KERNEL SYSTEM CALL

23 Ottobre 2010
Linux Day

Abstract

In questo breve articolo verrà analizzato il sistema di accesso al kernel mediante chiamate di sistema. Verranno descritte le due modalità di accesso dallo user-space al kernel previste dall'architettura x86 (o i386). Seguirà la descrizione dei file del kernel dove le funzioni di sisema sono definite e si concluderà con la scrittura di un modulo per identificare il vettore delle chiamate e aggiungere una nostra chiamata di sistema.

Da User-Space a Kernel-Space

Il passaggio da user space a kernel space in un sistema operativo viene eseguito mediante chiamate di sistema invocate normalmente con specifiche istruzioni dipendenti dall'architettura hardware.

Architettura x86

Storicamente, in Linux, l'architettura x86 prevede l'accesso al kernel dallo spazio utente tramite l'istruzione `INT 0x80`. Questa istruzione è un interrupt software che viene inizializzato in fase di boot. La specifica funzione di sistema viene discriminata in funzione del valore contenuto nel registro a 16 bit `AX` (o a 32 bit `EAX`).
La chiamata di un interrupt software è normalmente un'operazione molto pesante dal punto di vista del processore. L'architettura x86, che non brilla certo per ottimizzazione, vide un progressivo peggioramento delle prestazioni delle chiamate di sistema. Di fatto risultava che un Pentium II era più veloce di un Pentium III e ancora più di un Pentium IV nell'effettuare chiamate di sistema con questa tecnica.
Per questa ragione a partire al Pentium II, Intel ha aggiunto due istruzioni macchina chiamate `SYSENTER` e `SYSEXIT` le quali permettono di effettuare un salto all'interno del kernel in modo decisamente più rapido e ottimizzato.
Per poter utilizzare queste istruzioni sotto Linux, sono però necessari tre requisiti:

- Processore Intel Pentium II o AMD K7 o superiori.
- Kernel linux 2.6.x, anche se il sistema è stato aggiunto a partire dal kernel sperimentale 2.5.52.
- Le librerie *glibc* siano almeno alla versione 2.3.5 e siano state compilate adeguatamente per supportare queste istruzioni.

Praticamente tutte le distribuzioni fanno ormai uso delle istruzioni `SYSENTER`/`SYSEXIT`. Tra le poche (forse l'unica) a fare eccezione vi è Slackware che, al momento in cui scriviamo, è alla versione 13.1. Slackware sarà il sistema operativo usato per lo studio seguente in quanto di default implementa ancora le chiamate di sistema ad `INT 0x80`. Chiaramente gran parte di quello che verrà detto potrà essere testato su qualunque distribuzione.

Esempio di chiamata di sistema INT 0x80

L'esempio che verrà ora proposto è l'analisi di una chiamata di sistema qualunque. Compileremo il programma in modo statico ed andremo ad esaminarne il codice. Chiamiamo quindi due funzioni di esempio:

- `getpid`: ritorna il numero del processo. Non richiede parametri.
- `time`: restituisce il valore corrente della data misurata in secondi. Richiede il puntatore ad una variabile `time_t` oppure può essere `NULL`.

Queste chiamate vengono, come detto, identificate da un codice contenuto nel primo registro di accumulazione del processore, mentre negli altri registri si trovano i parametri o gli argomenti da passare alla funzione. Da [2] e [4] si legge in particolare che `getpid()` è identificato dal numero 20 (0x14), mentre `time()` dal numero 13 (0xD); per `time()` il parametro è contenuto in `EBX`.
Il programma *prova.c* implementa le due chiamate:

```
#include <stdio.h>
#include <time.h>          // per time()
#include <sys/time.h>
#include <sys/types.h>     // per getpid()
#include <unistd.h>
```

```
#define NOP3 { asm volatile ("nop;nop;"); }

time_t t;
int p;

int main()
{
	NOP3;
	p=getpid();
	NOP3;
	t=time(NULL);
	NOP3;
	return 0;
}
```

Compilare staticamente il programma con:

gcc -static -o prova prova.c

Questo permette di mantenere i simboli di debug. A questo punto disassembliamo il programma con gdb:

gdb prova

dal prompt del programma, disassembliamo la funzione `main()`, `getpid()` e `time()` con il comando:
`disassemble main` (o altra funzione).
Tramite le istruzioni `nop`, identifichiamo dove esattamente cominciano e finiscono le chiamate alle funzioni; di seguito è riportato uno stralcio della funzione `main`:

```
0x08048239 <main+17>:   nop
0x0804823a <main+18>:   nop
0x0804823b <main+19>:   call   0x804f0f0 <getpid>
0x08048240 <main+24>:   mov    %eax,0x80ce1e8
0x08048245 <main+29>:   nop
0x08048246 <main+30>:   nop
0x08048247 <main+31>:   sub    $0xc,%esp
0x0804824a <main+34>:   push   $0x0
0x0804824c <main+36>:   call   0x804f0c0 <time>
0x08048251 <main+41>:   add    $0x10,%esp
0x08048254 <main+44>:   mov    %eax,0x80ce1e4
```

Di fatto la chiamata è singola ad una sola funzione. Tale funzione è stata lincata staticamente quindi è già disponibile per il disassemblaggio.
Assembly di `getpid`:

```
0x0804f0f0 <getpid+0>:  mov    %gs:0x4c,%edx
0x0804f0f7 <getpid+7>:  cmp    $0x0,%edx
0x0804f0fa <getpid+10>: mov    %edx,%eax
0x0804f0fc <getpid+12>: jle    0x804f100 <getpid+16>
0x0804f0fe <getpid+14>: repz ret
0x0804f100 <getpid+16>: jne    0x804f10c <getpid+28>
0x0804f102 <getpid+18>: mov    %gs:0x48,%eax
0x0804f108 <getpid+24>: test   %eax,%eax
0x0804f10a <getpid+26>: jne    0x804f0fe <getpid+14>
0x0804f10c <getpid+28>: mov    $0x14,%eax
0x0804f111 <getpid+33>: int    $0x80
0x0804f113 <getpid+35>: test   %edx,%edx
0x0804f115 <getpid+37>: mov    %eax,%ecx
0x0804f117 <getpid+39>: jne    0x804f0fe <getpid+14>
0x0804f119 <getpid+41>: mov    %ecx,%gs:0x48
0x0804f120 <getpid+48>: ret
```

le prime due istruzioni valutano se il PID è già stato salvato in una variabile locata a `%gs:0x4c`. Se non lo è (ossia se è la prima volta che `getpid()` viene chiamata) allora viene invocata la chiamata di sistema. Questa nota è doverosa per quello che verrà descritto più avanti per i test. La parte più importante sono le riche +28 e +33 dove si vede che viene caricato il valore 0x14 nel registro `EAX` selezionando di fatto la system call `getpid()` e quindi viene invocato l'`INT 0x80`.
Analogamente disassembliamo la funzione `time`:

```
0x0804f0c0 <time+0>:    push   %ebp
0x0804f0c1 <time+1>:    mov    %esp,%ebp
0x0804f0c3 <time+3>:    mov    0x8(%ebp),%edx
0x0804f0c6 <time+6>:    push   %ebx
0x0804f0c7 <time+7>:    xor    %ebx,%ebx
0x0804f0c9 <time+9>:    mov    $0xd,%eax
0x0804f0ce <time+14>:   int    $0x80
0x0804f0d0 <time+16>:   test   %edx,%edx
0x0804f0d2 <time+18>:   je     0x804f0d6 <time+22>
0x0804f0d4 <time+20>:   mov    %eax,(%edx)
0x0804f0d6 <time+22>:   pop    %ebx
0x0804f0d7 <time+23>:   pop    %ebp
0x0804f0d8 <time+24>:   ret
```

all'indirizzo +9 e +14 si vede come, analogamente a prima, venga caricato in EAX un valore (0xD) e invocato l'interrupt. A questo punto occorre verificare quanto questo sistema inficia sulle prestazioni rispetto a `SYENTER/SYSEXIT`. Quindi, senza toccare alcun che (ne kernel, ne glibc) scriveremo dei benchmark per eseguire le chiamate di sistema con le istruzioni ottimizzate in questione. Per prima cosa però occorre valutare se il kernel supporta questo meccanismo e si da per scontato che il PC soddisfi i requisiti. Per farlo occorre eseguire questi procedimenti:

