Aláírt magmodulok

A rendszermag ma már a beillesztésük előtt képes ellenőrizni a modulok titkosított aláírásait. Ennek részleteibe avatjuk be olvasóinkat.

Másik operációs rendszerekben már évek óta ismert az aláírással ellátott rendszermagmodulok módszere. Bizonyos emberek és cégek hasznosnak tartják, ha kizárólag olyan modulokat (avagy meghajtókat, ahogy néha nevezik) telepítenek a rendszerükre, amelyre operációs rendszerük valamelyik hitelesítője (authority) az áldását adta. A Linux-rendszermag modulbetöltési módszerének változását figyelembe véve az aláírt rendszermagmodulokat ma már könnyedén a rendszermagba illeszthetjük. Ez az írás arról szól, hogy miként valósítottam meg ezt az elképzelést, valamint azt is bemutatja, hogy milyen módon kell a gyakorlatban használni.

Az aláírt rendszermagmodulok lényege, hogy valaki digitális aláírást szúr be a modulba, jelezvén, hogy az adott modulban megbízik. Senkit sem akarok meggyőzni arról, hogy ennek benne kellene lennie a Linuxban vagy feltétlenül szükség van rá, sem arról, hogy növeli a biztonságot. Egyszerűen csak azt mesélem el, hogy miképpen lehet megvalósítani, és bemutatok egy megoldást, ha esetleg valaki használni szeretné.

Az aláírt rendszermagmodulok működéséhez nyílt kulcsú titkosítást használunk. Az RSA nyílt kulcsú titkosítási

```
$ readelf -S visor.ko
There are 23 section headers, starting at offset 0x3954:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000040 0017e0 00  AX  0   0 16
  [ 2] .rel.text         REL             00000000 003cec 000cd0 08      21  1  4
  [ 3] .init.text        PROGBITS        00000000 001820 000210 00  AX  0   0 16
  [ 4] .rel.init.text    REL             00000000 0049bc 0001c8 08      21  3  4
  [ 5] .exit.text        PROGBITS        00000000 001a30 000030 00  AX  0   0 16
  [ 6] .rel.exit.text    REL             00000000 004b84 000030 08      21  5  4
  [ 7] .rodata           PROGBITS        00000000 001a60 000020 00   A  0   0 16
  [ 8] .rel.rodata       REL             00000000 004bb4 000028 08      21  7  4
  [ 9] .rodata.str1.1    PROGBITS        00000000 001a80 000449 01 AMS  0   0  1
  [10] .rodata.str1.32   PROGBITS        00000000 001ee0 0009c0 01 AMS  0   0 32
  [11] .modinfo          PROGBITS        00000000 0028a0 0006c0 00   A  0   0 32
  [12] .data             PROGBITS        00000000 002f60 000600 00  WA  0   0 32
  [13] .rel.data         REL             00000000 004bdc 0001e0 08      21  c  4
  [14] .gnu.linkonce.thi PROGBITS        00000000 003560 000120 00  WA  0   0 32
  [15] .rel.gnu.linkonce REL             00000000 004dbc 000010 08      21  e  4
  [16] __obsparm         PROGBITS        00000000 003680 000180 00  WA  0   0 32
  [17] .bss              NOBITS          00000000 003800 00000c 00  WA  0   0  4
  [18] .comment          PROGBITS        00000000 003800 00006e 00      0   0  1
  [19] .note             NOTE            00000000 00386e 000028 00      0   0  1
  [20] .shstrtab         STRTAB          00000000 003896 0000bd 00      0   0  1
  [21] .symtab           SYMTAB          00000000 004dcc 000760 10     22  58  4
  [22] .strtab           STRTAB          00000000 00552c 000580 00      0   0  1
```

