Kísérletezgetés a Ptrace-szel (2. rész)

Sorozatunk második részében Pradeep néhány komolyabb témát boncolgat: a töréspontok beállítását, illetve kód beszúrását a már futó folyamatokba.

Sorozatunk első részében (Linuxvilág 2003. január) láthattuk, hogyan lehet a Ptrace felhasználásával követni a rendszerhívásokat, illetve megváltoztatni a rendszerhívások jellemzőit. Most olyan összetettebb módszereket ismerhetünk meg, mint a töréspontok beállítása vagy a kódnak futó programokba történő beillesztése. A hibakeresők ezeket a módszereket töréspontok beállítására és hibakereső-kezelők futtatására használhatják. Akárcsak az előző részben, a mostani írásunkban található valamennyi kód is i386 architektúrára épül.

Csatlakozás a futó folyamatokhoz

Az első részben a követendő folyamatot gyermekként futtattuk a `ptrace(PTRACE_TRACEME, ..)` hívás után. Amennyiben csak arra vagyunk kíváncsiak, hogyan ad ki a program rendszerhívásokat, ez elegendő is. Ha azonban már futó folyamatot szeretnénk követni vagy elemezni, inkább a `ptrace(PTRACE_ATTACH, ..)` hívást használjuk.
Ha a `ptrace(PTRACE_ATTACH, ..)` függvénynek a követendő folyamatazonosítót (pidet) átadjuk, megközelítőleg azonos hatást érünk el, mintha a folyamat a `ptrace(PTRACE_TRACEME, ..)` hívást használná és a nyomkövető folyamat gyermekévé válna. A követett folyamat `SIGSTOP` üzenetet kap, így a szokásos módon megvizsgálhatjuk vagy módosíthatjuk. Miután végeztünk a módosítással vagy követéssel, a követett folyamatot a `ptrace(PTRACE_DETACH, ..)` hívással az útjára engedhetjük.
A következő kód egy kis követőprogramra mutat példát:

```
int main()
{    int i;
     for(i = 0;i < 10; ++i) {
          printf("SzÆmlÆl : %d\n", i);
          sleep(2);
     }
     return 0;
}
```

Mentsük a programot ***dummy2.c*** néven. Fordítsuk le és futtassuk:

```
gcc -o dummy2 dummy2.c
./dummy2 &
```

Ezekután az alábbi kód segítségével tudunk csatlakozni a dummy2-höz:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
/* Az user_regs_struct-hoz stb. */
int main(int argc, char *argv[])
{    pid_t traced_process;
     struct user_regs_struct regs;
     long ins;

     if(argc != 2) {
          printf("HasznÆlat: %s <k vetendı
                                  ➥pid>\n",
                 argv[0], argv[1]);
          exit(1);
     }

     traced_process = atoi(argv[1]);
     ptrace(PTRACE_ATTACH, traced_process,
            ➥NULL, NULL);
     wait(NULL);
     ptrace(PTRACE_GETREGS, traced_process,
            ➥NULL, &regs);
     ins = ptrace(PTRACE_PEEKTEXT,
                  ➥traced_process,
                  ➥regs.eip, NULL);
     printf("EIP: %lx vØgrehajtott utas tÆs:
            ➥%lx\n", regs.eip, ins);
     ptrace(PTRACE_DETACH, traced_process,
            ➥NULL, NULL);

