Pattern Recognition
and Applications Lab

Reverse Engineering

Dott. Ing. Davide Maiorca, Ph.D.

davide.maiorca@diee.unica.it

Corso di Sicurezza Informatica – A.A. 2015/2016

Università
di Cagliari, Italia

Dipartimento di
Ingegneria Elettrica
ed Elettronica

Sommario

- **Introduzione**
- **Struttura ELF**
  - Introduzione a *readelf*
  - Dall'ELF alla Memoria
- **Analisi Statica**
  - Elementi di Assembly X86 ed introduzione a *objdump*
  - Analisi della memoria durate chiamate di funzione
- **Analisi Dinamica del Codice**
  - Introduzione a *gdb*
  - Analisi dinamica della memoria

Introduzione

About Me

- **Febbraio 2012**
  - Laurea Specialistica – Ingegneria Elettronica *cum laude*
- **Febbraio 2012 – Dicembre 2012**
  - Borsa di Ricerca – Regione Sardegna
  - Studio e sviluppo di tecniche per la rilevazione di Malware in PDF
- **Novembre 2013 – Aprile 2014**
  - Visiting Student – Ruhr Universität Bochum (Prof. Dr. Thorsten Holz)
- **Marzo 2016**
  - Dottore di Ricerca (Ph.D. – Doctor Europaeus) – Università degli studi di Cagliari
- **Marzo 2016 – Ora**
  - PostDoctoral Fellow – Università degli Studi di Cagliari
- **Argomenti di Ricerca**
  - Rilevazione di Malware su documenti e files multimediali (PDF, Word, Flash…)
  - Rilevazione ed analisi di malware Android
  - Adversarial Machine Learning
- **Prototipi**
  - Malware Slayer
  - AdversariaLib
  - DexObfuscator
  - FlashBuster

Introduzione – Malware Analysis

- **Saper analizzare un malware è fondamentale per capire:**
  - Le azioni a danno del sistema
  - Il tipo di attacco impiegato
  - Eventuali elementi di novità
  - Le contromisure da adottare
- **La disciplina che regola lo studio e l'analisi dei malware prende, comunemente, il nome di *malware analysis***
- **Molto spesso non si dispone del codice *sorgente* del file che analizziamo**
  - Per questo motivo, ci vuole un cambio di prospettiva
  - Necessità di analizzare i files *senza sapere come sono stati creati*
- **Il *reverse engineering* è spesso l'unico modo per analizzare un eseguibile o un altro file *senza possederne il sorgente***
  - Una vera e propria arte, ricca di complessità
- **In questa lezione vedremo gli aspetti di base, concentrandoci anche sull'aspetto *pratico***

Introduzione - Reverse Engineering

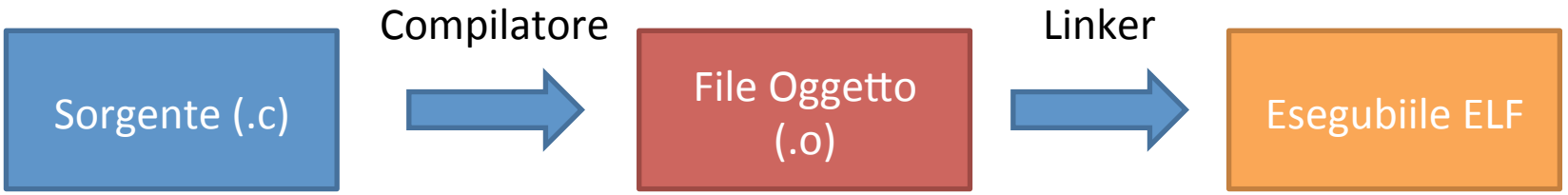
- Quando programmata, passate da un *sorgente* ad un *eseguibile*
- Problema di base: quello che fa il vostro programma è **esattamente** quello che volete?
  - Molto spesso non è così...
  - Il termine «bug» è spesso usato per definire un comportamento errato/non voluto del vostro programma
- **Reverse Engineering:** analizzare un programma *già compilato* con l'obiettivo di comprenderne a fondo i comportamenti
  - Questo significa, molto spesso, andare a vedere il funzionamento del programma ad un livello più basso del normale...
- In sicurezza informatica, il reverse engineering è una pratica fondamentale
  - Spesso è l'unico modo per analizzare un malware o per trovare vulnerabilità
  - Altissima complessità!
- Preparetevi a «sporcarvi le mani» ☺

Challenge!

- Le lezioni saranno ***challenge-driven***
- L'obiettivo è acquisire i concetti che vi consentono, *nella pratica*, di risolvere la challenge
- Dovreste avere un file chiamato *sum_number*
- Provate ed eseguirlo...
- Non avete il codice sorgente, pertanto non potete rispondere alla domanda posta dell'eseguibile
- Con i concetti che apprenderete in questa parte, sarete in grado di rispondere a questa e a molte altre domande!

Struttura ELF

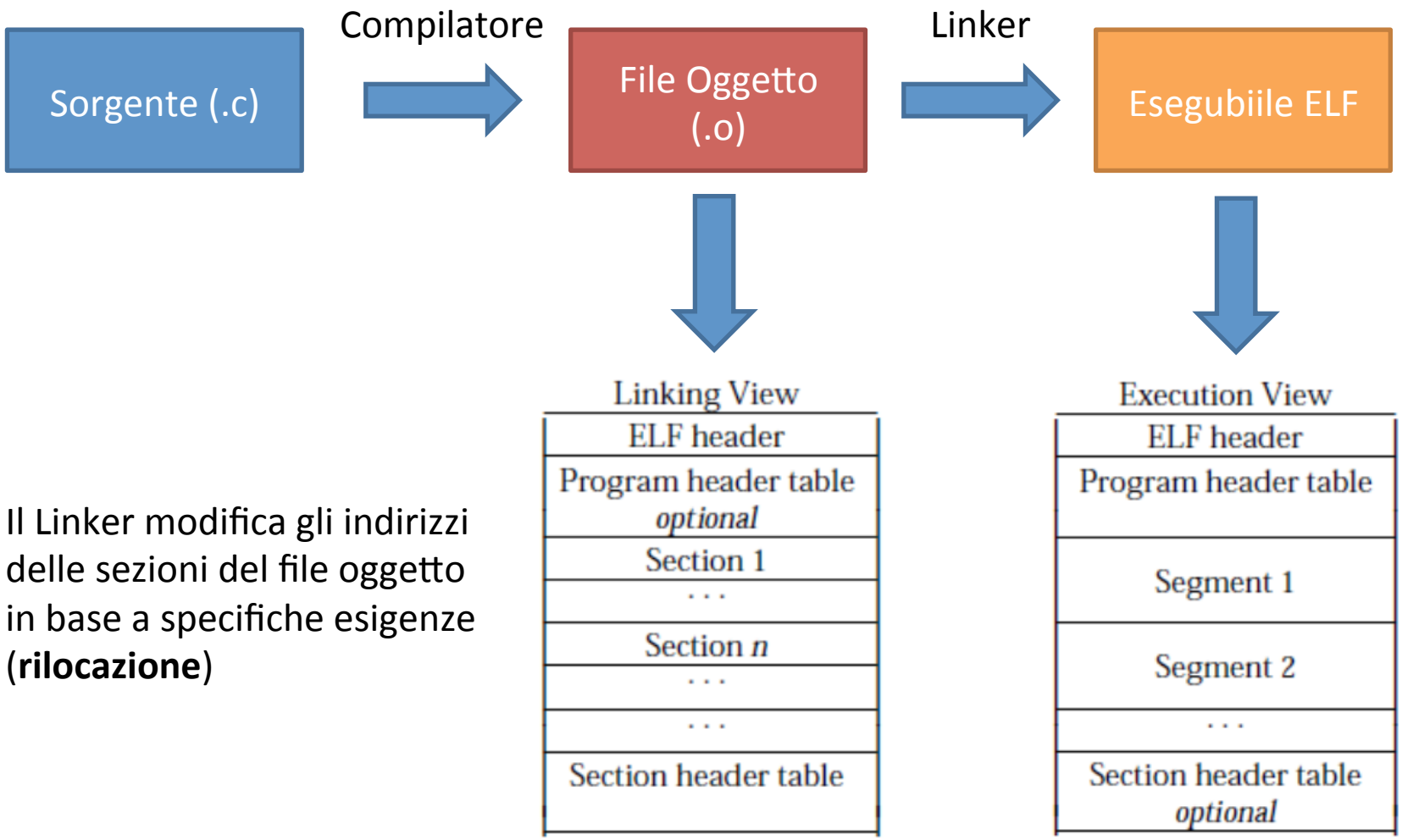
Creazione di un Eseguibile (Linux)



