

Asm coding with socket system calls – Primo approccio alla gestione dei socket

Publicato da *twiz* il 12/02/2002

Livello *intermedio*

Introduzione

Questo testo focalizzera' la sua attenzione sull' uso dei socket in asm su Linux, usando le (o meglio dire la) `sys_call` fornite/a dal kernel.

La sintassi utilizzata sara' la AT&T, il compilatore il GAS.

Il testo sara' diviso in due parti, la prima cerchera' di dare le basi teoriche su `sys_call` e dintorni, la seconda cerchera' di mettere in pratica le nozioni appena apprese applicandole ad un esempio pratico, la creazione di un bot in assembly.

La divisione in due parti dovrebbe risultare utile anche a chi gia' conosce il funzionamento dei socket e ricerca solo un esempio pratico, cio' non vuol dire "buttatevi subito su botz.s" per tutti, citando/parafrasando Tanenbaum :

"You have to eat your broccoli before you can have the double chocolate fudge cake dessert"

Se lo dice lui... :)

Programmi usati

- *as*: GNU Assembler (GAS)

Iniziamo

```
---[ Una sola sys_call ]
```

Sappiamo che ogniqualvolta vogliamo "passare il controllo" al kernel e mandare in esecuzione una `sys_call` dobbiamo richiamare l' "int 0x80".

Allo stesso modo sappiamo che il numero della `sys_call` va in `%eax` e i successivi parametri (se il numero totale di questi e' minore di 6) vengono passati nei registri `%ebx`, `%ecx`, `%edx` e cosi' via.

```
- Es.      xorl %ebx, %ebx      -
           movl %ebx, %eax  |  exit(0)   SYS_exit = 1 = %eax
           inc  %eax        |
           int  $0x80      -
```

Allo stesso modo siamo abituati a usare i socket in un high level language, sia questo il C o il perl, e ad avere a disposizione diverse chiamate quali ad esempio `socket`, `bind`, `connect`, `accept`, `listen`.

Ci aspetteremmo dunque di trovare una `SYS_bind`, una `SYS_connect` e cosi' via.

La realta' e' un po' diversa. Abbiamo infatti *una sola* `sys_call` "direttamente" accessibile, `SYS_socketcall`.

```
cat /usr/src/linux/include/asm/unistd.h | grep socketcall
```

```
#define __NR_socketcall      102
```

102. Bene. Abbiamo il numero da inserire in `%eax`.

La "sottochiamata" da eseguire viene definita da un numero in `%ebx`, che va ad identificare la funzione che ci interessa

```
cat /usr/src/linux/include/linux/net.h
```

```
[...]
#define SYS_SOCKET      1          /* sys_socket(2)          */
#define SYS_BIND       2          /* sys_bind(2)           */
#define SYS_CONNECT    3          /* sys_connect(2)        */
#define SYS_LISTEN    4          /* sys_listen(2)         */
#define SYS_ACCEPT     5          /* sys_accept(2)         */
#define SYS_GETSOCKNAME 6        /* sys_getsockname(2)    */
#define SYS_GETPEERNAME 7        /* sys_getpeername(2)    */
#define SYS_SOCKETPAIR 8         /* sys_socketpair(2)     */
#define SYS_SEND       9         /* sys_send(2)           */
#define SYS_RECV      10        /* sys_recv(2)           */
#define SYS_SENDTO    11        /* sys_sendto(2)         */
#define SYS_RECVFROM  12        /* sys_recvfrom(2)       */
#define SYS_SHUTDOWN  13        /* sys_shutdown(2)       */
```

```

#define SYS_SETSOCKOPT 14          /* sys_setsockopt(2)          */
#define SYS_GETSOCKOPT 15          /* sys_getsockopt(2)         */
#define SYS_SENDMSG    16          /* sys_sendmsg(2)            */
#define SYS_RECVMSG    17          /* sys_recvmsg(2)            */
[...]
```

Da 1 a 17, per ogni pseudo-sys_call che ci interessa (tutte queste chiamate sono documentate come le altre sys_call e come si vede dal cat precedente nella sezione 2 del man di linux).

Abbiamo il numero da mettere in %ebx.

