# Cap 2 - Parte 1 - Assembly

## Introduzione all'Assembly - Introduction to Assembly Language

⚜️ **Tag:** #assembly #bassoLivello #registri #programmazione

L'assembly è un linguaggio di programmazione di basso livello strettamente legato al codice macchina, utilizzato per interagire direttamente con l'hardware del computer. Ogni processore ha un proprio set di istruzioni assembly, come l'architettura x86 o ARM. L'assembly consente ai programmatori di accedere e manipolare i registri, gestire direttamente la memoria e controllare il flusso di esecuzione, risultando fondamentale in ambiti come la programmazione embedded, l'ottimizzazione del codice e l'analisi del malware.

---

## Struttura del Codice Assembly - Structure of Assembly Code

⚜️ **Tag:** #struttura #codice #istruzioni

Un programma in assembly è composto da istruzioni semplici che eseguono operazioni specifiche, come caricare valori nei registri o sommare due numeri. Ogni istruzione corrisponde a un'operazione eseguibile dal processore. Ecco alcuni elementi principali di un programma assembly.

### 1. Istruzioni di Base

Le istruzioni assembly sono brevi comandi mnemonici che rappresentano operazioni, come `MOV`, `ADD`, `SUB`, e `JMP`.

- **MOV**: Sposta i dati da un registro a un altro o tra un registro e una locazione di memoria.
- **ADD**: Somma due valori e memorizza il risultato in un registro.
- **SUB**: Sottrae un valore da un altro.
- **JMP**: Salta a un'altra parte del codice, utile per il controllo del flusso.

## 2. Registri

I registri sono piccole unità di memoria ultra-veloci integrate nel processore. I registri più comuni nell'architettura x86 includono:

- **EAX, EBX, ECX, EDX**: Registri generali usati per operazioni aritmetiche e manipolazione dati.
- **ESP, EBP**: Registri di stack, utilizzati per gestire il flusso delle funzioni e l'allocazione di memoria temporanea.
- **EIP**: Puntatore di istruzione, che indica la prossima istruzione da eseguire.

## Esempio di Codice Assembly di Base (x86):

Il seguente codice somma due numeri e memorizza il risultato.

```
section .data
    num1 db 5          ; Dichiarazione della variabile num1
con valore 5
    num2 db 10         ; Dichiarazione della variabile num2
con valore 10

section .text
global _start
_start:
    mov al, [num1]     ; Carica il valore di num1 nel
registro AL
    add al, [num2]     ; Somma il valore di num2 a quello in
AL
    mov [result], al   ; Memorizza il risultato nella
variabile result
```

```
    ; Exit
    mov eax, 1
    int 0x80
```

---

## Operazioni Aritmetiche e Logiche - Arithmetic and Logical Operations

⚜️ **Tag:** #aritmetica #logica #operazioni

L'assembly offre operazioni base per l'aritmetica e la logica, che consentono di manipolare i dati direttamente nei registri.

### Operazioni Aritmetiche

- **ADD**: Somma due operandi.
- **SUB**: Sottrae un operando da un altro.
- **MUL e DIV**: Moltiplicazione e divisione di numeri interi.

### Operazioni Logiche

- **AND, OR, XOR**: Operazioni bitwise che applicano maschere e trasformano i dati a livello di bit.
- **NOT**: Inverte tutti i bit dell'operando.

### Esempio di Operazione Logica XOR

L'operazione XOR è spesso utilizzata per cifrare e decifrare dati.

```
section .text
global _start
_start:
    mov al, 0xAA    ; Carica 0xAA nel registro AL
    xor al, 0xFF    ; Applica XOR con 0xFF (inverti i bit)
    ; AL ora contiene 0x55
```

# Gestione della Memoria e dello Stack - Memory and Stack Management

⚜️ **Tag:** #memoria #stack #gestioneMemoria

In assembly, la gestione della memoria è cruciale, poiché si ha accesso diretto allo stack, una struttura dati LIFO utilizzata per memorizzare indirizzi di ritorno e variabili temporanee durante l'esecuzione di funzioni.

