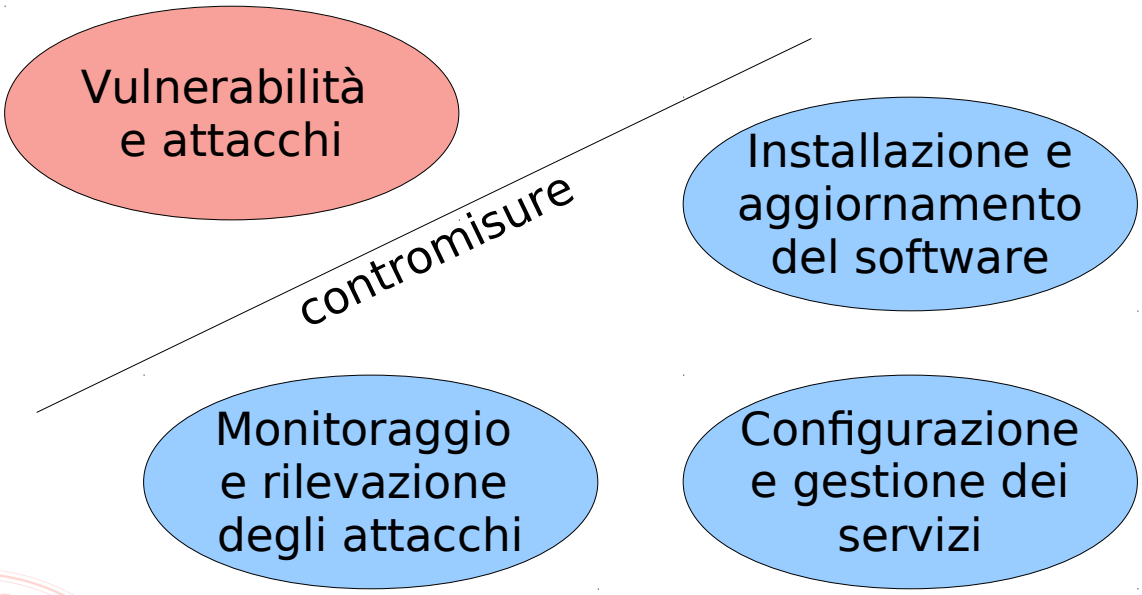
# Sicurezza dei sistemi

## Qualche esempio delle tecniche usate per violare i sistemi, e delle più comuni contromisure

Marco Prandini

## Panoramica



# Un po' di termini

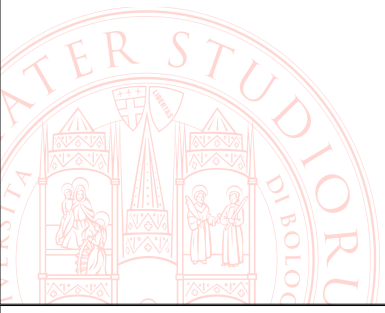
- Minaccia (threat)
  - Un atto ostile intenzionale o meno che ha un qualsiasi effetto negativo sulle risorse o sugli utenti del sistema
- Vulnerabilità (vulnerability)
  - Un difetto nelle misure a protezione del sistema o dei dati
    - Può essere strutturale nell'hardware o software
    - Puo dipendere dalla configurazione
    - Può dipendere da un uso scorretto (es. navigare come root)
- Exploit
  - Uno strumento per trarre vantaggio da una vulnerabilità concretizzando una minaccia
    - Tecnico (cracking)
    - Umano (social engineering)
- Rischio (risk)
  - Il potenziale danno immateriale, perdita economica, o distruzione di risorse che risulterebbe dall'azione di una minaccia che riuscisse a sfruttare una vulnerabilità

# Qualche rilevazione sugli attacchi

- Stima dei costi 100G$ (2015) → 6000G$ (2021)
- Vettori degli attacchi (2016)
  - 64% delle aziende attaccate via web
  - 62% via phishing / social engineering
  - 59% via malware (200k nuovi sample/giorno)
  - 51% denial of service
  - hot trend: ransomware (4000 attacchi/giorno)
- Vulnerabilità
  - OWASP Top Ten
    https://www.owasp.org/index.php/Top_10-2017_Top_10
  - CWE/SANS Top 25
    http://cwe.mitre.org/top25/

# Vulnerabilità del codice – origini (in sintesi)

- Codice di scarsa qualità
  - "Remotely exploitable buffer overflow vulnerabilities continue to be the number one issue that affects Windows services."
- Reazioni inadeguate dei produttori
  - "In many cases, the vulnerabilities were 0-days i.e. no patch was available at the time the vulnerabilities were publicly disclosed."

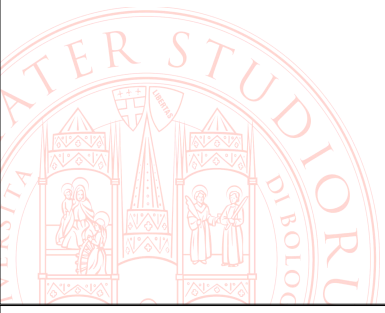


# Vulnerabilità causate dal codice

- Attacchi data-driven (spesso cross-platform)
  - Cross-site scripting
  - Condivisione e backup di file
  - Database query injection
  - Media players
  - External XML processors
- Logiche di controllo dell'accesso errate
  - Autenticazione fallata
  - Errori nella gestione delle autorizzazioni