- il kernel deve essere una versione 2.6.x.y: con il comando *uname -r* è possibile verificarlo e determinarne la versione.
- il vDSO deve essere abilitato: *cat /proc/sys/vm/vdso_enabled* deve essere maggiore di 0 per poter usare le `SYENTER/SYSEXIT`. Ad oggi è consuetudine che questo valore sia 1 (anche in SlackWare) che indica che il vDSO è dinamico.

È bene chiarire fin da subito che, se i requisiti sono verificati, l'invocazione della chiamata di sistema tramite il vDSO dipende dalla "configurazione" del vDSO stesso. In particolare se vDSO è dinamico (*/proc/sys/vm/vdso_enabled*=1) l'accesso al kernel viene fatto saltando ad un indirizzo, variabile da processo a processo, ma puntato da `*%gs:0x10`. A questo indirizzo viene scritto il valore del vDSO in fase di caricamento del processo, ossia è il kernel che provvede a decidere l'indirizzo e renderlo "noto" al processo stesso. Se invece il vDSO è statico (*/proc/sys/vm/vdso_enabled*=2) allora il jump sarà all'indirizzo fisso `0xffffe414`. Tale valore può essere diverso da kernel a kernel, da distribuzione a distribuzione. Quello riportato è quello letto in Slackware 13 e 13.1. Settando a 2 il vDSO (con *echo 2 > /proc/sys/vm/vdso_enabled*) ed eseguendo più volte *ldd* su un file eseguibile (per esempio *ldd /bin/bash*), si nota che gli indirizzi sono sempre costanti. In particolare sotto Slackware si ha la libreria virtuale *linux-gate.so.1 => (0xffffe000)*.

Nota: se un programma è scritto per vDSO statico (come i benchmark di esempio riportati), esso funzionerà anche per vDSO dinamico, ma non è vero il viceversa.

Detto questo si può procedere a scrivere (e capire) i benchmark. Ricordiamo che i programmi seguenti vanno compilati con alcune istruzioni obbligatorie:

gcc -static -o nomefile nomefile.c

ed eventualmente aggiungendo -DSTATIC_VDSO se volete fare il test per vDSO statico (chiaramente se il programma lo supporta); tutti i test che verranno fatti ora saranno con vDSO dinamico. Il primo che viene proposto è la misura della chiamata `getpid()`, file *misura_getpid.c*:

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#define NOP3 {asm volatile("nop; nop");}
#define K 1000000

#if !defined(STATIC_VDSO)
#define sys_getpid() __asm__( \
                     "movl $20, %eax    \n" \
                     "call *%gs:0x10    \n" \
                     "movl %eax, pid    \n" \
                     );
#else
#define sys_getpid() __asm__( \
                     "movl $20, %eax    \n" \
                     "call 0xffffe414   \n" \
                     "movl %eax, pid    \n" \
                     );
#endif
```

```
int pid=0;
unsigned long i;

int main(int c, char **v)
{
   if(c>1)
     {
      puts("GetPid()");
      while(i++ < K) {
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
         pid = getpid(); asm("mov %ebx, %gs:0x48;");
      }
     } else
     {
      puts("__asm__ ");
      while(i++ < K)
        {
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
             sys_getpid(); sys_getpid();
        }
     }

   printf("pid (from syscall) is %d\n", pid);
   return 0;
}
```

Il programma, lanciato da riga di comando con un parametro qualsiasi, esegue 1000000 di volte la chiamata di sistema classica, così come compilata nelle glibc. Lanciando il programma senza parametri si ha invece l'esecuzione della chiamata sfruttando il vDSO, ossia le istruzioni `SYSENTER`. L'unica nota da aggiungere è relativa all'istruzione "`mov %ebx, %gs:0x48;`": questa istruzione cancella la variabile che contiene il valore del PID in modo che `getpid()` venga chiamata continuamente e quindi non venga falsata la misura. Eseguendo ora il programma come segue:

time ./misura_getpid && time ./misura_getpid x

Si ottiene:

__asm__
pid (from syscall) is 23645

real 0m1.742s
user 0m0.696s
sys 0m1.044s
GetPid()

pid (from syscall) is 23646

real 0m3.325s
user 0m1.832s
sys 0m1.484s
Ossia con le istruzioni SYSENTER/SYSEXIT si impiega sul nostro sistema in oggetto quasi il dimezzamento dei tempi!!
Il secondo esempio proposto è la misura della funzione `time()`, in *misura_time.c*:

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#define NOP3 {asm volatile("nop; nop");}
#define K 1000000
#define TIMEVARPTR (&t)

// call 0xffffe414 va bene solo se non siamo in static
#define sys_time() __asm__(      \
      "leal t, %ebx;"           \
      "mov  $0xD, %eax;"  \
      "call *%gs:0x10;"         \
      "mov  %eax,t;"            \
      );

time_t t=0;
unsigned long i;

int main(int c, char **v)
{
   if(c>1)
     {
      puts("Time()");
      while(i++ < K) {
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
         t = time(TIMEVARPTR);  t = time(TIMEVARPTR);
      }
     } else {
      puts("__asm__ ");
      while(i++ < K)
        {
          sys_time();    sys_time();  sys_time();  sys_time();
          sys_time();    sys_time();  sys_time();  sys_time();
          sys_time();    sys_time();  sys_time();  sys_time();
          sys_time();    sys_time();  sys_time();  sys_time();
          sys_time();    sys_time();  sys_time();  sys_time();
        }
     }

   printf("Time is %d\n", t);
   return 0;
}
```

Questo programma è fatto solo per vDSO dinamico. Portarlo in statico è comunque piuttosto semplice. Anche in questo caso si lancia:

time ./misura_time && time ./misura_time x

e si ottiene
__asm__

Time is 1284753022

real   0m1.668s
user   0m0.500s
sys   0m1.168s
Time()
Time is 1284753025

real   0m3.183s
user   0m1.772s
sys   0m1.408s

ed anche in questo caso si vede circa un dimezzamento dei tempi.
L'ultimo esempio proposto è `gettimeofdate()`. Il codice è:

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

#define NOP3 {asm volatile("nop; nop;");}
#define K 1000000
#define TIMEVARPTR (&t)
// call 0xffffe414; solo se non compili come static
#define sys_gettimeofdate() __asm__( \
       "leal tv, %ebx;"    \
       "leal tz, %ecx;"    \
       "mov  $0x4E, %eax;" \
       "call *%gs:0x10;"   \
       "mov  %eax,r;"      \
       );

struct timeval  tv;
struct timezone tz;
unsigned long i;
int r;       // return value

int main(int c, char **v)
{
   if(c>1)
     {
      puts("GetTimeOfDay()");
      while(i++ < K) {
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
         r = gettimeofday(&tv, &tz);   r = gettimeofday(&tv, &tz);
      }
     } else
     {
      puts("__asm__ ");
      while(i++ < K)
        {
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
           sys_gettimeofdate(); sys_gettimeofdate();
```

