::. s0ftpj .::. HACKIT 03 .::. quest .::

:: La programmazione sikura secondo djb ::

PREMESSA

E' comune sentir dire, e, considerare qmail e tutti gli altri programmi di djb "sicuri". Sicuri poiche' si dice che djb si sia riscritto tutte le funzioni delle libc; sicuri perche' di sice che djb si sia reimplementato tutte le funzoni che manipolano le stringhe. Ed ancora per una qualche strana alchimia i suoi programmi risulterebbero essere anche molto performanti.
Questo speech vuole essere una introduzione alla programmazione secondo i canoni di djb alla scoperta delle leggende metropolitane chi si tramandano sui programmi di questo autore.

0: Premessa Operativa

Djb ha un modo di scrivere sporco, poco elegante dal punto di vista sintattico, ma semanticamene efficiente.
Iniziamo con l'osservare un suo classico sorgente (abbiamo scelto per i nostri esempi i src di ezmlm gestore di ml per qmail)

```
#include "stralloc.h"
#include "subfd.h"
#include "strerr.h"
#include "error.h"
#include "qmail.h"
#include "env.h"
#include "lock.h"
#include "sig.h"
#include "open.h"
#include "getln.h"
#include "case.h"
#include "scan.h"
#include "str.h"
#include "fmt.h"
...
```

La prima cosa che salta all'occhio e' il gran numero di include che vediamo dichiarati in testa al sorgente. Questo e' dovuto ad un tipico modo che djb ha di approcciarsi alla programmazione. Egli tende a separare le singole funzioni, o gruppi di funzioni

correlate in singoli file, ricordando uno stile di programmazione molto simile all'asm, dove per i programmi articolati si tende a scrivere poche procedure per file.
Cosi' facendo se noi dovessimo incontrare la funzione getln, al 90% esistera' un file getln.c che contine la suddetta funzione.

```
[belial:~/tmp/ezmlm-0.53]
$ cat getln.c
#include "substdio.h"
#include "byte.h"
#include "stralloc.h"
#include "getln.h"

int getln(ss,sa,match,sep)
register substdio *ss;
register stralloc *sa;
int *match;
int sep;
{
  char *cont;
  unsigned int clen;

        if (getln2(ss,sa,&cont,&clen,sep) == -1) return -1;
        if (!clen) { *match = 0; return 0; }
        if (!stralloc_catb(sa,cont,clen)) return -1;
        *match = 1;
        return 0;
}
```

Senza entrare ora nel merito della funzione notiamo subito che questa e' dichiarata con la specifica del tipo degli argomenti tra il testo ed il corpo della funzione.

Al momento della compilazione ogni singolo file verra' compilato singolarmente e trasformato in libreria statica che poi sara' a sua volta linkato al 'main'.
Questo attravero tre script:

```
[belial:~/tmp/ezmlm-0.53]
$ cat compile
#!/bin/sh
# WARNING: This file was auto-generated. Do not edit!
exec cc -O2 -c ${1+"$@"}
```

```
[belial:~/tmp/ezmlm-0.53]
$ cat makelib
#!/bin/sh
# WARNING: This file was auto-generated. Do not edit!
main="$1"; shift
rm -f "$main"
ar cr "$main" ${1+"$@"}
ranlib "$main"
```

```
belial:~/tmp/ezmlm-0.53]
$ cat load
#!/bin/sh
# WARNING: This file was auto-generated. Do not edit!
main="$1"; shift
exec cc -s -o "$main" "$main".o ${1+"$@"}
```

Script molto banali, ma per comprenderne meglio l'uso vi invito a leggere il Makefile.

1: Il fondamento di tutto

Due strutture portanti si possono riconoscere come pilastri della programmazione secondo djb:

struct stralloc

```
[belial:~/tmp/ezmlm-0.53]
$ head stralloc.h
#ifndef STRALLOC_H
#define STRALLOC_H

#include "gen_alloc.h"

GEN_ALLOC_typedef(stralloc,char,s,len,a)
```

