KERNEL SPACE VS USER SPACE

Una differenza vitale

La memoria in un sistema GNU/Linux viene divisa in due distinte aree: lo spazio utente (**user space**) e lo spazio kernel (**kernel space**). User space è quella parte di memoria dove vivono i processi utente: il ruolo del kernel è di gestirne le applicazioni (processi) ivi presenti controllando che un processo non invada celle di memoria che non gli competono nel qual caso viene "assassinato", di fatto viene inviato un segnale **SIGSEGV** generando un errore di segmentazione (**Segment Violation**) ad indicare un accesso illegale alla memoria. Il kernel space, invece, indica quella parte di memoria dove il codice del kernel viene memorizzato e eseguito dal processore. Per quanto detto, i processi in user space hanno solo una parte limitata di memoria e se tentano di occupare memoria non di loro pertinenza vengono chiusi immediatamente. Il kernel, invece, ha accesso a tutta la memoria. Processi che girano in user space non hanno accesso alla memoria in kernel space, ma possono accedervi in minima parte solo attraverso un'interfaccia messa a disposizione dal kernel attraverso le syscall. Ricordiamo, infine, che anche i processi dell'utente root (amministratore) girano in user space: non si confondano i due concetti!

breria C funge da tramite per la omonima chiamata di sistema **read()** e un interrupt software (istruzione **int 0x80**) viene generato cambiando la modalità kernel e passando il controllo al gestore delle syscall (**syscall handler**). Poiché ad ogni chiamata di sistema è associato un ID, il syscall handler, previo uso della chiamata **sys_call_table**, accede alla omonima struttura che contiene l'indirizzo delle funzione kernel che stiamo richiedendo. Al ritorno dalla chiamata di sistema l'esecuzione riprende nella libreria C la quale ritorna all'applicazione utente. In **/usr/include/asm/unistd_64.h** vi sono gli identificativi di tutte le syscall implementate nel kernel a cui corrisponde una specifica macro di nome **__NR_nome_syscall**. Il syscall handler è così in grado di identificare la funzione kernel da richiamare i cui nomi iniziano tutti con il prefisso **sys_**, ad esempio **sys_read** per la lettura di un file. Ad esempio, la chiamata ptrace ha identificativo 101. Una chiamata al kernel porta all'esecuzione di un'istruzione privilegiata. Il codice in kernel space, poiché viene lanciato in **kernel mode**, verrà eseguito nella modalità **Ring 0** livello nel quale si ha accesso a tutte le istruzioni macchina. Nell'architettura x86 vengono definiti 4 livelli gerarchici di protezione o anelli di protezione: Ring 0 detto kernel mode, **Ring 1** e **Ring 2** ad uso dei servizi del sistema operativo: ad esempio i device driver in Ring 1 e Ring2, hypervisor di sistemi virtuali in Ring 0 e Ring -1, quest'ultimo creato ad-hoc affinché un sistema operativo guest sia grado di eseguire le operazioni a Ring 0 nativamente senza influenzare

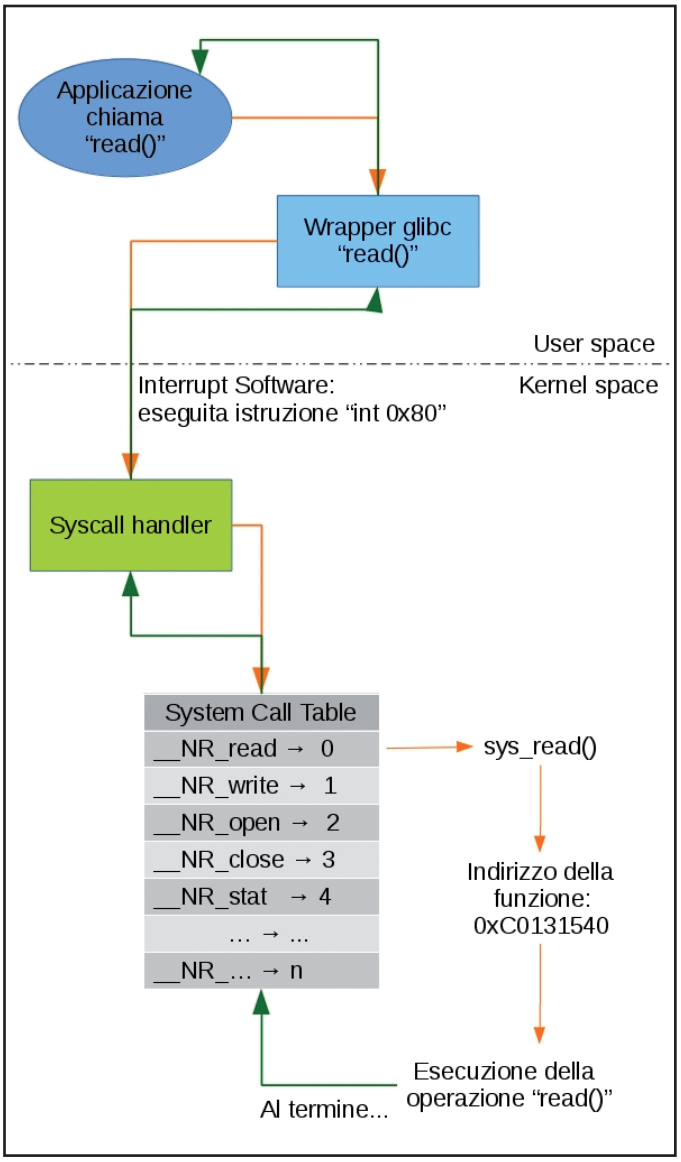

Fig. 4 • Invocazione di una chiamata di sistema

gli altri guest o il sistema host che lo ospita. Il livello con minori privilegi è **Ring 3** dove vengono eseguite le usuali applicazioni le quali hanno accesso solo ad un sottoinsieme molto limitato delle istruzioni del processore.

FINALMENTE YAMA!

Ora che abbiamo una panoramica di tutti gli attori in gioco dobbiamo tirare solo le somme e il testo sulla descrizione di Yama appare sicuramente più comprensibile: Yama estende il modello DAC introducendo un ulteriore livello di controllo, attraverso il framework LSM, sulla chiamata di sistema **ptrace().** Al momento di scrivere, il controllo su ptrace() è l'unica funzione disponibile, ma non si può escludere che di nuove verranno implementate in futuro visti i tempi necessari. Infatti, nel caso specifico, la prima proposta avvenne nel Giugno 2010