     return 0;
}
```

A fenti program egyszerűen csatlakozik a folyamathoz, megvárja, amíg megáll, megvizsgálja az eip (utasításszámláló) tartalmát, majd leválik.
A követett folyamat megállása után a `ptrace (PTRACE_POKETEXT, ..)` és a `ptrace(PTRACE_POKEDATA, ..)` függvényt használhatjuk kódbeillesztéshez.

Töréspontok beállítása

Hogyan állítanak be a hibakeresők töréspontokat? Általában a végrehajtandó utasítást egy csapdautasításra cserélik le, így amikor a követett program megáll, a követőprogram (azaz a hibakereső) nyugodtan vizsgálódhat. Amikor aztán a követőprogram folytatni akarja a követett kódot, egyszerűen visszahelyettesíti az eredeti utasítást. Íme egy példa:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
const int long_size = sizeof(long);

void getdata(pid_t child, long addr,
             ➥char *str, int len)
```

```
{    char *laddr;
     int i, j;
     union u {
             long val;
             char chars[long_size];
     }data;

     i = 0;
     j = len / long_size;
     laddr = str;
     while(i < j) {
          data.val = ptrace(PTRACE_PEEKDATA,
                            ➥child, addr +
                            ➥i * 4, NULL);
          memcpy(laddr, data.chars, long_size);
          ++i;
          laddr += long_size;
     }

     j = len % long_size;
     if(j != 0) {
          data.val = ptrace(PTRACE_PEEKDATA,
                            ➥child, addr
                            ➥+ i * 4, NULL);
          memcpy(laddr, data.chars, j);
     }
     str[len] = ·\0·;
}

void putdata(pid_t child, long addr,
             char *str, int len)
{    char *laddr;
     int i, j;
     union u {
             long val;
             char chars[long_size];
     }data;
     i = 0;
     j = len / long_size;
     laddr = str;
     while(i < j) {
          memcpy(data.chars, laddr, long_size);
          ptrace(PTRACE_POKEDATA, child,
                 ➥addr + i * 4, data.val);
          ++i;
          laddr += long_size;
     }
     j = len % long_size;
     if(j != 0) {
          memcpy(data.chars, laddr, j);
          ptrace(PTRACE_POKEDATA, child,
                 ➥addr + i * 4, data.val);
     }
}

int main(int argc, char *argv[])
{    pid_t traced_process;
     struct user_regs_struct regs, newregs;
     long ins;
     /* int 0x80, int3 */
     char code[] = {0xcd,0x80,0xcc,0};
     char backup[4];

     if(argc != 2) {
          printf("HasznÆlat: %s <k vetendı
                 ➥pid>\n", argv[0], argv[1]);
          exit(1);
     }
     traced_process = atoi(argv[1]);

     ptrace(PTRACE_ATTACH, traced_process,
            ➥NULL, NULL);
     wait(NULL);

     ptrace(PTRACE_GETREGS, traced_process,
            ➥NULL, &regs);

     /* Utas tÆs mÆsolÆsa ideiglenes tÆrol ba*/
     getdata(traced_process, regs.eip,
             ➥backup, 3);

     /* T rØspont elhelyezØse */
     putdata(traced_process, regs.eip, code, 3);

     /* Engedj k tovÆbb a folyamatot Øs vÆrjuk
     meg, am g vØgrehajtja az int 3 utas tÆst */
     ptrace(PTRACE_CONT, traced_process, NULL,
            ➥NULL);

     wait(NULL);
     printf("A folyamat leÆllt, visszarakjuk
            ➥az eredeti utas tÆsokat\n");
     printf("FolyatatÆshoz nyomja le az
            ➥<enter> billentyıt\n");
     getchar();
     putdata(traced_process, regs.eip, backup, 3);

     /* Az eip-t visszaÆll tjuk az eredeti
     utas tÆsra, hogy a folyamat tovÆbb
     futhasson */
     ptrace(PTRACE_SETREGS, traced_process,
            ➥NULL, &regs);

     ptrace(PTRACE_DETACH, traced_process,
            ➥NULL, NULL);
     return 0;
}
```

Az előbb tehát három bájtot a csapdautasítás kódjával helyettesítettünk, majd amikor a folyamat megállt, az eredeti utasítást visszahelyettesítettük, azután az eredeti helyére állítottuk vissza az eip-t. Az *1–4. ábra* segít megérteni, hogyan néz ki az utasításfolyam a fenti program végrehajtásakor.
Most, hogy már tisztáztuk a töréspont-beillesztés módszerének a hátterét, szúrjunk be pár kódbájtot a futó programba. A kódbájtok a „hello world” üzenetet fogják megjeleníteni.
Következő programunk csak egy egyszerű, de az igényeinkhez igazított „hello world” program lesz. A programot a következő paranccsal fordítsuk le:

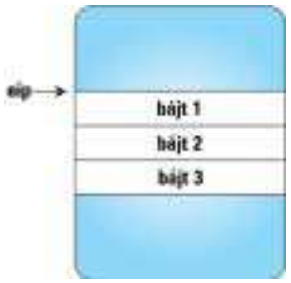
```
gcc -o hello hello.c

void main()
{
__asm__("
       jmp forward
```