ELF

- **Executable and Linkable Format**
- **Eseguibile nei sistemi Linux (a 32 e a 64 bit)**
  - Gli eseguibili a 32 e 64 bit **NON** sono gli stessi
  - Differenze, ad esempio, nella gestione della memoria
- **Eseguibile diviso (generalmente) in quattro parti**
- ***ELF Header***
  - Descrive le informazioni di base sul file
  - Esempio: Tipo di architettura, indirizzi e dimensioni delle altre parti
- ***Section Header***
  - Descrive la posizione di tutte le sezioni dell'eseguibile
  - **Obbligatoria nel file .o (prima del linking)**
- ***Program Header***
  - Descrive le sezioni dell'eseguibile che vengono caricate in memoria durante l'esecuzione del programma (*segmenti)*
  - **Obbligatoria nel file eseguibile**
- ***Data***
  - I dati veri e propri del file

ELF (2)



Il Linker modifica gli indirizzi delle sezioni del file oggetto in base a specifiche esigenze (**rilocazione**)

ELF Header

- **Per analizzare un elf nella pratica possiamo usare diversi tools**
- **Noi cominceremo con l'usare *readelf***
  - E' un tool che trovate in qualsiasi distribuzione di Linux
  - Fornisce informazioni molto utili sulla organizzazione del file
- **Iniziamo allora...**
  - ***readelf –h sum_number***
- **Congratulazioni, avete appena ottenuto l'header dell'eseguibile!**
- **Diverse informazioni sono «auto-evidenti»**
- **Magic number**
  - Definisce il tipo di file (primi quattro byte) ed altre proprietà
- **Entry point**
  - Identifica l'indirizzo di memoria **VIRTUALE** di inzio del programma
  - Se siete convinti che l'inizio sia il main, preparetevi ad un cambio di fede...☺
- **Informazioni su inizio e dimensione sezioni dell'ELF**

Section Header

- **Vediamo di rendere le cose più interessanti...**
  - ***readelf –S sum_number***
  - 35 sezioni!
  - Naturalmente non ci interessano tutte...
  - Stiamo analizzando sezioni già rilocate (eseguibile completo)
- **.text**
  - Istruzioni del processo e dati in sola lettura
  - Modifiche a questi valori -> Segmentation Fault!
  - I valori in sola lettura hanno generalmente un marker
- **.data**
  - Dati statici inizializzati
- **.bss**
  - Dati statici non inizializzati

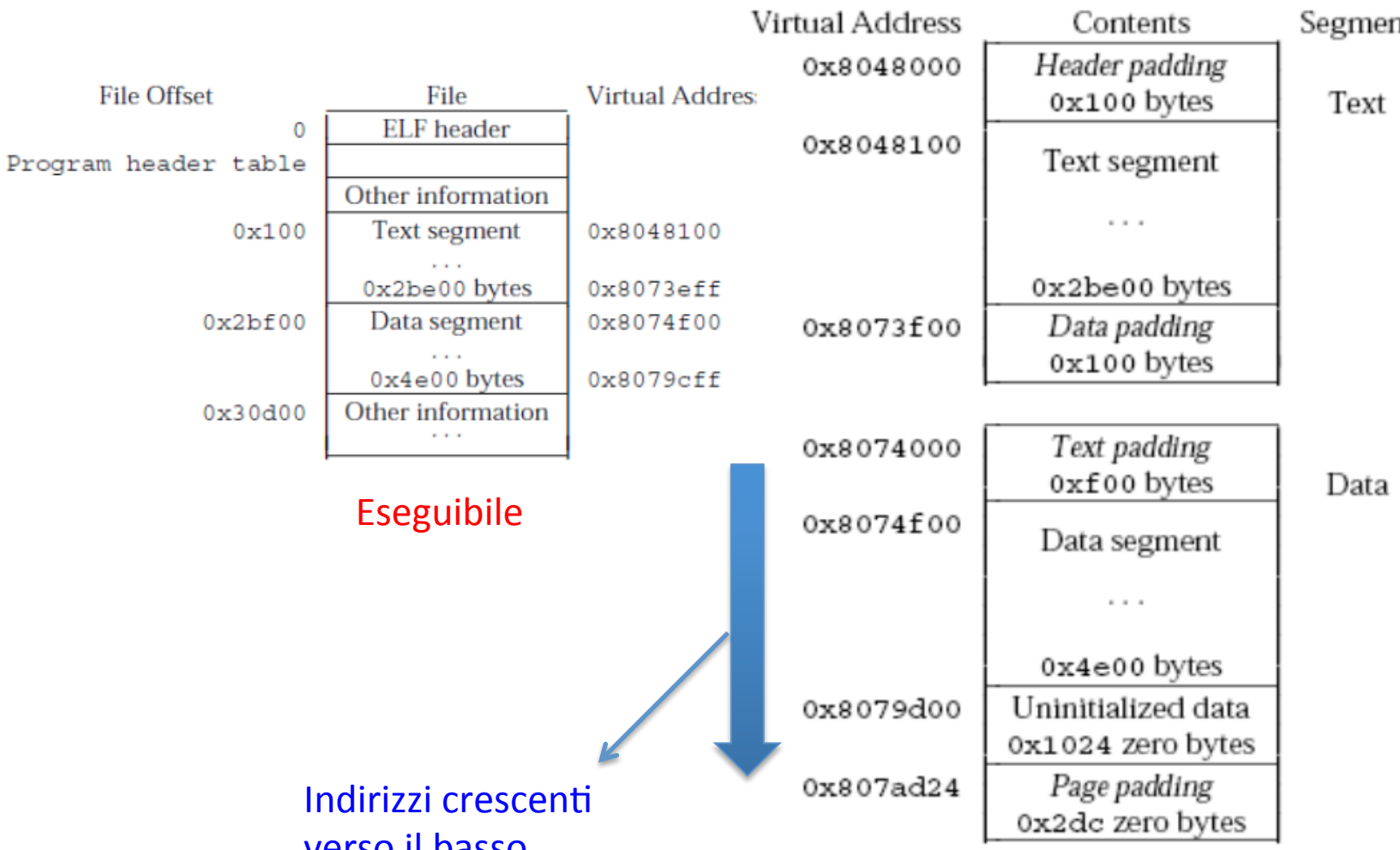
Section Header (2)

- **Tipi**
  - **PROGBITS**: sezioni contenenti dati effettivamente usati dal programma
  - **NOTE**: dati extra non utili per l'esecuzione del programma
  - **SYMTAB/DYNSYM**: contengono informazioni sui *simboli*, ovvero, rappresentare con dei nomi i dati manipolati dal codice macchina
  - **STRTAB**: contiene le stringhe usate dall'eseguibile (ad esempio quelle che stampate a video)
  - **REL**: Tabella di rilocazione
- **Altri parametri**
  - **Address**: indirizzo memoria virtuale della sezione
  - **Size**: dimensione della sezione
  - **Offset**: posizione all'interno del file
  - **Flag**: flags di esecuzione
  - Potete tralasciare gli altri parametri

