Nella scarna documentazione di Linux, i device file /dev/mem e /dev/kmem vengono descritti come dispositivi a caratteri che rispecchiano l'attuale contenuto della RAM.
Vogliamo mostrare in questo breve articolo come tali dispositivi possano essere usati in modo relativamente agevole per estrarre informazioni dal kernel.
Questo può risultare utile quando non esiste già un proc file o un kobject in grado di visualizzare le stesse informazioni. Per concretezza, illustriamo l'approccio facendo riferimento ad un caso concreto, ossia al problema di visualizzare il contenuto delle tabelle di paginazione di un generico processo.

Introduzione

Come è noto, la directory /dev include i device file, ossia i dispositivi di ingresso/uscita quali i dischi, le stampanti e quant'altro. Una caratteristica affascinante dei sistemi di tipo Unix è che lo stesso paradigma usato per accedere a file regolari viene usato per accedere ai device file. In altre parole, dopo avere aperto un device file è possibile usare le API read(), write() e lseek() per accedere alle informazioni contenute nel dispositivo. Come è ovvio, possono essere usate soltanto le API compatibili col tipo di dispositivo. Ad esempio, non è possibile effettuare una write() su un CD-ROM (il nome di tale device file è del tipo /dev/hdc) né è possibile effettuare una lseek() su un mouse (il nome di tale device file è del tipo /dev/input/mouse0).

Inoltre, per potere aprire alcuni device file critici, ad esempio quelli che rappresentano le partizioni di disco o quelli come /dev/mem che rappresentano il contenuto della RAM, il programma deve avere privilegi di root, ossia è necessario diventare superuser prima di compilare il programma.

Il device file sul quale lavoreremo si chiama /dev/mem. Come il nome indica, esso rappresenta l'attuale contenuto della RAM. Un altro device file simile a /dev/mem. è chiamato /dev/kmem.. A differenza del primo che considera gli indirizzi dei byte all'interno del file come indirizzi fisici, esso considera gli stessi indirizzi come indirizzi lineari. La differenza tra indrizzo fisico e indrizzo lineare verrà chiarita nella prossima sezione.

Prima di procedere, ricordiamo che esiste anche un proc file chiamato /proc/kcore che fornisce le stesse informazioni di /dev/mem. L'unica differenza è che questo proc file inizia con una testata ELF lunga 4 KB, per cui nel calcolare l'offset di una variabile è necessario sottrarre 4 KB dall'indirizzo fornito dal linker. Per ovvii motivi di semplicità, scegliamo di interagire con /dev/mem..

Poiché il kernel è contenuto nella RAM in indirizzi prefissati scelti dal linker di Linux, conoscendo gli indirizzi in cui sono contenute le variabili globali, è possibile partendo da esse attraversare le varie strutture di dati del kernel ed estrarre informazioni che non possono essere ricavate dai proc file o kobject esistenti.

Rimane da chiarire come si ottengono gli indirizzi forniti dal linker. La risposta sta in una tabella ASCII chiamata System.map che viene preparata durante il linkaggio del kernel. Tale tabella è contenuta nella directory linux contenente il codice sorgente del kernel.

Se intendete interagire con /dev/mem, è quindi necessario disporre della System.map corrispondente al kernel che state usando. Alcune distribuzioni inseriscono nella directory /boot la System.map del kernel da esse utilizzate. Nel caso in cui la vostra distribuzione preferita non includa tale file, suggeriamo di scaricare un kernel vanilla, compilarlo ed installarlo. In questo modo avrete la certezza che la System.map a cui fate riferimento è compatibile con il kernel che state utilizzando.

Mettiamo ora in pratica quanto appena detto e presentiamo un semplice programma che stampa il valore di una variabile globale del kernel.

Come stampare il contenuto di una variabile globale

Linux fa uso di una variabile globale di tipo `int` chiamata `jiffies`; tale variabile contiene il numero di tick trascorsi da quando è stato inizializzato il sistema operativo. Tale variabile è incrementata dopo ogni interruzione di timer. A seconda della opzione di compilazione, essa è incrementata 1000, 250 oppure 100 volte al secondo. La voce "Timer frequency" del menu di configurazione "Processor type and features" consente di impostare la frequenza di timer desiderata.

Usiamo un editor per cercare il simbolo `jiffies` nella `System.map`. Dopo avere scartato una decina di simboli contenenti la stringa `jiffies`, troviamo la voce che ci interessa:

```
c0364678 D jiffies
```

Questa voce ci dice che il linker ha riservato l'indirizzo lineare esadecimale `c0364678` per la variabile `jiffies`. Attenzione però: l'indirizzo calcolato dal linker non è l'indirizzo fisico della variabile `jiffies`, bensì l'indirizzo lineare.