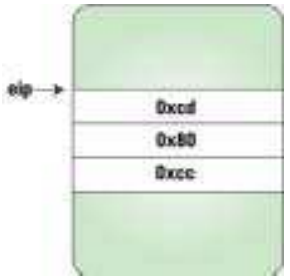
```
backward:
    popl    %esi        # A hello world sz veg
                        # c mØnek lekØrØse
    movl    $4, %eax    # ´rÆsi rendszerh vÆs
                        # kØrØse
    movl    $2, %ebx
    movl    %esi, %ecx
    movl    $12, %edx
    int     $0x80
    int3                # T rØspont. Itt a
                        # program megÆll,
                        # Øs a vezØrlØst
                        # visszaadja
                        # a sz lınek
forward:
    call    backward
    .string \"Hello World\\n\""
    );
}
```

Az előre-hátra (forward/backward címekre) ugrálásra azért van szükségünk, hogy a „hello world” szöveg címét megállapíthassuk.
A fenti assemblyhez tartozó gépi kódot lekérhetjük a GDB-ből. Indítsuk be a GDB-t, és fejtsük vissza a programot:

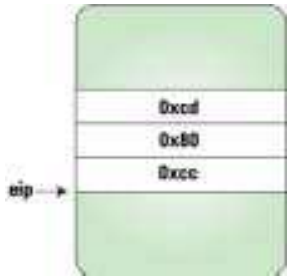
```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483e0 <main>:   push   %ebp
0x80483e1 <main+1>: mov    %esp,%ebp
0x80483e3 <main+3>: jmp    0x80483fa <forward>
End of assembler dump.
(gdb) disassemble forward
Dump of assembler code for function forward:
0x80483fa <forward>:        call   0x80483e5
<backward>
0x80483ff <forward+5>:      dec    %eax
0x8048400 <forward+6>:      gs
0x8048401 <forward+7>:      insb
(%dx),%es:(%edi)
0x8048402 <forward+8>:      insb
(%dx),%es:(%edi)
0x8048403 <forward+9>:      outsl
%ds:(%esi),(%dx)
0x8048404 <forward+10>:     and
%dl,0x6f(%edi)
0x8048407 <forward+13>:     jb     0x8048475
0x8048409 <forward+15>:     or
%fs:(%eax),%al
0x804840c <forward+18>:     mov    %ebp,%esp
0x804840e <forward+20>:     pop    %ebp
0x804840f <forward+21>:     ret
End of assembler dump.
(gdb) disassemble backward
Dump of assembler code for function backward:
0x80483e5 <backward>:       pop    %esi
0x80483e6 <backward+1>:     mov    $0x4,%eax
0x80483eb <backward+6>:     mov    $0x2,%ebx
0x80483f0 <backward+11>:    mov    %esi,%ecx
0x80483f2 <backward+13>:    mov    $0xc,%edx
0x80483f7 <backward+18>:    int    $0x80
0x80483f9 <backward+20>:    int3
End of assembler dump.
```



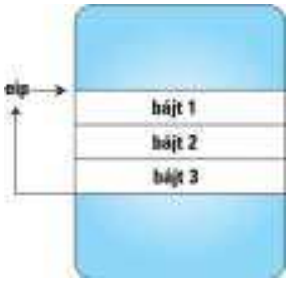
1. ábra
Miután a folyamat megállt



2. ábra Miután a csapda-utasításbájtokat elhelyeztük



3. ábra A csapda működésbe lépett, és a vezérlés a nyomkövető programhoz került



4. ábra Az eredeti utasítások visszahelyettesítése után az eip-t az eredeti helyre állítjuk vissza

Nekünk a `main+3`-tól a `backward+20`-ig van szükségünk a kódra, ami összesen 41 bájtot jelent. A gépi kódot GDB alatt az x paranccsal jeleníthetjük meg:

```
gdb x/40bx main+3
<main+3>:     eb 15 5e b8 04 00 00 00
<backward+6>:bb 02 00 00 00 89 f1 ba
<backward+14>:       0c 00 00 00 cd 80 cc
<forward+1>: e6 ff ff ff 48 65 6c 6c
<forward+9>: 6f 20 57 6f 72 6c 64 0a
```

Megszereztük a végrehajtandó utasítások bájtjait. Mire várunk akkor? Az előző példában látott módszerrel be tudjuk illeszteni őket a futó programba. A forráskód a következőképpen néz ki (most csak a `main` függvényt adjuk meg):

```
int main(int argc, char *argv[])
{   pid_t traced_process;
    struct user_regs_struct regs, newregs;
    long ins;
    int len = 41;
    char insertcode[] =
"\xeb\x15\x5e\xb8\x04\x00"

"\x00\x00\xbb\x02\x00\x00\x00\x89\xf1\xba"

"\x0c\x00\x00\x00\xcd\x80\xcc\xe8\xe6\xff"

"\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f"
        "\x72\x6c\x64\x0a\x00";
    char backup[len];
```

