Scusate la memcmp – considerazioni sull'uso della memcmp(3) in C (e C++)

Tonino: *Vincè, io mi uccido: meglio un giorno da leone che cento giorni da pecora! O no Vincè? Meglio un giorno da leone!*
Vincenzo: *Tonì, che ne saccio io d''a pècura o d''o leone? Fa' cinquanta juórne da orsacchiotto.*

Nonostante l'intenzione *(giuro)* di scrivere un altro articolo non propriamente di programmazione (come ho fatto negli gli ultimi due, **qui** e **qui)** sono stato improvvisamente travolto da un problema reale (*di codice, eh!*) e ho deciso di soprassedere (*e rimandiamo, rimandiamo…*) e di tornare al nostro amato **C**.

Il problema reale citato riguardava l'uso (*e il mal uso*) della funzione della **libc memcmp(3)**, che è utilissima, ben fatta e, a volte, indispensabile, ma che può anche provocare dei notevoli mal di testa. E quindi ci vuole un articolo: inizialmente avevo pensato a uno della serie **"No, Grazie!"** (*ne ho scritti già un po' e vi invito a leggerli o rileggerli,* ***qui****,* ***qui****,* ***qui****,* ***qui****,* ***qui*** *e* ***qui***), ma sarebbe stato ingeneroso verso la **memcmp(3)**, che, come detto sopra, è una buona funzione, basta usarla bene. E allora, invece di un **"No, grazie!"** sarà uno **"Scusate"**, ispirato nuovamente al grande **Massimo Troisi** che con il suo **Scusate il Ritardo** si scusava per il tempo passato dal suo ultimo film (ma il gioco di parole significava anche altro).



…ma quante volte ti ho detto di non usare la memcmp?…
E allora, andiamo al dunque: la **memcmp(3)** esegue, come dice il nome, una comparazione di memoria, ossia ci dice se due blocchi di memoria sono uguali (o diversi). Vediamone una semplice implementazione tanto per chiarire di cosa stiamo parlando. Vai col codice!

```
#include <string.h> int memcmp(const void *vl, const void *vr, size_t n)
{ const unsigned char *l=vl, *r=vr; for (; n && *l == *r; n--, l++, r++); return n ? *l-*r : 0;
}
```

questa è l'implementazione della (notevole e raccomandabile) **musl libc**, ed è, come spesso accade in questa libreria, ridotta all'osso privilegiando semplicità e funzionalità. La **memcmp(3)** della **glibc**, pur eseguendo lo stesso compito, è implementata in una maniera abbastanza più complicata (per correggere alcune vulnerabilità della funzione) e quindi ve la risparmio: per rendere l'idea va benissimo la versione della **musl libc**.

Come si nota, stiamo parlando di una funzione abbastanza semplice (nell'esempio sono 3 linee!) che compara *"byte-a-byte"* (anzi, *"char-a-char"*) due zone di memoria, ritornando

zero se sono uguali, e un valore diverso da zero (positivo o negativo) se sono diverse: il codice canta. Apparentemente non ci possono essere grossi problemi, e infatti, normalmente, la funzione è utilissima e funzionale, ma… in alcuni casi i problemi ci sono! Eccome! Il trucco è ben descritto nello **standard del C11** (cap. 6.2.6.1/6):

When a value is stored in an object of structure or union type, including in a member
object, the bytes of the object representation that correspond to any padding bytes
take unspecified values.

Ecco, il problema principale della **memcmp(3)** (a parte le vulnerabilità corrette dalla **glibc**) è questo, e si chiama **"Structure Padding"**. È noto (o almeno dovrebbe esserlo) che una **C** *struct* viene, per *default,* allineata dal compilatore in maniera che il *size* della struttura sia un multiplo di 4. È evidente, quindi, che una *struct,* come la vediamo noi, può essere abbastanza diversa da come la rappresenta il compilatore. E quindi? Quindi, bisogna usare la **memcmp(3)** con molta cautela quando si comparano i contenuti di strutture, i bug sono in agguato e possono provocare comportamenti molto inaspettati. Vediamo un semplicissimo esempio:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h> // una struttura di esempio
typedef struct { uint8_t achar; uint32_t anint;
} Test; // testmemcmp - funzione main()
int main(int argc, char *argv[])
{ // definisco e inizializzo test1 e test2 Test test1;
test1.achar = 11; test1.anint = 222222222; Test test2;
test2.achar = 11; test2.anint = 222222222; // comparo test1 e
test2 con memcmp(3) if (memcmp(&test1, &test2, sizeof(Test))
== 0) printf("%s: test1 e test2 sono uguali\n", argv[0]); else
printf("%s: test1 e test2 sono diverse\n", argv[0]); return 0;
}
```

Ecco, questo esempio, una volta compilato ed eseguito produce

questo risultato:

```
aldo@Linux $ gcc testmemcmp.c -o testmemcmp
aldo@Linux $ ./testmemcmp
./testmemcmp: test1 e test2 sono diverse
aldo@Linux $
```

Ma come? Le due strutture *test1* e *test2* hanno un contenuto diverso? Ma se le ho inizializzate nella stessa maniera! Ma che cavolo dice la **memcmp(3)**?… Ebbene si! Sono diverse, perché io ho inizializzato solo i campi che visibili (*achar* e *anint*) ma non ho inizializzato i campi invisibili, e cioè i *byte* di *padding* aggiunti dal compilatore! Infatti, se aggiungete una **printf(3)** che vi stampi i *sizeof* di *test1* e *test2* scoprirete che il *size* è 8, anche se, a prima vista, dovrebbe essere 5 (un *char* da 1 *byte* + un *int* da 4 *byte):* i 3 *byte* in più sono proprio i *byte* di *padding* aggiunti dal compilatore per allineare il membro *achar,* che ora occupa 4 *byte. Maledetto compilatore, mi hai fregato un'altra volta!*

Il riassunto di tutto questo potrebbe essere: "L*a **memcmp(3)** è una ottima funzione per comparare memoria, ma non usatela per comparare strutture*!". Ma questo è un po' semplicistico, *eddài, sicuramente si può trovare una soluzione!*. Ok, e allora vediamo alcune possibili soluzioni:

1) fare un reset della memoria della struct prima di usarla

Avete mai sentito parlare della **memset(3)**? Immagino di si. Può tornare utile per far funzionare l'esempio mostrato sopra. Vediamo come:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h> // una struttura di esempio
typedef struct { uint8_t achar; uint32_t anint;
} Test; // testmemcmp - funzione main()
int main(int argc, char *argv[])
{ // definisco e inizializzo test1 e test2 Test test1; //
```

```
oppure, senza memset(3): Test test1 = {0}; memset(&test1, 0, sizeof(Test)); test1.achar = 11; test1.anint = 2222222222; Test test2; memset(&test2, 0, sizeof(Test)); test2.achar = 11; test2.anint = 2222222222; // comparo test1 e test2 con memcmp(3) if (memcmp(&test1, &test2, sizeof(Test)) == 0) printf("%s: test1 e test2 sono uguali\n", argv[0]); else printf("%s: test1 e test2 sono diverse\n", argv[0]); return 0; }
```

Ecco, questo esempio dà il seguente risultato:

```
aldo@Linux $ gcc testmemcmp.c -o testmemcmp
aldo@Linux $ ./testmemcmp
./testmemcmp: test1 e test2 sono uguali
aldo@Linux $
```

il trucco usato è abbastanza evidente: con la **memset(3)** azzeriamo anche i *byte* di *padding,* quindi la successiva chiamata a **memcmp(3)** funziona bene, visto che ora i campi nascosti sono uguali nelle due strutture. Il prezzo da pagare è, però, relativamente alto, perché bisogna ricordarsi di usare la **memset(3)** ogni volta che si definisce (o si alloca) una nuova *struct*. da comparare.

