Segítség a bajban: User-Mode Linux

A rendszermagtérben való programozás mindig is a guruk szakterülete volt. Kevés embernek van elegendő bátorsága, tudása és türelme a megszakítások, az eszközök és a sötét árnyként lebegő „magpánik” világában.

Amennyiben felhasználói térben írunk programot, a legrosszabb, ami történhet, hogy programunk „coredumpol”, vagyis leáll és memórialenyomata a lemezre kerül. Ilyenkor programunk valamit nagyon rosszul csinált, ezért az operációs rendszer úgy döntött, hogy a program által lefoglalt teljes memóriát és állapotadatot *core* fájl formájában visszaadja nekünk. A *core* fájlt ezután felhasználhatjuk a hibakereséshez, és kijavíthatjuk a problémát.
Amikor azonban a rendszermagot programozzuk, nincs semmilyen operációs rendszer, amely közbeléphetne és biztonságosan leállíthatná a programunkat, majd megmondaná, hogy mi volt a gond. Igaz, a Linux-rendszermag elég „rendes” saját kódjához, ugyanis néha még a pánikot is képes túlélni, amennyiben az elkövetett hiba viszonylag kicsi (ezeket a pánikokat általában „oops”-oknak nevezik). Ugyanakkor programunkat semmi sem akadályozhatja meg, hogy felülírja vagy elérje a rendszermag címterében elhelyezkedő bármely memóriaterületet. Másrészt ha a modul lefagy, a rendszermag is így tesz (gyakorlatilag a pillanatnyi rendszermagszál fagy le, azonban az eredmény általában ugyanaz).
Ezek a nehézségek esetleg nem tűnhetnek túl veszélyesnek, pedig komolyan kell vennünk őket. Ha a rendszermag pánikol, a legritkább esetben tudjuk meg, hogy tulajdonképpen mi is okozta. Gyakori megoldás, hogy `printk`-kat pakolunk mindenhová, és reménykedünk, hogy belebotlunk a hibába, mielőtt az üzenetek újraindításkor eltűnnének. Mindez természetesen feltételezi, hogy a fájlrendszert nem károsítottuk. Egyszer már elvesztettem a teljes fájlrendszeremet egy rosszkor érkező rendszermagpánik során (és főként amiatt, hogy egy rosszul meghatározott mutató az `ext2` belső adatszerkezeteinek egy részét felülírta).
Az első dolog, amit nem árt megjegyezni, ha rendszermag-programozásra adjuk fejünket: minden forrást tartsunk NFS-en. A másik gépen biztonságban maradnak a fájlok. Azonban ez sem ment meg minket attól, hogy minden egyes pánik után futtatnunk kelljen az `e2fsck`-t. Ráadásul így is elveszthetjük fájlrendszerünket, még ha a forráskód a másik gépen biztonságban is van.
Így aztán talán nem meglepő, hogy milyen kevesen vágnak bele a rendszermag programozásába. Most azonban mindez megváltozhat.

Virtuális gépek és az UML

Réges-régen, a nagygépes időkben, amikor még az időmegosztásos gépek voltak a legelterjedtebbek, megszületett a virtuális gép gondolata. A virtuális gép egy olyan beágyazott számítógép, amely teljes egészében a rendelkezésünkre áll. A virtuális gép fizikai alkatrészeket közvetlenül nem érheti el. A vassal való összes kapcsolatot a számítógép vagy egy emulátor ellenőrzi.
A VMware (➲ http://www.vmware.com) például egy meglehetősen hatékony virtuális gépet készített, amelyen minden x86-alapú operációs rendszer futtatható Windows NT, 2000, XP vagy Linux alatt. A SoftPC (egy 8086 emulátor, amely Windows- és DOS-programok futtatását teszi lehetővé) Motorola 68 k-alapú számítógépeken (vagyis Macintoshon) 1988 óta használható.
A valódi virtuális gépek néha túlságosan drágák lehetnek az érdeklődők költségvetéséhez képest (a VMware Workstation for Linux ára a honlapjukon közzétettek alapján: 299 dollár). Szerencsére immár azok számára is létezik ingyenes lehetőség, akik Linux alatt szeretnének dolgozni: a User-Mode Linux (UML).
A User-Mode Linux (➲ http://user-mode-linux.sourceforge.net) nem teljes értékű virtuális gép. Nem emulálja a különböző alkatrészeket, és nem teszi lehetővé, hogy más operációs rendszereket futtassunk. Megengedi viszont, hogy a rendszermagot a felhasználói térben futtassuk. A fejlesztés során számos előnyre tehetünk szert ezáltal: a gazdagép fájlrendszere védett marad, a virtuális fájlrendszer visszavonhatatlan (ez a károsodástól óvja meg), több gépet is futtathatunk egyetlen számítógépen (ez különösen gépközi kapcsolattartás, például hálózati üzenetek ellenőrzése során lehet hasznos, hiszen nem kell több számítógépet használnunk), illetve a rendszermagot igen könnyen futtathatjuk a hibakeresőben.