1. lista

```
map start-mapend      protection   offset       device     inode      process file
08048000-0804d000     r-xp         00000000     03:08      66111      /opt/kde2/bin/kdeinit
```

```
    if(argc != 2) {
        printf("HasznÆlat: %s <k vetendı
            ➥pid>\n",
               argv[0], argv[1]);
        exit(1);
    }
    traced_process = atoi(argv[1]);
    ptrace(PTRACE_ATTACH, traced_process,
           ➥NULL, NULL);
    wait(NULL);
    ptrace(PTRACE_GETREGS, traced_process,
           ➥NULL, &regs);
    getdata(traced_process, regs.eip, backup,
            ➥len);

    putdata(traced_process, regs.eip,
            ➥insertcode, len);
    ptrace(PTRACE_SETREGS, traced_process,
           ➥NULL, &regs);
    ptrace(PTRACE_CONT, traced_process,
           ➥NULL, NULL);

    wait(NULL);
    printf("The process stopped, Putting back
            ➥the original instructions\n");
    putdata(traced_process, regs.eip, backup,
            ➥len);
    ptrace(PTRACE_SETREGS, traced_process,
           ➥NULL, &regs);
    printf("Letting it continue with
            ➥original flow\n");
    ptrace(PTRACE_DETACH, traced_process,
           ➥NULL, NULL);
    return 0;
}
```

Kód beszúrása szabad helyekre

Az előző példában a kódot közvetlenül a végrehajtott utasításfolyamba szúrtuk be. Sajnos az ilyesmi könnyen összezavarhatja a hibakeresőket, ezért keressünk inkább egy üres helyet a folyamatban, és ide helyezzük a kódot. Az üres helyet a követett folyamathoz tartozó */proc/pid/maps* fájl vizsgálatával találhatjuk meg. A következő függvény e térkép indulócímét keresi ki:

```
long freespaceaddr(pid_t pid)
{
    FILE *fp;
    char filename[30];
    char line[85];
    long addr;
    char str[20];

    sprintf(filename, "/proc/%d/maps", pid);
    fp = fopen(filename, "r");
    if(fp == NULL)
        exit(1);
    while(fgets(line, 85, fp) != NULL) {
        sscanf(line, "%lx-%*lx %*s %*s %*s",
               ➥&addr, str, str, str, str);
        if(strcmp(str, "00:00") == 0)
            break;
    }
    fclose(fp);
    return addr;
}
```

A */proc/pid/maps* minden sora a folyamat egy-egy lefoglalt tartományának felel meg.
A */proc/pid/maps* bejegyzést az *1. listában* láthatjuk.

A következő program a kódot az üres területre szúrja be. A szerkezete hasonlít az előző beszúróprogramhoz, de azzal az eltéréssel, hogy az új kódunk tárolásához az üres terület címét használjuk majd fel. A `main` függvény forrását az alábbiakban olvashatjuk:

```
int main(int argc, char *argv[])
{   pid_t traced_process;
    struct user_regs_struct oldregs, regs;
    long ins;
    int len = 41;
    char insertcode[] =
"\xeb\x15\x5e\xb8\x04\x00"

"\x00\x00\xbb\x02\x00\x00\x00\x89\xf1\xba"

"\x0c\x00\x00\x00\xcd\x80\xcc\xe8\xe6\xff"

"\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f"
        "\x72\x6c\x64\x0a\x00";
    char backup[len];
    long addr;

    if(argc != 2) {
        printf("HasznÆlat: %s <k vetendı
            ➥pid>\n",
               argv[0], argv[1]);
        exit(1);
    }

    traced_process = atoi(argv[1]);

    ptrace(PTRACE_ATTACH, traced_process,
           ➥NULL, NULL);
    wait(NULL);

    ptrace(PTRACE_GETREGS, traced_process,
```

```
            ➥NULL, &regs);
        addr = freespaceaddr(traced_process);
        getdata(traced_process, addr, backup, len);

        putdata(traced_process, addr, insertcode,
            ➥len);
        memcpy(&oldregs, &regs, sizeof(regs));
        regs.eip = addr;
        ptrace(PTRACE_SETREGS, traced_process,
            ➥NULL, &regs);
        ptrace(PTRACE_CONT, traced_process,
            ➥NULL, NULL);
        wait(NULL);
        printf("A folyamat megÆllt,
            ➥visszahelyezz k az eredeti
            ➥utas tÆsokat.\n");
        putdata(traced_process, addr, backup, len);
        ptrace(PTRACE_SETREGS, traced_process,
            NULL, &oldregs);
        printf("TovÆbbengedj k az eredeti
            ➥folyamot\n");
        ptrace(PTRACE_DETACH, traced_process,
            ➥NULL, NULL);
        return 0;
}
```