## Comandi di Stack

- **PUSH**: Inserisce un valore nello stack, decrementando ESP.
- **POP**: Rimuove un valore dallo stack, incrementando ESP.
- **CALL**: Salva l'indirizzo di ritorno nello stack e salta alla funzione chiamata.
- **RET**: Estrae l'indirizzo di ritorno dallo stack e lo carica nel registro EIP.

## Esempio di Uso dello Stack

Ecco un esempio di come si utilizza lo stack per gestire una chiamata di funzione:

```
section .text
global _start
_start:
    mov eax, 5        ; Carica 5 in EAX
    push eax          ; Inserisce EAX nello stack
    call funzione     ; Chiama la funzione

funzione:
    pop ebx           ; Estrae il valore dallostack in EBX
    add ebx, 2        ; Somma 2 al valore
    ret               ; Ritorna alla chiamata precedente
```

# Esempio di Programma Assembly Completo - Complete Assembly Program Example

⚜️ **Tag:** #programmaCompleto #esempio

Di seguito è riportato un semplice programma che stampa un messaggio su schermo utilizzando una syscall in ambiente Linux. La syscall `write` viene usata per scrivere una stringa specifica sullo standard output.

```
section .data
    msg db 'Hello, Assembly!', 0xA  ; Messaggio e newline
    len equ $ - msg                 ; Calcola la lunghezza
del messaggio

section .text
global _start
_start:
    mov eax, 4          ; syscall numero 4 (write)
    mov ebx, 1          ; file descriptor 1 (stdout)
    mov ecx, msg        ; indirizzo del messaggio
    mov edx, len        ; lunghezza del messaggio
    int 0x80            ; chiamata a sistema

    mov eax, 1          ; syscall numero 1 (exit)
    xor ebx, ebx        ; codice di uscita 0
    int 0x80            ; chiamata a sistema
```

**Note sul Codice**:

- `mov eax, 4`: Imposta il codice syscall `write`.
- `mov ebx, 1`: Specifica `stdout` come file descriptor.
- `int 0x80`: Invia la chiamata a sistema, utilizzata su architetture Linux x86.

# Strumenti per la Programmazione e Debugging in Assembly - Tools for Programming and Debugging in Assembly

⚜️ **Tag:** #strumenti #debugging #programmazione

L'assembly richiede strumenti specifici per scrivere, assemblare e fare il debugging del codice.

- **Assembler**: NASM (Netwide Assembler) e MASM sono tra gli assembler più usati per assemblare il codice assembly in un eseguibile.
- **Debugger**: GDB (GNU Debugger) permette di tracciare l'esecuzione del codice, ispezionare i registri e monitorare lo stack.
- **Disassemblatori**: IDA Pro, Ghidra e Radare2 permettono di convertire il codice macchina binario in assembly, fondamentale per il *reverse engineering*.

## Esempio di Uso di NASM e GDB:

```
# Assembla e linka il programma
nasm -f elf64 esempio.asm -o esempio.o
ld esempio.o -o esempio

# Esegui con GDB
gdb ./esempio
```

---

🔑 **Chiavi:**
[assembly, registri, stack, NASM, GDB, mov, syscall, memoria, programmazione a basso livello]

---

## Suggerimenti per Approfondimenti - Suggestions for Further Study

- **Sistemi Embedded e Assembly**: Scopri l’uso dell’assembly in dispositivi embedded e come gestire hardware specifico.
- **Ottimizzazione delle Prestazioni**: Esplora le tecniche di ottimizzazione dell'assembly per migliorare le prestazioni di applicazioni critiche.
- **Exploit e Sicurezza**: Approfondisci l’uso dell’assembly per comprendere exploit di buffer overflow e altre vulnerabilità di sicurezza informatica.

---

# Approfondimento: Exploit e Sicurezza in Assembly

---

## Introduzione agli Exploit in Assembly - Introduction to Exploits in Assembly

⚜️ **Tag:** #exploit #assembly #sicurezza #bufferOverflow

Gli exploit in assembly rappresentano tecniche di attacco che sfruttano vulnerabilità a livello di codice o memoria, come buffer overflow, manipolazione di puntatori e esecuzione arbitraria di codice. Grazie all’accesso diretto alla memoria e ai registri, l’assembly è spesso usato per studiare e scrivere exploit a basso livello, particolarmente nei contesti di sicurezza informatica e *reverse engineering*. La comprensione di questi exploit è cruciale per proteggere sistemi e applicazioni, nonché per sviluppare tecniche di mitigazione.