algoritmus alapelveiről és működéséről a Linux Journal ➲ http://www.linuxjournal.com/article/6826 hálózati cikkében olvashatunk. Ez a cikk feltételezi, hogy az olvasó jártas a nyílt kulcsú titkosítás alapjaiban és képes egy új Linux-rendszermagot elkészíteni és betölteni a gépébe. Az új rendszermag elkészítéséről és betöltéséről bővebben a Linux Kernel HOWTO-ban (➲ http://www.tldp.org) olvashatunk.
A 2.5-ös fejlesztői rendszermagsorozatban ***Rusty Russell*** újraírta a Linux-magmodulok működését. A korábbi rendszermagokban a modulbetöltési logika nagyobbik része a felhasználói térben zajlott – Rusty változtatásai révén a teljes logika a rendszermagtérbe került; ilyen módon csökkent a kiépítésfüggetlen logika, valamint a felhasználói felület is lényegesen egyszerűsödött. Ennek egyik kellemes mellékhatásaként mostantól a rendszermag a teljes modulhoz hozzáférhet nyers formában. A magmodul egy egyszerű ELF formátumú program. Az ELF az executable and linking format (végrehajtható és csatoló formátum) szavak rövidítése és a végrehajtható programokat jelöljük vele. Az ELF szabványt a ➲ http://www.muppetlabs.com/~breadbox/software/ELF.txt címen találjuk meg szöveges formátumban.
Az ELF-állományok különböző szakaszokra bonthatók. Ezeket a szakaszokat tekinthetjük meg a `readelf` program futtatásakor, lásd az *1. listát*.
Minthogy az ELF-fájlok szakaszokból épülnek fel, nem túl nehéz egy újabb szakaszt felvenni a modul állományában, amit így modultöltés közben a rendszermag a memóriába olvas. Ha a modulba egy RSA-aláírásos szakaszt helyezünk,

```
for (i = 1; i < hdr->e_shnum; i++) {
    name = secstrings+sechdrs[i].sh_name;

    /* Csak azokkal a szakaszokkal foglalkozunk,
       ➥amelyeknek "text" vagy "data"
       olvasható a nevükben*/
    if ((strstr(name, "text") == NULL) &&
        (strstr(name, "data") == NULL))
        continue;
    /* a ".rel.*" szakaszok nem érdekelnek
       ➥bennünket. */
    if (strstr(name, ".rel.") != NULL)
        continue;

    temp = (void *)sechdrs[i].sh_addr;
    size = sechdrs[i].sh_size;
    do {
        memset(&sg, 0x00, sizeof(*sg));
        sg.page = virt_to_page(temp);
        sg.offset = offset_in_page(temp);
        sg.length = min(size,
                        (PAGE_SIZE - sg.offset));
        size -= sg.length;
        temp += sg.length;
        crypto_digest_update(sha1_tfm, &sg, 1);
    } while (size > 0);
}
```

a rendszermag visszakódolhatja az aláírást és összehasonlíthatja az éppen betöltött fájl aláírásával. Ha a kettő egyezik, az aláírás érvényes és a modul sikeresen beilleszthető a rendszermag memóriájába. Ha az aláírások nem egyeznek, akkor valaki vagy változtatott a modulon, vagy nem a megfelelő kulccsal írták azt alá. Ezután a modult elutasíthatjuk – erről szól ez a folt.

Hogyan működik a rendszermagkód?

Amikor a rendszermagot valamilyen modul betöltésére utasítjuk, a ***kernel/module.c*** állományban megadott kód fut le. Itt szinte minden feladatot a `load_module` függvény hajt végre: a modult megfelelő szakaszokra tördeli, ellenőrzi a memóriahelyzeteket és a szimbólumokat, illetve elvégzi azokat a további feladatokat, amelyeket egyébként az összeszerkesztő program (linker) szokott. A folt ezt a programot módosítja a következő sorok beszúrásával:

```
if (module_check_sig(hdr, sechdrs, secstrings)) {
    err = -EPERM;
     goto free_hdr;
}
```

A `module_check_sig` nevű új függvény tartalmazza a modul-aláírási logikával kapcsolatos összes részletet. Ha hibát ad vissza, akkor ***Improper Permission*** hibaüzenetet adunk vissza a felhasználónak és megszakítjuk a modulbetöltést. Ha a függvény 0 értékkel tér vissza, azaz nem történt hiba, s a modulbetöltés folyamata sikeresen folytatódhat.
A `module_check_sig` függvény a ***kernel/module-sig.c*** állományba kerül. Ez a függvény először is ellenőrzi, hogy van-e aláírás a modulban – ezt a következő kódsorokkal végezzük el:

```
sig_index = 0;
for (i = 1; i < hdr->e_shnum; i++)
    if (strcmp(secstrings+sechdrs[i].sh_name,
               "module_sig") == 0) {
        sig_index = i;
        break;
}
if (sig_index <= 0)
    return -EPERM;
```