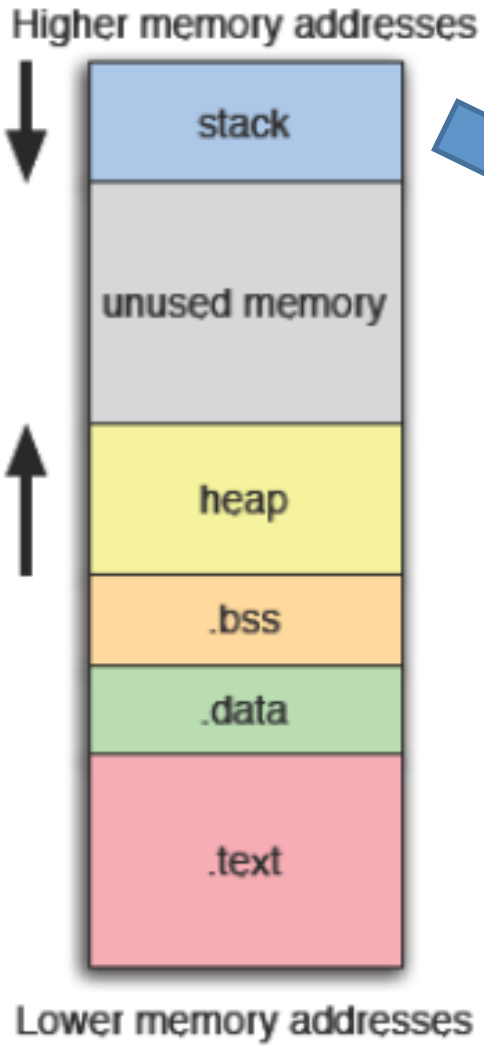
Program Header

- **Da un punto di vista didattico, è più utile vedere il Program Header dopo il Section Header...**
  - ***readelf –l sum_number***
- **Il Program Header è composto da *segmenti***
  - Ogni segmento è composto da un insieme di sezioni
- **I segmenti LOAD sono quelli che vengono caricati in memoria quando il programma è eseguito**
- **Segmento contenente la sezione .text (02 nel nostro esempio)**
  - Contiene le istruzioni macchina da eseguire (**flags: Lettura/Esecuzione**) -> La parte legata alla sezione .text prende il nome di segmento **.text** in memoria.
- **Segmento contenente le sezioni .data e .text (03 nel nostro esempio)**
  - Contiene le sezioni .data e .bss, cioè i dati inizializzati e non (flags: Lettura e Scrittura)
  - L'intero segmento *senza la sezione .bss* prende il nome di **.data** in memoria
  - In memoria la sezione **.bss** viene considerata come un segmento a parte
- **Attenzione agli offset!**
  - PHDR indica la program header table (che nel nostro esempio inizia all'offset 52)
  - LOAD inizia all'offset 0 (cioè dall'inizio del file), ma usa soltanto i **primi 0x6a4 bytes**, nonostante ne carichi 0x1000 (cioè 4096 - valore di allineamento a causa della paginazione della memoria)

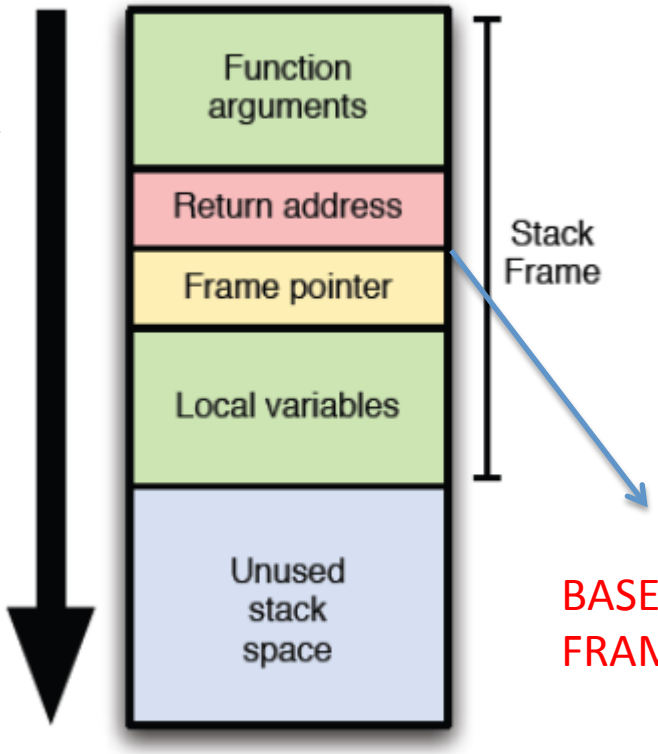
Dall'ELF alla Memoria



Struttura della Memoria Processo Linux X86



Rispetto all'immagine della slide precedente la sezione .text è in basso (indirizzi crescenti verso l'alto)



BASE DELLO STACK FRAME

Lo stack accumula verso gli indirizzi inferiori della memoria!

Struttura della Memoria Processo Linux X86 (2)

- **Heap**
  - Memoria allocata dinamicamente
- **Stack**
  - Composto da **frames**
  - Contiene numerose informazioni relative alle funzioni
  - Parametri funzioni, indirizzo di ritorno, variabili locali...
- **Ogni volta che una funzione viene invocata, un *frame* viene allocato in memoria**
- **Function arguments**
  - Gli argomenti della funzione in memoria
- **Indirizzo di ritorno**
  - L'indirizzo a cui la funzione deve ritornare al termine della sua esecuzione
- **Puntatore al frame**
  - E' considerato la "base" del frame
- **Variabili Locali**
  - Le variabili definite nella funzione

Analisi Statica

Disassemblare un eseguibile – Analisi Statica

- **Finora abbiamo visto le informazioni strutturali dell'eseguibile**
- **Ma ora vogliamo guardare più in profondità**
- **Vogliamo, cioè, capire quali sono le *istruzioni* che il processore *esegue realmente***
- **Questa operazione prende il nome di *disassembling***
- **Per fare questo, possiamo usare il tool *objdump***
  - ***objdump –d sum_number***
- **L'analisi del codice effettuata in questo modo prende il nome di *analisi statica***
- **L'analisi statica ha molti vantaggi:**
  - E' generalmente l'analisi più rapida (specie se fatta in maniera automatica)
  - Fornisce un'ampia quantità di informazioni in maniera immediata
  - Evita l'esecuzione dell'applicazione!

Elementi di Assembly X86

- **Intel CISC**
  - Complex Instruction Set Computer
  - Numero elevato di istruzioni (ma a noi ne interessano solo alcune!)
- **Convenzione AT&T per le istruzioni (opcode, sorgente, destinazione)**
  - Usata su Linux! (Su Windows si usa la convenzione Intel con sorgente e destinazione invertite)
- **Indirizzamento a 32 bit**
- **Little endian!**
  - La memorizzazione avviene «al contrario»
  - Il byte MENO significativo viene memorizzato nell'indirizzo PIU' BASSO
- **Esempio per la parola 0x90AB12CD**

Indirizzo Memoria	Byte Salvato
1003	90
1002	AB
1001	12
1000	CD

Indirizzamento memoria

- **Fate MOLTA attenzione alla little endianess**
- **La cosa può portare molta confusione!**
- **Dato un indirizzo che punta ad un blocco …**
- **…tale indirizzo punta SEMPRE alla parte INFERIORE del blocco**

Immaginate di avere due indirizzi: 0xbfff0000 e 0xbfff0004. Sul primo Indirizzo inserite l'array di caratteri '123' (in esadecimale rappresentato da 0x31 0x32 0x33) e nel secondo il NUMERO 123 (rappresentato da 7b)

LA PAROLA SI LEGGE, IN QUESTO CASO DA INIZIO BLOCCO + 4 fino alla fine

0xbfff0000: 0x00333231
0xbfff0004: 0x0000007B

	Indirizzo Memoria	Byte Salvato
Fine secondo blocco	0xbfff0007	00
	0xbfff0006	00
	0xbfff0005	00
Inizio secondo blocco	0xbfff0004	7B
Fine primo blocco	0xbfff0003	0x00
	0xbfff0002	0x33
	0xbfff0001	0x32
Inizio primo blocco	0xbfff0000	0x31