# La fase progettuale

- Non esistono metodi formali che consentano una verifica completa delle caratteristiche di sicurezza di un sistema - purtroppo la pigrizia dei progettisti tipicamente va ben oltre: esempi plateali includono
  - Code injection
  - WEP



# Attacchi locali – Exploit

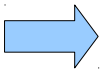
- Gli attacchi locali sfruttano le vulnerabilità di:
  - Sistema Operativo
  - Hardware sottostante
  - Software in esecuzione locale
- Non sono coinvolti:
  - Protocolli di rete
  - Router e infrastruttura di rete
- Focus on:
  - Architettura Intel IA32
  - Sistema Operativo Linux
- Considerazioni analoghe per altri OS e architetture

# Obiettivo Exploit

- Lo scopo di un exploit è far eseguire ad un processo operazioni per cui non era stato pensato
- Tre obiettivi (non mutuamente esclusivi):
  - Fermare il processo (Denial Of Service – DOS)
  - Dirottare il flusso di esecuzione (Esecuzione di codice maligno)
  - Ottenere i privilegi dell'utente root

# Esempio di SQL injection

- Un esempio di code injection: scavalcare un sistema di autenticazione SQL-based, o usarlo per tutt'altro

```
SELECT * FROM Users
WHERE uid='$username'
AND pwd='$password';
```

... e se digito come password:

```
' OR 'a'='a
'; DROP DATABASE WebApp;  --
```

SQL injection cheat sheet

- https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/

# Esempio di cross-site scripting

- Un esempio di code injection: cross site scripting (XSS)
  - https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

http://www.yourdomain.com/welcomedir/welcomepage.php?name=John

```
<?php
        echo 'Welcome to our site ' . stripslashes($_GET['name']);
?>
```

http://www.yourdomain.com/welcomedir/welcomepage.php?name=
<script language=javascript>alert('Hey, you are going to be hijacked!');</script>

- Esecuzione nel dominio di sicurezza associato al sito vulnerabile
  - può manipolare cookie di autenticazione come già consentito al sito
  - se la pagina vulnerabile è locale può eseguire codice privilegiato

# All'estremo opposto: CPU bugs

- Mancanza di igiene (Spectre)
  - la CPU precarica codice (prefetching) per essere certa di non restare mai senza operazioni da svolgere
  - esecuzione speculativa: se si avvicina un branch e la CPU ha cicli disponibili, vengono eseguite le istruzioni di entrambi i rami e messi in cache i risultati
  - anche se i dati del ramo di branch che poi si rivela inutile sono inaccessibili, restano in cache, e misurando il tempo di accesso a diversi dati, si può risalire a quale fosse il valore di certi parametri che hanno causato la scelta di un ramo o dell'altro
- Errore concettuale (Meltdown):
  - non applicare agli "internals" della CPU gli stessi criteri di separazione imposti tra sistema operativo e user space.
- Dettagli

https://googleprojectzero.blogspot.it/2018/01/reading-privileged-memory-with-side.html
https://meltdownattack.com/ - https://spectreattack.com/

# I grandi classici: exploit binari

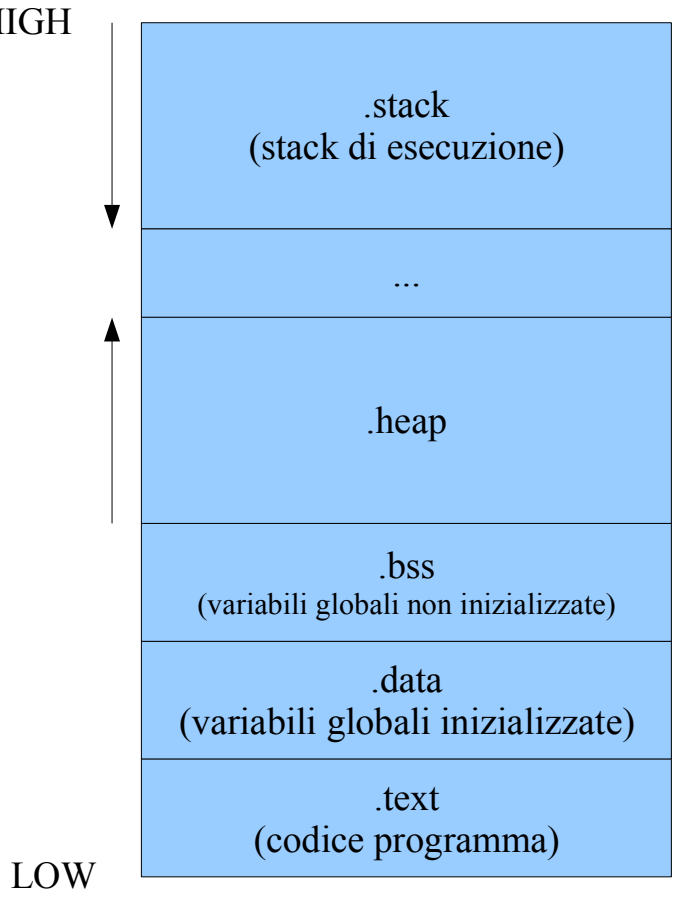
- La comprensione dei meccanismi alla base delle tecniche di Exploit necessita di una conoscenza basilare di:
  - Disposizione in memoria di un processo
  - Funzionamento dell'architettura del processore su cui lavora il sistema vittima (nel nostro caso IA32)
- Si noti che la maggior parte delle vulnerabilità sono relative a codice scritto in C/C++ e linguaggi derivati. Infatti tali linguaggi, più vicini all'hardware, non realizzano controlli (in automatico) sui dati
- Linguaggi di più alto livello (come ADA, ad es.) in fase di compilazione aggiungono controlli ad ogni istruzione/gruppi di istruzioni