Az UML beállítása

Az UML futtatása egyszerű: valamelyik bináris csomagot (rendszermagbináris és néhány további eszköz) avagy a rendszermagfoltot is letölthetjük. Ezenkívül szükségünk lesz még a fájlrendszerre is. Azt javaslom, először játszunk el a binárisokkal, és csak azután fogjunk egy olyan saját rendszermag összeállításába, amely igényeinket jobban kielégíti. A HOWTO ezeket a témaköröket bővebben taglalja.
Az UML egyik nagy előnye a „Copy-on-Write” fájlok alkalmazásában rejlik. Ezek a fájlok lehetővé teszik, hogy a virtuális fájlrendszert az alapfájlrendszer módosítása nélkül változtassuk meg. A fájlrendszeren végrehajtott minden írási művelet vagy módosítás ezekbe az (általában *.cow* végződésű) fájlokba kerül. Így ha munka közben sikerült pánikba kergetni a fájlrendszert, mindössze annyit kell tennünk, hogy töröljük a *.cow* fájlt (ez újra létre fog jönni), és meghibásodott fájlrendszerünk máris eredeti állapotában tért vissza (léteznek olyan eszközök is, amelyekkel a *.cow* fájlban tárolt változásokat az eredeti fájlrendszerbe dolgozhatjuk be – abban az esetben, ha meg szeretnénk tartani őket).

Hibakereső modulok

Most, hogy az UML már fut, ideje alkotnunk valamit. Kipróbálás céljából írtam egy nagyon egyszerű rendszermagmodult. Négy eszközt használ: */dev/gentest[0-3]*. A modul minden eszközt egy kicsit másképpen kezel. Az első eszköz a süllyesztő (mint a */dev/null*). A második későbbi felhasználásra tárol karaktersoroza-

```
GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
0xa00d5631 in __kill ()
Breakpoint 1 at 0xa000e36b: file panic.c, line 52.
Breakpoint 2 at 0xa00c867e: file user_util.c, line 104.
Breakpoint 3 at 0xa00035bf: file init/main.c, line 553.

Breakpoint 3, start_kernel () at init/main.c:553
553             printk(linux_banner);
(gdb) c
Continuing.
```

1. kép Az UML futtatása GDB alatt

```
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0xa00e8f91 in __libc_nanosleep ()
(gdb) print *module_list
$1 = {size_of_struct = 84, next = 0xa0158720, name = 0xa3008b00 "genTest",
  size = 3664, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
  nsyms = 10, ndeps = 0, syms = 0xa3008e00, deps = 0x0, refs = 0x0,
  init = 0xa3008054, cleanup = 0xa3008078, ex_table_start = 0x0,
  ex_table_end = 0x0, persist_start = 0x0, persist_end = 0x0, can_unload = 0,
  runsize = 0, kallsyms_start = 0x0, kallsyms_end = 0x0,
  archdata_start = 0x83e58955 <Address 0x83e58955 out of bounds>,
  archdata_end = 0xa26810ec "",
  kernel_data = 0x68a30087 <Address 0x68a30087 out of bounds>}
(gdb) printf "0x%08x\n", (int)module_list + module_list->size_of_struct
0xa3008054
(gdb) add-symbol-file ~/work/kHacking/genTest/genTest.o 0xa3008054
add symbol table from file "/home/chaos/work/kHacking/genTest/genTest.o" at
        .text_addr = 0xa3008054
(y or n) y
Reading symbols from /home/chaos/work/kHacking/genTest/genTest.o...
done.
(gdb)
```