Notare che, come indicato nel commento del codice qui sopra, si può azzerare la *struct* anche senza usare **memset(3)** inizializzandola nella definizione (questo però non vale se definiamo un *pointer* da usare successivamente con **malloc(3)**). E non fatevi ingannare da falsi risultati provocati dalla fortuna: può succedere che, casualmente, i *byte* di *padding* abbiano lo stesso valore anche senza eseguire la **memset(3)**, sia dopo una definizione sia dopo una allocazione (e **malloc(3)** non inizializza la memoria!), *quindi oggi il codice funziona, e magari domani no… occhio a questi dettagli!*

2) istruire il compilatore su come usare il padding

Questo si può fare modificando minimamente il codice, dicendo al compilatore di non fare il *padding* per quella particolare

struct. Si fa così (riporto solo il dettaglio del codice modificato rispetto al primo esempio di *testmemcmp.c):*

```
// una struttura di esempio "packed"
typedef struct __attribute__((__packed__)) { uint8_t achar;
uint32_t anint;
} Test;
```

questo funziona, ma ha alcune controindicazioni: la prima è che il *padding* serve a gestire più efficientemente la memoria e quindi ometterlo potrebbe peggiorare le prestazioni. E poi bisogna ricordasi di farlo sempre (stesso problema del punto 1). L'ultima controindicazione è la scarsa portabilità di questo trucco, che dipende dal compilatore (*__packed__* è un attributo del solo **GCC**) e dalla CPU (potrebbe non funzionare su CPU diverse da **x86** o **amd64**). Tra l'altro in rete si trovano esempi di casi particolari di malfunzionamento e, *dulcis in fundo*, bisogna tener presente che **GCC,** anticamente, trattava questo caso come codice pericoloso segnalando dei *Warning* preoccupanti (adesso non più, probabilmente hanno trovato la maniera di renderlo meno pericoloso). In ogni caso vi ricordo che scrivere codice non portabile non è mai una buona idea.

3) comparare le struct senza usare memcmp(3) (spoiler: questa è la mia opzione preferita).

Come anticipato dallo *spoiler,* questa è la mia opzione preferita, anche se non è a costo zero, visto che bisogna scrivere un po' di codice in più (ma neanche tanto). Vi avverto: è una soluzione un po' lapalissiana ed è molto semplice: bisogna scrivere una versione *custom* della **memcmp(3)** per ogni tipo di struttura da comparare, e usarla sempre al posto della versione generica. Senza ulteriori giri di parole, facciamo cantare un'altra volta il codice!

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h> // una struttura di esempio
typedef struct { uint8_t achar; uint32_t anint;
```

```
} Test; // prototipi locali
bool myMemcmp(const Test *s1, const Test *s2); // testmemcmp - funzione main()
int main(int argc, char *argv[])
{ // definisco e inizializzo test1 e test2 Test test1; test1.achar = 11; test1.anint = 2222222222; Test test2; test2.achar = 11; test2.anint = 2222222222; // comparo test1 e test2 con myMemcmp() if (!myMemcmp(&test1, &test2)) printf("%s: test1 e test2 sono uguali\n", argv[0]); else printf("%s: test1 e test2 sono diverse\n", argv[0]); return 0;
} // myMemcmp - versione specializzata di memcmp(3) per il tipo Test
bool myMemcmp(const Test *s1, const Test *s2)
{ if (s1 && s2 && (s1->achar == s2->achar) && (s1->anint == s2->anint)) return false; // sono uguali: la memcmp(3) ritornerebbe 0 else return true; // sono diverse: la memcmp(3) ritornerebbe !0
}
```

Che ne dite? La versione specializzata della **memcmp(3)** confronta *campo-a-campo* invece che *byte-a-byte* (*e grazie al… che funziona: per questo è lapalissiana*) è il risultato è sempre sicuro. È un po' più laborioso, perché per ogni tipo di *struct* bisogna scrivere la versione specializzata, però è a prova di errore, e ne vale la pena. In più c'è il vantaggio di poter ristornare direttamente un *bool,* il che è molto comodo. Io normalmente seguo questa via.

Direi che per oggi può bastare. Ho introdotto questo nuovo tipo di articolo *"Scusate"*, che è meno drastico del *"No Grazie!"*, visto che la **memcmp(3)** ha alcuni difetti a livello di uso, ma, nel complesso, è una funzione buona e utile, basta usarla come e quando si deve. Non so esattamente di cosa parlerò nel prossimo articolo, ma sarà di sicuro interessante, ve lo prometto! *E, come sempre, non trattenete il respiro nell'attesa!*

Ciao, e al prossimo post!

https://italiancoders.it/scusate-la-memcmp-considerazioni-sull

uso-della-memcmp3-in-c-e-c/