni operazione di moltiplicazione e divisione.

L'istruzione di moltiplicazione ha il seguente formato. Moltiplica il valore con eax e memorizza il risultato in **edx:eax** come valore a 64 bit. Qui sono richiesti due registri perché il risultato della moltiplicazione di 2 valori a 32 bit può spesso essere superiore a 32 bit. I 32 bit inferiori del risultato vengono memorizzati nel registro eax, e i 32 bit superiori vengono memorizzati nel registro edx.

mul valore

Il valore può essere un altro registro o una costante come operando immediato.

Per l'istruzione di divisione, il caso è opposto. Divide il valore a **64 bit in edx:eax** e salva il risultato in eax e il resto in edx.

div valore

**Istruzioni di Incremento e Decremento**

Come suggerisce il nome, le istruzioni di incremento e decremento incrementano o decrementano il registro dell'operando di 1. La sintassi per incrementare eax di 1 è la seguente:

**inc eax**

Allo stesso modo, la sintassi per decrementare eax di 1 utilizzando l'istruzione di decremento è la seguente:

**dec eax**

**Istruzioni Logiche**

Le istruzioni logiche vengono utilizzate per eseguire operazioni logiche. Esamineremo alcune operazioni logiche comuni eseguite dalla CPU.

**Istruzione AND**

L'istruzione AND esegue un'operazione AND bit a bit sugli operandi. Un'operazione AND restituisce un output di 1 quando entrambi gli input sono 1; altrimenti restituisce 0. Un esempio di istruzione è il seguente:

**and al, 0x7c**

In questo esempio, 0x7c si converte in 01111100 in binario. Se al avesse un valore di 0xfc, che è 11111100 in binario, l'output dell'istruzione sopra sarà 01111100. Tuttavia, se al ha un valore di 0x8c, 10001100 in binario, il risultato dell'istruzione sopra sarà 00001100 o 0xc.

**Istruzione OR**

L'istruzione OR esegue un'operazione OR bit a bit. Un'operazione OR restituisce 1 se almeno uno degli operandi è 1. Restituisce 0 se nessuno degli operandi è 1. Un esempio di istruzione è il seguente:

**or al, 0x7c**

In questo esempio, se al avesse un valore di 0xfc o 11111100 in binario, l'output dell'istruzione sopra sarà 11111100. Allo stesso modo, se al ha un valore di 0x8c o 10001100 in binario, il risultato sarà ancora 11111100 in binario o 0xfc.

**Istruzione NOT**

L'istruzione NOT prende un operando. Semplicemente inverte i bit dell'operando, sostituendo 1 con 0 e viceversa. Nell'esempio seguente, se al avesse un valore di 11110000, il risultato sarà 00001111.

**not al**

**Istruzione XOR**

L'operazione XOR restituisce 1 se entrambi gli input sono opposti. Restituisce 0 quando entrambi gli input sono uguali. Questa operazione viene eseguita dall'istruzione XOR nel linguaggio assembly, che esegue un'operazione XOR bit a bit sugli operandi. Ha la seguente sintassi.

**xor al, 0x7c**

Se al ha un valore di 0xfc, che è 11111100, il risultato di questa istruzione sarà 10000000 o 0x80. Allo stesso modo, se al ha un valore di 0x8c, che è 10001100, il risultato di questa istruzione sarà 11110000 o 0xf0. Se al ha un valore di 0x7c, il risultato sarà 0x00. Questo mostra che XORing di un registro con se stesso dà come risultato 0. Pertanto, l'istruzione XOR viene spesso utilizzata per azzerare un registro, il che è più ottimizzato rispetto a un'istruzione MOV.

**Condizionali**

Spesso una CPU deve determinare se due valori sono uguali, maggiori o minori l'uno dell'altro. Per eseguire tali operazioni, la CPU utilizza alcune istruzioni condizionali.