2. kép Modullista

```
  runsize = 0, kallsyms_start = 0x0, kallsyms_end = 0x0,
  archdata_start = 0x83e58955 <Address 0x83e58955 out of bounds>,
  archdata_end = 0xa26810ec "",
  kernel_data = 0x68a30087 <Address 0x68a30087 out of bounds>}
(gdb) printf "0x%08x\n", (int)module_list + module_list->size_of_struct
0xa3008054
(gdb) add-symbol-file ~/work/kHacking/genTest/genTest.o 0xa3008054
add symbol table from file "/home/chaos/work/kHacking/genTest/genTest.o" at
        .text_addr = 0xa3008054
(y or n) y
Reading symbols from /home/chaos/work/kHacking/genTest/genTest.o...
done.
(gdb) print *module_list
$2 = {size_of_struct = 84, next = 0xa0158720, name = 0xa3008b00 "genTest",
  size = 3664, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
  nsyms = 10, ndeps = 0, syms = 0xa3008e00, deps = 0x0, refs = 0x0,
  init = 0xa3008054 <init_module>, cleanup = 0xa3008078 <cleanup_module>,
  ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
  persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start = 0x0,
  kallsyms_end = 0x0,
  archdata_start = 0x83e58955 <Address 0x83e58955 out of bounds>,
  archdata_end = 0xa26810ec "",
  kernel_data = 0x68a30087 <Address 0x68a30087 out of bounds>}
(gdb)
```

3. kép Szimbólumfájl betöltése

tokat. A modul állapotát a harmadik eszközről olvashatjuk ki, végül a nullás eszköz a másik három eszköz bármelyike lehet aszerint, hogyan állítottuk be (a beállítást `ioctl`-hívásokkal módosíthatjuk). A rendszermagmodult megtalálhatjuk a 34. CD Magazin/UML könyvtárában, vagy a ➲ http://www.frascone.com/kHacking/gentest-0.1.tar.gz címről tölthetjük le.

Hibakeresés printkval

Nos, akkor idézzünk elő egy gonosz hibát! Tegyük fel, hogy amikor valaki megnyitja a negyedik eszközt (***cat /dev/gentest4***), a modul aljas módon végtelen ciklusba kerül: `for(;;) i++;` (lásd az 1. listát: 34. CD Magazin/UML könyvtár). A `deadlocks` leállásokat vagy fagyásokat jelent ezek gyakori hibák programírás során. Néha bizony igen nehéz felfedezni őket. Egy programozó ilyenkor egyszerűen `printk`-kat használna a hiba felderítésére: `printk("IdeØrtem!\n");`. Ez a fajta hibakeresés valóban működik, azonban a rendszer így is jó párszor kifagy, mire a hibát megtaláljuk. Folyamatos `fsck` futtatást feltételezve ez elég kellemetlen lehet. Az UML segítségével

```
This GDB was configured as "i386-redhat-linux"...
0xa00d5631 in __kill ()
Breakpoint 1 at 0xa000e36b: file panic.c, line 52.
Breakpoint 2 at 0xa00c867e: file user_util.c, line 104.
Breakpoint 3 at 0xa00035bf: file init/main.c, line 553.

Breakpoint 3, start_kernel () at init/main.c:553
553             printk(linux_banner);
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0xa00e8f91 in __libc_nanosleep ()
(gdb) printf "0x%08x", (int)module_list + module_list->size_of_struct
0xa3008054(gdb) add-symbol-file ~/work/kHacking/genTest/genTest.o 0xa3008054
add symbol table from file "/home/chaos/work/kHacking/genTest/genTest.o" at
        .text_addr = 0xa3008054
(y or n) y
Reading symbols from /home/chaos/work/kHacking/genTest/genTest.o...
done.
(gdb) print *module_list
$1 = {size_of_struct = 84, next = 0xa0158720, name = 0xa3008b00 "genTest",
  size = 3664, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
  nsyms = 10, ndeps = 0, syms = 0xa3008e00, deps = 0x0, refs = 0x0,
  init = 0xa3008054 <init_module>, cleanup = 0xa3008078 <cleanup_module>,
  ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
  persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start = 0x0,
  kallsyms_end = 0x0,
  archdata_start = 0x83e58955 <Address 0x83e58955 out of bounds>,
  archdata_end = 0xa26810ec "éu\aJG\030kkkk\213Ué\212\002<;t\t\204Ât\téØþkkB\211
Ué\213ué\200>", kernel_data = 0x68a30087 <Address 0x68a30087 out of bounds>}
(gdb) break gOpen
Breakpoint 4 at 0xa30082ce: file gentest.c, line 254.
(gdb) c
Continuing.

Breakpoint 4, gOpen (Inode=0xa22ae640, fp=0xa2360620) at gentest.c:254
254             printk("%s: (%d:%d) open(%p, %p)\n",
(gdb)
```