# Processo in memoria

- Nell'OS Linux lo spazio di indirizzamento di un processo in memoria è suddiviso in un insieme di segmenti:
  - Segmento .text, che contiene il codice eseguibile
  - Segmento .data, contenente i dati inizializzati (variabili statiche inizializzate)
  - Segmento .bss, contenente le variabili non inizializzate
  - Stack d'esecuzione, con i record d'attivazione del processo e le variabili locali
  - Heap, segmento di memoria contenete le variabili dinamiche (può crescere dinamicamente)
  - Altri segmenti necessari al funzionamento (.got, .ctor, .dtor)

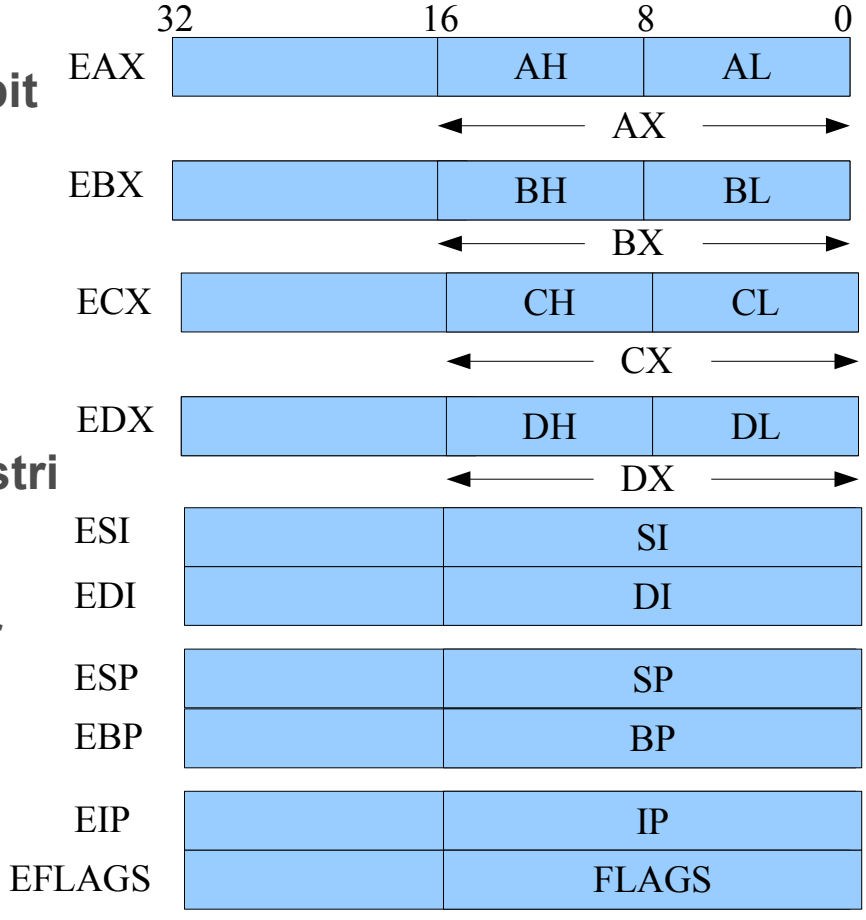
# Processo in memoria

- Si noti che il programmatore “vede” gli indirizzi virtuali. Questi ultimi sono tradotti dall'OS (+ HW) in indirizzi fisici:
  - I segmenti non necessariamente sono contigui
- Andamento indirizzi:
  - Lo stack cresce verso il basso
  - L'heap cresce verso l'alto




# Cenni architettura IA32

- L'architettura IA32 è dotata di 8 registri 32 bit “general purpose”:
  - EAX, EBX, ECX, EDX
  - ESP: stack pointer
  - EBP: base pointer
  - ESI: source
  - EDI: destination
- Inoltre è dotata di registri speciali:
  - EIP: Instruction Ptr
  - EFLAGS: Status register



# Convenzioni di chiamata C IA32

- La traduzione C → Assembly adotta convezioni standard (a differenza di altri linguaggi)
- Tre convenzioni di chiamata:
  - __cdecl (Convenzione di Default)
    - Inserimento sullo stack di tutti i parametri attuali di chiamata in ordine inverso rispetto alla signature del metodo.
    - Chiamata tramite “CALL” salvando l'indirizzo di ritorno sullo stack.
    - Il chiamato salva il contenuto del registro EBP ponendolo sullo stack e aggiorna il valore di EBP al contenuto attuale di ESP (in altri termini il nuovo EBP punterà alla locazione dello stack in cui è salvato il vecchio EBP).
    - Il chiamato ritorna il risultato delle proprie computazioni nel registro EAX.
    -

# Convenzioni di chiamata C IA32

  - __cdecl (continuo)
    - E' compito del chiamato ripristinare il valore originario di EBP, con quello (da lui) salvato sullo stack in precedenza.
    - E' compito del chiamante rimuovere i parametri attuali dallo stack
  - __stdcall
    - L'unica differenza con la modalità precedente è che è responsabilità del chiamato eliminare i parametri dallo stack.
  - __fastcall
    - La differenza con la __cdecl è che I parametri attuali non sono passati via stack bensì tramite i registri generali (da EAX a EDX e ESI e EDI, quindi con un limite di 6 parametri di 32 bit)