La macro GEN_ALLOC_typedef genera in realta' la struct stralloc

```
[belial:~/tmp/ezmlm-0.53]
$ head gen_alloc.h
#ifndef GEN_ALLOC_H
#define GEN_ALLOC_H

#define GEN_ALLOC_typedef(ta,type,field,len,a) \
  typedef struct ta { type *field; unsigned int len; unsigned
int a; } ta;

#endif
```

per cui la nostra struct stralloc sara' cosi composta:

```
typedef struct stralloc {
	/* puntatore ad una stringa o 0 se non e' ancora
allocata */
	char *s;
	/* sono il numero di byte della stringa puntata da s se
	 * questa e' allocata */
		int len;
	/* i numero di byte allocate per la stringa s */
	unsigned int a;
} stralloc;
```

```
[belial:~/tmp/ezmlm-0.53]
$ head substdio.h
#ifndef SUBSTDIO_H
#define SUBSTDIO_H

typedef struct substdio {
  /* e' un puntatore ad un array di caratteri */
  char *x;
  /* quato e' occupato di x */
  int p;
  /* lunghezza di x */
  int n;
  /* file descriptor */
  int  fd;
  /* e' un puntatore ad una funzione che deve essere
   * invocata come op(fd, x, n) */
  int (*op)();
} substdio;
```

2: stralloc - dynamically allocated strings

Stralloc rappresenta una stringa dinamicamente allocata in memoria; la lunghezza della stringa e' limitata solo dalla grandezza della memoria;

La struct stralloc, come abbiamo gia' visto, e' composta da tre elementi:
char *s e' un puntatore ad una stringa, o 0 se questa non e' ancora allocata;
int len e' la lunghezza della stringa s se questa e' allocata;
unsigned int a e' il numero dei byte effettivamente allocati per la stirnga se questa e' allocata.

Una struct stralloc viene inizializzata come :

```
struct stralloc sa = {0}; /* una struc stralloc ancora non allocata */
```

3: stralloc - Operazioni

Diamo nello sguardo nel dettaglio ad funzioni sulle stralloc:

int stralloc_ready(&sa,len);
stralloc sa;
int len;

Questo funzione si assicura che nella stralloc sa, ci sia sufficientemente spazio allocato per len caratteri. Se cosi' non fosse lo alloca.

Uno sguardo alla funzioneche si trova dentro il file gen_allocdefs.h :

```
#define
GEN_ALLOC_ready(ta,type,field,len,a,i,n,x,base,ta_ready) \
int ta_ready(x,n) register ta *x; register unsigned int n; \
{ register unsigned int i; \
  if (x->field) { \
    i = x->a; \
    if (n > i) { \
      x->a = base + n + (n >> 3); \
      if (alloc_re(&x->field,i * sizeof(type),x->a * sizeof(type))) return
1; \
      x->a = i; return 0; } \
    return 1; } \
  x->len = 0; \
  return !!(x->field = (type *) alloc((x->a = n) * sizeof(type))); }
```

dentro il file stralloc_eady.c la troviamo poi cosi' istanziata:

```
GEN_ALLOC_ready(stralloc,char,s,len,a,i,n,x,30,stralloc_ready)
```

il che trasforma la nostra funzione in:

```
/* controlla se c'e' n spazio dentro x */
int stralloc_ready(x, n) register stralloc x; unsigned int n;
{
        register unsigned int i;
        /* se la stringa e' gia allocata */
        if (x->s) {
/* i uguale alla lunghezza di quanto e' effettivamente allocato */
                i = x->a;
                /* n e' maggiore dello spazio gia' allocato allora */
                if (n > i) {
/* settiamo a, contatore dello spazio allocato basa + n + (n >> 3) */
                        x->a = 30 + n + (n >> 3);
        /* dopo di che si realloca la stringa s di a spazio per il tipo */
                        if(alloc_re(&x->s, i * sizeof(char), x->a *
                                                        sizeof(char))){

                                /* se la cosa riesce torna 1 */
                                return 1;
                        }
                        /* altrimenti ripristina a ed esci */
                        x->a = i;
                        return 0;
                }
/* se n e' minore dello spazio gia' allocato e disponibile torna 1; */
                return 1;
        }
        /* la la stralloc non era stata ancora allocata allocala. */
        x->len 0;
        return !!(x->s = (char *) alloc(x->a = n) * sizeof(char)));
}
```

Questa semplice fuzione di verifica, allocazione e riallocazione di un buffer di memoria come vedremo viene invocata da tutte le altre funzioni che lavorano sulle stralloc come un sorta di preambolo o check dello spazio dispobile; ed e' il "vero cuore della sicurazza nell'uso delle stringhe" per djb.
Controllare sempre che sia allocato lo spazio necessario prima di una qualsiasi operazione e caso mai allocarlo o riallocarlo fa si che difficilmente si incorrera' in buffer overflow.