Dobbiamo tenere presente che nelle architetture IA-32 a 32 bit sono disponibili 2^{32}, ossia circa 4 miliardi di indirizzi diversi. Questo spazio degli indirizzi viene diviso da Linux in due parti: la prima parte è riservata ai processi User Mode mentre la seconda parte è riservata al kernel. Una opzione di compilazione consente di specificare la frazione di spazio degli indirizzi da riservare al kernel.
La macro `PAGE_OFFSET` specifica il primo degli indirizzi lineari riservati al kernel.
Nell'esempio che stiamo illustrando `PAGE_OFFSET` è stata impostata al valore esadecimale `0xc0000000`, ossia a 3 GB. Ciò significa che i primi tre giga di indirizzi lineari sono riservati ai processi User Mode mentre il quarto giga è riservato al kernel.

Per trasformare un indirizzo lineare usato dal kernel nel corrispondente indirizzo fisico dobbiamo semplicemente sottrarre all'indirizzo lineare la costante `PAGE_OFFSET`. Tornando al nostro esempio, ciò significa che l'indirizzo fisico in RAM della variabile `jiffies` è `364678`.

A questo punto possiamo presentare un programma che stampa il valore attuale di `jiffies`:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define PAGE_OFFSET         0xc0000000
#define JIFFIES_OFFSET      0xc0364678 - PAGE_OFFSET

int main(void)
{
        int fd, buffer;

        fd = open("/dev/mem", O_RDONLY);
        lseek(fd, JIFFIES_OFFSET, SEEK_SET);
        read(fd, &buffer, 4);
        close(fd);
        printf("jiffies= %d\n", buffer);
        return 0;
}
```

Come si può vedere, il programma è molto semplice: dopo avere aperto il file `/dev/mem`, esso effettua una `lseek()` con l'offset appropriato e legge i 4 byte della variabile binaria `jiffies`. La `printf()` è utilizzata per convertire in decimale il valore binario di `jiffies`. Eseguendo ripetutamente tale

programma, osserverete che esso non stampa mai lo stesso valore bensì una serie di valori crescenti. Questo è corretto in quanto, come detto in precedenza, Linux incrementa jiffies ogni volta che si verifica una interruzione di timer.

Come leggere i campi di una struttura di dati

L'esempio precedente era volutamente semplice in quanto la variabile globale scelta è uno scalare che occupa 4 byte, per cui è sufficiente leggere 4 byte consecutivi a partire da un indirizzo prefissato. Il nostro obiettivo è più ambizioso: vogliamo essere in grado di "navigare" nelle strutture di dati del kernel sfruttando indirizzi contenuti in vari campi di descrittori di risorse.

Per fare ciò non basta conoscere l'indirizzo iniziale della struttura di dati che codifica il descrittore ma dobbiamo anche conoscere gli spiazzamenti dei campi che ci interessano all'interno della struttura.

Facciamo un esempio concreto. Ogni processo gestito da Linux ha un suo descrittore di processo codificato dalla struttura C task_struct Tale descrittore include decine di campi diversi. Quelli che ci interessano maggiormente sono:

- Il campo tasks contenente una coppia di puntatori usati per collegare tra di loro i descrittori di tutti i processi esistenti
- Il campo mm contenente un puntatore ad un descrittore di tipo mm_struct che identifica lo spazio degli indirizzi del processo
- Il campo pid che contiene il PID del processo

I programmi del kernel accedono agevomente ai vari campi di un descrittore poiché sono scritti nel linguaggio C. Ad esempio, se current è una variabile contenente l'indirizzo di un descrittore di processo, current->pid identifica il campo pid del descrittore di processo.

Usando il device file /dev/mem non possiamo identificare un campo di una struttura mediante un nome ma dobbiamo conoscerne lo spiazzamento **ancora spiazzamento** rispetto all'indirizzo iniziale della struttura. Per conoscere il valore di un generico campo, bisogna quindi effettuare un lseek() specificando come indirizzo l'indirizzo iniziale della struttura più lo spiazzamento del campo.

Per ricavare gli spiazzamenti corretti dei campi che ci interessano abbiamo due strade a disposizione:

- Studiare il codice di Linux e ricavare a mano gli spiazzamenti. La definizione di task_struct, ad esempio, si trova nello header file include/linux/sched.h
- Scrivere un apposito modulo da linkare al kernel il cui unico scopo è quello di realizzare delle printk() che visualizzano gli spiazzamenti desiderati.

Per calcolare gli spiazzamenti dei suddetti campi possiamo, ad esempio, definire una semplice funzione ts_offsets() come segue:

```
static int ts_offset(void)
{
        struct task_struct ts;
        printk("tasks offset= %d\n", (int)&ts.tasks - (int)&ts);
        printk("pid offset= %d\n", (int)&(ts.pid) - (int)&ts);
        printk("mm offset= %d\n", (int)&(ts.mm) - (int)&mm);
    return 0;
}
```

Tale funzione deve essere invocata dalla funzione di inizializzazione del modulo.

Per ricavare i dati richiesti, è sufficiente lanciare il comando insmod per inserire il modulo e quindi dmesg per leggere i dati generati dalle printk().

Consigliamo il secondo approccio in quanto è facile commettere errori nel calcolare gli spiazzamenti di campi contenuti in strutture di dati complesse.

Reperimento di un descrittore di processo

Abbiamo parlato di descrittori di processo, ma come facciamo a conoscere gli indirizzi di tali strutture di dati? Al solito, System.map ci fornisce un primo prezioso indirizzo di descrittore di processo. Partendo da esso e percorrendo la lista realizzata mediante il campo tasks, è possibile accedere ai descrittori di tutti i processi esistenti. La variabile globale che ci interessa si chiama init_task e specifica l'indirizzo lineare del padre di tutti i processi, il processo avente PID uguale a 0, anche chiamato *processo 0* o *swapper process* o *idle process*. Come suggerito dall'ultimo nome, questo processo va in esecuzione soltanto quando il kernel non trova nessun altro processo pronto ad andare in esecuzione.

Nella System.map che stiamo usando, troviamo la voce:

```
c035ef80 D init_task
```

per cui l'indirizzo fisico del descrittore di processo da usare è 0x35ef80.

La funzione get_task_struct(), illustrata in seguito restituisce l'indirizzo lineare del descrittore di processo avente un PID prefissato. I due parametri utilizzati contengono, rispettivamente il descritto di file di /dev/mem aperto da un'altra funzione e il PID del processo da cercare.

La macro INIT_TASK_OFFSET denota l'indirizo fisico del processo 0 (nel nostro esempio, 0x35ef80). Le macro NEXT e PID denotano gli spiazzamenti dei campi tasks.next e pid all'interno di un descrittore di processo (per Linux 2.6.17, tali spiazzamenti sono uguali a 25 e 40, rispettivamente). La macro PAGE_OFFSET descritta in precedenza è impostata nella nostra macchina a 0xc0000000.

```
int get_task_struct(int fd, int target_pid)
{
	int rc, tsk_offset, next_offset;
	int pid, tasks_next;

	tsk_offset = INIT_TASK_OFFSET;
	while (1) {
		rc = lseek(fd, tsk_offset + PID*4, SEEK_SET);
		rc = read(fd, &pid, 4);
		if (pid == target_pid) {
			return (tsk_offset + PAGE_OFFSET);
		}
		next_offset = tsk_offset + NEXT*4;
		rc = lseek(fd, next_offset, SEEK_SET);
		if (rc < 0) {
			printf("bad next_offset lseek\n");
			_exit(0);
			}
		rc = read(fd, &tasks_next, 4);
```

```
            tsk_offset = tasks_next - PAGE_OFFSET - NEXT*4;
            if (tsk_offset == INIT_TASK_OFFSET)
                  return -1;
            }
}
```

Stampa delle tabelle di paginazione di un processo

Le tabelle di paginazione sono strutture di dati molto critiche: esse sono lette milioni di volte al secondo dal circuito hardware MMU (Memory Management Unit) della CPU che traduce indirizzi lineari in indirizzi fisici. Allo stesso tempo, esse sono modificate occasionalmente dal kernel, ad esempio quando viene assegnata una nuova page frame ad un processo, oppure quando una pagina viene swappata su disco, oppure quando viene tolta una page frame ad un processo.
Nell'architettura IA-32 che stiamo considerando sono usati due livelli di paginazione: una tabella principale chiamata *Page Global Directory* ed un insieme di *Page Table*. A prescindere dal tipo, ogni voce di una tabella di paginazione che mappa una page frame contiene due tipi di informazione:

I bit più significativi contengono l'indirizzo fisico della page frame
I rimanenti bit contengono una serie di flag (vedi dopo)

Il nostro obiettivo è quello di realizzare un programma che stampa in modo intelleggibile il contenuto delle tabelle di paginazione di un processo. Più precisamente, il programma dovrà stampare gli indirizzi fisici nel formato esadecimale e dovrà premettere ad ogni flag binario il suo nome. Ad esempio, una voce di tabella di paginazione potrebbe essere stampata nel formato seguente:

```
i=0032 linear=08000000 physical=03113000 G=0 PS=0 U/S=1 R/W=1 P=1
```

In questo esempio si tratta della 33-esima voce della Page Global Directory di un processo. Poiché ognuna delle 1024 voci di tale tabella mappa 4 MB di indirizzi lineari, l'indirizzo lineare associato alla i-esima voce è: `i*0x400000`. L'indirizzo fisico è impostato dai programmi di gestione della memoria del kernel. Nell'esempio appena illustrato il flag `PS` (Page Size) è uguale a 0. Questo significa che la voce di Page Global Directory fa uso di page frame da 4 KB. In questo caso, l'indirizzo fisico `0x03113000` è quello di una Page Table da 1024 voci, ognuna delle quali punta eventualmente ad una page frame da 4 KB.
Nella maggior parte dei casi i processi User Mode fanno uso di page frame da 4 KB quindi dobbiamo aspettarci di trovare nelle voci della Page Global Directory riservate al processo soltanto voci aventi il flag `PS` uguale a 0. [1]
Viceversa, il kernel fa ampio uso di page frame da 4 MB per mappare la RAM disponibile, per cui possiamo aspettarci che le ultime voci della Page Global Directory abbiano un contenuto del tipo:

```
i=0862 linear=d7800000 physical=17800000 G=1 PS=1 U/S=0 R/W=1 P=1
```

In questo caso, il flag `PS` è impostato a 1 (page frame da 4 MB); inoltre il flag `U/S` (User/Supervisor) che nell'esempio precedente valeva 1 vale ora 0 per siginificare che soltanto i programmi del kernel sono abilitati ad indirizzare l'intervallo di indirizzi lineari che inizia a `0xd7800000`. Si noti come in questo secondo esempio l'indirizzo fisico è correlato a quello lineare dalla semplice relazione:

```
indirizzo lineare = indirizzo fisico + PAGE_OFFSET
```

Ogni processo possiede un suo spazio degli indirizzi, e quindi un proprio gruppo di tabelle di paginazione. Il descrittore dello spazio degli indirizzi di un processo è una struttura di tipo `mm_struct` definita nello header file `include/linux/sched.h` Il campo `mm` contenuto nel descrittore di processo punta al descrittore di tipo `mm_struct` che definisce lo spazio degli indirizzi

[1] L'unica eccezione è costituita da programmi che fanno uso del hugetlbfs filesystem basato a sua volta sul ramfs filesystem.

del processo.
Nei processori Intel 80x86, il registro di controllo cr3 punta alla Page Global Directory del processo attualmente in esecuzione. Quando Linux salva il contesto hardware di un processo, esso non salva l'attuale contenuto di cr3 insieme agli altri registri. In effetti, Linux assegna una apposita Page Global Directory ad ogni processo che viene creato e tale tabella rimane in RAM finché il processo non viene eliminato. Si noti però che il contenuto di tale tabella può cambiare drasticamente – ad esempio, in seguito ad una API di tipo execve(). Per questo motivo, Linux usa un campo chiamato pgd all'interno del descrittore di tipo mm_struct che punta alla Page Global Directory del processo. Il contenuto di tale campo non viene mai modificato.

Tornando al nostro programma, dobbiamo quindi conoscere lo spiazzamento PGD di tale campo all'interno del descrittore di tipo mm_struct per potere accedere alla Page Global Directory. Fatto ciò, scandiamo le voci da 32 bit della tabella scartando quelle che contengono tutti 0 poiché tali voci mappano intervalli di indirizzi lineari che non fanno parte dello spazio degli indirizzi del processo. Se il processo prova ad eseguire una istruzione che richiede un indirizzo non valido, ad esempio, un indirizzo mappato da una voce nulla di una tabella di paginazione, il circuito MMU genera una eccezione di tipo Page Fault, il kernel riprende il controllo e provvede ad eliminare il processo.

Il resto è normale amministrazione. Dobbiamo consultare la documentazione Intel per scoprire la posizione ed il significato dei vari flag all'interno di ogni voce. Dobbiamo inoltre ricordarci di interpretare correttamente i 20 bit di indirizzo. Se il flag PS è uguale a 1, i 10 bit più significativi concatenati con 22 bit uguali a 0 rappresentano l'indirizzo fisico di una page frame da 4 MB. Se il flag PS è uguale a 0, i 20 bit dell'indirizzo concatenati con 12 bit uguali a 0 rappresentano l'indirizzo fisico di una page frame da 4 KB.
L'intero programma opportunamente documentato è allegato nel CD-ROM di questa rivista.
Ci limitiamo quindi ad illustrarne un possibile uso. Come sappiamo, Linx utilizza la paginazione su richiesta (demand paging). Possiamo verificare sperimentalmente tale fatto con il seguente programma di prova:

```
int main(void)
{
	int pid;
	unsigned int init_task_addr, i;
	int *bufp;

	pid = getpid();
	init_task_addr = INIT_TASK;
	dump_pgt(init_task_addr, pid, "dump0");
	bufp = malloc(8000000);
	for (i=0; i<2000000; i++)
		*(bufp + i) = 999;
	dump_pgt(init_task_addr, pid, "dump1");
	return 0;
}
```

La funzione principale del programma realizzato si chiama dump_pgt() e fa uso di due parametri: il PID del processo di cui si vogliono stampare le tabelle di paginazione ed il nome del file dove scrivere tali tabelle.
Nell'esempio riportato, la funzione dump_pgt() viene invocata due volte creando due file diversi chiamati dump0 e dump1. Tra la prima e la seconda invocazione, il programma di prova invoca una malloc() ottenendo un'area di memoria dinamica di 8 MB ed effettua operazioni di scrittura su di essa. In questo modo, il kernel è costretto ad allocare 2048 page frame supplementari al processo. Confrontando i due file dump0 e dump1, si verifica che la Page Global Directory descritta in dump1 contiene effettivamente due nuove voci non nulle per mappare gli 8 MB supplementari nonché due Page Table supplementari le cui 2048 voci puntano ad altrettante page frame.

Per saperne di più
Per stampare qualche altro flag ignorato da `dump_pgt()` oppure per avere ulteriori informazioni sull'architettura dei microprocessori basati sull'architettura IA-32, suggeriamo di consultare il testo *Intel Architecture Software Developer's Manual, vol. 3: System Programming* che può essere scaricato dal sito: http://developer.intel.com/design/pentium4/manuals/25366815.pdf
Per realizzare programmi che si basano sull'uso del device file `/dev/kmem`, è necessario sapere come funziona il kernel e conoscere le principali strutture di dati da esso utilizzate.
Al rischio di sembrare immodesti, suggeriamo il testo Understanding the Linux Kernel (3rd edition), O'Reilly 2005 scritto dall'autore di questo articolo e da Marco Cesati. Anche se tale testo è ormai superato in alcune parti (descrive il kernel Linux 2.6.11), esso presenta in modo sintetico le principali strutture di dati e gli algoritmi utilizzati da Linux 2.6 e può costituire una fonte di ispirazione per realizzare altri programmi simili a quello appena illustrato.

Avvertenze finali

Il codice descritto in precedenza avrà forse qualche pregio , ma certamente non ha quello della portabilità. In effetti, esso ispezionaalcune strutture di dati del kernel quindi può funzionare correttamente soltanto per uno specifico kernel Linux compilato con specifiche opzioni.

Abbiamo quindi incluso in un header file chiamato `dump_pgt.h` tutte le macro che dipendono dal tipo di kernel utilizzato. Nel sistema descritto in quest'articolo, tali macro sono impostate nel modo seguente:

```
#define PAGE_OFFSET 0xc0000000  /* verificare gli indirizzi lineari
                                   usati dalla System.map */
#define INIT_TASK       0xc035ef80  /* verificare la System.map */
#define INIT_TASK_OFFSET    INIT_TASK - PAGE_OFFSET
/* spiazzamenti di task_struct espressi in int
  (valido per linux 2.6.17)              */
#define NEXT            25
#define MM              31
#define PID             40
/* spiazzamenti di mm_struct espressi in int
  (valido per linux 2.6.17)              */
#define PGD             9
```

Prima di compilare il programma allegato, può quindi essere necessario modificare lo header file `dump_pgt.h` in modo da renderlo compatibile col kernel utilizzato.

Ricordiamo infine che il programma opera correttamente per microprocessori Intel 80x86 con al più 4 GB di RAM. Esso non è stato esteso a sistemi con più di 4 GB di RAM che fanno uso del PAE (Physical Address Extension).