```
                sys_gettimeofdate(); sys_gettimeofdate();
            }
        }

        printf("GetTimeOfDate = %d\n", r);
        return 0;
    }
```

che lanciato al solito come:

time ./misura_gettimeofdate && time ./misura_gettimeofdate x

restituisce:

__asm__
GetTimeOfDate = 0

real 0m33.568s
user 0m1.276s
sys 0m32.286s
GetTimeOfDay()
GetTimeOfDate = 0

real 0m35.636s
user 0m2.900s
sys 0m32.734s

Ora, come si nota il processore si cuoce in quanto 32s di sys (ossia esecuzione in spazio kernel) sono particolarmente tanti. Solo il tempo in spazio utente tende ad essere inferiore, ma ciò è una magra consolazione. Questo esempio serve per far vedere come ogni funzione può subire ottimizzazioni più o meno sensibili, ma alla fine ciò che deve essere implementato in modo ottimizzato è il codice nello spazio kernel.

Note sulla compatibilità

La domanda che viene spontanea è: se la quasi totalità dei sistemi Linux x86 usa `SYSENTER/SYSEXIT`, ciò implica che la retrocompatibilità sia compromessa? Cosa accadrebbe su un Pentium 1? La risposta è: nulla; il kernel infatti capisce se determinate istruzioni sono supportate oppure no e sceglie di conseguenza come devono avvenire le invocazioni. Ad ogni modo è sempre possibile disabilitare il vDSO sin dal boot passando al kernel il parametro **`nosep`**.
Quindi è tendenzialmente insensato non utilizzare queste istruzioni le quali possono essere sfruttate a pieno solo se anche le *glibc* lo permettono. Queste possono essere ricompilate opportunamente seguendo quanto indicato in [6]. L'unica controindicazione trovata è stata relativamente al driver proprietario delle schede NVIDIA®. Infatti tale driver è l'unica componente del kernel che è stato necessario ricompilare una volta aggiornate le librerie sotto Slackware. Non avendo schede AMD/ATI® con driver proprietario non è stato testato se il problema affligge anch'essi.

La seconda domanda che ci si potrebbe porre è: ma quanto questa cosa accelererebbe il mio sistema? In altre parole, il gioco vale la candela? Secondo chi scrive sì perchè è insensato non utilizzare qualche cosa che, seppur minimamente, porta benefici prestazionali al sistema. È però altrettanto evidente che i programmi testati eseguono 1000000 di chiamate... e fanno solo quello! La stragrande maggioranza dei programmi chiamerà poche volte il kernel in proporzione al resto del codice, quindi non bisogna aspettarsi che il sistema raddoppi le prestazioni. Anzi nella maggior parte dei casi l'aumento di prestazioni dovrebbe essere minimo.
...ma poco è meglio di niente.

Come fare per abilitare queste ottimizzazioni in un sistema coma SlackWare? Occorre ricompilare solo le librerie glibc in particolare aggiungendo le opzioni di compilazione *--enable-bind-now* e *--enable-shared*. Per magiori informazioni o per avere lo slackbuild per la Slackware consultare il sito dell'associazione, sezione Tips&Tricks [6].

VDSO

Virtual **D**ynamically-linked **S**hared **O**bject, è l'equivalente di una libreria dinamica (da cui *virtual*) fornita dal kernel che permette allo spazio utente di eseguire poche azioni a livello kernel senza l'appesantimento dovuto alle normali system call provvedendo nel contempo a scegliere la modalità di accesso al kernel più efficiente. Si è detto non a caso "equivalente di una libreria" perchè questo meccanismo si trova contenuto nella libreria *linux-gate.so.1* che è possibile visualizzare per esempio lanciando:

ldd /bin/bash (o qualsiasi programma compilato non staticamente)

Se ora si cercasse nel sistema questa libreria, non la si troverebbe in quanto, come detto, è virtuale e fornita dal kernel,

come verificheremo più avanti.
Per quanto detto fino ad ora, ogni kernel 2.6 è configurato di default per avere il vDSO dinamico in modo da innalzare la sicurezza del sistema rendendo un po' più laborioso l'accesso al kernel da parte di eventuali "attaccanti".
A livello kernel il codice del vDSO è identificato dall'indirizzo `__kernel_vsyscall`. Tale indirizzo è un indirizzo virtuale scelto dal sistema operativo in modo casuale e passato al processo tramite il parametro `AT_SYSINFO` del formato elf ([8]), oppure può essere statico in funzione del fatto che il valore del vDSO sia 1 o 2 nel file virtuale */proc/sys/vm/vdso_enabled*. La cosa importante è che la scelta è effettuata in fase di caricamento del programma e fino ad allora quell'indirizzo non esisterà.
Ora proveremo ad esaminare cosa c'è a questo indirizzo. Per farlo vi sono principalmente due vie: la prima è la più semplice e richiede il programma *ddd* o *gdb*, mentre la seconda prevede la scrittura di un programma apposito che copi e scriva in un file il contenuto del vDSO per poi disassemblarlo in seguito (il secondo metodo è presente negli esempi).
Di seguito viene documentato il primo metodo; si procede disassemblando per esempio il programma *misura_time* (ma si potrebbe disassemblare qualsiasi programma del sistema):

gdb misura_time

gdb riconosce l'autocompletamento dei comandi e dei simboli premendo TAB (esattamente come una shell) quindi se proviamo a scrivere dal prompt di *gdb*

disassemble __kernel_v<TAB>

oppure tutto intero

disassemble __kernel_vsyscall

non solo non si completerà nulla, ma scrivendo per intero il comando, gdb restituirà:

No symbol "__kernel_vsyscal" in current context.

Questo perchè l'indirizzo in questione, come già detto, non è stato caricato perchè il programma non è in esecuzione.
Quindi lanciando i comandi

break main
run
Starting program: /home/.../misura_time

Breakpoint 1, 0x08048236 in main ()
Current language: auto; currently asm

il programma viene caricato e quindi ora è noto l'indirizzo del vDSO; a questo punto si può disassemblare:

disassemble __kernel_vsyscall
Dump of assembler code for function __kernel_vsyscall:

```
0xb78af414 <__kernel_vsyscall+0>:       push   %ecx
0xb78af415 <__kernel_vsyscall+1>:       push   %edx
0xb78af416 <__kernel_vsyscall+2>:       push   %ebp
0xb78af417 <__kernel_vsyscall+3>:       mov    %esp,%ebp
0xb78af419 <__kernel_vsyscall+5>:       sysenter
0xb78af41b <__kernel_vsyscall+7>:       nop
0xb78af41c <__kernel_vsyscall+8>:       nop
0xb78af41d <__kernel_vsyscall+9>:       nop
0xb78af41e <__kernel_vsyscall+10>:      nop
0xb78af41f <__kernel_vsyscall+11>:      nop
0xb78af420 <__kernel_vsyscall+12>:      nop
0xb78af421 <__kernel_vsyscall+13>:      nop
0xb78af422 <__kernel_vsyscall+14>:      jmp    0xb78af417 <__kernel_vsyscall+3>
0xb78af424 <__kernel_vsyscall+16>:      pop    %ebp
0xb78af425 <__kernel_vsyscall+17>:      pop    %edx
0xb78af426 <__kernel_vsyscall+18>:      pop    %ecx
0xb78af427 <__kernel_vsyscall+19>:      ret
```

End of assembler dump.

Questo indirizzo si troverà sempre a 0x414 dall'entry point della libreria *linux-gate.so.1* che in questo caso si troverà all'indirizzo `0xb78af000` (vedere *arch/x86/vdso/vdso32-sysenter-syms.lds*). Se invece il vDSO fosse statico, l'entry point sarebbe `0xffffe414`. Questo codice viene eseguito in user-space fino alla `sysenter` che invoca il kernel. Il kernel poi, quando avrà terminato le sue operazioni tornerà allo user-space all'indirizzo `__kernel_vsyscall+16`, ossia a 0x424 come indicato da `VDSO32_SYSENTER_RETURN` in *arch/x86/vdso/vdso32-syms.lds*.

Nota: è noto che il codice del kernel si trova agli indirizzi virtuali supperiori a `0xc0000000`. La cosa singolare è che, anche se la libreria è codice a livello kernel, essa viene caricata al di sotto dell'indirizzo sopra citato solo se il vDSO è dinamico.

Ma cosa accade se il vDSO non è supportato? Per vederlo basta disabilitarlo con:

echo 0 >/proc/sys/vm/vdso_enabled

ed eseguendo subito un

ldd /bin/bash

si vede che la libreria *linux-gate.so.1* è scomparsa. Andando a disassemblare quindi *misura_time* eseguendo le operazioni menzionate in precedenza, non si riesce ad accedere all'indirizzo del vDSO, giustamente. Nonostante ciò, tutto funziona. Questo perchè la chiamata al vDSO (`__kernel_vsyscall`) ora punta all'indirizzo statico seguente:

```
0x08053290 <_dl_sysinfo_int80+0>:       int    $0x80
0x08053292 <_dl_sysinfo_int80+2>:       ret
```

A questo punto, per capire come funziona il tutto sarà sufficiente trovare nel kernel il codice che realizza tutto questo. La directory che contiene tutti i files relativi è *arch/x86/vdso*. Il file principale è *vdso32-setup.c* il quale configura la modalità di accesso al vDSO, o meglio se utilizzare l'istruzione `SYSENTER` o `SYSCALL` in funzione dell'architettura, oltre che a rilocare il vettore delle chiamate si sistema, ecc. Ad ogni modo tutti e tre i metodi di accesso al kernel sono inglobati nel file *vdso32.S*:

```
        .globl vdso32_int80_start, vdso32_int80_end
vdso32_int80_start:
        .incbin "arch/x86/vdso/vdso32-int80.so"
vdso32_int80_end:

        .globl vdso32_syscall_start, vdso32_syscall_end
vdso32_syscall_start:
#ifdef CONFIG_COMPAT
        .incbin "arch/x86/vdso/vdso32-syscall.so"
#endif
vdso32_syscall_end:

        .globl vdso32_sysenter_start, vdso32_sysenter_end
vdso32_sysenter_start:
        .incbin "arch/x86/vdso/vdso32-sysenter.so"
vdso32_sysenter_end:
```

e quindi sono presenti tutti i segmenti di codice. Il codice di queste librerie .so incluse da questo file, è implementato nella sottodirectory *vdso32/* rispettivamente nei files assembly *int80.S*, *sysenter.S* e *syscall.S* dove è possibile individuare gli indirizzi `__kernel_vsyscall` visti in precedenza.

Nota su architettura AMD x86_64:

Per i sistemi a 64 bit di tipo x86_64 le cose sono un po' differenti. Quando questo tipo di architettura fu introdotta, tra i vari miglioramenti introdotti da AMD vi furono anche le istruzioni `SYSCALL/SYSRET`. Questo semplifica enormemente le cose perchè non c'è bisogno di alcun metodo di arbitraggio per capire quale tipo di istruzione deve essere usata per dare l'accesso al kernel ed inoltre il vDSO è sempre presente, quindi anche a livello di codice il sistema è (o dovrebbe essere) un po' più semplice.
Le prestazioni rimangono pari o leggermente superiori all'istruzione `SYSENTER`.

KERNEL: organizzazione e moduli

Vedremo ora come il kernel organizza le proprie chiamate di sistema, dove le implementa e come sia possibile aggiungerne di nuove. Esamineremo un kernel x86, analisi facilmente estendibile ad un kernel x86_64 con relativamente poche differenze. Il kernel in analisi è il 2.6.35.7 e tutti i file comprensivi di percorso sono da intendersi a partire dalla radice dei sorgenti del kernel, nel nostro caso */usr/src/linux-2.6.35.7/*. I file principali da esaminare sono:

- *include/linux/syscalls.h*: questo file elenca tutte le funzioni di sistema il cui nome è indicato con il suffisso *sys_*. Per esempio `getpid()` sarà `sys_getpid()`. All'inizio di questo file vengono definite alcune macro per la scrittura e/o definizione delle varie chiamate di sistema anche in funzione del tipo di architettura.
- *arch/x86/include/asm/syscalls.h*: questo file ridefinisce tutte le chiamate di sistema dipendenti dall'architettura (in questo caso x86). Qualche tempo fa venne deciso da Linus Torvals di unificare le architetture x86 e x86_64, quindi vi sono alcuni file (tra cui questo) nel quale vengono definite alcune macro che tendono a differenziare il codice in funzione dal fatto che il kernel venga compilato per x86_64. Ad ogni funzione dichiarata vi è poi il percorso che indica dove è stata implementata. Normalmente ogni funzione è implementata in un file specifico.
- *arch/x86/kernel/syscall_table_32.S*: questo file è scritto in assembly e va a definire manualmente tutti i puntatori alle chiamate di sistema. In particolare il file comincia con la dichiarazione `ENTRY(sys_call_table)` che automaticamente definisce anche la sezione di memoria nella quale questo vettore verrà allocato. Leggendo il file si ritrovano le funzioni localizzate ad una posizione pari al loro codice identificativo, come per esempio `getpid()`:

```
.long sys_getpid    /* 20 */
```

- *arch/x86/kernel/syscall_64.c*: in questo file viene semplicemente implementato il vettore `sys_call_table` che contiene le chiamate di sistema a 64bit. Di fatto inizialmente "azzera" il vettore ponendo ogni puntatore a `&sys_ni_syscall` ossia punta alla funzione nulla (*ni* sta per *not implemented*). Questa funzione è così implementata (*kernel/sys_ni.c*):
```
asmlinkage long sys_ni_syscall(void)
{
   return -ENOSYS;
}
```
Successivamente il vettore viene completato includendo il file *arch/x86/include/asm/unistd_64.h* il quale sfrutta la macro `__SYSCAL` per fare in modo che le chiamate di sistema corrispondano ad un valore esatto. Tale valore corrisponderà al valore da passare al registro `EAX` (o meglio `RAX` a 64bit) per invocare la chiamata di sistema dallo spazio utente. A tal proposito, andando a vedere dove sono collocate le chiamate si scopre che i valori sono completamente differenti tra x86 e x86_64. Per esempio in precedenza si è visto che `getpid()` era identificata dal numero 20 (0x14) su architettura x86, mentre per x86_64 questa chiamata è al valore 39:
```
#define __NR_getpid                39
__SYSCALL(__NR_getpid, sys_getpid)
```
L'allocazione di questo vettore va fatta poi manualmente nella sezione desiderata mediante una apposita macro definita nei file menzionati.
- *arch/x86/include/asm/unistd_32.h*: versione a 32 bit di *unistd_64.h*. Sicuramente più ordinata e comprensibile, elenca l'insieme delle define che identificano i numeri delle varie system call. La define è `__NR_<nome_syscall>`, per esempio `__NR_getpid`.
- *arch/x86/vdso/vdso32/**: In questa sezione è implementato principalmente il codice per l'invocazione delle chiamate di sistema con le istruzioni `SYSENTER` e `SYSCALL` (nei file *sysenter.S e syscall.S*).

A questo punto è sufficiente andare a vedere dove sono state implementate le singole chiamate di sistema. Mantenendo gli esempi sopra citati, vediamo solo dove e come è implementata la `getpid()`. Questa è definita in *kernel/timer.c*:

```
SYSCALL_DEFINE0(getpid)
{
      return task_tgid_vnr(current);
}
```

Di fatto la variabile `current` interna al kernel è una struttura che contiene tutte le informazioni necessarie per la gestione del processo da parte del kernel stesso (per esempio è usata moltissimo dallo scheduler). La macro `SYSCALL_DEFINE0` è definita in *include/linux/syscalls.h*. Il numero che la contraddistingue indica solo quanti parametri prevedere in ingresso la funzione. In questo caso 0.

Esempi

Gli esempi che andremo ora a vedere sono fondamentalmente due, e riguardano entrambi come sostituire una chiamata di sistema. In entrambi i casi è evidente che occorre modificare il vettore delle chiamate di sistema identificato da `sys_call_table`. Dal kernel 2.6 questo puntatore non è più esportato come simbolo e quindi non è più possibile sostituire le chiamate, come invece si faceva nel kernel 2.4, nel modo seguente:

```
sys_old_ptr = sys_call_table[__NR_getpid];
sys_call_table[__NR_getpid] = sys_new_ptr;
```

Oggi occorre ricostruire la posizione di questo puntatore per potervi accedere. Questo puntatore, come per altri, è verificabile sempre tramite il file *System.map* presente nei sorgenti compilati del kernel o qualche volta nel file */proc/kallsyms*.
Gli esempi che seguiranno sono sviluppati su kernel 2.6.35.x salvo diversa comunicazione.

sys_call_table from System.map

Il primo esempio permette di individuare manualmente gli indirizzi delle funzioni o variabili di interesse. Avendo i sorgenti disponibili e compilati, avremo a disposizione anche il file *System.map*. In questo file sono presenti tutti gli indirizzi delle funzioni e variabili compresi quelli definiti negli script del linker per esempio `_text` e `_etext`.
Detto questo, il modulo che verrà proposto è basato su [16] e prevede di individuare l'indirizzo della `sys_call_table` la quale però in alcuni sistemi può essere memorizzata in una sezione di codice protetta in scrittura e quindi occorrono anche le funzioni `set_page_ro()` e, soprattutto, `set_page_rw()`. Tutto questo lo si può trovare all'interno del file *System.map* come già detto e, per il kernel e il sistema in esame, il risultato è il seguente:

```
c1319110 R sys_call_table
c10132f7 T set_pages_ro
c10133da T set_pages_rw
```

Gli indirizzi sono ovviamente virtuali ed il loro valore è superiore sicuramente a `0xC0000000` che è l'indirizzo oltre il quale il sistema alloca la memoria e il codice dello spazio kernel[1]. In particolare tutti rientrano nella memoria al di sopra di `0xC1000000` ovvero dove il linker riloca il codice eseguibile del kernel (`_text`).
Il modulo è *replace_syscall.c* e prevede appunto che vengano configurati manualmente gli indirizzi principali sopra citati. Tali indirizzi sono parametrizzabili in fase di load del modulo qualora quelli inseriti non vadano bene su altri kernel. Nell'esempio verrà sostituita la chiamata `getpid()`. La nuova funzione non farà altro che inviare messaggi (riportati nei file di log) con il numero di volte che questa è stata chiamata.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/version.h>

#include <asm/unistd.h>   // contiene i numeri delle chiamate __NR_xxx
#include <asm/page.h>     // per avere le funzioni per la memoria come virt_to_page
#include <asm/linkage.h>
//#include <asm/memory_model.h>  // Get pfn_to_page e page_to_pfn
#include <linux/mm.h>            // Get struct page
#include <linux/syscalls.h>      // Get syscall sys_xxx

MODULE_LICENSE("GPL");
MODULE_VERSION("0.1");
MODULE_AUTHOR("Calzo");
MODULE_DESCRIPTION("Syscall Hijaking: sostituzione della chiamata 'getpid'");

// PARAMETERS
static unsigned long sct_addr = 0xc1319110;   // Syscall Table address
module_param(sct_addr, ulong, S_IRUGO);    // see LDD3 cap 2, pag 36 o linux/stat.h
MODULE_PARM_DESC(sct_addr, "Syscall Table address");

static unsigned long pages_rw_addr = 0xc10133da;     // Indirizzo set_pages_rw
module_param(pages_rw_addr, ulong, S_IRUGO);
MODULE_PARM_DESC(pages_rw_addr, "Address of set_pages_rw()");

static unsigned long pages_ro_addr = 0xc10132f7;     // Indirizzo della funzione
module_param(pages_ro_addr, ulong, S_IRUGO);
MODULE_PARM_DESC(pages_rw_addr, "Address of set_pages_ro()");

static void *original_syscall;         // puntatore alla syscall originale
static unsigned long *sys_call_table;
static void (*pages_rw)(struct page *page, int numpages);
static void (*pages_ro)(struct page *page, int numpages);
struct page *sys_call_table_page;      // puntatore alla pagina

int __init repsys_init_module(void);
void repsys_clenup_module(void);
static int counter = 0;
```

questa funzione andrebbe definita anteponendo la macro asmlinkage*, come per tutte le syscall, ma non dovendogli passare parametri si può anche omettere*

```
asmlinkage pid_t (*old_getpid)(void);
```

Nuova chiamata getpid()*: scrive il numero di chiamate effettuate da quando il modulo è stato caricato, e poi chiama la vecchia funzione di sistema* getpid()

```
asmlinkage pid_t sys_new_getpid(void)
{
       printk(KERN_ALERT "getpid chiamata %d volte\n", ++counter);
       return old_getpid();
}
```

Inizializzazione del modulo: prima di tutto esegue un controllo blando sugli indirizzi di memoria passati controllando che siano

1 In realtà è possibile cambiare questo indirizzo in fase di configurazione del kernel, ma 0xC0000000 è il valore di default

indirizzi in spazio kernel. Ciò comunque non da sicurezze perchè se gli indirizzi fossero sbagliati, i disastri sono assicurati. Fatto questo si procede al salvataggio dei puntatori alle funzioni, quindi viene calcolato l'indirizzo virtuale in termini di "pagine"[2] *dove risiede la* sys_call_table *e reso read-and-write tale indirizzo. Quindi si procede al salvataggio della vecchia syscall (*sys_getpid()*) e sostituita con quella nuova.*

```
int __init repsys_init_module(void)
{
	if(sct_addr<=0xC000000 ) {
		printk(KERN_ALERT "Invalid SysCall Table address\n");
		return -1;
	}

	if(pages_rw_addr<=0xC000000 ) {
		printk(KERN_ALERT "Invalid set_page_rw() address\n");
		return -2;
	}

	if(pages_ro_addr<=0xC000000 ) {
		printk(KERN_ALERT "Invalid set_page_ro() address\n");
		return -3;
	}
	// --------- //
	printk(KERN_ALERT "Setting sys_call_table RW...\n");
	sys_call_table = (unsigned long*)sct_addr;
	pages_rw = (void*)pages_rw_addr;
	pages_ro = (void*)pages_ro_addr;
```