# Esempio di chiamata (1/3)

Codice **chiamante** (convenzione __cdecl)

```
…
int result;
…
result = sum(4,5);
…
```

Compilazione

```
0x80401000 result: DW 1
…
0x8040200A PUSH    0x5
0x8040200F PUSH    0x4
0x80402015 CALL    _sum
0x8040201A ADD     ESP, 0x8
0x8040201F MOV     result, EAX
…
```

Il chiamante immette i parametri della funzione in cima allo stack in ordine inverso

Il chiamante invoca la funzione tramite l'operazione CALL, che pone in cima allo stack l'indirizzo di ritorno

Di ritorno dalla funzione, il chiamante rimuove dallo stack i parametri passati

Il chiamante prende il risultato dal registro EAX

# Esempio di chiamata (2/3)

Codice **chiamato** (convenzione __cdecl)

```
int __cdecl sum(int a, int b) {
    int c;
    c = a+b;
    return c;
}
```

Compilazione

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0X4
MOV     [EBP-4], [EBP+8]
ADD     [EBP-4], [EBP+12]
MOV     EAX, [EBP-4]
MOV     ESP, EBP
POP     EBP
RET
```

Salva il contenuto di EBP, e memorizza in EBP il puntatore al frame pointer (contenuto di ESP)

Riserva sullo stack lo spazio per la variabile locale c

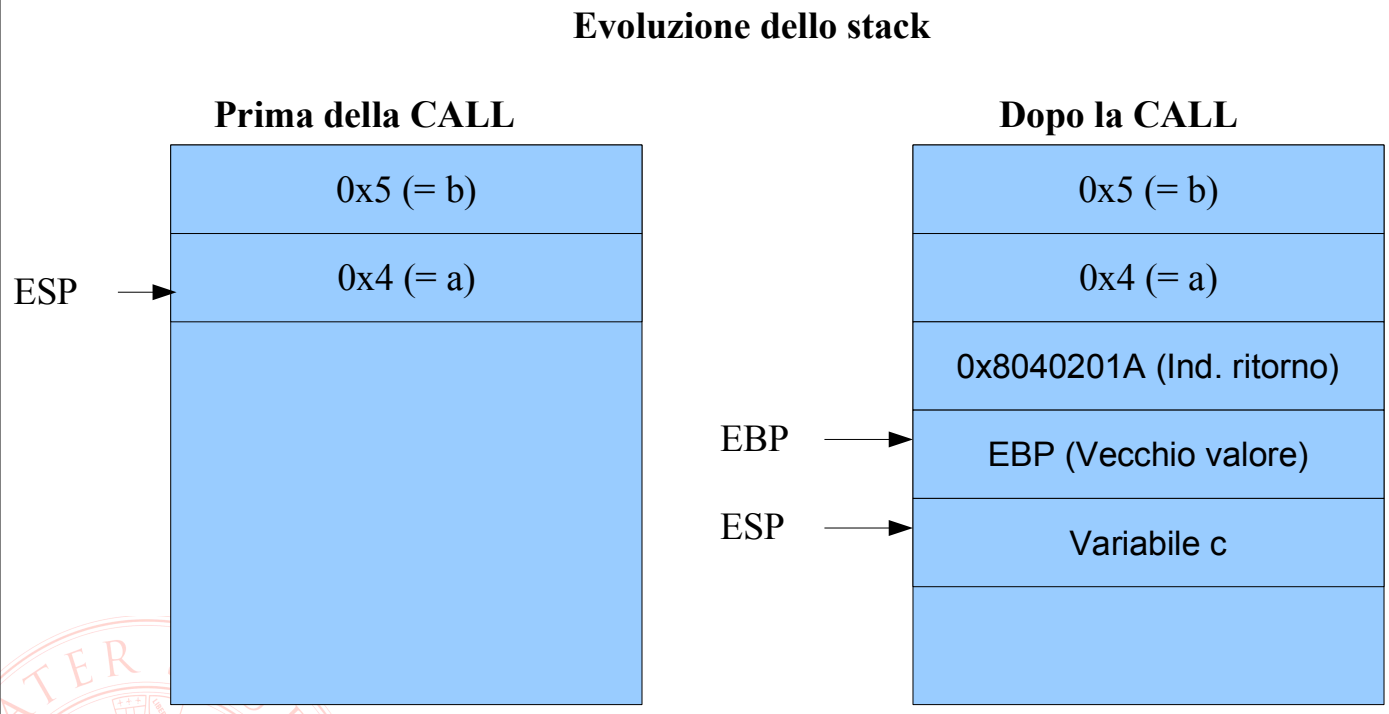
Effettua le operazioni usando come Riferimento (in base a cui muoversi sullo stack) il registro EBP

Salva il risultato in EAX

Ripristino di EBP e ESP

Ritorno al chiamante

# Esempio di chiamata (3/3)




# Stack Overflow (1/9)

- Prerequisiti per l'attuazione:
  - Presenza di un buffer locale ad una funzione (tale buffer si troverà quindi sullo stack).
  - Possibilità di alimentare il buffer con input esterni (ad es. da tastiera, da file, da rete).
- Modalità di attuazione:
  - L'attaccante riempie il buffer con dati fittizi (padding), sforandone i limiti, fino ad arrivare a porre sullo stack “nella posizione giusta” un nuovo indirizzo di ritorno dalla chiamata (sovrascrivendo quello preesistente).
- Risultato:
  - Il flusso di controllo non procede con il ritorno al chiamante “lecito” bensì al “nuovo” indirizzo di ritorno posto dall'attaccante.

# Stack Overflow (2/9)

- Un buffer locale ad una funzione/routine è realizzato con un blocco di memoria riservato sullo stack (di lunghezza finita e predeterminata).
- Il linguaggio C non controlla che i dati con cui alimentare i buffer/array abbiano una lunghezza congrua (tale controllo è a carico del programmatore).
- Si ricordi che:
  - Lo stack cresce con indirizzi decrescenti (cioè i dati "più vecchi" hanno indirizzi più alti)
  - Il riempimento del buffer avviene invece per indirizzi crescenti (sebbene questo si trovi sullo stack)

# Stack Overflow (3/9)

- Conseguenza della strategia di riempimento dello stack:
  - In assenza di controlli di overflow, i dati in eccesso (immessi nel buffer) vanno a sovrascrivere i dati dello stack immessi in precedenza.

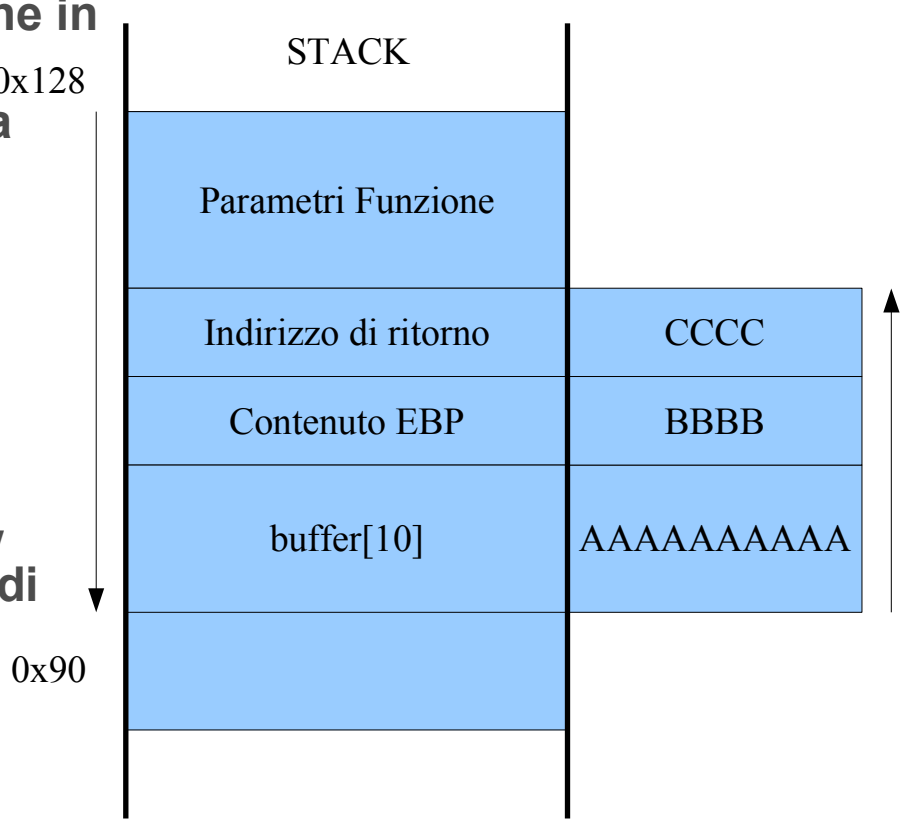
- Esempio