A kódrészlet a magmodul valamennyi ELF-szakaszán végiglépked, és kikeresi közülük a `module_sig` nevűt.
Ha az aláírást nem találja meg, hibát ad vissza és megakadályozza a modul betöltését. Ha megtalálja, folytatódhat a függvény végrehajtása.
Miután a rendszermag megtalálta a modul aláírását, meg kell állapítania, hogy milyen nyalábolóérték (hash value) tartozik a betöltendő modulhoz. Ehhez létrehozza a rendszermag által használt végrehajtható vagy adat-ELF szakaszok SHA1 nyaláboló értékét. A rendszermagban már megtalálható az SHA1 nyalábolók létrehozásához használt kód (egyéb nyalábolókkal, az MD5-tel és MD4-gyel együtt), így e lépés belső logikájának a nagy része már eleve készen van.
A függvény először az SHA1 algoritmus meghívásával foglalja le a titkosításátalakítási szerkezetet, majd ezt a következő kódsorokkal alaphelyzetbe állítja:

```
sha1_tfm = crypto_alloc_tfm("sha1", 0);
if (sha1_tfm == NULL)
    return -ENOMEM;
crypto_digest_init(sha1_tfm);
```

Az ELF-fájl számunkra érdekes részeinek SHA1 nyalábolóértékének előállítására az `sha1_tfm` változót használjuk, mint azt a *2. listán* látható kód mutatja.
A kódban csak azokkal az ELF-szakaszokkal foglalkozunk, amelyeknek a neve tartalmazza a „text” vagy a „data” szót, de amelyek nem foglalják magukban a „.rel.” szövegrészletet. Miután az összes szakaszt megtaláltuk és betöltöttük az SHA1 algoritmusba, a következő sorokkal az SHA1 nyalábolót áthelyezzük az `sha1_result` változóba:

```
crypto_digest_final(sha1_tfm, sha1_result);
crypto_free_tfm(sha1_tfm);
```

Az SHA1 nyaláboló kiszámításának és az aláírt nyaláboló helyének a megállapítása után már csak annyi a dolgunk, hogy visszafejtsük az aláírt nyalábolót és összehasonlítsuk a számított értékkel. Ezt a lépést a függvény utolsó sorával végezzük el:

```
return rsa_check_sig(sig, &sha1_result[0]);
```

Az `rsa_check_sig` függvény a ***security/rsa/rsa.c*** fájlban található és a rendszermagban futtathatóvá alakított eredeti GnuPG-kódot használja – ennek segítségével vissza lehet fejteni az aláírásokat és össze lehet hasonlítani az értékeket. Működésének pontos ismertetése azonban már meghaladná cikkünk kereteit.

Hogyan működik a felhasználói tér kódja?

Miután láttuk, hogy milyen módon dönti el a rendszermag, hogy a modul helyes aláírással rendelkezik-e, már csak az a kérdés, hogy miképpen tesszük azt az aláírást a modulba? Két, a felhasználói térben futó program – az `extract_pkey` és a `mod` –, valamint egy apró parancsfájl, a `sign` található a rendszermagfoltban (a ***security/rsa/userspace/*** könyvtárban). A két programot a könyvtárban található `Makefile` futtatásával fordíthatjuk le. Az `extract_pkey` felhasználásával a nyílt kulcsot helyezhetjük el a rendszermagban, a `mod` programot pedig a ***sign*** parancsfájl használja a magmodulok aláírására.
A modulok aláírásához először is létre kell hoznunk egy RSA-aláíró kulcsot – ezt a `gnupg` program segítségével tehetjük meg. Az RSA aláíró kulcs létrehozásához a `gpg`-nek adjuk meg a `--gen-key` kapcsolót, lásd a *3. listát*.
Sorra válaszolgassuk meg a feltett kérdéseket, és végül elkészül az RSA-kulcsunk. A kulcs használatához ugyanakkor el kell készítenünk annak titkosított változatát is. Ehhez ismét futtassuk le a `gpg`-t, és szerkesszük át az imént készített kulcsot, lásd a *4. lista* (a kulcsomat a listában található szövegben `testkey`-nek neveztem el).
Mivel mi új kulcsot szeretnénk felvenni, az `addkey` parancsot gépeljük be:

```
Command> addkey
Please select what kind of key you want:
```

```
$ gpg --gen-key
gpg (GnuPG) 1.2.1; Copyright (C) 2002 Free
Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to
redistribute it under certain conditions.
See the file COPYING for details.

Please select what kind of key you want:
   (1) DSA and ElGamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection?

RSA kulcsot akarunk készíteni, így először az 5-
öst választjuk, majd 1024-et adjuk meg.

Your selection? 5
What keysize do you want? (1024)
Requested keysize is 1024 bits
```

```
$ gpg --edit-key testkey
gpg (GnuPG) 1.2.1; Copyright (C)
2002 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to
redistribute it under certain conditions.
See the file COPYING for details.

Secret key is available.

gpg: checking the trustdb
gpg: checking at depth 0
➥signed=0 ot(-/q/n/m/f/u)=0/0/0/0/0/1
pub  1024R/77540AE9  created:
➥2003-10-09 expires: never trust: u/u
(1). testkey

Command>
```

```
   (2) DSA (sign only)
   (3) ElGamal (encrypt only)
   (5) RSA (sign only)
   (6) RSA (encrypt only)
Your selection?
```