Un altra funzione che possiamo considerare cardine nell'uso delle stringhe e' :

```
int stralloc_copyb(&sa,buf,len);
```

Questa funzione copia len caratteri dal buffer buf alla stralloc sa. Quasi ogni altra funzione che lavora sulle stralloc viene richiamata a questa, che possiamo leggere dentro il file stralloc_opyb.c :

```
int stralloc_copyb(sa,s,n)
stralloc *sa;
char *s;
unsigned int n;
{
/* se la stringa e' pronta continua altrimenti esci tornando
 * 0. */
  if (!stralloc_ready(sa,n + 1)) return 0;
  /* copya n byte dal buffer s al buffer della stralloc. */
  byte_copy(sa->s,n,s);
```

```
  /*imposta la lunghezza della buffer usato dentro la
   * stralloc. */
  sa->len = n;
  /* metti un segno sull'ultimo byte della stringa allocata.*/
  sa->s[n] = 'Z'; /* ``offensive programming" */
  return 1;
}
```

Ancora importante per l'utilizzao delle stringhe sono le funzioni di append o concatenazione; un esempio tipico e' la

```
int stralloc_catb(&sa,buf,len);
```

Questa funziona che prende come argomenti una puntatore ad una struct stralloc sa, un putatore ad un buffer buf, e un intero len, appende, o, concatena alla stralloc sa quella contenuta dentro buf, per len caratteri.

Eseminiamo piu' nel dettaglio la funzione che si trova dentro stralloc_catb.c :

```
int stralloc_catb(sa,s,n)
stralloc *sa;
char *s;
unsigned int n;
{
  /* se il buffer della stralloc non e' allocato ci possiamo
   * rifare alla stralloc_copy, in quanto cominciamo
   * "dall'inizio". */
  if (!sa->s) return stralloc_copyb(sa,s,n);
```

```
  /* la stralloc_readyplus controlla che la stralloc abbia
   * spazio per n caratteri oltre quelli gia' allocati. Se la
   * stralloc non e' allocata si comporta esattamente come
    * una stralloc_ready(); */
  if (!stralloc_readyplus(sa,n + 1)) return 0;
  /* una volta verificato il buffer e la sua grandeza
   * possiamo appendere la nostra stringa copiando da
   * sa->s (il buffer) + sa->len (la lunghezza della stringa
    * sa->s) il buffer s per n byte. */
  byte_copy(sa->s + sa->len,n,s);
  /* aggiorniamo la lunghezza di sa->s che sara'
   * sa->len + n appunto. */
  sa->len += n;
  /* mettiamo come primaun segnale nell'ultimo byte dopo
    * la stringa sa->s. */
  sa->s[sa->len] = 'Z'; /* ``offensive programming'' */
  return 1;
}
```

Questo e' quanto bisogna sapere per poter leggere con cognizione di causa il codice di djb relativo all'uso delle stringhe. Quache esempio pratico lo vedremo in seguito andando ad analizzare alcuni esempi di programmazione tratti da programmi del 'nostro eroe'.

4: substdio - l'input output

Sub-Standard I/O Library, o substdio, contiene un po di funzioni base o meglio un metodo operativo veloce e flessibile per lavorare con lo Standard I/O.

Il componente basilare del substdio, come abbiamo gia' visto, e' la struct substdio essa contiene:

int (*op) (); 'un operazione', per leggere o scrivere dentro il buffer di I/O (stdio lavora in buffered mode);
int fd un file descriptor, indice del file che stiamo leggendo o scrivendo;
char *x un puntatore ad un buffer non nullo;
int n la lunghezza di del buffer puntato da x;
int p quanto e' effettivamente occupato del buffer puntato da x;

Un volta dichiarata un substdio, essa viene inizializzata attraverso la funzione:

```
void substdio_fdbuf(&s,op,fd,buf,len);
```

che troviamo dentro il file substdio.c

```
void substdio_fdbuf(s,op,fd,buf,len)
register substdio *s;
register int (*op)();
register int fd;
register char *buf;
register int len;
{
  s->x = buf;
  s->fd = fd;
  s->op = op;
  s->p = 0;
  s->n = len;
}
```

Come possiamo vedere questa funzione e' banale, fa solo un opera di "associazione" di alcuni elmenti inerenti alle operazioni di I/O all'interno di una struttura. Djb, tende ad allocare come variabili globali tutti gli elementi che andranno a "comporre" la struct.

op in particolare e' un puntatore ad una funzione che viene invocata come:

op(fd, x, n)

dove x e' un buffer di lunghezza n; op leggera' da fd e scrivera' sul buffer, o leggera' dal buffer e scrivera' su fd,

o semplicemente flushera' il buffer se e' necessario, o, utilizzera' semplicemente l'operazione op sul descrittore fd.

5: substdio - le operazioni di out

Le due funzioni cardini delle operazioni di output sono:

```
int substdio_put(&s,from,len);
int substdio_bput(&s,from,len);
```

dove s e' un puntatore ad una struct substdio, from e' un puntatore ad un buffer, len e' un intero.