---

# Exploit di Buffer Overflow - Buffer Overflow Exploits

⚜️ **Tag:** #bufferOverflow #vulnerabilità #overflow

Un buffer overflow si verifica quando dati in eccesso sono scritti in un buffer, sovrascrivendo la memoria adiacente. Questo può portare a eseguire codice arbitrario se la memoria sovrascritta include puntatori di ritorno o altre variabili di controllo.

## Principi Base di un Buffer Overflow

1. **Individuazione del Buffer**: L'attaccante riempie il buffer oltre il suo limite.
2. **Sovrascrittura dell'EIP (Extended Instruction Pointer)**: Manipolando l'indirizzo di ritorno sullo stack, l'attaccante può indirizzare il programma verso il proprio codice.
3. **Inserimento di uno Shellcode**: Un piccolo pezzo di codice scritto in assembly che, quando eseguito, fornisce accesso all'attaccante.

## Esempio di Exploit di Buffer Overflow in C e Assembly

Il seguente esempio mostra un overflow buffer su un array in C e il modo in cui è possibile sfruttarlo con assembly per eseguire uno shellcode.

**Codice Vulnerabile in C**:

```
#include <stdio.h>
#include <string.h>

void funzione_vulnerabile(char *input) {
    char buffer[10];
    strcpy(buffer, input);  // Vulnerabilità: non controlla
la lunghezza dell'input
}
```

```c
int main() {
    char input[50];
    printf("Inserisci il payload: ");
    gets(input);  // gets è vulnerabile a overflow
    funzione_vulnerabile(input);
    return 0;
}
```

**Shellcode in Assembly** (Esempio per ottenere una shell su Linux):

```
section .text
global _start
_start:
    xor eax, eax                ; pulisce EAX
    push eax                    ; termina la stringa con
NULL
    push 0x68732f2f             ; '//sh'
    push 0x6e69622f             ; '/bin'
    mov ebx, esp                ; puntatore a '/bin//sh'
    push eax                    ; argomenti NULL
    push ebx
    mov ecx, esp                ; puntatore agli argomenti
    mov al, 0xb                 ; syscall execve
    int 0x80                    ; esegue la syscall
```

**Nota**: Il codice sopra è un esempio di shellcode in Linux e potrebbe essere rilevato come potenzialmente dannoso da strumenti di sicurezza.

### Esecuzione dell'Exploit:

L'attaccante costruisce un payload che riempie il buffer, sovrascrive l'EIP con l'indirizzo del proprio shellcode e riesce a eseguire comandi arbitrari.

---

## Tecniche di Manipolazione del Flusso di Esecuzione - Manipulating Control Flow

⚜️ **Tag:** #flussoDiEsecuzione #saltoCondizionale #ROP

L'assembly permette di manipolare direttamente il flusso di esecuzione tramite istruzioni condizionali e salti, cruciali per la realizzazione di exploit complessi.

## 1. Salti Condizionali

Le istruzioni di salto come `JMP`, `JE`, `JNE`, `JG`, `JL` permettono di controllare la direzione del flusso di esecuzione. Manipolare queste istruzioni consente di saltare a specifiche sezioni di codice.

## 2. Return-Oriented Programming (ROP)

Il *Return-Oriented Programming* sfrutta pezzi di codice esistente nel programma (detti *gadget*), combinandoli per creare una catena di operazioni che simula un payload senza bisogno di uno shellcode personalizzato. Ogni gadget termina con un'istruzione `RET`, che reindirizza al gadget successivo.

**Esempio di ROP**:
Supponiamo di voler eseguire una sequenza di operazioni sfruttando gadget esistenti.

```
Gadget 1: mov eax, [esp+4]; ret
Gadget 2: add eax, ebx; ret
Gadget 3: push eax; ret
```

Questi gadget, concatenati, possono creare una sequenza di operazioni senza inserire nuovo codice, eludendo molte protezioni contro buffer overflow.