Di seguito la memoria contenente sys_call_table *vettà forzata a read-and-write (rw); questo perchè alcune distribuzioni*[3] *rendono questa zona di memoria read-only (ro). In realtà in Slackware, con un kernel vanilla che ricordiamo essere 2.6.35.7, si riesce tranquillamente a sovrascrivere il vettore della system call. La cosa particolare è che in System.map, la* sys_call_table *è marcata in sola lettura (R), quindi anche in un normale kernel questo dovrebbe essere vero, ma non lo è e andrebbe verificato anche sulle altre distribuzioni*

```
	printk(KERN_ALERT "Ricalcolo indirizzi:\n");
	sys_call_table_page = virt_to_page(*sys_call_table);
	pages_rw(sys_call_table_page, 1);
	original_syscall = (void*)sys_call_table[__NR_getpid];
	old_getpid = original_syscall;
	printk(KERN_ALERT   "   sys_call_table (virtual): 0x%X\n"
	                    "   sys_call_table (page): 0x%X\n"
	                    "   old sys_call (virtual): %X\n",
		(int)sys_call_table, (int)sys_call_table_page, (int)original_syscall);

	sys_call_table[__NR_getpid] = (unsigned long)sys_new_getpid;

	return 0;
}
```

Unload del modulo: ripristino della system call originale

```
void repsys_clenup_module()
{
	printk(KERN_INFO "Ripristino chiamata di sistema originale.\n");
	pages_ro(sys_call_table_page, 1);
	sys_call_table[__NR_getpid] = (unsigned long)original_syscall;
}

module_init(repsys_init_module);
module_exit(repsys_clenup_module);
```

Il test del modulo lo si esegue semplicemente caricandolo ed eseguendo un qualsiasi processo che richieda una `getpid()` o utilizzando il programma *testGetPid* presente nei sorgenti. Quando questa chiamata verrà invocata, verrà scritto nei log del kernel quante volte essa è stata chiamata: per SlackWare il file da osservare è */var/log/syslog* per tutti i messaggi marcati `KERN_ALERT`.

Nota: avendo disposizione il file *System.map*, è possibile estrarre particolari indirizzi come `_text`, `_etext`, ecc e tramite questi ricalcolare gli indirizzi o cercare particolari stralci di codice per individuare gli indirizzi `sys_call_table` e/o altri che possano servire. Ci sono numerosi esempi in internet a riguardo, ma il

2 Normalmente (default) ogni pagina è di 4kb, ma può essere anche di 8kb. Dipende dalla configurazione del kernel

3 Dalla documentazioni in internet, le distribuzioni che proteggono la sys_call_table sono quelle RedHat

concetto è di base lo stesso di quello appena illustrato.

sys_call_table detection using IDT

Gli esempi postati in precedenza sono propedeutici, ma non risolvono il problema generico. Infatti essi prevedono che chi carica il modulo abbia i sorgenti del kernel compilati e possa accedere alla *System.map*. Questa però è una situazione rara, soprattutto se chi esegue il codice è il famoso "malintenzionato" menzionato negli aggiornamenti di un noto sistema operativo non oggetto di questo documento.
Quindi ora l'obiettivo è capire come accedere alla `sys_cal_table` senza passare il valore da utente, ossia in automatico.
Per fare ciò la Intel ci è venuta in aiuto fin dai tempi del 80286 con le istruzioni `SGDT/SIDT`[4] ([12], [4]). In particolare l'istruzione che verrà utilizzata sarà `SIDT`. Prima di entrare nel dettaglio chiariamo cosa occorre fare:

1. L'obiettivo è ottenere l'indirizzo della `sys_call_table`
2. Per farlo occorrerà prima trovare l'indirizzo della funzione `system_call()` la quale esegue un accesso diretto alla tabella delle chiamate di sistema
3. trovato questo indirizzo, verrà analizzato il codice per trovare la `sys_call_table`

Per capire ciò si procede all'analisi del codice del kernel contenuto in *arch/x86/kernel/*[5]. In particolare il codice che ci interessa è relativo ai "punti di ingresso" per le syscall definiti in *entry_32.S*[6]. In questo file sono previste tutte le modalità di accesso al kernel ovvero l'implementazione vera e propria della funzione `system_call()`. Noi preleviamo a titolo di esempio la sezione legata alla `SYSENTER`:

```
sysenter_do_call:
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)
        LOCKDEP_SYS_EXIT
        DISABLE_INTERRUPTS(CLBR_ANY)
        TRACE_IRQS_OFF
        movl TI_flags(%ebp), %ecx
        testl $_TIF_ALLWORK_MASK, %ecx
        jne sysexit_audit
sysenter_exit:
```

Il codice riportato valuta se la chiamata di sistema contenuta nel registro `EAX` è valida, ossia non eccede il massimo numero di syscall (`jae`: jump if above or equal, [12]). Se la syscall esiste ecco che il sistema la chiama accedendo direttamente al puntatore a tale funzione contenuto nel vettore `sys_call_table`. Quindi è verificata la presenza di questa chiamata e quindi è chiaro che se noi conoscessimo il puntatore alla funzione `system_call()` potremmo esaminarne il codice macchina ed estrarre l'indirizzo della tabella. Ora l'opcode dell'istruzione call è noto da [12], ma noi per avere la certezza del formato (opcode) dell'istruzione call, disassembleremo proprio *vmlinux* presente nella root dei sorgenti dopo averli compilati. Quindi si procede eseguendo (anche da utente):

gdb vmlinux

e dal suo prompt disassembliamo proprio la funzione `system_call()`:

(gdb) disassemble system_call

che restituisce il seguente codice:

```
0xc1316abc <system_call+0>:     push   %eax
0xc1316abd <system_call+1>:     cld
[...]
0xc1316abe <system_call+2 to 14>:     push di tutti i registri
[...]
0xc1316acb <system_call+15>:    mov    $0x7b,%edx
0xc1316ad0 <system_call+20>:    mov    %edx,%ds
0xc1316ad2 <system_call+22>:    mov    %edx,%es
0xc1316ad4 <system_call+24>:    mov    $0x0,%edx
0xc1316ad9 <system_call+29>:    mov    %edx,%fs
0xc1316adb <system_call+31>:    mov    $0xffffe000,%ebp
0xc1316ae0 <system_call+36>:    and    %esp,%ebp
0xc1316ae2 <system_call+38>:    testl  $0x100001d1,0x8(%ebp)
```

4 Store Global/Interrupt Descriptor Table Register descritto nel manuale dei processori Intel per x86 e/o IA64.
5 Attenzione che da una certa versione in avanti le architetture x86 e x86_64 si sono unificate. Quindi se state utilizzando un kernel vecchio, la directory potrebbe essere *arch/i386/kernel* per sistemi a 32 bit
6 Per vecchi kernel x86 il nome del file è semplicemente *entry.S*

```
0xc1316ae9 <system_call+45>:    jne    0xc1316bc4 <syscall_trace_entry>
0xc1316aef <system_call+51>:    cmp    $0x152,%eax
0xc1316af4 <system_call+56>:    jae    0xc1316c0d <syscall_badsys>
0xc1316afa <system_call+62>:    call   *-0x3ece6ef0(,%eax,4)
0xc1316b01 <system_call+69>:    mov    %eax,0x18(%esp)
0xc1316b05 <syscall_exit+0>:    cli
0xc1316b06 <syscall_exit+1>:    mov    0x8(%ebp),%ecx
0xc1316b09 <syscall_exit+4>:    test   $0x1000feff,%ecx
0xc1316b0f <syscall_exit+10>:   jne    0xc1316be4 <syscall_exit_work>
0xc1316b15 <restore_all_notrace+0>:     mov    0x38(%esp),%eax
0xc1316b19 <restore_all_notrace+4>:     mov    0x40(%esp),%ah
0xc1316b1d <restore_all_notrace+8>:     mov    0x34(%esp),%al
0xc1316b21 <restore_all_notrace+12>:    and    $0x20403,%eax
0xc1316b26 <restore_all_notrace+17>:    cmp    $0x403,%eax
0xc1316b2b <restore_all_notrace+22>:    je     0xc1316b3c <ldt_ss>
[...]
0xc1316b2d <restore_nocheck+0 to 9>: pop di tutti i registri
[...]
0xc1316b38 <restore_nocheck+11>:        add    $0x8,%esp
0xc1316b3b <irq_return+0>:              iret
0xc1316b3c <ldt_ss+0>:                  lar    0x40(%esp),%eax
[...]
```