Elementi di Assembly X86 (2)

- **8 Registri «General purpose» + 1 che punta all'istruzione successiva (noi vediamo solo quelli relativi al nostro esempio!)**
  - **EAX, EDX**: Registri «accumulatori»
  - **ESP**: Puntatore allo stack
  - **EBP**: Puntatore alla base dello stackframe (quando una funzione viene chiamata)
  - **EIP**: Puntatore alla istruzione successiva
- **Istruzioni fondamentali (quelle che noi vedremo)**
  - **PUSH**: Inserisce una parola nello stack
  - **POP**: Rimuove una parola dallo stack
  - **MOV**: Sposta un valore da registro a registro o da registro a memoria (e viceversa)
  - **MOVL**: Sposta una word di dimensione 4 byte su un registro (e viceversa)
  - **AND**: Effettua una operazione di AND logico
  - **ADD/SUB**: aggiunge/sottrae un valore ad/da un registro
  - **LEAVE**: Completa alcune operazioni sullo stack (vedi slides successive)
  - **RET**: Equivalente di return
  - **CALL**: Invoca una funzione
  - **NOP**: Una istruzione che non esegue alcuna operazione
  - L'operazione **xcgh %ax %ax** è uguale alla NOP (ma non entriamo nei dettagli)

DISCLAIMER

I VALORI DEI REGISTRI (ebp, esp, eax…) POSSONO ESSERE DIVERSI A SECONDA DELLA VOSTRA ARCHITETTURA E NELLE SLIDES VIENE RIPORTATO UN ESEMPIO OTTENUTO DA UNA ESECUZIONE SU MACCHINA VIRTUALE

Una prima occhiata…

- **Dall'output di *objdump* dobbiamo guardare la sezione .text**
- **Ci sono diverse funzioni (e molte istruzioni!)**
- **Un programma C inizia dalla funzione main...**
  - Andiamo dunque a cercarla!
- **Prima di «lanciarci» ad analizzare istruzione per istruzione, cosa possiamo dire guardando la funzione main «a naso?»**
- **Un modo interessante è quello di individuare eventuali istruzioni di *chiamata a funzione***
  - Cerchiamo, quindi, le istruzioni «call»
- **Vediamo che vengono invocate tre funzioni: *sum, printf, puts***
  - Puts è come una printf senza formattazione. Viene usata, ad esempio per stampare caratteri come la newline
- **Risultato: il nostro programma chiama una funzione detta «sum» e stampa qualcosa a video, con una newline!**

Analisi Statica del codice - Main

Stack Frame (Per la funzione main)

esp = 0xbffff078

Situazione iniziale

Ogni «blocco» è costituito da 4 byte
La funzione main è sempre chiamata
da una routine chiamata _start (in Assembly il programma NON inizia dal main…)

L'ultimo elemento che una funzione inserisce
nello stack prima di chiamare la funzione successiva
è l'*indirizzo di ritorno*

Analisi statica del codice - Main

Stack Frame (Per la funzione main)

esp = 0xbffff078

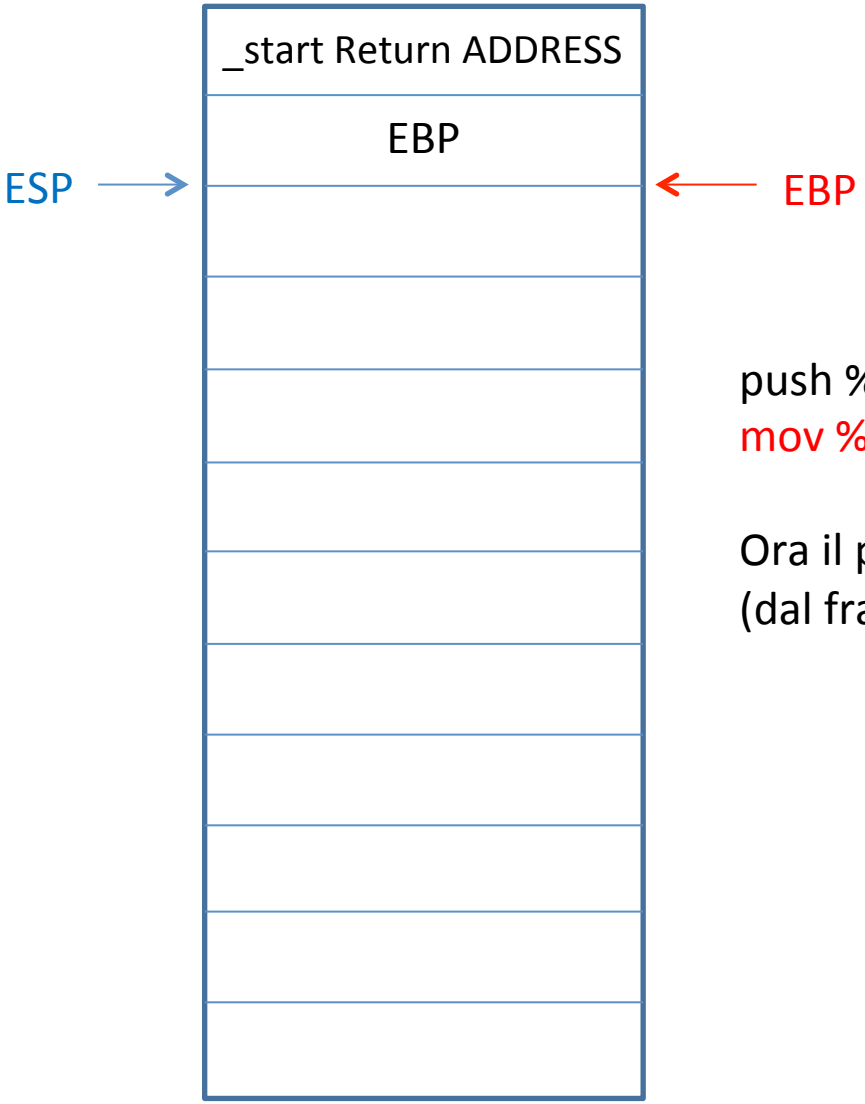
push %ebp

Il vecchio puntatore al frame viene salvato

PUSH PRIMA SPOSTA IL PUNTATORE di 4 BYTES, POI INSERISCE L'ELEMENTO!

Analisi statica del codice - Main

Stack Frame (Per la funzione main)