---

# Tecniche di Evasione e Anti-Exploit - Evasion and Anti-Exploit Techniques

⚜️ **Tag:** #evasione #bypassSicurezza #protezione

Gli attaccanti adottano tecniche per evadere le protezioni moderne come ASLR, DEP, e stack canaries. Allo stesso modo, i difensori implementano patch e mitigazioni per prevenire tali exploit.

### 1. Bypass di ASLR (Address Space Layout Randomization)

ASLR randomizza gli indirizzi di memoria, complicando il processo di sfruttamento. Tuttavia, gli attaccanti possono utilizzare *leak di memoria* per individuare gli indirizzi del processo in esecuzione.

### 2. Evasione di DEP (Data Execution Prevention)

DEP impedisce l'esecuzione di codice in regioni designate solo per dati. Gli attaccanti possono bypassare DEP utilizzando ROP, che sfrutta codice legittimo esistente.

---

## Protezioni Moderne contro Exploit - Modern Protections Against Exploits

⚜️ **Tag:** #mitigazioni #protezione #stackCanary #controlFlowGuard

Per contrastare gli exploit, i sistemi operativi e i compilatori moderni includono protezioni che rendono più difficile eseguire attacchi con successo.

### 1. Stack Canaries

Un valore (canarino) è inserito nello stack prima del puntatore di ritorno. Se viene alterato, indica una sovrascrittura, e il programma termina l'esecuzione.

### 2. Control Flow Guard (CFG)

CFG protegge il flusso di esecuzione limitando le istruzioni di salto a indirizzi validi, prevenendo gli attacchi basati su puntatori di ritorno manipolati.

### 3. SEH Overwrite Protection

Su Windows, la protezione *Structured Exception Handler (SEH)* impedisce la manipolazione degli handler di eccezioni per controllare il flusso del programma, proteggendo da alcuni tipi di buffer overflow.

---

## Esempio Completo: Exploit e Mitigazioni

⚜️ **Tag:** #casoStudio #exploit #protezione

Vediamo un caso studio che combina un buffer overflow con tecniche di mitigazione come stack canaries e ASLR.

### Codice Vulnerabile:

```
#include <stdio.h>
#include <string.h>

void funzione_vulnerabile(char *input) {
    char buffer[16];
    strcpy(buffer, input); // buffer overflow potenziale
}

int main() {
    char input[128];
    printf("Inserisci input: ");
    fgets(input, sizeof(input), stdin);
    funzione_vulnerabile(input);
    return 0;
}
```

## Exploit senza Protezioni

L’attaccante riempie il buffer e sovrascrive l’EIP con un indirizzo di shellcode. Senza stack canaries, ASLR o DEP, questo exploit può eseguire codice arbitrario.

## Con Stack Canaries

Con i canarini abilitati, una sovrascrittura del buffer modifica il valore del canarino, causando un errore di sicurezza e bloccando l’exploit.

**Attivazione dei Canarini in GCC**:

```
gcc -fstack-protector-all -o programma_sicuro programma.c
```

## Con ASLR e DEP

Con ASLR, gli indirizzi di memoria del buffer e dello stack cambiano a ogni esecuzione, rendendo difficile trovare l’indirizzo di destinazione. Con DEP attivo, l'esecuzione dello shellcode nel buffer dati fallisce.

---

🔑 **Chiavi:**
[exploit, buffer overflow, ROP, sicurezza, mitigazioni, DEP, ASLR, stack canary, reverse engineering, assembly]

---

# Suggerimenti per Approfondimenti - Suggestions for Further Study

- **Strumenti di Analisi ROP**: Studia strumenti come *ROPgadget* o *pwntools* per identificare gadget nei binari e creare catene ROP.
- **Sviluppo di Exploit Sicuri in Ambienti Isolati**: Esegui exploit in ambienti virtualizzati

per studiare gli effetti e l'efficacia delle protezioni come DEP e ASLR.

- **Analisi di Exploit su Sistemi Embedded**: Analizza l'efficacia delle mitigazioni nei dispositivi embedded, dove le protezioni possono essere limitate.