A színfalak mögött

De valójában mi történik ezalatt a rendszermagban? Hogyan működik a Ptrace? Ez a rész önmagában megérne egy külön cikket – mégis, nézzük meg röviden, hogyan zajlanak a dolgok!
Amikor a folyamat `PTRACE_TRACEME`-mel hívja meg a Ptrace-t, a rendszermag beállítja a folyamatzászlókat, hogy jelezze, a folyamat követés alatt áll:

```
Source: arch/i386/kernel/ptrace.c

if (request == PTRACE_TRACEME) {

    /* mÆr k vetnek minketı */
    if (current->ptrace & PT_PTRACED)
        goto out;

    /* Æll tsuk be a ptrace bitet
       a folyamatzÆszl kban. */
    current->ptrace |= PT_PTRACED;
    ret = 0;
    goto out;
}
```

Amikor a rendszerhívás-bejegyzés véget ér, a rendszermag megvizsgálja a zászlót, és amennyiben a folyamatot követik, meghívja a követő rendszerhívást. A nyers assembly-részeleteket itt találhatjuk meg: arch/i386/kernel/entry.S.

Beléptünk a `sys_trace()` függvénybe, ezt a *arch/i386/kernel/ptrace.c*-ben találhatjuk. A függvény megállítja a gyerekfolyamatot, és jelet küld a szülőnek, értesítve őt arról, hogy a gyermeket megállította. Az alvó szülő így felébred, és végrehajthatja a Ptrace-varázslatokat. Amint a szülő végzett, és meghívja a `ptrace(PTRACE_CONT, ..)` vagy a `ptrace(PTRACE_SYSCALL, ..)` függvényt, a `wake_up_process()` ütemező függvény meghívásával a gyermeket is felébreszti. Más architektúrák ezt úgy oldják meg, hogy `SIGCHLD` üzenetet küldenek a gyereknek.

Összegzés

A Ptrace néhány ember számára varázslatos tudománynak tűnhet, hiszen képes egy futó program vizsgálatára és követésére. Általában hibakeresők és rendszerhíváskövető programok (például a Ptrace) használják ezt a lehetőséget, ugyanakkor érdekes távlatot nyit meg egy felhasználómódú kiterjesztés létrehozására is. Számos próbálkozás napvilágot látott már, ami az operációs rendszert felhasználószinten próbálja meg bővíteni. A ***Kapcsolódó címek*** között olvashatunk az UFO-ról, azaz a felhasználószintű fájlrendszerbővítésről (User-level extension to Filesystems). A Ptrace-t ezenkívül biztonsági rendszerek megvalósítására is fel szokták használni.
A cikkben (a jelenlegi és az előző részben) található összes kód (eredeti angol változata) tar-állományként elérhető a Linux Journal FTP lapján [ftp.ssc.com/pub/lj/listings/issue104/6210.tgz].

Linux Journal 2002. december, 104. szám

Pradeep Padala (p_padala@yahoo.com)
Jelenleg a Floridai Egyetemen diplomája megszerzésén munkálkodik. Érdeklődési területei között a rácsos kiépítésű és osztott rendszerek szerepelnek. Honlapját a ➲ http://www.cise.ufl.edu/~ppadala címen lehet elérni.

KAPCSOLÓDÓ CÍMEK

Extending the Operating System at User Level: The UFO Global File System
➲ http://www.cs.ucsb.edu/projects/ufo/97-usenix-ufo.ps

A Ptrace-kézikönyv (Secure, User-Level Resource-Constrained Sandboxing)
➲ http://csdocs.cs.nyu.edu/Dienst/Repository/2.0/Body/ncstrl.nyu_cs%2fTR1999-795/pdf

A Ptrace súgóoldala
➲ http://www.die.net/doc/linux/man/man2/ptrace.2.html