Programmazione sicura

Laboratorio 2

Considerazioni sugli esercizi precedenti

C Memory layout in pratica

Espressioni regolari in C

Esercizi lab_1 1/2

- S01.c
  - Division by zero. Esempio banale, facile da testare su linea di comando.
- S02.c
  - Assegnazione invece di confronto.
  - Che valore assume (intRand = 5)?
  - Quando si entra nell'if?
  - Se la condizione fosse stata (intRand = 0)?
  - Provate printf(“%d”, (i = 2));

Esercizi lab_1 2/2

- S03.c
  - Divizione per zero.
  - Simile a s01.c, ma col %
- S04.c
  - Confronto invece di assegnazione.
- S05.c
  - Divisione per zero, ma il valore è casuale, bisogna creare un ciclo while(1) per forzare la condizione di errore.
- S06.c
  - Assegnazione invece di confronto
  - Il ciclo for è finto, viene eseguit una sola volta.
  - L'if viene sempre eseguito in quanto l'espressione ha valore 5.
  - Il risultato stampato a schermo è nella maggior parte dei casi errato.

C Memory Layout 1/2

- Come ottenere semplicemente informazioni sull'organizzazione della memoria di un programma C?
- La memoria è sostanzialmente occupata da codice e dati.
- Per ciò che concerne le aree di memoria utilizzate, i dati sono classificati in base ai qualificatori di tipo (const, static), lo scope (globale o locale a una funzione), il fatto di usare malloc o alloca e alcune convenzioni che riguardano le architetture delle CPU (valori di ritorno di una funzione)
- Aprire sorgente lab_2_ mem_layout.c

C Memory layout 2/2

- Tramite printf stampiamo informazioni sulle variabili allocate.
- La conversionie %p stampa un indirizzo in esadecimale, ai puntatori bisognerebbe fare il cast a (void *). Il cast non c'è per chiarezza, in quanto gli indirizzi sono sempre della stessa dimensione (32 o 64 bit).
- Provate a costruire un grafico con gli indirizzi di memoria, identificate le aree coinvolte.
- Provate ad aggiungere variabili e stringhe col qualificatore “const”.
- Sia globali che locali.
- In quale area di memoria si trovano?

Indirect selection

- Scrivete l'esempio 5.9 del libro di testo.
- Installate i giochi col comando:
  - `sudo apt-get install bsdgames fortune`
  - I giochi vengono installati in /usr/games, aggiungetene altri alla lista di giochi supportati.
  - Alcuni di quelli presenti sul libro non sono presenti.
- Per i dettagli sulla funzione execlp: `man execl`
- Aggiungete il file .c al makefile, compilatelo ed eseguitelo.
- Provate a dare in input comandi non previsti che forzino comportamenti non voluti.
- Eseguite un check con *Flawfinder* includendolo nel makefile.

Espressioni regolari

- PCRE (Perl Compatible Regular Expression) è una libreria in linguaggio C per utilizzare le regular expression nei propri programmi.
- Tramite wrapper viene implementata in numerosi linguaggi. PHP ad esempio ha un wrapper per PCRE.
- Non molto ben documentata.
- I pacchetti necessari risultano già installati nella nostra macchina virtuale. Possiamo usarli includendo semplicemente gli headerfile e le librerie per il linker.
- Nella nostra macchina virtuale sono presenti le man pages per tutte le funzioni e per la grammatica dei pattern.

PCRE (Perl Compatible Regular Expression)

- Useremo le funzioni:
  - pcre_compile
  - pcre_exec
- Esempio *lab_1_hello_pcre.c*
  - Aggiungerlo al makefile e compilarlo
- Farlo girare, provate a variare il pattern.
- `man pcre_compile`
- `man pcre_exec`
- Nei sorgenti della lezione di oggi è incluso lab_2_pcredemo.c, un esempio d'uso esteso delle PCRE. Potete studiarlo.