```
void function() {
   char buffer[10];
   gets(buffer);
   ...
}
```

- La funzione gets legge una stringa dallo stdin ma non effettua controlli sulla lunghezza.

# Stack Overflow (4/9)

- Esempio di disposizione in memoria.
- L'attaccante scrive una stringa lunga più del dovuto
- Nell'esempio di prima:
  - 10 Byte di padding
  - 4 byte per coprire EBP
  - 4 byte per sovrascrivere l'indirizzo di ritorno
- Si noti come l'overflow sovrascriva l'indirizzo di ritorno.




**Stringa: "AAAAAAAAAABBBBCCCC"**

# Stack Overflow (5/9)

- Conseguenze dell'attacco:
  - Denial Of Service
    - Se l'indirizzo di ritorno "illecito" è relativo ad una posizione in memoria non accessibile (dal programma in esecuzione) avviene un crash per "segmentation fault".
  - Controllo del flusso. L'attaccante prende il controllo dell'esecuzione. Si hanno due alternative:
    - Shell Coding: si "inietta" (come parte della stringa) un pezzo di codice maligno preparato in precedenza. In questo modo, è possibile eseguire codice con i privilegi del processo vittima.
    - Return to LibC: la stringa iniettata per overflow scrive sullo stack i dati necessari per effettuare una invocazione di una funzione di libreria C.

# Stack Overflow (6/9)

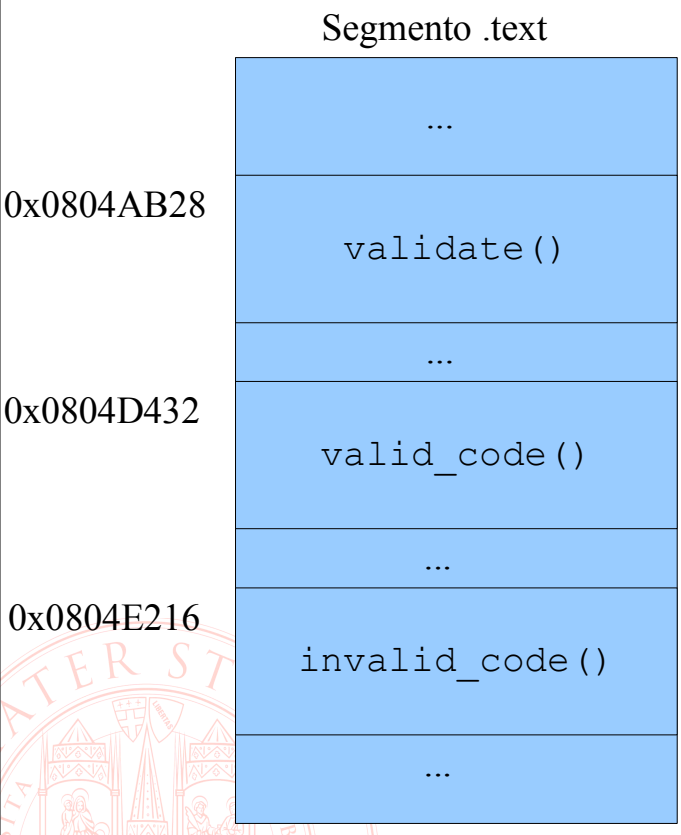
- Un semplice esempio di controllo del flusso
- In questo programma l'input copiato in un buffer determina se un utente ha i diritti di eseguire una porzione di codice o meno (ad es. potrebbe essere richiesta la password da linea di comando)