**L'istruzione TEST**

L'istruzione test esegue un'operazione AND bit a bit, e invece di memorizzare il risultato nell'operando di destinazione come fa l'istruzione AND, imposta il Flag Zero (ZF) se il risultato è 0. Questa istruzione viene spesso utilizzata per verificare se un operando ha un valore NULL, ad esempio, testando l'operando contro se stesso. Questo viene fatto perché utilizzare l'istruzione test richiede meno byte rispetto al confronto con 0. L'istruzione test ha la seguente sintassi:

**test destinazione, origine**

**L'istruzione CMP**

In base al risultato, l'istruzione CMP confronta i due operandi e imposta il Flag Zero (ZF) o il Carry Flag (CF). Ha la seguente sintassi:

**cmp destinazione, origine**

L'istruzione di confronto funziona in modo simile a un'istruzione di sottrazione. L’unica differenza è che gli operandi non vengono modificati. Il Flag Zero (ZF) viene impostato se entrambi gli operandi sono uguali. Se l'operando origine è maggiore dell'operando destinazione, il Carry Flag (CF) viene impostato. Il ZF e il CF vengono cancellati se l'operando destinazione è maggiore dell'operando origine.

**Branching**

Quando non c'è branching, l'Instruction Pointer passa da un'istruzione all'altra nell'ordine in cui sono posizionate in memoria. Il flusso di controllo rimane in una linea retta a meno che non ci sia un'operazione di branching. Le operazioni di branching cambiano il valore dell'Instruction Pointer e modificano il flusso di controllo del programma da lineare a ramificato.

**L'istruzione JMP**

L'istruzione JMP fa saltare il flusso di controllo a una posizione specificata. Ha la seguente sintassi:

**jmp posizione**

Qui, l'operando posizione verrà spostato nell'Instruction Pointer, rendendolo l'indirizzo da cui verrà recuperata la prossima istruzione per l'esecuzione.

**Salti Condizionali**

Spesso, il codice richiede di spostarsi se viene soddisfatta una specifica condizione. Nei linguaggi di programmazione di livello superiore, ci sono condizioni if che aiutano a soddisfare questo requisito. Tuttavia, non c'è alcuna istruzione if nel linguaggio assembly. Questo requisito viene soddisfatto utilizzando salti condizionali.

I salti condizionali decidono se saltare in base al valore dei Flag Registers. La loro sintassi è simile all'istruzione di salto.

|  |  |
| --- | --- |
| **Istruzione** | **Spiegazione** |
| **jz** | Salta se il ZF è impostato (ZF=1). |
| **jnz** | Salta se il ZF non è impostato (ZF=0). |
| **je** | Salta se uguale. Spesso usato dopo un'istruzione CMP. |
| **jne** | Salta se diverso. Spesso usato dopo un'istruzione CMP. |
| **jg** | Salta se la destinazione è maggiore dell'operando origine. Esegue un confronto con segno ed è spesso usato dopo un'istruzione CMP. |
| **jl** | Salta se la destinazione è minore dell'operando origine. Esegue un confronto con segno ed è spesso usato dopo un'istruzione CMP. |
| **jge** | Salta se maggiore o uguale a. Salta se l'operando destinazione è maggiore o uguale all'operando origine. Simile alle istruzioni precedenti. |
| **jle** | Salta se minore o uguale a. Salta se l'operando destinazione è minore o uguale all’operando origine. Simile alle istruzioni precedenti. |
| **ja** | Salta se sopra. Simile a jg, ma esegue un confronto senza segno. |
| **jb** | Salta |

**Lo Stack**

Nella stanza Panoramica dell'Architettura x86, abbiamo appreso dello stack e della sua importanza. Abbiamo anche appreso alcuni dei registri utilizzati per fare riferimento alla posizione dello stack in memoria. Lo stack è una memoria di tipo Last In, First Out (LIFO). Ciò significa che l'ultima variabile inserita nello stack è la prima a essere estratta. Queste operazioni di push e pop vengono eseguite seguendo le istruzioni nel linguaggio assembly.

**L'istruzione PUSH**

L'istruzione push ha la seguente sintassi:

**push origine**

Come accennato in precedenza, l'istruzione push inserirà l'operando origine nello stack. Il valore dell'operando viene memorizzato nella posizione di memoria indicata dal puntatore dello stack (ESP), diventando efficacemente il nuovo elemento in cima allo stack. Il puntatore dello stack viene quindi aggiornato (decrementato) per riflettere la posizione aggiornata in cima allo stack. Le seguenti istruzioni pushano anche tutti i registri a scopo generale nello stack.