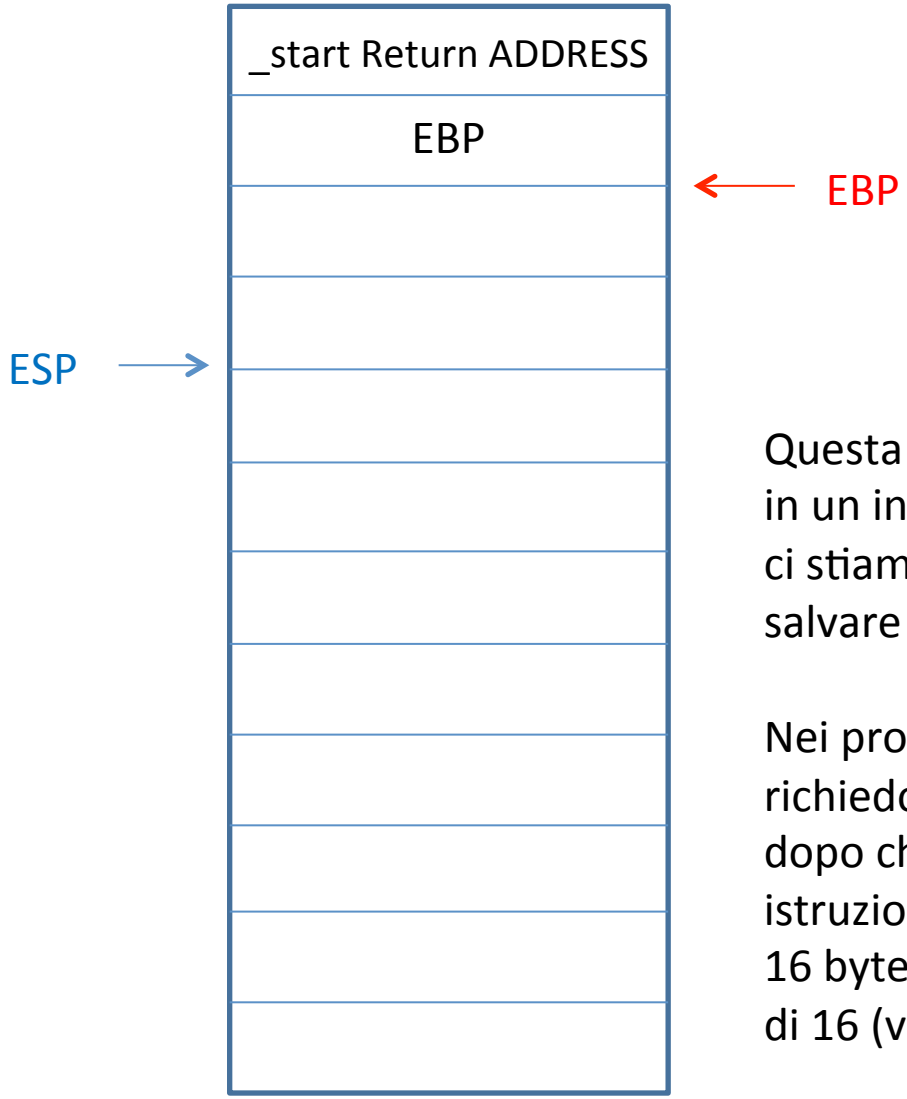
esp = 0xbffff078
ebp = 0xbffff078

push %ebp
mov %esp %ebp

Ora il puntatore al frame punta al frame attuale
(dal frame _start a quello del main)

Analisi statica del codice – Allocazione Memoria

Stack Frame (Per la funzione main)



esp = 0xbffff070
ebp = 0xbffff078

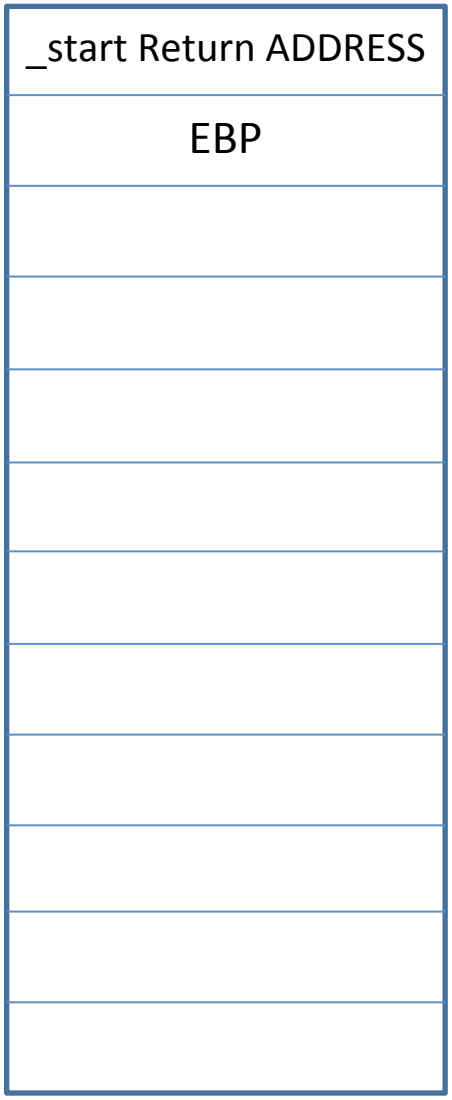
```
push %ebp
mov %esp %ebp
and 0xfffffff0 %esp
```

Questa istruzione «speciale» porta esp in un indirizzo certamente multiplo di 16. Questo perché ci stiamo apprestando a liberare della memoria per salvare i parametri della funzione da chiamare.

Nei processori intel ci sono delle istruzioni speciali che richiedono che esp si trovi in un indirizzo multiplo di 16 dopo che la memoria è stata liberata. Con questa istruzione il compilatore può aggiungere un multiplo di 16 bytes ed essere SICURO che esp si trovi in un multiplo di 16 (vedi slide successiva)

Analisi statica del codice – Allocazione Memoria

Stack Frame (Per la funzione main)



esp = 0xbffff050
ebp = 0xbffff078

push %ebp
mov %esp %ebp
and 0xffffff0 %esp
sub 0x20, %esp

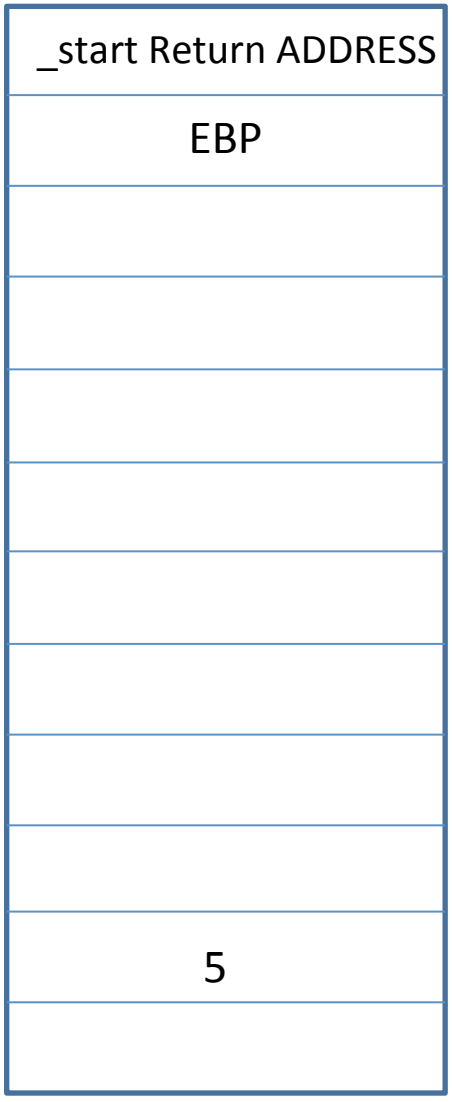
Scendo di 32 byte nello stack (0x20)

Notare come stia scendendo di un multiplo di 16 (vedi istruzione precedente)

PUSH+MOV+(AND)+SUB -> Quasi certamente vi trovate di fronte ad una preparazione a chiamata di funzione.

Analisi statica del codice – Chiamata Funzione

Stack Frame (Per la funzione main)



EBP

esp = 0xbffff050
ebp = 0xbffff078

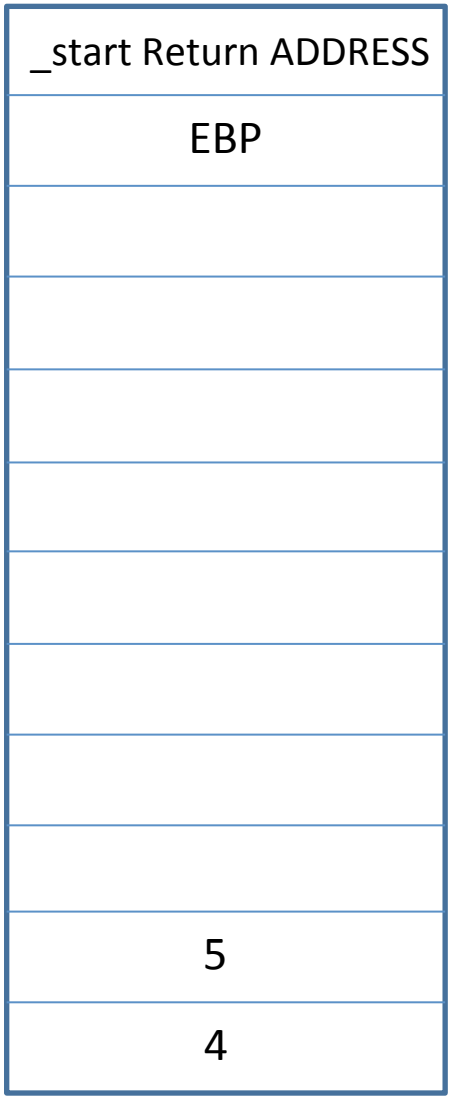
```
push %ebp
mov %esp %ebp
and $0xfffffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(esp)
```

Di norma, dopo aver liberato la memoria, inizia Il posizionamento dei parametri della funzione da chiamare (vanno sempre ALLA FINE dello stack, in ordine inverso-> Il primo parametro va esattamente alla fine dello spazio liberato).