PCRE (Perl Compatible Regular Expression)

- `pcre *pcre_compile(const char *pattern, int options, const char **errptr, int *erroffset, const unsigned char *tableptr)`
- pattern: stringa che rappresenta l'espressione regolare secondo lasintassi pcre.
- options: opzioni per modificare il comportamento del match, vanno combinate con l'OR (|). Esempio: PCRE_UNGREEDY | PCRE_DOTALL
- **errptr: l'indirizzo di un puntatore in cui salvare un messaggio di errore.
- *erroffset: la posizione nel pattern in cui l'errore è stato trovato.
- *tableptr: contiene una tabella che codifica i caratteri. Esiste una utility nei sorgenti di PCRE per crearla, dftables. Di solito NULL.

PCRE (Perl Compatible Regular Expression)

- `int pcre_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize);`
- *code: oggetto PCRE resetituito da pcre_compile.
- *extra: oggetto pcre_extra, si ottiene con pcre_study, serve a velocizzare, se possibile, l'esecuzione in molti utilizzi successivi di un oggetto pcre.
- *subject: la stringa su cui eseguiamo il match.
- length: la lunghezza di subject (\0 non è fine stringa).
- startoffset: la posizione da cui iniziare il match.
- options: flag combinati in OR simili a pcre_compile. Man page per i dettagli.
- *ovector: sottostringhe catturate durante il match. Ovecsize è la dimensione di ovector, multipli di 3, triple composte da (start, end, ?).
- ???Nella demo ovec usato in coppie???.

Espressioni Regolari - Esercizio

- Scrivete un programma in C, sulla falsa riga dell'esempio 5.12 del libro di testo.
- Il ciclo principale deve leggere continuamente da input una linea di testo e matcharla con 3 espressioni regolari.
- Le 3 espressioni devono metchare un codice fiscale, una data e una email.
- In base al match che va a buon fine il programma stampa il tipo di input inserito (data, CF o email).

Espressioni Regolari – Esercizio 2

- Usate fgets per l'input del testo. `man fgets` per i dettagli.
- L'input di fgets include il newline, rimuovetelo prima di fare il match e aggiustate la lunghezza della stringa di conseguenza.
- Pagine di manuale utili: fgets, pcre_compile, pcre_exec, pcrepattern, pcre
- Compilare il programma come segue:

```
gcc -o main main.c -lpcre
```

- Iniziate con il match delle date, fatelo funzionare, poi aggiungete gli altri.

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

int main(int argc, char* argv[]) {
    pcre* re_email;
    pcre* re_cf;
    pcre* re_date;
    const char* err;
    int   errOffset;
    char  *regex_date = "^[0-9]{2}/[0-9]{2}/[0-9]{4}$";
    char  *regex_cf = "^[a-zA-Z]{6}[0-9]{2}[a-zA-Z][0-9]{2}[a-zA-Z][0-9]{3}[a-zA-Z]$";
    char  *regex_email = "^[0-9a-zA-Z._]{2,}@[0-9a-zA-Z._]*.[a-zA-Z]{2,3}$";
    int   i;
    char  buff[256];

    re_email = pcre_compile(regex_email, 0, &err, &errOffset, NULL);
    if (re_email == NULL) {
        printf("PCRE email compilation failed\n");
        return 1;
    }

    re_cf = pcre_compile(regex_cf, 0, &err, &errOffset, NULL);
    if (re_cf == NULL) {
        printf("PCRE CF compilation failed\n");
        return 1;
    }

    re_date = pcre_compile(regex_date, 0, &err, &errOffset, NULL);
    if (re_date == NULL) {
        printf("PCRE date compilation failed\n");
        return 1;
    }

    while (1){
        printf("type ->");
        fgets(buff, 256, stdin);
        int len = strlen(buff);
        buff[len-1] = '\0'; //elimino il neewline
        len--;
        int match = 0;

        int rc = pcre_exec(re_email, NULL, buff, len, 0, 0, NULL, 0);
        if (rc >= 0) {
            printf("%s: is email\n", buff);
            match++;
        }

        rc = pcre_exec(re_cf, NULL, buff, len, 0, 0, NULL, 0);
        if (rc >= 0) {
            printf("%s: is fiscal code\n", buff);
            match++;
        }

        rc = pcre_exec(re_date, NULL, buff, len, 0, 0, NULL, 0);
        if (rc >= 0) {
            printf("%s: is date\n", buff);
            match++;
        }

        if (!match){
            printf("%s: invalid input\n", buff);
        }
        match = 0;
    }
}
```