RSA-kulcsra van szükségünk, ezért válasszuk a hatost, és válaszoljunk a fennmaradó kérdésekre. Miután a kulcs elkészült, gépeljük be a `quit` parancsot:

```
Command> quit
Save changes? yes
```

```
#!/bin/bash
module=$1
key=$2

# emeljük ki a bennünket érdeklő részeket
./mod $module $module.out

# sha1 alkalmazása a szakaszokra
sha1sum $module.out | awk "{print \$1}" >
➥$module.sha1

# szakaszok titkosítása
gpg --no-greeting -e -o - -r $key $module.sha1 >
➥$module.crypt

# a titkosított adat hozzáfűzése a modulhoz
objcopy --add-section module_sig=$module.crypt
➥$module

# ideiglenes fájlok eltávolítása
rm $module.out $module.sha1 $module.crypt
```

Most már van egy kulcsunk, amivel aláírhatjuk a modulokat. A modulok aláírásához a `sign` nevű egyszerű parancsfájlt használjuk, lásd az *5. listát*.
A parancsfájl először is a `mod` programot lefuttatja a magmodulon. Ez a program az ELF-állományból kiemeli a bennünket érdeklő részeket, majd egy ideiglenes fájlba menti őket. A `mod` programot a későbbiek folyamán ismertetjük részletesebben.
Miután létrehoztuk a bennünket érdeklő részeket tartalmazó ELF-fájlt, az `sha1sum` programmal SHA1 nyalábolót készítünk az állományról. Ezt az SHA1 nyalábolót fogjuk azután a GPG segítségével titkosítani, átadjuk a kulcsot, majd a titkosított állományt ideiglenes fájlba mentjük.
A titkosított állományt `module-sig` néven új ELF-szakaszként hozzáadjuk az eredeti modulhoz, amit az `objcopy` programmal végzünk el. Ennyi az egész. A linuxos gépeken már eleve létező szokásos programok segítségével nem volt nehéz az SHA1 nyalábolót előállítani, titkosítani és az ELF-fájlba illeszteni.
A `mod` program is igen egyszerű. Kihasználja, hogy az `objdump` nevű, `binutils` programon alapuló ***libbfd*** könyvtár tudja, hogyan kell kezelni az ELF-állományokat és többféleképpen is módosítani tudja őket. Mivel a ***libbfd*** könyvtár valamennyi nehéz ELF-logikát megvalósítja, a `mod` programnak nincs más dolga, mint a következő kód segítségével végiglépdelni az ELF-állomány kívánt szakaszain, lásd a *6. listát*.
Miután a magmodulokat alá tudjuk írni, a rendszermag pedig képes ezeket az aláírásokat ellenőrizni, már csak annyi maradt hátra, hogy a sikeres visszafejtés érdekében nyílt kulcsunkat a rendszermagba töltsük. A Linux-rendszermag levelezőlistán elég sok szó esik mostanában arról, hogyan is kell helyesen kezelni a kulcsokat. A vita során több jó ajánlás is született, amelyeket a 2.7-es rendszermagsorozat fog felhasználni. Egyelőre azonban nem kell aggódnunk a kulcsok helyes és rugalmas felhasználása miatt, úgyhogy egyszerűen közvetlenül beforagain:

```
for (section = abfd->sections;
     section != NULL;
     section = section->next) {
    if (section->flags & SEC_HAS_CONTENTS) {
        if (bfd_section_size(abfd, section)
        ➥== 0)
            continue;

        /* Csak azokkal a szakaszokkal
           foglalkozunk, melyeknek
           "text" vagy "data" olvasható
            a nevükben*/
        name = section->name;
        if ((strstr(name, "text") == NULL) &&
            (strstr(name, "data") == NULL))
            continue;

        size = bfd_section_size(abfd, section));
        data = (bfd_byte *)malloc(size);

        bfd_get_section_contents(abfd, section,
                                 (PTR)data, 0, size);

        stop_offset = size / opb;

        for (addr_offset = 0;
             addr_offset < stop_offset;
             ++addr_offset) {
            fprintf(out, "%c",
            ➥data[addr_offset]);
        }
        free(data);
    }
}
```

díthatjuk.
Először is szükség lesz nyilvános kulcsunknak egy másolatára. Utasítsuk a GPG-t, hogy a kulcsot egy ***public_key*** nevű állományba csomagolja ki:

```
$ gpg --export -o public_key
```

