Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
15
Caccia al bug/Processi/System Calls

Codici sorgenti

- Scompattiamo l'archivio di codici sorgenti che troviamo sotto /tmp/caccia_al_bug.tgz con il comando:

  $ tar xvfz /tmp/caccia_al_bug.tgz

- Nella directory caccia_al_bug troviamo i file d'esempio di questa lezione
- È possibile scaricare l'archivio anche da

  http://www.na.icar.cnr.it/~oliva/labos-2011/caccia_al_bug.tgz

esempio1.c

- Come funziona il seguente codice C ?

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int x,y;
  x=0;
  printf("Inserisci il primo numero: ");
  scanf("%d",&x);
  printf("Inserisci il secondo numero: ");
  scanf("%d",&y);
  if ( x = y )
    printf("Uguali\n");
  else if ( x > y )
    printf("Decrescenti\n");
  else
    printf("Crescenti\n");
  exit(0);
}
```

esempio1.c

- Compiliamo il codice d'esempio con il comando

  $ gcc -o esempio1 esempio1.c

- Eseguiamo il programma, provando ad inserire diversi interi e osserviamo come (non) funziona

esempio1.c

- La condizione dell'if è implementata utilizzando l'operatore di assegnazione “=” invece dell'operatore di confronto “==”
- L'operatore “=” assegna il valore alla variabile x e restituisce il valore assegnato come valore di ritorno dell'operazione effettuata
- Questo fa sì che la verifica della condizione abbia successo a seconda che y sia positivo o nullo

esempio1.c

- La versione corretta del codice C è la seguente:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int x,y;
  x=0;
  printf("Inserisci il primo numero: ");
  scanf("%d",&x);
  printf("Inserisci il secondo numero: ");
  scanf("%d",&y);
  if ( x == y )
    printf("Uguali\n");
  else if ( x > y )
    printf("Decrescenti\n");
  else
    printf("Crescenti\n");
  exit(0);
}
```

esempio2.c

- Il seguente programma compilato termina con errore. Perche'?

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int x;
	printf("Inserisci un intero: ");
	scanf("%d", x);
	printf("Hai inserito %d\n", x);
	exit(0);
}
```

esempio2.c

- L'errore generato sui sistemi Unix e' Segmentation Fault (segfault)
- L'errore ha luogo quando un programma tenta di accedere ad una locazione di memoria alla quale non gli è permesso accedere, oppure quando tenta di accedervi in una maniera che non gli è concessa (ad esempio in scrittura su memoria read-only)
- Questo tipo di errore è dovuto, in genere, all'uso improprio dei puntatori
- In questo caso scanf si aspetta un indirizzo di memoria su cui scrivere il dato inserito dall'utente ma riceve il contenuto di x

esempio2.c

- La versione corretta del programma è la seguente:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int x;
	printf("Inserisci un intero: ");
	scanf("%d", &x);
	printf("Hai inserito %d\n", x);
	exit(0);
}
```

L'opzione -Wall

- L'opzione -Wall utilizzata in fase di compilazione abilita la visualizzazione di tutti i warning e spesso consente di rilevare alcuni casi di sintassi lecite, ma sospette
- Utilizziamo tale opzione per compilare il file esempio1.c e visualizzeremo il messaggio
  warning: suggest parentheses around assignment used as truth value
- Utilizziamo tale opzione per compilare il file esempio2.c e visualizzeremo il messaggio
  warning: format ‘%d’ expects type ‘int *’, but argument 2 has type ‘int’
- L'opzione -Wall è vostra amica: usatela sempre!

esempio3.c

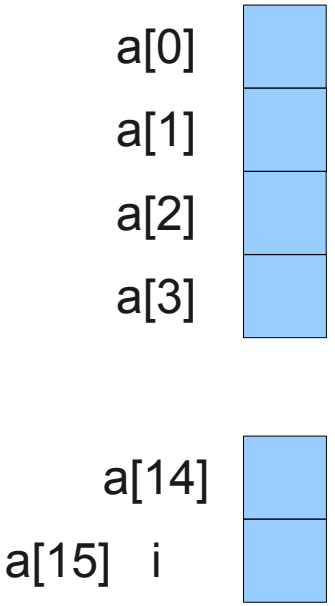
- Qual e' l'errore in questo codice ?
- ```
  #include <stdio.h>
  #include <stdlib.h>
  int main()
  {
    int a[15], i;
    for(i=1; i<=15; i++)
      a[i] = 0;
    for(i=1; i<=15; i++)
      printf("a[%d]=%d\n", i, a[i]);
    exit(0);
  }
  ```
- Cosa succede quando eseguiamo il codice?
  ```

esempio3.c

- Gli indici di un array partono da zero pertanto gli elementi in un array a con 10 elementi sono:

  a[0], a[1], ... a[14]

- L'aria di memoria indirizzata con a[15] non è contenuta nell'array a e quindi anche se non abbiamo errori di esecuzione o di compilazione stiamo scrivendo nell'area di memoria destinata ad i



esempio4.c

- Qual e' l'errore in questo codice ?
- ```
  #include <stdio.h>
  #include <string.h>
  int main(int argc, char *argv[])
  {
     char *stringa;
     strcpy(stringa, argv[1]);
     printf("%s", stringa);
     return 0;
  }
  ```
  ```

esempio4.c

- Dichiarare una variabile puntatore non alloca lo spazio necessario per la memorizzazione
- ```
  #include <stdio.h>
  #include <string.h>
  int main(int argc, char *argv[])
  {
      char stringa[30];
      strcpy(stringa, argv[1]);
      printf("%s", stringa);
      return 0;
  }
  ```
- Ma c'è un nuovo problema in questo codice, quale?
  ```

esempio4.c

- La stringa passata come argomento potrebbe essere più lunga di 30 caratteri quindi nel copiarla all'interno dell'array stringa[30] potremo sforare l'area di memoria passata all'array stringa
- ```
  #include <stdio.h>
  #include <string.h>
  int main(int argc, char *argv[])
  {
     char stringa[30];
     strncpy(stringa, argv[1],30);
     printf("%s", stringa);
     return 0;
  }
  ```
- Dalla pagina di manuale di strcpy:
- If the destination string of a strcpy() is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine.
  ```