4. kép Töréspontok beállítása

```
add symbol table from file "/home/chaos/work/kHacking/genTest/genTest.o" at
        .text_addr = 0xa3008054
(y or n) y
Reading symbols from /home/chaos/work/kHacking/genTest/genTest.o...
done.
(gdb) print *module_list
$2 = {size_of_struct = 84, next = 0xa0158720, name = 0xa3008b00 "genTest",
  size = 3664, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
  nsyms = 10, ndeps = 0, syms = 0xa3008e00, deps = 0x0, refs = 0x0,
  init = 0xa3008054 <init_module>, cleanup = 0xa3008078 <cleanup_module>,
  ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
  persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start = 0x0,
  kallsyms_end = 0x0,
  archdata_start = 0x83e58955 <Address 0x83e58955 out of bounds>,
  archdata_end = 0xa26810ec "",
  kernel_data = 0x68a30087 <Address 0x68a30087 out of bounds>}
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0xa3008314 in gOpen (Inode=0xa229ca00, fp=0xa2360620) at gentest.c:269
269             for (;;) i++;
(gdb) list
264
265         // Hang when device 4  is opened
266         if (MINOR(Inode->i_rdev) == 4) {
267             int i;
268             printk("Computing pi to the last decimal position . . .\n");
269             for (;;) i++;
270         }
271
272         return 0;
273     } // gOpen
(gdb) where
#0  0xa3008314 in gOpen (Inode=0xa229ca00, fp=0xa2360620) at gentest.c:269
#1  0xa00308b6 in chrdev_open (inode=0xa229ca00, filp=0xa2360620)
    at devices.c:153
#2  0xa005f02f in devfs_open (inode=0xa229ca00, file=0xa2360620) at base.c:2739
#3  0xa002f8a7 in dentry_open (dentry=0xa22eecc0, mnt=0xa08853a0, flags=32768)
    at open.c:688
#4  0xa002f7ab in filp_open (filename=0xa2660000 "/dev/gentest4", flags=32768,
    mode=0) at open.c:646
#5  0xa002faef in sys_open (filename=0x9ffffe7d "/dev/gentest4", flags=32768,
    mode=0) at open.c:788
#6  0xa00c58b3 in execute_syscall (regs=
        {regs = {2684354173, 32768, 0, 1073819696, 1074649107, 2684353628, 42949
67258, 43, 43, 0, 0, 5, 1074419060, 35, 518, 2684353580, 43}})
    at syscall_kern.c:410
#7  0xa00c599e in syscall_handler (unused=0x0) at syscall_user.c:70
(gdb)
```

5. kép Fagyáspróba

egyszerűen csak beírjuk a `printk`-kat, és mindig új fájlrendszert indítunk a kipróbáláshoz.

Az UML a `printk`-kal segített ezt a hibát megtalálni, de ez a hiba igazából néhány újraindításnál komolyabb bosszúságot nem okozott volna. Itt az ideje, hogy elkészítsük első igazán gonosz hibánkat. Tegyük fel, hogy valaki az ötös eszközről olvas (azaz ***cat /dev/gentest5***), és a modul elkezdi a teljes memóriát felülírni: `memset(0, 0, 0xffffffff);` (lásd *a 2. listát*). A memória felülírása elég általános hiba a C programokban. Rendszermagban különösen kellemetlen, és néha azonnali újraindítást eredményezhet, így a kiadott `printk`-kból az égvilágon semmit sem látunk. Ezeket a hibákat is fel lehet `printk`-kal deríteni, csakhogy ez igen időigényes feladat.