```
int validate() {
  char buffer[10];
  gets(buffer);
  if(//Valid input)
    return 1;
  else
    return 0;
}

void valid_code() {
  // Codice per utenti autorizzati
  ...
}

void invalid_code() {
  // Codice per utenti non aut.
  ...
}
```

# Stack Overflow (7/9)




- L'obiettivo di un attaccante che non ha le credenziali affinché `validate` ritorni 1, è di forzare l'esecuzione del codice di `valid_code`
- `validate` utilizza la funzione `gets` per ottenere la password
- La `gets` è non sicura
- In tali condizioni un attacco di stack overflow è molto semplice

# Stack Overflow (8/9)

- Nel momento in cui `validate` richiede l'input all'utente l'attaccante prepara una stringa così composta
  - 10 byte di padding
  - 4 byte per “scavalcare” l'EBP
  - 4 byte per scrivere l'indirizzo di `valid_code`: **0x0804D432**

E' necessario tradurre l'indirizzo di `valid_code` nella sua rappresentazione ASCII. Il risultato comprende anche caratteri non stampabili. Esistono varie tecniche per passare al processo tali caratteri (ad es. printf di bash).

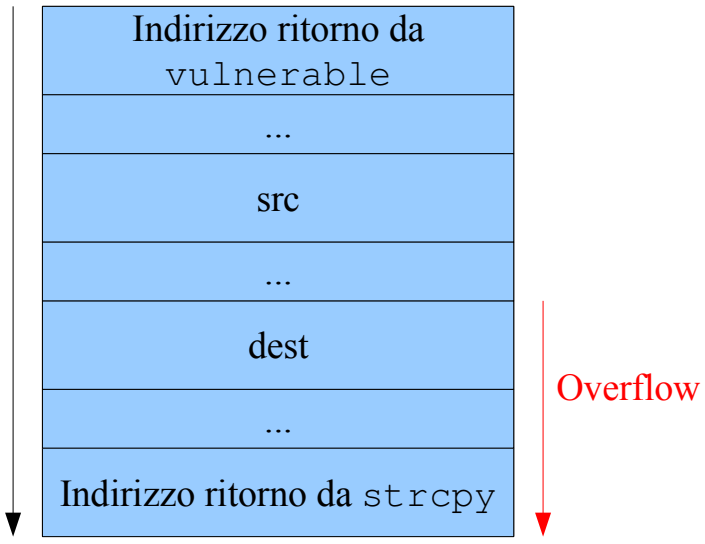
**Stringa d'attacco: “AAAAAAAAAABBBB\x32\xD4\x04\x08”**

L'indirizzo è fornito con i byte in ordine inverso: l'architettura IA32 è infatti **little endian**

# Stack Overflow (9/9)

- L'attacco di Stack Overflow può essere utilizzato **anche su architetture che presentano stack con indirizzi crescenti**