Breve riepilogo sui puntatori

- Un puntatore è una variabile che memorizza un indirizzo di memoria

  di una variabile, di un elemento di un array, di una funzione, ...
- La dichiarazione di un puntatore avviene utilizzando la sintassi:

  tipo *nome;
- dove il tipo può essere nativo o derivato (struct, enum, union);
- L'operatore & preposto al nome di una variabile (&var) ci consente di ottenere l'indirizzo di una variabile scalare mentre per gli array è sufficiente il nome
- L'operatore * si utilizza a indirizzare lo spazio puntato: l'assegnazione *p=val, assegna alla variabile puntata da p il valore val.
- Come abbiamo visto in esempio3.c la dichiarazione di un puntatore non alloca lo spazio per contenere il valore della variabile
- Un puntatore, quindi, deve essere utilizzato:
  - per referenziare una area di memoria allocata staticamente
  - per referenziare una area di memoria allocata dinamicamente

Aritmetica dei puntatori

- Se p è un puntatore ad una variabile che occupa n byte, allora p+1 corrisponde al valore di p incrementato di n
- Si supponga di avere un array di caratteri memorizzato a partire dalla locazione `0xbffff406`
- Se cp è un puntatore a caratteri che punta al primo elemento dell'array (contiene `0xbffff406`) allora cp+1 punterà all'indirizzo `0xbffff406+1=0xbffff407` poiché ogni carattere occupa 1 byte in memoria
- Se invece supponiamo di avere un array di interi memorizzato a partire dalla stessa locazione ed un puntatore ip ad interi allora ip+1 punterà all'indirizzo `0xbffff406+4=0xbffff410` poiché ogni intero occupa 4 byte in memoria

Elementi di un array

- Dal momento che gli elementi di un array sono memorizzati in sequenza, l'aritmetica dei puntatori favorisce un accesso semplice agli indirizzi degli elementi
- Elementi a[n], *(a+n) Indirizzi &(a[n]), a+n
- Si tenga presente che il C non effettua nessun controllo sulla validità dell'indirizzo di un puntatore, quindi un puntatore non correttamente inizializzato, oppure erroneamente assegnato, potrebbe modificare o provare a modificare un'area di memoria qualsiasi all'interno dello spazio di indirizzamento del nostro processo

Esempio

- Qual'e' il problema di questa funzione?
- char *messaggio ()
  {
    char r[]=“Benvenuti”;

    …

    return r;

  }

Esempio

- Qual'e' il problema di questa funzione?
- char *messaggio ()
  {
    char r[]=“Benvenuti”;

    …

    return r;

  }
- r è una variabile automatica che esiste soltanto durante l'esecuzione della funzione

esempio5.c

- Qual'è l'errore in questo codice?
- ```
  #include <stdio.h>
  #include <stdlib.h>
  void swap(int a, int b) {
       int temp;
       temp=a;
       a=b;
       b=temp;
       return;
  }
  int main () {
     int uno;
     int due;
     printf("Inserisci il primo numero: ");
     scanf ("%d",&uno);
     printf("Inserisci il secondo numero: ");
     scanf ("%d",&due);
     swap(uno,due);
     printf("primo=%d secondo=%d\n",uno,due);
     exit(0);
  }
  ```
  ```

esempio5.c

- Il passaggio dei parametri avviene esclusivamente per valore: eventuali modifiche all'interno della funzione NON hanno effetto al suo esterno perché sono effettuate su una copia delle variabili passate come argomento
- Per simulare un passaggio per riferimento si utilizzano i puntatori in questo modo vengono passati alle funzioni gli indirizzi delle variabili ed eventuali modifiche all'interno della funzione hanno effetto sul valore della variabile nella funzione chiamante

esempio5.c

- La versione giusta del codice è:
- ```
  #include <stdio.h>
  #include <stdlib.h>
  void swap(int *a, int *b) {
       int temp;
       temp=*a;
       *a=*b;
       *b=temp;
       return;
  }
  int main () {
      int uno;
      int due;
      printf("Inserisci il primo numero: ");
      scanf ("%d",&uno);
      printf("Inserisci il secondo numero: ");
      scanf ("%d",&due);
      swap(&uno,&due);
      printf("primo=%d secondo=%d\n",uno,due);
      exit(0);
  }
  ```
  ```

esempio6.c

- Cosa fa il seguente programma?

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    printf("Hello world!");
    sleep(5);
    exit(0);
}
```

I/O buffering

- Le funzioni della liberia di I/O standard utilizzano il buffering
- Un buffer è un'area di memoria per memorizzare le informazioni prima di inviarle al device
- L'utilizzo di un buffer riduce il numero di system call “effettive” e migliora le performance
- La scrittura su standard output ha una politica di buffering a linea: le operazioni di I/O avvengono non appena viene inserito nel buffer un carattere newline '\n'
- Vale sia per printf nell'output sul terminale che da scanf nell'input da tastiera

esempio6.c

- Il seguente codice si comporterà, quindi, come atteso:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   printf("Hello world!\n");
   sleep(5);
   exit(0);
}
```