Hibakeresés GDB-vel

Mint korábban említettem, az UML kitűnő hibakereső eszköz. Biztonságban tarthatjuk fájlrendszerünket, mialatt hibát keresünk a modulokban, de még ennél is többet tud, hiszen itt van nekünk a GDB.

A legtapasztaltabb rendszermag-programozók tudják, hogy létezik egy mód a rendszermag hibakeresésére GDB-vel, mégpedig a soros vonalon keresztül. Csakhogy – legalábbis tapasztalataim szerint – ez nem működik valami jól. A GDB csak pislákol a rendszermagban, néha ki is fagy, ráadásul mindehhez

```
        .text_addr = 0xa3008054
(y or n) y
Reading symbols from /home/chaos/work/kHacking/genTest/genTest.o...
done.
(gdb) c
Continuing.

Breakpoint 1, panic (
    fmt=0xa0137ee0 "Kernel mode fault at addr 0x%lx, ip 0x%lx") at panic.c:52
52              bust_spinlocks(1);
(gdb) where
#0  panic (fmt=0xa0137ee0 "Kernel mode fault at addr 0x%lx, ip 0x%lx")
    at panic.c:52
#1  0xa00c6bac in segv (address=0, ip=2734719432, is_write=2, is_user=0)
    at trap_kern.c:94
#2  0xa00c7800 in segv_handler (sig=11, sc=0xa22abbc0, usermode=0)
    at trap_user.c:400
#3  0xa00c7992 in sig_handler (sig=11, sc=
      {gs = 0, __gsh = 0, fs = 0, __fsh = 0, es = 43, __esh = 0, ds = 43, __dsh
= 0, edi = 0, esi = 1024, ebp = 2720710320, esp = 2720710296, ebx = 2720623584,
edx = 0, ecx = 1073741823, eax = 0, trapno = 14, err = 6, eip = 2734719432, cs =
 35, __csh = 0, eflags = 66178, esp_at_signal = 2720710296, ss = 43, __ssh = 0,
fpstate = 0xa22abc18, oldmask = 0, cr2 = 0}) at trap_user.c:459
#4  <signal handler called>
#5  0xa30081c8 in gRead (fp=0xa2354620, userBuffer=0x804b220 "", bufSiz=1024,
    offset=0xa2354640) at /usr/src/uml/linux/include/asm/arch/string.h:488
#6  0xa0030096 in sys_read (fd=3, buf=0x804b220 "", count=1024)
    at read_write.c:162
#7  0xa00c58b3 in execute_syscall (regs=
      {regs = {3, 134525472, 1024, 1024, 134525472, 2684353564, 4294967258, 43
, 43, 0, 0, 3, 1074419252, 35, 514, 2684353516, 43}}) at syscall_kern.c:410
#8  0xa00c599e in syscall_handler (unused=0x0) at syscall_user.c:70
(gdb) frame 5
#5  0xa30081c8 in gRead (fp=0xa2354620, userBuffer=0x804b220 "", bufSiz=1024,
    offset=0xa2354640) at /usr/src/uml/linux/include/asm/arch/string.h:488
488                     default: COMMON("\n\tstosw\n\tstosb"); return s;
(gdb)
```

6. kép Memóriafelülírás

két gépre is szükségünk van. Sikeresen kerestem rendszermag-hibákat oly módon, hogy a VMWare alatt futó virtuális gép soros kapuját egy fájlba irányítottam át, azonban így is elég nehézkes volt, mivel a GDB rendszermagrésze néha kifagyott. Az UML segítségével mindez a már múlté: az UML lehetővé teszi, hogy a teljes virtuális gépet GDB-be helyezzük, és futás közben vagy akár a pánik után is csatlakozzunk. A UML GDB alatti elindításának legegyszerűbb módja, ha indításkor megadjuk a `debug` parancssori kapcsolót. Az UML ezután a GDB-t egy Xterm-ablakban elindítja, majd megállítja a rendszermagot. Ilyenkor a rendszermag indulásának folytatásához általában egyszerűen csak ***c***-t szokás ütni (lásd *1. képünkön*).

A modul hibakereséséhez először be kell tölteni a modult, aztán meg kell mondani a GDB-nek, hol találja a szimbólumfájlt, végül beállíthatjuk a szükséges töréspontokat.