Il codice che interessa maggiormente è quello evidenziato dove si riconosce il codice esaminato in *entry_32.S*. Quindi ora vediamo l'opcode dei comandi processando il file con *objdump* e cercando il simbolo `system_call`:

objdump -D vmlinux | less

cercando ora il simbolo della funzione (con "/" seguita da *system_call*) ecco che vedremo il codice macchina della chiamata:

```
c1316afa <syscall_call>:
c1316afa:       ff 14 85 10 91 31 c1    call   *-0x3ece6ef0(,%eax,4)
c1316b01:       89 44 24 18             mov    %eax,0x18(%esp)
```

come si nota gli indirizzi del codice sono gli stessi ed identifichiamo l'opcode della call (`FF1485`). Il numero che segue è l'indirizzo di memoria nel quale è contenuto l'indirizzo della `sys_call_table`.
Verranno quindi proposti due esempi per sostituire la funzione `getpid()` (`sys_getpid()` per il kernel) e `open()` (`sys_open()` per il kernel).

Vediamo allora più nel dettaglio come fare per fare ciò che è stato descritto. Innanzi tutto occorre ottenere quella che nei moduli di empio è chiamata `idt_struct`:

```
struct {
        unsigned short    limit;
        unsigned int      base;
} __attribute__ ((packed)) idt_struct;
```

e questa struttura viene caricata tramite l'istruzione

```
asm("sidt %0" : "=m" (idt_struct));
```

Dalla Figura 1 si vede come con questa struttura è possibile accedere al vettore degli interrupt.

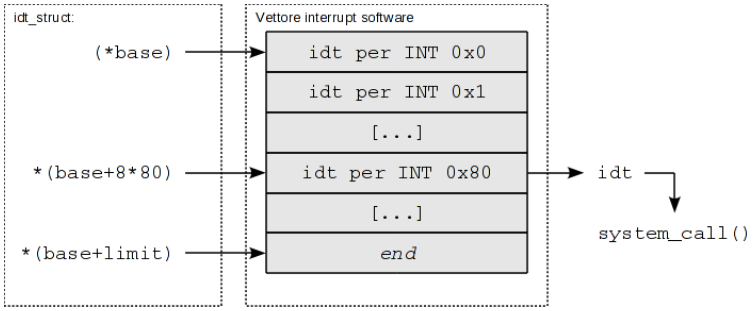

Figura 1: Struttura IDT per l'accesso al vettore degli interrupt

Il vettore degli interrupt contiene una struttura di 8byte (chiamata `idt` negli esempi) tramite la quale è possibile

ottenere l'indirizzo della funzione `system_call()`. Per farlo occorre copiare quanto contenuto all'indirizzo `*(base+8*0x80)` nella struttura `idt` così descritta:

```
struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;
```

da cui si ottiene l'indirizzo della funzione `system_call()` come:

```
system_call_ptr = (idt.off2 << 16) + idt.off1;
```

A questo punto si procede come preannunciato, ossia si cerca l'opcode della call per estrarne il puntatore alla funzione `system_call()`.

Il primo esempio proposto *replace_syscall_idt_getpid.c* prevede la sostituzione della system call `getpid()` (`sys_getpid()` per il kernel). Il codice è il seguente:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/version.h>

#include <asm/unistd.h>   // contiene i numeri delle chiamate __NR_xxx
#include <asm/page.h>     // per avere le funzioni per la memoria come virt_to_page
#include <asm/linkage.h>
#include <linux/mm.h>                   // Get struct page
#include <linux/syscalls.h>             // Get syscall sys_xxx
#include <linux/mm_types.h>

#define MAX_SEEK    500

MODULE_LICENSE("GPL");
MODULE_VERSION("0.1");
MODULE_AUTHOR("Calzo");
MODULE_DESCRIPTION("Detect sys_call_table uging IDT");

// Struttura ottenuta tramite l'istruzione "sidt" per sistemi a 32bit
struct {
        unsigned short limit;
        unsigned int base;
} __attribute__ ((packed)) idt_struct;

//struttura che permette l'individuazione l'indirizzo della system_call()
struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;

static unsigned long *sys_call_table;
static unsigned long system_call_ptr;
static int counter=0;

int __init repsysidt_init_module(void);
void repsysidt_cleanup_module(void);
```

Prototipo della getpid. Si noti che la macro asmlinkage *indica al compilatore che questa funzione riceverà i parametri tramite lo stack e non tramite i registri. Per* getpid(), *che non ha nessun parametro, questa macro potrebbe essere omessa, ma è caldamente consigliabile di scriverla.*

```
asmlinkage pid_t    (*old_getpid)(void);
```

Nuova funzione getpid()*: non fa altro che indicare quante volte è stata chiama questa funzione da quando il modulo è stato caricato.*

```
asmlinkage pid_t sys_new_getpid(void)
```

```
{
        printk(KERN_ALERT "getpid chiamata %d volte\n", ++counter);
        return old_getpid();
}
```

La funzione che segue "scopre" l'indirizzo della sys_call_table *esaminando il codice macchina della funzione* system_call. *L'analisi prevede l'dentificazione dell'opcode FF 14 85, come già detto. La ricerca si limita ragionevolmente ai primi 500* (MAX_SEEK) *byte*

```
void *repsysidt_findout_sct_addr(void *system_call_addr)
{
        unsigned char *p;   // puntatore generico accessibile byte per byte
        int i;

        p = (unsigned char*)system_call_addr;  // p = indirizzo funzione
        for(   i=0; (!(p[0]==0xFF && p[1]==0x14 && p[2]==0x85)) && i<MAX_SEEK;
               p++, i++);

        if(i<MAX_SEEK) {
               p += 3;            // avanzo nel codice per selezionare il puntatore
               return (void *)*(unsigned long*)p;     // indirizzo sys_call_table
        }

        return NULL; // qui non ci devo arrivare
}
```

Inizializzazione del modulo: qui si ritrova tutto il codice visto fino ad ora, ossia la determinazione di idt_struct*, di* idt *e infine si chiama la funzione* repsysidt_findout_sct_addr(). *Si prosegue quindi salvando il vecchio indirizzo della syscall da sostituire e sostituendo la syscall in oggetto con quella nuova.*

```
int __init repsysidt_init_module(void)
{
        asm("sidt %0" : "=m" (idt_struct));
        memcpy(&idt, (void*)(idt_struct.base+8*0x80), sizeof(idt));
        system_call_ptr = (idt.off2 << 16) + idt.off1;

        printk("system_call() address: 0x%X\n", (int)system_call_ptr);

        // a questo punto si cerca la sys_call_table
        sys_call_table =
               (unsigned long*)repsysidt_findout_sct_addr((void*)system_call_ptr);

        if(!sys_call_table)
        {
               printk(KERN_WARNING "sys_call_table address not found!\n");
               return -1;
        }

        printk("system_call() address: 0x%X\n", (int)system_call_ptr);
```

ora si cerca la sys_call_table*: questo indirizzo è verificabile tramite il file System.map, così come tutti gli altri indirizzi*

```
        printk("sys_call_table address: 0x%X\n", (int)sys_call_table);
        printk("sys_getpid     address: 0x%X\n", (int)(sys_call_table[__NR_getpid]));

        old_getpid   = (void*)sys_call_table[__NR_getpid];
        sys_call_table[__NR_getpid]      = (unsigned long)sys_new_getpid;

        printk(KERN_ALERT "Sostituzione sys_getpid - address: 0x%X\n",
               (int)(sys_call_table[__NR_getpid]));

        return 0;
}
```

Infine vi è il ripristino della vecchia chiamata nel caso in cui il modulo venga rimosso. È obbligatorio ripristinare la syscall onde evitare che nel kernel rimangano indirizzi che, una volta rimosso il modulo, non esisterebbero più. Provare per credere.