Per quanto riguarda gli argomenti della chiamata viene passato in %ecx un puntatore al primo di questi argomenti... questo potrebbe farvi venire in mente come linux gestisce ad esempio mmap, ovvero le sys_call con piu' di 5 argomenti. Ricapitolando:

```

%eax      -----> 102  ( __NR_socketcall )
%ebx      -----> SYS_* (bind, connect, ...) [1-17]
%ecx      -----> puntatore al primo degli argomenti.
```

Bene ora abbiamo in mano tutti gli strumenti necessari per iniziarea a mangiare la torta al cioccolato.

```
---[ Implementazione - Pratica - botz.s ]
```

Dopo questa prima (noiosa ma necessaria) introduzione possiamo finalmente passare alla parte piu' direttamente interessante, ovvero la stesura di codice.

A mio avviso il secondo modo migliore per imparare e' avere qualcosa da toccare con mano, da leggere e quanto piu' possibile "spiegato".

Per questa ragione cerchero' di commentare il piu' dettagliatamente e chiaramente possibile (per quanto mi e' umanamente possibile a mezzanotte e mezza) le parti piu' importanti.

... ah il primo metodo migliore e' ovviamente sperimentare, sperimentare e ancora sperimentare , quindi scrivete, scrivete e scrivete :) Tutti i vostri programmi inizieranno con la dichiarazione di un socket :

```
- C ---> socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Vediamo di scriverlo in asm :

```

movl $0x66, %eax          # __NR_socketcall = 102 (66 in hex)
movl $0x01, %ebx          # SYS_SOCKET = 1

pushl $0x06               # Passiamo gli argomenti sullo stack (guardare gli
pushl $0x01               # include per i valori di AF_INET ecc ) in ordine
pushl $0x02               # inverso sullo stack in modo da avere %esp che
                          # punta al primo argomento -> AF_INET = 2

movl  %esp,%ecx           # Mettiamo il puntatore in %ecx e chiamiamo
int   $0x80               # l' int $0x80 come e' solito per le syscall
```

In questo esempio abbiamo usato lo stack come base per passare gli argomenti e dovremo aver cura di pulirlo dopo (con addl 0x0b). La stessa procedura puo' essere scritta anche riservandosi lo spazio sullo stack (anche main e' una funzione con tanto di variabili locali ;)):

```

pushl %ebp                # Prologo della funzione
movl  %esp,%ebp
subl  $0x0b, %esp         # Facciamo spazio per i parametri

movl $0x02, (%esp)       # Copiamo gli argomenti sullo stack
movl $0x01, 4(%esp)      # equivalente a movl $0x01, -8(%ebp)
movl $0x06, 8(%esp)      # movl $0x06, -4(%ebp)

movl $0x66, %eax         # __NR_socketcall = 102 (66 in hex)
movl $0x01, %ebx         # SYS_SOCKET = 1

movl %esp,%ecx           # Passiamo il puntatore al primo degli argomenti
int  $0x80
```

In entrambi i casi lo stack pointer (%esp) contiene l' indirizzo del primo argomento della funzione socket(), che e' esattamente quello che ci serve da mettere in %ecx.

C'e' un secondo modo per passare gli argomenti, senza usare lo stack ed e' quello di usare un' area di memoria ad-hoc e copiare l' indirizzo del primo argomento in %ecx.

Vediamo l' implementazione in assembly :

```
.bss

args:                # L' equivalente di una dichiarazione di un array
arg0: .long 0        # di 3 int, int args[3];
arg1: .long 0
arg2: .long 0

[...]

.text

movl $0x02, arg0     # Soliti argomenti della funzione socket
movl $0x01, arg1
movl $0x06, arg2

movl $0x66, %eax     # __NR_socketcall = 102 (66 in hex)
movl $0x01, %ebx     # SYS_SOCKET = 1

movl $args, %ecx     # copiamo l' indirizzo in %ecx
int 0x80
```

La procedura qui sopra va ripetuta per ogni chiamata a SYS_socketcall, e puo' essere utile implementare dunque una macro che riassume i passaggi comuni ad ogni chiamata.

Le funzioni con piu' argomenti tra le 17 possibili sono :

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);

int recvfrom(int s, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

E' quindi sufficiente dichiarare un array di 6 int (arg0, arg1 ... arg6) per poter chiamare praticamente tutte le pseudo-socketcall con la stessa macro.

Vediamo l' implementazione :