---

# Approfondimento: Strategie e Ottimizzazione per Scrivere Buon Codice Assembly

---

## Introduzione all'Ottimizzazione del Codice Assembly - Introduction to Assembly Code Optimization

⚜️ **Tag:** #assembly #ottimizzazione #prestazioni #efficienza

Scrivere codice assembly efficiente richiede attenzione a dettagli di basso livello, ottimizzando ogni singola istruzione per ottenere le migliori prestazioni. L'assembly permette di interagire direttamente con l'hardware e i registri, consentendo un controllo fine del flusso di esecuzione, dell'uso della memoria e dell'efficienza delle operazioni aritmetiche e logiche. In ambiti come la programmazione embedded, l'ottimizzazione del codice assembly è essenziale per ridurre l'uso della memoria e aumentare la velocità.

---

## Strategie di Ottimizzazione di Base - Basic Optimization Strategies

⚜️ **Tag:** #strategieBase #usoRegistri #caricoEfficiente

L'ottimizzazione di base del codice assembly si concentra sull'uso efficiente dei registri, la minimizzazione delle operazioni e il miglioramento del flusso di esecuzione.

## 1. Utilizzo Efficiente dei Registri

I registri sono più veloci della memoria; pertanto, massimizzare l'uso dei registri per conservare variabili temporanee e risultati intermedi riduce le operazioni di memoria.

- **Mantenere i Dati nei Registri**: Carica i dati nei registri una sola volta e utilizzali finché necessario per evitare accessi ripetuti alla memoria.
- **Evita il Salvataggio Ridondante**: Evita di spostare i dati nei registri senza necessità, per esempio riutilizzando lo stesso registro per operazioni successive.

## 2. Minimizzare il Carico di Memoria

Le operazioni di accesso alla memoria sono relativamente lente rispetto a quelle sui registri. Ridurre il numero di caricamenti e memorizzazioni dirette migliora significativamente le prestazioni.

**Esempio**:

```
mov eax, [num1]         ; Carica num1 in EAX
add eax, [num2]         ; Somma num2 a EAX
mov [result], eax       ; Salva il risultato in memoria
```

Questo esempio evita operazioni di caricamento e memorizzazione non necessarie usando `EAX` per entrambe le operazioni.

---

# Tecniche di Ottimizzazione Intermedia - Intermediate Optimization Techniques

⚜️ **Tag:** #loopUnrolling #calcoloPrecedente #evitaIstruzioniLente

Per migliorare ulteriormente il codice assembly, si possono applicare tecniche come l'ottimizzazione dei cicli, la pre-calcolazione e la sostituzione delle istruzioni lente.

## 1. Loop Unrolling (Srotolamento dei Cicli)

Il loop unrolling riduce il numero di salti condizionali nei cicli replicando le istruzioni. Questo approccio è utile per cicli con un numero ridotto di iterazioni fisse.

**Esempio**:
Supponiamo di voler sommare un array di quattro elementi:

```
; Prima versione con ciclo
mov ecx, 4              ; Numero di iterazioni
mov esi, array
xor eax, eax            ; Azzera EAX per il totale

ciclo:
    add eax, [esi]      ; Somma elemento all'accumulatore
    add esi, 4          ; Sposta al prossimo elemento
loop ciclo

; Versione srotolata
mov esi, array
xor eax, eax            ; Azzera EAX
add eax, [esi]          ; Somma array[0]
add eax, [esi+4]        ; Somma array[1]
add eax, [esi+8]        ; Somma array[2]
add eax, [esi+12]       ; Somma array[3]
```

## 2. Pre-calcolo di Espressioni

Calcola in anticipo le espressioni costanti o semi-costanti, riducendo il numero di calcoli runtime.

**Esempio**:
Invece di calcolare `x * 2` in un ciclo, utilizza uno shift bitwise:

```
shl eax, 1       ; Efficiente, equivalente a moltiplicare
per 2
```

### 3. Evitare Istruzioni Lente

Alcune istruzioni, come `DIV` e `MUL`, sono relativamente lente. Dove possibile, sostituisci la divisione e la moltiplicazione con operazioni di shift o addizione.

---

## Tecniche di Ottimizzazione Avanzata - Advanced Optimization Techniques

⚜️ **Tag:** #saltoCondizionale #evitaBranching #pipelinizzazione

Ottimizzare a livello avanzato richiede considerare il flusso di esecuzione condizionale e la pipelinizzazione, evitando salti condizionali inefficaci e sfruttando il parallelismo del processore.