**pusha (pusha tutte le parole): Inserisce tutti i registri a scopo generale a 16 bit nello stack, cioè AX, BX, CX, DX, SI, DI, SP, BP**

**pushad (pusha tutti i double word): Inserisce tutti i registri a scopo generale a 32 bit nello stack, cioè EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP**

Quando ci imbattiamo in queste istruzioni, è spesso un segno di qualcuno che inietta manualmente istruzioni assembly per salvare lo stato dei registri, come spesso accade con il shellcode.

**L'istruzione POP**

L'istruzione pop ha la seguente sintassi:

**pop destinazione**

L'istruzione pop recupera il valore in cima allo stack e lo memorizza nell'operando destinazione. Di conseguenza, il puntatore dello stack (ESP) viene incrementato per riflettere l'aggiustamento fatto dopo aver estratto il valore. Le seguenti istruzioni popano anche tutti i registri a scopo generale dallo stack.

**popa (popa tutte le parole): Estrae i valori sequenzialmente dalla cima dello stack ai registri a scopo generale nell'ordine seguente: DI, SI, BP, BX, DX, CX, AX. Lo SP o ESP viene aggiustato per riflettere la nuova posizione in cima allo stack.**

**popad (popa tutti i double word): Estrae i valori sequenzialmente dalla cima dello stack ai registri a scopo generale nell'ordine seguente: EDI, ESI, EBP, EBX, EDX, ECX, EAX. Lo SP o ESP viene aggiustato per riflettere la nuova posizione in cima allo stack.**

**L'istruzione CALL**

L'istruzione call viene utilizzata nel linguaggio assembly per eseguire un'operazione di chiamata di funzione per svolgere un compito specifico. Ha la seguente sintassi:

**call posizione**

A seconda della convenzione di chiamata, gli argomenti vengono posizionati nello stack o nei registri in una chiamata di funzione. Il prologo della funzione prepara lo stack aggiustando l'EBP e l'ESP e pushando l'indirizzo di ritorno nello stack. Allo stesso modo, quando la funzione restituisce, l'epilogo ripristina lo stack per la funzione chiamante. Apprenderemo di più sulle convenzioni di chiamata, sul prologo e sull'epilogo nelle stanze successive.

**ESERCIZIO**

**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>**

**mov EAX,0x20**: Questa istruzione sposta il valore esadecimale 0x20 (32 in decimale) nel registro EAX. Il registro EAX è comunemente utilizzato per le operazioni aritmetiche e per contenere valori temporanei.

**mov EDX,0x38**: Questa istruzione sposta il valore esadecimale 0x38 (56 in decimale) nel registro EDX. Il registro EDX è spesso utilizzato come registro generale per le operazioni aritmetiche.

**add EAX,EDX**: Questa istruzione aggiunge il contenuto del registro EDX al registro EAX e memorizza il risultato in EAX. Quindi, dopo questa istruzione, il registro EAX conterrà la somma dei valori precedentemente caricati.

**mov EBP,EAX**: Questa istruzione sposta il contenuto del registro EAX nel registro EBP. Il registro EBP viene spesso utilizzato come base del puntatore per accedere ai dati nello stack della memoria.

**cmp EBP,0xa**: Questa istruzione confronta il contenuto del registro EBP con il valore esadecimale 0xa (10 in decimale). Il confronto viene eseguito sottraendo il valore in EBP da 10 e impostando i flag del processore in base al risultato.

**jge 0x1176 <main+61>**: Questa istruzione salta a 0x1176 (linea di codice successiva) se il risultato del confronto precedente è maggiore o uguale a zero (indicato dal flag di segno). Ciò significa che se il valore in EBP è maggiore o uguale a 10, verrà eseguita la linea di codice successiva.

**mov EAX,0x0**: Questa istruzione sposta il valore esadecimale 0x0 (0 in decimale) nel registro EAX. Il registro EAX spesso contiene il valore di ritorno per le funzioni.

**call 0x1030 <printf@plt>**: Questa istruzione chiama la funzione printf dalla libreria standard di C, il cui indirizzo in memoria è 0x1030. La funzione printf è comunemente utilizzata per stampare dati sullo standard output.