Entrambe le funzioni scrivono len characters in s, usando il fd e l'operazione in esso descritti, dal buffer puntato da from.

Le due funzioni differenziano nel flushing del buffer associato ad s quando questo si riempie:
substdio_put flusha il buffer prima prima di scrivere dati nuovi, al contrario substdio_bput prima scrive e solo dopo flusha.

Queste due funzioni sono descritte dentro il file substdo.c :

```
int substdio_put(s,buf,len)
register substdio *s;
register char *buf;
register int len;
{
  register int n;

  /* poniamo n uguale alla lunghezza del buffer associato alla
   * nostra struct substdio s. */
  n = s->n;
  /* se len, la grandezza di buf, e' maggiore della differenza tra il
   * buffer associato, n, e lo spazio gia' utilizzato, s->p */
  if (len > n - s->p) {
    /* allora flusha il buffer associato, in modo da liberarlo */
    if (substdio_flush(s) == -1) return -1;
    /* now s->p == 0 */
   /* se n e' minore DI SUBSTDIO_OUTSIZE allora poni n
    * uguale a SUBSTDIO_OUTSIZE */
    if ( n < SUBSTDIO_OUTSIZE ) n = SUBSTDIO_OUTSIZE;
   /* finche' la lunghezza del buffer puntato da buf, len, e'
    * maggiore della  grandezza del buffer associato */
    while (len > s->n) {
         /* se n e' maggiore di len, allora poni n = len. */
      if (n > len) n = len;
         /* chiama allwrite vero cuore delle operazioni di out. */
      if (allwrite(s->op,s->fd,buf,n) == -1) return -1;
```

```
        /* aggiorana buf e len dopo la scrittura. */
      buf += n;
      len -= n;
    }
  }
  /* now len <= s->n - s->p */
  /* quindi scriviamo semplicemente sul buffer associato. */
  byte_copy(s->x + s->p,len,buf);
  /* aggiorniamo il contatore dello spazio utilizzato. */
  s->p += len;
  return 0;
}


int substdio_bput(s,buf,len)
register substdio *s;
register char *buf;
register int len;
{
  register int n;

 /* finche' la grandezza del buffer buf, len e' maggiore della
  * differenza tra la grandzza del buffer associato, s->n, e
  * lo spazio dello stesso effettivametne utilizzato s->p,
  * allora... */
  while (len > (n = s->n - s->p)) {
       /*copia dentro il buffer associato n byte di buf. */
    byte_copy(s->x + s->p,n,buf);
```

```
    s->p += n;
    buf += n;
    len -= n;
    /* flusha la struct substdio s. */
    if (substdio_flush(s) == -1) return -1;
  }
  /* now len <= s->n - s->p */
  /* copia dentro il buffer associato, s->x, len byte di buf. */
  byte_copy(s->x + s->p,len,buf);
  /*  aggiorana il contatore dello spazio utilizzato dentro la
   * struct substdio s */
  s->p += len;
  return 0;
}


static int allwrite(op,fd,buf,len)
register int (*op)();
register int fd;
register char *buf;
register int len;
{
  register int w;

  /* finche len != 0 */
  while (len) {
       /*esegui op */
    w = op(fd,buf,len);
       /* se op ha dato errore */
    if (w == -1) {
```

```
        /* se errno e' uguale a error_intr */
      if (errno == error_intr) continue;
        /* altrimenti torna -1 */
      return -1; /* note that some data may have been
written */
    }
    if (w == 0) ; /* luser's fault */
      /* aggiorna buf e len */
    buf += w;
    len -= w;
  }
  // se tutto e' andato bene ritorniamo 0
  return 0;
}
```

6: substdio - le operazioni di in

Le operazioni di input sono analoghe a quelle di output.

7: il codice di ezmlm

Quello di cui finora 'astrattamente' abbiamo parlato, lo vediamo applicato sul codice di ezmlm, il gestore di ml per qmail. In particolare esaminiamo il codice di ezmlm-send.c

torino :: hackit03 :: quest <at> s0ftpj <dot> org

8: ringraziamenti in ordine sparso

s0ftpj, freaknet, dyne, brigata giuoconda, autistici, bfi, asbesto, lobo, smaster, antani, valvoline, nail, vecna, nextie, smilzo, martin, naif, koba, fusys, vodka, l3chuck, newmark, magone, sandman, blicero, mr. ortomio, cyberz, scai, tripz, z0rz0rz, && others...

9: links && riferimenti vari

http://www.hackmeeting.org
http://www.s0ftpj.org
http://cr.yp.to
http://www.freaknet.org
http://www.dyne.org