```
void repsysidt_cleanup_module()
{
        sys_call_table[__NR_getpid] = (unsigned long)old_getpid;
        printk(KERN_INFO "Ripristino syscall originale\n");
}
```

```
module_init(repsysidt_init_module);
module_exit(repsysidt_cleanup_module);
```

Per testare il modulo è sufficiente caricarlo da root con

insmod replace_syscall_idt_getpid.ko

e quindi chiamare un programma ad hoc che esegua questa chiamata di sistema (per esempio *testGetPid* distribuito insieme ai sorgenti del modulo) oppure aspettando un po' fino a che qualche processo non invochi questa chiamata di sistema. In SlackWare i log del modulo verranno scritti in */var/log/messages* o */var/log/syslog* rispettivamente per i messaggi marcati `KERN_INFO` e `KERN_ALERT`.

Il secondo esempio proposto è strutturalmente uguale al primo ed è *replace_syscall_idt_open.c* che sostituirà la chiamata `open()` (`sys_open()` nel kernel) impedendo a chiunque, anche a root, l'accesso ad uno specifico file. La prima aggiunta rispetto a prima è il parametro `filename`

```
static char *filename = "file_importantissimo.txt";
module_param(filename, charp, 0640);          // see LDD3 cap 2, pag 36
MODULE_PARM_DESC(filename, "Nome del file a cui bloccare l'accesso");
```

Questo parametro posto al valore di default *file_importantissimo.txt* indica quale file non sarà accessibile al caricamento del modulo. Questo valore lo si può trovare in */var/module/replace_syscall_idt_open/parameters/filename* e tale file (che ha permessi pari a 640) può essere modificato con nome del file che si preferisce, ma da root. In alternativa lo si può cambiare anche in fase di caricamento del modulo con il comando:

insmod replace_syscall_idt_open filename="nome_file"

La chiamata *open* prevede tre parametri in ingresso quindi ora l'uso della macro `asmlinkage` nel prototipo di tutte le funzioni (vecchia e nuova) open sarà obbligatorio:

```
asmlinkage long     (*old_open)(const char __user *fn, int flags, int mode);

asmlinkage long sys_new_open(const char __user *fn, int flags, int mode)
{
       if( strcmp(fn, filename) == 0 ) {
              printk(KERN_ALERT "Impossibile accedere a '%s'\n", filename);
              return -EACCES;
       }
       return old_open(fn, flags, mode);
}
```

Per testare questo modulo occorre caricarlo esattamente come il precedente. A questo punto per verificarne il funzionamento è sufficiente aprire il file *file_importantissimo.txt* che per esempio si suppone essere in /media/disk. Aprendo una console in /media disk è sufficiente dare un qualsiasi comando, per esempio

cat file_importantissimo.txt

e il sistema non permetterà l'apertura neppure da root.

Nota: in realtà passando il percorso completo (ossia *cat /media/disk/file_importantissimo.txt*) il file viene aperto.

A questo punto esaminando i log dei due moduli si vede che vengono riportati, a puro titolo informativo, gli indirizzi delle chiamate si sistema. Per verificare se sono corretti è sufficiente esaminare il file *System.map* presenti nei sorgenti del kernel una volta compilato.

Nota: il codice degli esempi precedenti mostra come rendere scrivibili gli indirizzi di memoria flaggati in read-only. La `sys_call_table` dovrebbe essere contenuta ad un indirizzo read-only come si può vedere anche dal file System.map. Nonostante questo è stato possibile sovrascrivere senza problema gli indirizzi delle syscall. Dalla documentazione trovata in rete sembra che il flag read-only venga messo da alcune distribuzioni come Red Hat o altre. La cosa che attualmente ipotizzo è che il kernel vanilla non abbia questo tipo di patch, ma andrebbe verificato con certezza.

Links:

[1] http://www.tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/ Implementing a System Call on Linux 2.6 for i386

[2] http://osinside.net/syscall/system_call_table.htm System call table per kernel 2.6

[3] http://pradeepkumar.org/2010/01/implementing-a-new-system-call-in-kernel-version-2-6-32.html come aggiungere una nuova syscall

[4] http://www.fermi.mn.it/linux/quarta/syscalls.htm chiamate di sistema al kernel

[5] http://kernelnewbies.org/KernelGlossary definizioni, tra cui quella del vDSO

[6] http://www.lugman.net/mediawiki/index.php?title=Sysenter/sysexit_VS_int_%240x80 Lugman tips

[7] http://tldp.org/LDP/khg/HyperNews/get/khg.html Linux Kernel Hackers' Guide

[8] http://articles.manugarg.com/systemcallinlinux2_6.html Sysenter Based System Call Mechanism in Linux 2.6

[9] http://kerneltrap.org/node/5793 System call replacement

[10] http://www.epanastasi.com/?page_id=52 Trovare l'indirizzo della sys_call_table

[11] http://www.logix.cz/michal/doc/i386/chp09-04.htm spiegazione del *base* e *limit* dell'IDT

[12] http://www.intel.com/design/pentiumii/manuals/24319102.pdf Intell Architecture Software Developer Manual Volume 2 – Instruction Set

[13] http://www.penguin.cz/~literakl/intel/intel.html principali istruzioni per 286, valide anche per le famiglie di processori attuali

[14] http://www.slideshare.net/fisher.w.y/rootkit-on-linux-x86-v26 Questo è potente: spiega bene l'uso di *sidt* per trovare system_call()

[15] http://kerneltrap.org/node/5793 e http://people.gnome.org/~lcolitti/gnome-startup/linux-iolog/readlog.c interessante esempio, anche se non è chiaro come viene linkata sys_open().

[16] *LINUX&CO n° 40 – Syscall Hijacking* – Marco Caimi

[17] *Linux Device Driver 3th Edition* – Jonatan Corbet, Alesandro Rubini and Greg Kroah-Hartman – O'REILLY

[18] *Understanding The Kernel Linux 3th Edition* – Daniele P. Boved and Marco Cesati - O'REILLY

Info & Credits

Articolo scritto e presentato al Linux Day 2010 – decima giornata nazionale di Linux e del Software Libero – dall'associazione culturale LUGMan (Linux Users Group Mantova). L'articolo è disponibile in formato PDF insieme ai sorgenti dei moduli e dei programmi descritti. Il tutto è scaricabile liberamente dal sito dell'associazione (www.lugman.org sezione *Documentazione*). Il documento è stato scritto con OpenOffice 3.2 sotto SlackWare da Calzoni Pietro aka *Calzo*.
Chiunque volesse altri formati del documento o avesse problemi nel reperire il documento stesso o i sorgenti, volesse segnalare eventuali errori, richiedere informazioni, ecc può contattare liberamente l'associazione a info@lugman.org o direttamente l'autore a calzog@gmail.com.

Chiunque volesse contattare l'associazione LUGMan può consultare il sito www.lugman.org nella sezione “Contatti”.