```
void vulnerable() {
    …
    char src[10];
    char dest[10];
    …
    gets(src);
    …
    strcpy(dest, src);
    …
}
```




- **In questo caso, inserendo una stringa più lunga del dovuto in `src`, nel momento in cui verrà richiamata la `strcpy` per copiarla in `dest`, essa andrà a sovrascrivere l'indirizzo di ritorno della `strcpy` stessa e non di `vulnerable`**

# Stack Overflow – Canarini (1/3)

- **Canarini**
  - Prima forma di contromisura
  - Deve il suo nome alla similitudine storica con i canarini dei minatori in grado di rilevare fughe di gas
- Il concetto alla base è di porre sullo stack, prima di un buffer, un dato di riferimento.
  - Il processo è in grado di rilevare un tentativo di attacco verificando l'integrità di tale dato.
  - In caso di attacco con overflow il dato viene sovrascritto e la verifica da esito negativo
  - Tale protezione è realizzata tramite la collaborazione congiunta di compilatore e libreria standard
  - Non è un meccanismo fornito dall'OS o dall'HW
  - Generazione del dato e verifica causano overhead

# Stack Overflow – Canarini (2/3)

- Funzionamento in GCC
  - Tale protezione era inizialmente fornita come patch ProPolice (a partire dalla versione 3.x)
  - Inclusa nel main branch a partire dalla versione 4.1
- Abilitazione:
  - Non è abilitato di default su tutti gli OS / Distro Linux
  - `-fstack-protector` abilita solo per buffer di stringhe
  - `-fstack-protector-all` abilita per tutti i tipi di buffer
  - `--param ssp-buffer-size=` imposta una soglia di dimensione del buffer oltre la quale la protezione viene attivata. Questo evita che la protezione venga attivata per tutte le chiamate a funzione, riducendo l'overhead.

# Stack Overflow – Canarini (3/3)


- Ad ogni chiamata di funzione:
  - Si genera e scrive un valore casuale di 4 byte
- L'attaccante dovrebbe sovrascrivere anche il canarino per modificare l'indirizzo di ritorno (non può "scavalcarlo")
- La modifica del canarino viene rilevata nel momento in cui si ritorna dalla chiamata
- L'azione di default in tal caso è la terminazione del processo
- Tale evento può essere inoltre catturato tramite segnali dell'OS

# Shellcoding (1/5)

- Non è una forma di attacco alternativa, bensì è un tecnica per sfruttare lo stack overflow.
- Consiste nell'iniettare (tramite stack overflow):
  - Codice maligno
  - Indirizzo di ritorno che punti al codice di cui sopra
- L'obiettivo tipico di tale approccio è l'apertura di una shell (da cui il nome), con i privilegi del processo attaccato (tipicamente root o con il bit setuid attivato).
- Il principio alla base di tale tecnica è utilizzabile anche con altri tipi di attacco alternativi allo stack overflow.

# Shellcoding (2/5)

- Un esempio di codice maligno che realizza l'invocazione della system call exit con il valore 0, terminando il processo è il seguente:

```
mov EBX, 0
mov EAX, 1
int 0x80
```

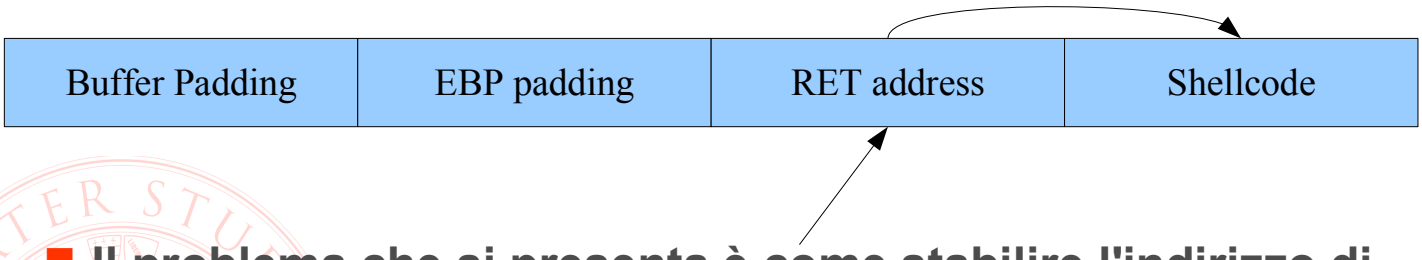
I parametri di una system call devono essere caricati nei registri in ordine EBX, ECX, EDX, ESI, EDI. Nel nostro caso l'unico parametro è il valore 0.

Il registro EAX va predisposto con l'identificativo numerico della system call. nel nostro caso exit=1

Si genera un interrupt sulla linea 0x80
(che corrisponde al gestore delle system call in Linux)

# Shellcoding (3/5)

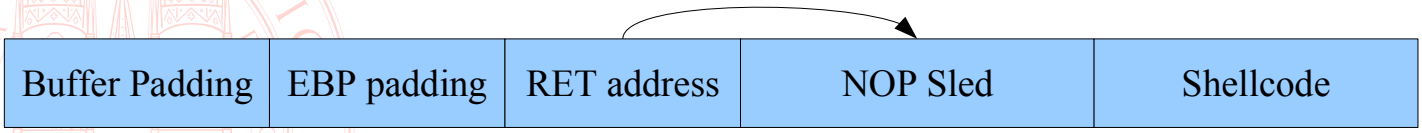
- Dopo aver preparato il codice assembly da iniettare si procede come segue:
  - Si compila il codice di cui sopra.
  - Si genera la rappresentazione esadecimale del risultato compilato.
  - Si genera la stringa da iniettare come visto per lo stack overflow:




- Il problema che si presenta è come stabilire l'indirizzo di ritorno (ovvero l'indirizzo in cui andrà a posizionarsi il codice maligno iniettato).

# Shellcoding (4/5)

- Per calcolare/intuire l'indirizzo di ritorno esistono varie tecniche:
  - Tipicamente il segmento .stack di un processo comincia sempre a partire dallo stesso indirizzo
  - L'attaccante può procurarsi il codice sorgente del programma da attaccare e la sua immagine binaria (facile se la vittima usa una qualsiasi distro Linux)
- Per avere maggiori garanzie sulla riuscita si può ricorrere ad un "NOP Sled" (slitta di NOP):
  - Il NOP Sled è una sequenza di NOP (operazione assembly che non effettua nulla) che precede lo Shellcode.
  - E' sufficiente che il RET Address cada in un punto qualsiasi del NOP Sled:




# Shellcoding (5/5)

- Non sempre un codice maligno è iniettabile
  - Tornando all'esempio di prima:

```
mov EBX, 0          BB 00 00 00 00
mov EAX, 1    ⟶     B8 01 00 00 00
int 0x80            CD 80
```

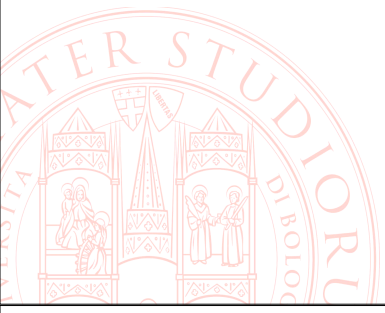
  - Nel risultato esadecimale c'è una serie di byte uguali a 0
  - Il primo di questi verrà interpretato dalle routine di lettura stringhe come **terminatore di stringa**
  - Uno shellcode, per poter essere iniettabile, deve essere sottoposto ad un'operazione di **Zeros Cut-off**
  - Nell'esempio precedente si può agire in questo modo:

NB: AL è l'insieme di 8 bit meno significativi Del registro EAX

```
xor EBX, EBX          31 DB
mov AL, 1     ⟶      B0 01
int 0x80              CD 80
```

# Shellcoding / ASLR

- Una prima contromisura contro l'iniezione di codice maligno è l'**ASLR: Address Space Layout Randomization**
  - E' una protezione offerta dall'OS
  - Si tratta di rendere casuale l'indirizzo di partenza dei segmenti che compongono un processo
  - Questa randomizzazione interessa solo gli indirizzi virtuali di un processo non la sua disposizione in RAM
  - In questo modo l'attaccante non ha modo di trovare un plausibile indirizzo assoluto di ritorno a cui puntare
- Per Linux tale feature è offerta dalla patch grsecurity/PAX che **solo in alcune distribuzioni è inserita nel kernel**



# Shellcoding / NX Stacks

- Una seconda contromisura alla possibilità di eseguire codice maligno dallo stack è fornita dagli **Stack Non Eseguibili**
- E' una feature **fornita dall'HW ma che deve essere supportata dall'OS**
- Alcune architetture permettono di impostare un flag associato a pagine di memoria
  - Tale flag, che deve essere impostato dall'OS, indica se la pagina contiene codice che può essere eseguito o meno
  - Intel commercializza questa feature con il nome di XD bit (eXecution Disable bit) ed è stata introdotta solo a partire dai processori a 64 bit
  - Esistono alcuni OS (Solaris) che implementano tale feature in software, ma ciò causa un leggero overhead

# Shellcoding / NX Stacks

- Tale feature **non è presente sui processori INTEL/AMD a 32 bit**
- **Linux supporta gli stack non eseguibili a partire dal kernel 2.6.8 per architettura x86_64**
  - **Introdotta con la patch PAX/grsecurity**
  - **La versione x86 a 32 bit può sfruttare tale feature solo se in esecuzione su un processore x86_64**
- Un'evoluzione di tale tecnologia è la **W^X**
  - **Piuttosto che marcare una pagina come non eseguibile, ogni pagina viene marcata come Eseguibile (eXecutable) o Scrivibile (Writable) ma non entrambe**
  - **Offre una sicurezza maggiore rispetto al bit XD, che comunque permette di eseguire codice da pagine scrivibili**
  - **In Linux è implementato parzialmente da ExecShield**

# Altre possibilità di attacco

- **RET2LIBC / RET2SYSCALL**
  - **Tramite Stack Overflow si dirotta il flusso di esecuzione, piuttosto che verso codice sullo stack protetto da NX, verso una (o più) funzioni della onnipresente libreria C, oppure verso una system-call del s. o.**
- **Format Strings**
  - **Tramite la stringa di formato passata a printf si inietta codice e si forza il salto che lo esegue**
- **Heap Overflow**
  - **Si sfruttano vulnerabilità specifiche dei metadati inseriti dalle librerie C per l'allocazione dinamica di memoria**
- **Return Oriented Programming**
  - **Si assembla il codice da eseguire come sequenza di "gadget" (brevi sequenze di linguaggio macchina prese dai binari già in memoria)**

# Rootkit (1/2)

- Gli attacchi mostrati in precedenza consentono ad un intruso di controllare un sistema
  - Tale controllo può avvenire ad es. con una shell con i diritti del proprietario del processo attaccato
- Il problema dell'attaccante è nascondere la propria presenza
- Per Rootkit si intende proprio un sistema software in grado di nascondere la compromissione di un sistema

# Rootkit (2/2)

- Tipologie di rootkit
  - Application level
    - Trojan, sostituzione di utility di sistema o demoni con versioni contenenti backdoor
  - Kenel level
    - Moduli kernel o driver maligni che consentono di controllare (anche a livello hardware tramite i device driver) la macchina attaccata o di sostituire le system call del sistema operativo (ad esempio per nascondere tra i file presenti in una directory quelli maligni o tra i processi in esecuzione i trojan installati)
  - Library Level
    - Sostituzione di librerie con una propria versione

# Rootkit – rilevazione

- I rootkit hanno l'obiettivo di mascherare l'attacco
  - La rilevazione della presenza del rootkit è un'operazione difficile poiché deve far uso degli stessi strumenti di diagnostica che il rootkit ha alterato (questo perchè tipicamente la ricerca dei rootkit non dovrebbe comportare lo spegnimento e l'eventuale reinstallazione dell'OS).
  - Esistono varie tecniche di rilevazione
    - Ricerca euristica: tipica degli antivirus, consiste nella ricerca di pattern binari ricorrenti nei processi maligni
    - Verifica della memoria di massa del sistema effettuando il boot da un OS pulito (ad es. tramite una distribuzione live). Tale soluzione è indicata per rootkit di livello kernel