0x4(esp) -> posiziona l'elemento alla posizione esp + 4 (MA ESP NON SI SPOSTA)

Analisi statica del codice – Chiamata Funzione

Stack Frame (Per la funzione main)



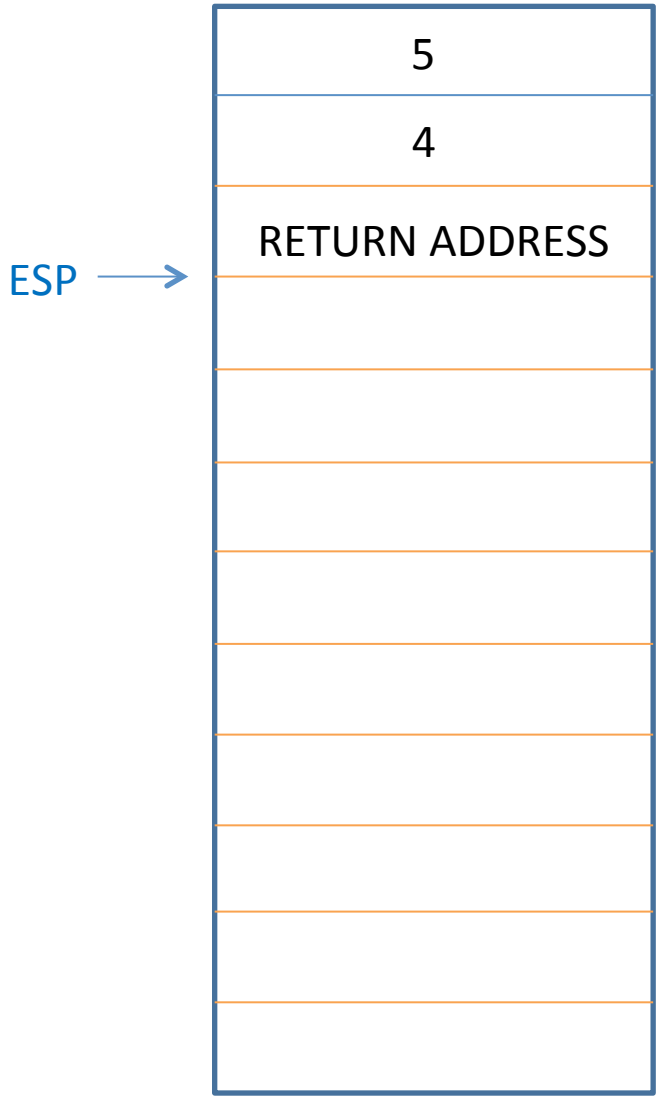
esp = 0xbffff050
ebp = 0xbffff078

push %ebp
mov %esp %ebp
and $0xfffffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(esp)
movl $0x4, (esp)

Inserisco il secondo parametro (cioè il primo nel rispettivo codice C). La funzione prende quindi due parametri che valgono 4 e 5 -> func(4, 5)

Analisi statica del codice – Chiamata Funzione

Stack Frame (Funzioni main e sum)



Frame Funzione Main

esp = 0xbffff04c
ebp = 0xbffff048

```
push %ebp
mov %esp %ebp
and $0xfffffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(esp)
movl $0x4, (esp)
call 804844d <sum>
```

Frame Funzione Sum

(N.B. anche se concettualmente il frame contiene anche i parametri passati, di norma l'inizio del frame corrisponde alla fine del return address – vedi linea gialla)

Chiamare la funzione significa inserire nello stack il suo indirizzo di ritorno (cioè, in questo caso, l'indirizzo della prossima istruzione del main) ed andare all'indirizzo della funzione

Analisi statica del codice – Funzione Sum

Stack Frame (Funzioni main e sum)

ESP →

← EBP

Frame Funzione Main

esp = 0xbffff048
ebp = 0xbffff048

```
...
movl $0x5, 0x4(esp)
movl $0x4, (esp)
call 804844d <sum>
```

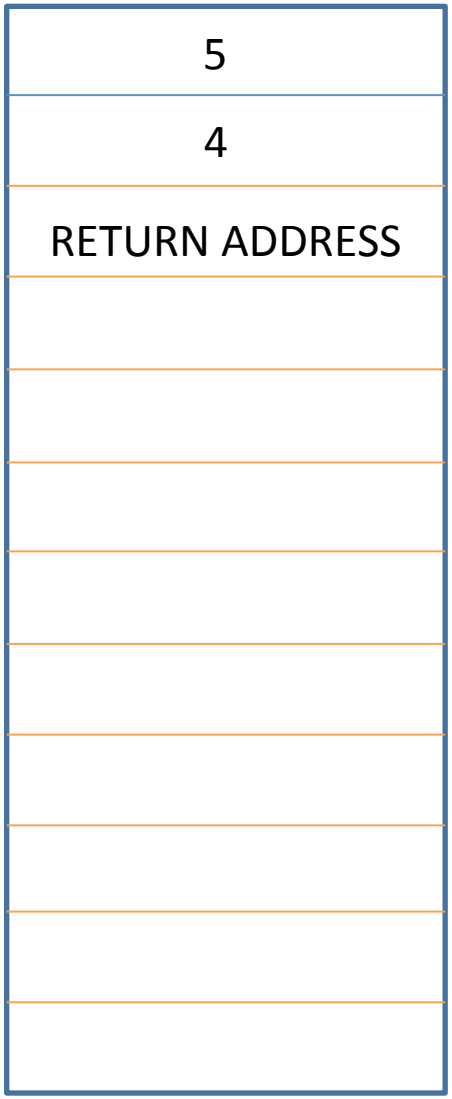
Frame Funzione Sum

```
push ebp
mov %esp, %ebp
....
```

La funzione carica sempre, nel suo frame, L'EBP al frame della funzione chiamante (che sta esattamente sotto il return address del chiamante)

Analisi statica del codice – Funzione Sum

Stack Frame (Funzioni main e sum)



Frame Funzione Main

esp = 0xbffff04c
ebp = 0xbffff078

```
...
movl $0x5, 0x4(esp)
movl $0x4, (esp)
call 804844d <sum>
```

Frame Funzione Sum

```
push ebp
mov %esp, %ebp
....
leave
```

leave svuota lo stack frame relativo alla funzione dalla quale si esce e ripristina l'ebp

Analisi statica del codice – Funzione Sum

Stack Frame (Funzioni main e sum)

Frame Funzione Main

esp = 0xbffff030
ebp = 0xbffff058

```
...
movl $0x5, 0x4(esp)
movl $0x4, (esp)
call 804844d <sum>
```