Menjünk sorban: először töltsük be a modult. A forráskódhoz mellékelve találunk egy `loadModule` nevű egyszerű héjprogramot, amely betölti a modult, és amennyiben még nem léteznének, az eszközöket is elkészíti.

A modul betöltése után a GDB-ablakban nyomjuk le a CTRL-C billentyűket, ezzel a rendszermagot megállítjuk, majd nézzük meg a `module_list` mutatót. Az utoljára betöltött modulnak a lista elején kell lennie. A modul címét egy egyszerű `printf` paranccsal lekérhetjük. Erre szükségünk is lesz majd, amikor a szimbólumfájlt betöltjük (lásd *2. képünkön*).

Most töltsük be a szimbólumfájlt az `add-symbol-file MODUL_EL R SI_ T MEM RIAC˝M` paranccsal. A felhasznált fájlnév a gazdagép rendszerén értendő és nem a virtuális gépen. Miután ***y***-t válaszoltunk az „Are you sure you know what you're doing?" (Biztos, hogy tudod, mit csinálsz?) kérdésre, a szimbólumfájl betöltődik. Ha újra megnézzük a `module_list` mutatót, ellenőrizhetjük, hogy valóban helyesen töltődött-e be. Figyeljük meg, hogy az `init` és `cleanup` mutatók most már a címüknek megfelelő függvénynevekkel jelennek meg (lásd *3. képünkön*).

Attól kezdve, hogy a modult betöltöttük, tetszés szerinti töréspontot állíthatunk be. Én a megnyitáshoz állítottam be egy töréspontot, majd megpróbáltam `cat`-elni az egyik eszközt (lásd *4. képünkön*).

Futtassuk le két tesztünket, és nézzük meg, mennyire lesz nehéz meglelni a hibát, amennyiben GDB-t használunk. Az első próba során a rendszer lefagy. Csakhogy most a hibakeresőben leüthetjük a CTRL-C billentyűket, és megtekinthetjük, hol állt le. A leállaspróbában (lásd *5. képünkön*) teljesen nyilvánvaló, hogy a megállás helye a `for` cikluson belül van. Ha játszani akarunk, az `i` változó tartalmát ki is írathatjuk, hogy láthassuk, mi történik.

A memóriafelülírás egy kicsit bonyolultabb. Nem a pánik miatt, hanem mert a `memset`-et használtam. A `memset` a GNU `libc`-ben beágyazott assemblykódok beszúrásával végződik, így aztán úgy néz ki, mintha a hiba a `string.h`-ban, és nem pedig a modulunkban lenne. Ennek ellenére azért megmutatja, melyik függvényben fordult elő a hiba, így megtudhatjuk, hogy a `memset` a bűnös (lásd *6. képünkön*).

Ezenkívül a hiba felfedéséhez a pillanatnyi függvény bármelyik helyi változóját megvizsgálhatjuk (`gRead`), vagy megtekinthetjük a globális változókat.

Összefoglalás

Bár az UML egy eszközmeghajtó írásában valószínűleg nem sokat segít (mivel nem éri el a gép fizikai alkatrészeit), azért felbecsülhetetlen segítséget nyújt a rendszermagmodulok hibakeresésében. Lehetővé teszi, hogy olyan könnyedén írjunk rendszermagmodulokat, mintha bármilyen más C-programot fejlesztenénk, ráadásul pánikoktól, fagyástól vagy adatvesztéstől sem kell tartanunk. Hasznos eszköz minden rendszermag-programozó fegyvertárában.

Linux Journal május, 97. szám

David Frascone
(dave@frascone.com) jelenlegi munkahelye a SunLabs division of Sun Microsystems, Inc. Pillanatnyilag a Diameter (AAA) kivitelezése a feladata. Az IETF AAA munkacsoportjának tagja.

További érdekességek

Az O'Reilly & Associates, Inc. kiadásában 1998-ban jelent meg *Alessandro Rubini* Linux Device Drivers című könyve.

The Linux Kernel Hackers' Guide
➲ http://www.linuxdoc.org/LDP/khg/HyperNews/get

Szintén az O'Reilly & Associates, Inc., kiadásában olvasható *Daniel P. Bovet* és *Marco Cesati* Understanding the Linux Kernel 2001 című műve.

A The User-Mode Linux Kernel honlapja
➲ http://user-mode-linux.sourceforge.net