```
.macro __socketcall call          # L' idea di questa macro non e' mia, ma l' ho
    movl $0x66, %eax              # vista implementata da wojtek kaniewski
    movl \call, %ebx              # <wojtekka@irc.pl> nel suo ph.s
    movl $args, %ecx
    int $0x80
.endm
```

E vediamo come richiamare l' oramai classica socket(): [supponiamo definito sia l' array args sia la macro __socketcall]

```
movl $0x02, s_arg0              # socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
movl $0x01, s_arg1
movl $0x06, s_arg2
__socketcall $1                  # 1 -> sys_socket
```

Sia la macro che il suo uso sono di facile comprensione e non mi dilungo ulteriormente a riguardo (ricordo solo che \param e' il modo per accedere al valore del parametro "param" all' interno di una macro). Avremmo potuto, invece che usare una macro, definire una procedura ed usare ad esempio lo stack... come spesso capita programmando le soluzioni possono essere molteplici, a voi provarle, sperimentarle, padroneggiarle e scegliere la migliore implementazione :) Ora facciamo un passo oltre, visto che il continuare ad invocare sys_socket non credo sia molto appagante. Scriveremo un semplicissimo bot per irc utilizzando le nozioni apprese si qui e applicandole.

Il codice dovrebbe essere, a mio avviso, abbastanza chiaro e sufficientemente commentato. I passaggi, sempre a mio avviso, piu' importanti saranno ripresi e analizzati subito dopo il sorgente :) Iniziate a sentire un po' di gusto di cioccolata ?

```
# botz.s
# Simple and Useless bot for irc
```

```

# Written by twiz - twiz@email.it
# This bot actually don't have any special function except connect to irc and
# answer to server's pings.

```

```

.data
    port = 6667
    ircserver = 0x745d33e
nick: .ascii "NICK asmbotz1\r\n"
    nick_l = .-nick
user: .ascii "USER asm asm asm\r\n"
    user_l = .-user
channel: .ascii "JOIN #cuni\r\n"
    channel_l = .-channel
ping: .ascii "PING"
pong: .ascii "PONG\r\n"
    pong_l = .-pong

.bss
buffer: .space 512, 0
sock: .long 0

s_args:
s_arg0: .long 0
s_arg1: .long 0
s_arg2: .long 0
s_arg3: .long 0

server: .space 128, 0
server_l = .-server
.macro __socketcall call
    movl $0x66, %eax
    movl \call, %ebx
    movl $$s_args, %ecx
    int $0x80
.endm

.text
.global _start

_start:
    movl $0x02, s_arg0
    movl $0x01, s_arg1
    movl $0x06, s_arg2
    __socketcall $1
    cmp $0x01, %eax
    jb exitproc
    movl %eax, sock
    movw $0x02, server
    movw $((port & 0xff00) >> 8 | (port & 0xff) << 8), server+2
    movl $ircserver, server+4
    movl %eax, s_arg0
    movl $server, s_arg1
    movl $server_l, s_arg2
    __socketcall $0x03
    cmp $0, %eax
    jb exitproc
    movl $0x04, %eax
    movl sock, %ebx
    movl $nick, %ecx
    movl $nick_l, %edx
    int $0x80
    movl $0x04, %eax
    movl sock, %ebx
    movl $user, %ecx
    movl $user_l, %edx
    int $0x80
    movl $0x04, %eax
    movl sock, %ebx
    movl $channel, %ecx
    movl $channel_l, %edx
    int $0x80

Loop:
    movl $0x03, %eax

```

```

    movl sock, %ebx
    movl $buffer, %ecx
    movl $512, %edx
    int $0x80
    movl $buffer, %esi          # Confrontiamo le stringhe provenienti dal
    movl $ping, %edi           # server con la nostra stringa per vedere se
    movl $0x04, %ecx           # siamo stati pingati.

    repe
    je pongroutine             # Ricevuto un ping, invochiamo la procedura
    jmp Loop

pongroutine:
    movl $0x04, %eax           # Semplicemente inviamo un PONG di risposta
    movl sock, %ebx
    movl $pong, %ecx
    movl $pong_l, %edx
    int $0x80
    jmp Loop

exitproc:
    movl $0x01, %eax
    xorl %ebx, %ebx
    int $0x80

```

Bene. Questo era il nostro rudimentale bot. Sa di cioccolato? Passiamo ad analizzare le parti piu' interessanti