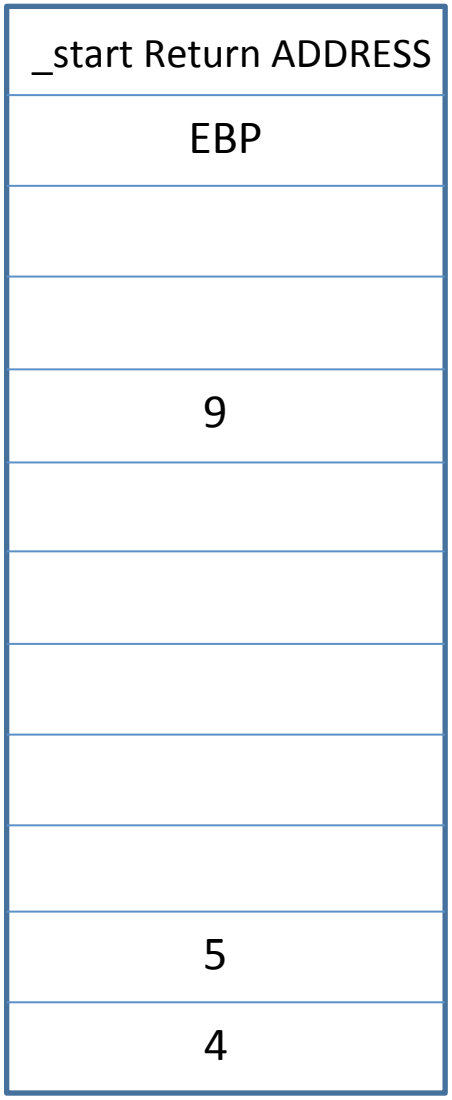
Frame Funzione Sum

```
push ebp
mov %esp, %ebp
....
leave
ret
```

ret carica, attraverso una POP (il contrario di PUSH, rimuove l'elemento dallo stack e torna indietro di 4) il return address su eip (registro istruzione successiva)

Analisi statica del codice – Funzione Sum

Stack Frame (Per la funzione main)



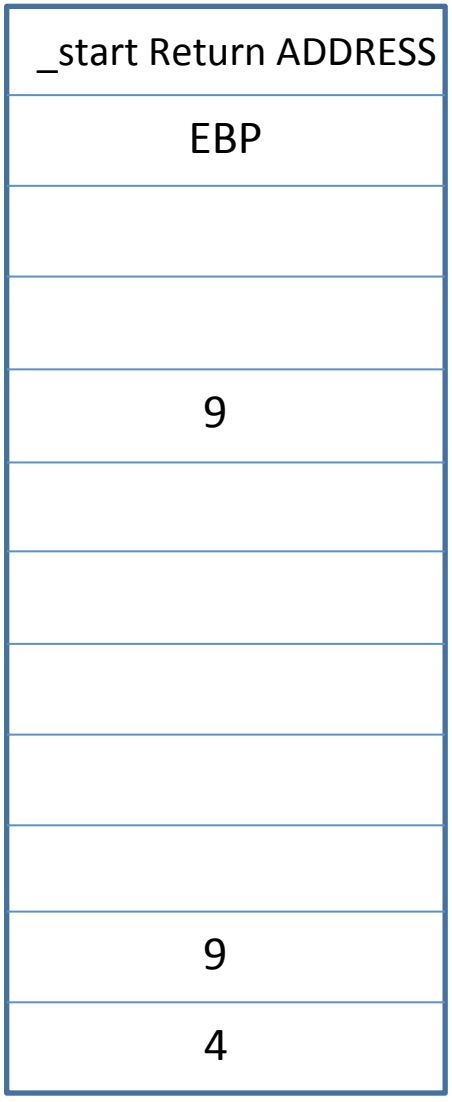
esp = 0xbffff050
ebp = 0xbffff078

```
push %ebp
mov %esp %ebp
and $0xfffffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(esp)
movl $0x4, (esp)
call 804844d <sum>
mov %eax, 0x1c(esp)
```

%eax è il risultato della funzione sum, che viene memorizzato sotto EBP riservato alle variabili locali.
In teoria sarebbe alla posizione EBP-4...
MA non dobbiamo scordarci l'istruzione di allineamento (in blu) che, in questo caso, trasla tutto di 8 bytes...

Analisi statica del codice – Chiamata printf

Stack Frame (Per la funzione main)



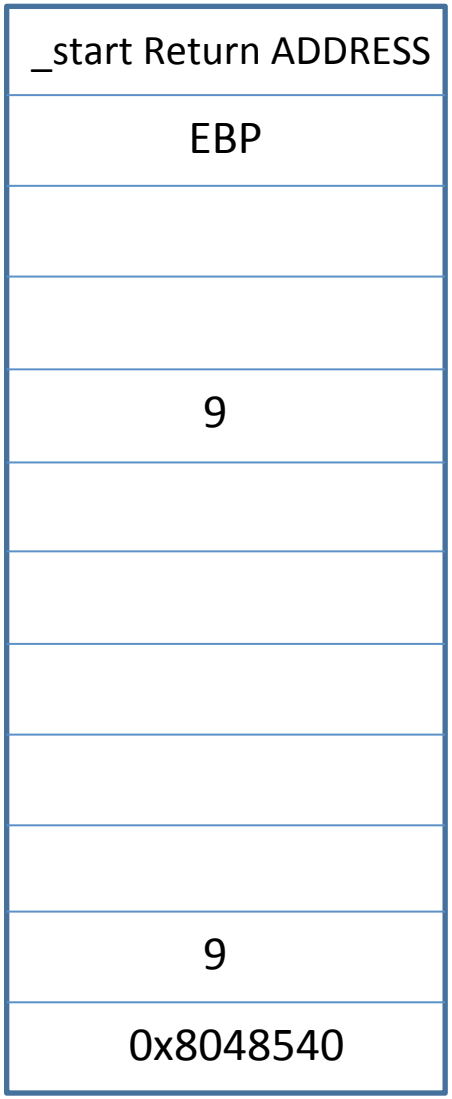
esp = 0xbffff050
ebp = 0xbffff078

```
push %ebp
mov %esp %ebp
and $0xfffffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(%esp)
movl $0x4, (%esp)
call 804844d <sum>
mov %eax, 0x1c(%esp)
mov %eax, 0x4(%esp)
```

Carico i parametri per la prossima chiamata

Analisi statica del codice – Chiamata printf

Stack Frame (Per la funzione main)



esp = 0xbffff050
ebp = 0xbffff078

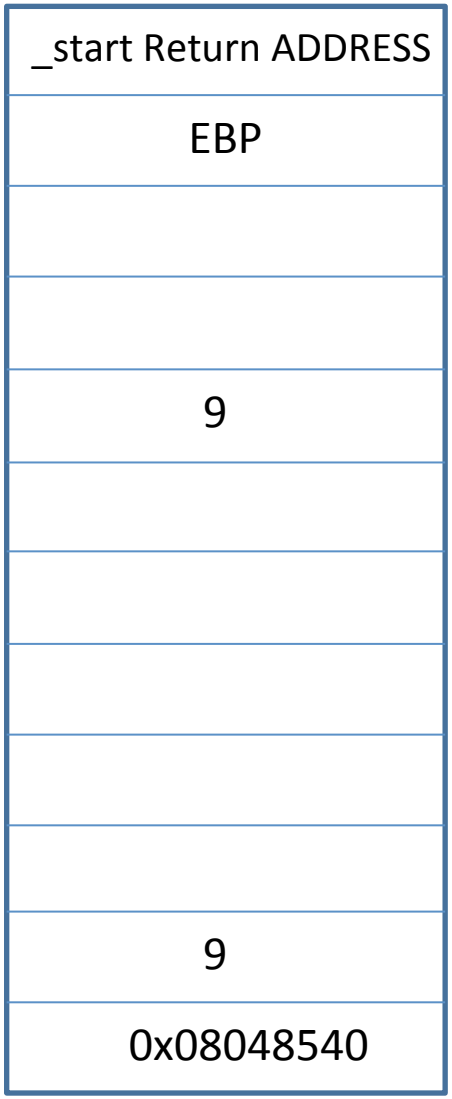
EBP

```
push %ebp
mov %esp %ebp
and $0xfffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(%esp)
movl $0x4, (%esp)
call 804844d <sum>
mov %eax, 0x1c(%esp)
mov %eax, 0x4(%esp)
movl $0x8048540, (%esp)
```

Questo indirizzo si riferisce ad una stringa (le stringhe sono memorizzate in apposite sezioni del file)

Analisi statica del codice – Chiamata printf

Stack Frame (Per la funzione main)



esp = 0xbffff050
ebp = 0xbffff078

EBP

```
push %ebp
mov %esp %ebp
and $0xfffff0 %esp
sub $0x20, %esp
movl $0x5, 0x4(%esp)
movl $0x4, (%esp)
call 804844d <sum>
mov %eax, 0x1c(%esp)
mov %eax, 0x4(%esp)
movl $0x8048540, (%esp)
call 8048310 <printf@plt>
...
```

Chiama la funzione printf (che prende come parametri la stringa ed un valore da stampare...)