### 1. Minimizzare il Branching (Salti Condizionali)

I salti condizionali interrompono il flusso del processore, causando uno *stalling* della pipeline. Riduci l'uso di salti condizionali e sostituiscili con operazioni di mascheramento dove possibile.

**Esempio**:
Supponiamo di voler impostare `eax` a 1 se `ebx` è zero, altrimenti impostarlo a zero. Un approccio senza salti:

```
mov eax, ebx        ; Copia ebx in eax
neg eax             ; Se ebx è zero, eax rimane zero;
altrimenti diventa -ebx
sbb eax, eax        ; Imposta eax a 1 se ebx è zero,
altrimenti a zero
```

## 2. Ottimizzazione della Pipeline

Assicurati che le istruzioni siano organizzate per sfruttare al meglio le pipeline del processore. Alterna tipi di istruzioni (come carichi e operazioni logiche) per evitare conflitti.

## 3. Inline Assembly nelle Funzioni Critiche

Se stai lavorando in un linguaggio di alto livello, come C o C++, inserire codice assembly inline in funzioni critiche può migliorare le prestazioni. Ottimizza piccole porzioni di codice, come funzioni matematiche o manipolazioni di stringhe, direttamente con assembly.

**Esempio di Assembly Inline in C**:

```
int moltiplica_per_due(int x) {
    int risultato;
    __asm__ ("shl %1, 1; mov %1, %0" : "=r" (risultato) :
"r" (x));
    return risultato;
}
```

---

# Caso Studio di Ottimizzazione in Assembly

⚜️ **Tag:** #casoStudio #ottimizzazione #assembly

Vediamo un esempio di ottimizzazione di una funzione per sommare numeri in un array, prima in modo non ottimizzato, poi applicando le tecniche sopra descritte.

## Codice Non Ottimizzato

```
section .data
    array db 1, 2, 3, 4, 5

section .text
global _start
_start:
    mov ecx, 5           ; Numero di elementi
    mov esi, array       ; Puntatore all'array
    xor eax, eax         ; Risultato inizia a zero

ciclo:
    add eax, [esi]       ; Somma elemento
    add esi, 1           ; Avanza al prossimo elemento
    loop ciclo           ; Decrementa ecx, salta a ciclo se
non zero
```

## Codice Ottimizzato

Applichiamo loop unrolling, uso efficiente dei registri e minimizzazione dei salti:

```
section .data
    array db 1, 2, 3, 4, 5

section .text
global _start
_start:
    mov esi, array       ; Puntatore all'array
    xor eax, eax         ; Azzera EAX per accumulare la
somma

    ; Srotolamento del ciclo
    add eax, [esi]       ; Somma array[0]
    add eax, [esi+1]     ; Somma array[1]
    add eax, [esi+2]     ; Somma array[2]
    add eax, [esi+3]     ; Somma array[3]
```

```
    add eax, [esi+4]     ; Somma array[4]

    ; Exit
    mov ebx, eax         ; Metti il risultato in EBX per
verifica
    mov eax, 1           ; syscall exit
    int 0x80             ; Esegui la chiamata di sistema
```

Con queste ottimizzazioni, la somma dell'array viene eseguita senza cicli e senza salti, migliorando notevolmente le prestazioni.

---

🔑 **Chiavi:**
[assembly, ottimizzazione, loop unrolling, registri, branch minimization, pipelinizzazione, inline assembly, prestazioni]

---

## Suggerimenti per Approfondimenti - Suggestions for Further Study

- **Approfondisci

la Pipelinizzazione del Processore**: Scopri come le pipeline funzionano in architetture specifiche e come adattare il codice assembly.

- **Profiling del Codice Assembly**: Usa strumenti di profiling, come `perf` su Linux, per identificare colli di bottiglia e migliorare ulteriormente le prestazioni.
- **Tecniche di Ottimizzazione SIMD**: Sfrutta le istruzioni SIMD (Single Instruction, Multiple Data) per gestire grandi quantità di dati in parallelo su CPU moderne.