```
ircserver = 0x745d33e          # Server in network byte order
```

E' irc.tin.it :) Ora che sapete che server e' suppongo che la successiva domanda sia "Come lo cambio/riproduco ?" Credo che chiunque di voi abbia gia' avuto a che fare con l' asm e con un qualsiasi tutorial, documento o libro sia gia' stato oltremisura indottrinato riguardo a Little Endian e Big Endian e relativa architettura degli x86.

Allo stesso modo credo siate stati almeno altrettanto indottrinati, usando i socket in qualsiasi HLL (High Level Language) su Network Byte Order e Host Byte Order, quindi non mi tocchero' il discorso :) Vediamo come ottenere l' unsigned long int necessario per il server.

Metodo comodo :

ircaddr.c

```

<-- taglia qui -->
main(int argc, char **argv)
{
    printf("Addr = 0x%x\n", inet_addr(argv[1]));
}
<-- qui finisce -->

```

Compile, eseguite passando in argv[1] l' ip del server che vi interessa.

Ma se programmate in asm questa soluzione di comodo non credo vi soddisfi :) Vediamo di fare un passo in piu' attraverso un esempio :

ircaddr.c

```

<-- taglia qui -->
main(int argc, char **argv)
{
    unsigned long int tot;
    printf("Addr = 0x%x\n", inet_addr("127.0.0.1"));
    tot = 127 << 24 | 0 << 16 | 0 << 8 | 1;
    __asm__("bswap %0" : "+r" (tot));
    printf("Addr = 0x%x\n", tot);
}
<-- qui finisce -->

```

Compile e avviate...

```
twiz@twiz:~$ ./ex
```

```
Addr = 0x100007f
Addr = 0x100007f
twiz@twiz:~$
```

Tombola! Ecco come funziona `inet_addr` (piccolo doveroso ringraziamento a nextie e' d'obbligo). Lascio a voi il "compito" (semplice semplice) di scriverla in assembly.
Dati per scontati gli shift la linea interessante rimane :

```
__asm__( "bswap %0" : "+r" (tot));
```

Per chi non fosse abituato alla inline-syntax del C, "+r" significa che la variabile `tot` verra' usata sia in lettura sia in scrittura. Se volete sapere di piu' a riguardo consultate "info gcc" nella sezione "Machine Descriptions : Constraints : Modifiers" e il testo di rmyagi sull' inline assembly (<http://www.geocities.com/SiliconValley/Ridge/2544/>). Ora avete tutti i mezzi per "svolgere" il vostro "compito" ;) Stesso discorso vale per la porta, alla riga :

```
movw $((port & 0xff00) >> 8 | (port & 0xff) << 8), server+2
```

Andiamo oltre

```
cmp $0x01, %eax
jb exitproc
```

Con queste due istruzioni controlliamo il valore di ritorno (in `%eax`) della chiamata (per sapere quali valori di ritorno indicano errore come sempre consultate `man `funzione``). Puo' essere utile introdurre una diversa "exitproc" , magari che stampi a video su `stderr` (`movl $0x02, %ebx`) un messaggio di errore prima di uscire.

Il Loop e' in se' molto semplice, ci limitiamo a riceverne le stringhe dal server (per dubbi sulle stringhe inviate e ricevute puo' essere utile consultare le rfc relative) e a confrontarle con "PING", se combaciano chiamiamo la pingroutine, che altro non fa che inviare un PONG! :) Per server (quali ad esempio `undernet` al join o altri) un semplice pong puo' non essere sufficiente, in quanto si aspettano oltre al pong in risposta o un numero inviato dal ping (PING 212338456 ad es. su `undernet`) o il nome del server, come sempre, se interessati all' argomento, leggete le rfc.

Conclusioni

Come avete visto il programma e' semplice, potete modificarlo, aggiungerne funzioni (risposta a determinati comandi ecc. ecc.... se lo fate magari fatemelo sapere :)) giocare un po' come volete.

Mi auguro questa semplice introduzione vi sia servita, per ogni chiarimento, feedback (molto apprezzato), critica e/o altro mi potete scrivere a:

```
twiz@email.it
```

Oppure mi potete trovare (esami permettendo) di sera in giro per le reti irc.
Alla prossima

```
-twiz
```