Ulteriori osservazioni…

- Per capire completamente la soluzione dell'enigma, dovrete analizzare anche la funzione sum...
- Il principio è lo stesso della funzione main (ancora più semplice)!
- Riuscite a trovare la soluzione dell'enigma dall'analisi statica?
- Domanda aggiuntiva: potete dire qual è la soluzione soltanto guardando la funzione main?

Analisi Dinamica

Analisi Dinamica

- **L'analisi statica fornisce diverse informazioni sull'eseguibile**
- **Tuttavia, questo a volte non è sufficiente!**
  - Ad esempio, ci sono dei casi in cui certe informazioni sono disponibili solo a runtime...
  - Oppure, altre volte, comprendere i valori dei registri in certi istanti dell'esecuzione del programma è troppo complesso!
  - Oppure, un attaccante ha modificato l'eseguibile in modo da rendere difficile l'analisi statica…
- **In questo, *l'analisi dinamica* può essere di grande aiuto**
- **Analisi Dinamica = monitora l'esecuzione del programma, consentendo di analizzare la memoria, le istruzioni ed il flusso del programma a *runtime***

Introduzione a GDB

- **GDB = Gnu DeBugger**
- **E' il programma open source «per eccellenza» per effettuare l'analisi dinamica di eseguibili X86/X64**
- **Funziona su Linux, Windows, OSX**
- **Una miriade di funzionalità!**
- **Consente di *interrompere* l'esecuzione di un programma ad una specifica istruzione (breakpoints)**
  - Potete analizzare la memoria e i registri!
- **Consente anche di impostare delle interruzioni condizionate all'esecuzione di certi eventi (*conditional breakpoints)***
- **GDB consente di trovare bugs in un programma (o anche di sfruttarli a nostro vantaggio...)**

Usare GDB

- Riprendiamo il nostro sum_number
- *gdb sum-number*
- Con *run* eseguirete il programma
- Bello, ma forse poco interessante per noi...
- Da objdump, vediamo che la funzione comincia con 0x804844d
- Col comando x/i potete vedere l'istruzione ad un certo indirizzo
- Se date x/i 0x804844d trovate la prossima istruzione da eseguire
- Proviamo a vedere cosa succede dentro la funzione «sum»
- L'indirizzo della prima istruzione è 0x0804844d
- *break *0x0804844d*
  - **NON DIMENTICATE L'ASTERISCO**
- *run*
- L'esecuzione si ferma ***PRIMA DI ESEGUIRE L'ISTRUZIONE***

Comando X

- **Molto potente, ma merita qualche spiegazione aggiuntiva**
- **X serve a mostrare il contenuto della memoria seguendo un certo tipo di codifica (cioè se sapete che quella in memoria è, ad esempio, una istruzione o un byte)**
- **Se date x/ni potete vedere n istruzioni a partire da un certo indirizzo...**
- **Se date x/nb potete vedere n bytes a partire dalla parte inferiore del blocco...**
  - Esempio: x/4b $(ebp+4) vi mostra l'indirizzo della funzione di ritorno (in questo caso, il chiamante di sum), dato da: **0x08048480**
  - Vi verrà visualizzato così: 0x80 0x84 0x04 0x08 (IL BYTE MENO SIGNIFICATIVO E' A SINISTRA)
- **Una visualizzazione più conveniente è quella a word**
  - x/w $(ebp+4) vi mostra il risultato come word, partendo però dal **byte più significativo**
- **ATTENZIONE A NON USARE soltanto x (senza gli slash), perché adotterà la modalità di visualizzazione della sua ultima chiamata**

Analisi della Memoria con GDB

- **Possiamo ottenere informazioni sugli stack frames disponibili al momento**
- **Date *frame***
- **C'è solo un frame disponibile al momento (quello della funzione sum)**
- **Selezionate il frame con *f 0***
- ***info f***
  - Visualizza tutte le informazioni sui registri che vi servono
  - Visualizza ebp attuale, l'ebp precedente, l'indirizzo di ritorno (saved eip)
- **Vediamo cosa contengono i registri!**
- ***info registers ebp***
  - Vi mostra il valore di EBP (ATTENZIONE, IL NUOVO EBP NON E' STATO ANCORA AGGIORNATO, VEDETE DUNQUE ANCORA IL VALORE VECCHIO
- ***info registers esp***
  - Vedete l'attuale puntatore allo stack

Analisi della Memoria con GDB

- **Una delle funzionalità più interessanti di GDB è quella di mostrare il *contenuto* della memoria**
- **Vediamo cosa c'è nella locazione puntata da esp**
- ***info registers esp* restituisce «0xbffff04c»**
- **Quindi date *x/w 0xbffff04c***
  - X restituisce il valore di memoria presente ad un certo indirizzo
- **Il risultato è «0x08048480», che è l'indirizzo della istruzione dopo la chiamata a sum**
  - Quindi esp sta puntando alla locazione contenente l'indirizzo di ritorno!
  - E questo è giusto, perché dobbiamo ancora inserire nello stack il nuovo ebp
- **Andiamo avanti, ora, istruzione per istruzione**
  - Usate il comando ***ni***
- **Usatelo, ora, per tre volte**
  - Notate come gdb riesca a «tradurre» le informazioni macchina in righe di codice
  - Questo perché, durante la compilazione, sono state elaborate delle informazioni di «debugging»

Analisi della Memoria con GDB

- **Vedete, quindi, come nella funzione sum vengano sommati due parametri e memorizzati in una nuova variabile**
- **I parametri vengono memorizzati, per la somma, nei registri eax ed edx**
  - Vengono eseguite «mov 0xc(%ebp), %eax», «mov 0x8(%ebp), %edx»
- **Quali sono i valori che sono stati memorizzati su eax ed edx?**
- **Primo modo:**
  - ***info registers ebp*** -> 0xbffff048
  - ***x/b 0xbffff048+(0xc)*** -> POTETE VISUALIZZARE LA MEMORIA ANCHE A DETERMINATI OFFSETS! ☺ -> Ottenete 5 (cioè il SECONDO parametro)
  - ***x/b 0xbffff048+(0x8)*** -> Ottenete 4 (cioè il PRIMO parametro)
- **Secondo modo:**
  - Date ***ni*** due volte
  - ***info registers eax***, ***info registers edx***
- **Terzo modo:**
  - ***print a*** e ***print b***, dato che la funzione prende in ingresso a e b
  - Funziona SOLO se avete informazioni di debugging...☺

Riassumendo...

- **In queste lezioni avete imparato diverse cose**
- **Avete compreso la struttura di un eseguibile Linux**
- **Avete visto come l'eseguibile viene caricato in memoria**
- **Avete analizzato l'eseguibile, apprendendo i fondamenti di assembly x86 e utilizzando due tecniche:**
  - Analisi statica
  - Analisi dinamica
- **Tutto bello...**
- **Ma la domanda ora è: come fa un attaccante a sfruttare tutte le informazioni che abbiamo visto noi *a suo vantaggio?***
- **Stay tuned for the next lesson!**