A GPG nyílt kulcsok kezelésének megkönnyítésére néhány Ericsson-fejlesztő egy ***extract_pkey*** nevű egyszerű programot készített, ami segít részeire bontani a kulcsot. Egy kicsit módosítottam ezen a programon olyan módon, hogy C-kódot készítsen a nyílt kulcshoz.
Futtassuk az ***extract_pkey*** programot, megadva neki a korábban elkészített ***public_key*** állományt. A kimenetet mentsük az ***rsa_key.c*** nevű fájlba:

```
$ extract_pkey public_key > rsa_key.c
```

Ezután ezt az állományt helyezzük a ***security/rsa/*** könyvtár gyökerébe, lecserélve a saját nyilvános kulcsunkat:

```
$ mv rsa_key.c ~/linux/linux-2.6/security/rsa/
```

Létrehoztuk a szükséges RSA kulcspárt és a saját nyilvános kulcsunkat a rendszermagkönyvtárba helyeztük. Készítsük el a foltozott rendszermagot – ne felejtsük el bejelölni a ***Module signature checking*** lehetőséget –, majd telepítsük. Amint ezt a rendszermagot betöltjük, már csak a saját kulcsunkkal aláírt modulokat tudjuk betölteni, úgyhogy óvatosan járjunk el: kizárólag fejlesztői gépen használjuk a módszert.

Mi maradt még hátra?

Mint azt a cikkben bemutattuk, számos lépés szükséges a kulcs előállításához, a rendszermag aláírásához és a nyílt kulcs rendszermagba történő illesztéséhez. Ez ebben a formában egy meglehetősen csiszolatlan fejlesztői projekt. Amennyiben a rendszermagfejlesztők és általában a Linux-közösség számára elfogadhatóbbá szeretnénk tenni, a lépéseket önműködővé kell tenni. Egyszerűbb megoldásra van szükség az összes modul aláírásához és a nyílt kulcs kezeléséhez is. A képesség leegyszerűsítésének nyilvánvaló igénye mellett néhány egyéb elérendő célt is megfogalmazhatunk a projektben:

- Az RSA-kódot általános titkosító keretrendszerbe kell helyezni, hogy azt más rendszermagmodulok is használhassák.
- Lehetővé kell tenni, hogy a rendszermagban egyszerre több nyílt kulcs is létezhessen, illetve egyetlen gépen többféle forrásból származó modulok is futtathatóak legyenek.
- Érdemes egyszerűsíteni az aláírási logikát, hogy a GPG saját aláírási képességét, esetleg a `bsign` program megoldását használhassuk az egyedi `mod` program helyett.

Köszönetnyilvánítás

Szeretném megköszönni az Ericsson fejlesztőinek, hogy elkészítették a `digsig` nevű programot és rendszermagfoltot, ezzel téve lehetővé, hogy GPG-változatukat felhasználjam a rendszermaghoz. Korábban is készítettem ilyet, de a megvalósítás borzasztó lett; szerencsére kiadták a megoldásukat, ami nagy segítséget jelentett. A `digsig` rendszermagfolt segítségével a felhasználók programokat írhatnak alá, a rendszermag pedig semmilyen aláíratlan programot nem fog futtatni. A projektről további adatok a ➲ http://sourceforge.net/projects/disec címen érhetők el. Szeretnék továbbá köszönetet mondani alkalmazómnak, az IBM-nek, amiért lehetővé tette, hogy a projekten dolgozzak, és ***Don Marti***-nak, aki folyton piszkált, hogy készítsem el és fejezzem be ezt a cikket.

Linux Journal 2004. február, 117. szám

Greg Kroah-Hartman (greg@kroah.com)
Jelenleg a Linux-rendszerek USB és gyors csatlakoztatású (PCI Hot Plug) egységei rendszermagba épített meghajtóprogramjainak a fejlesztője. Az IBM-nél dolgozik és rendszermaggal kapcsolatos kérdésekkel foglalkozik.

Frissítés 2.6-os sorozatú rendszermagra

Mint azt a Programvadászatban is írtam, több dolgot is frissíteni kell, ha a rendszermagnak ezt az új vonalát akarjuk használni.
Az első ilyen nálam (Debian/GNU Linux Woody) a `modconf` program kényszerű frissítése volt. Bizonyára vannak olyanok, akik emlékeznek még a Potato hiányosságaira a 2.4-es sorozat kezelésében – úgy látszik, ez megismétlődik.
Miután a modullistát üresen találtam, a ***/etc/apt/sources.list*** fájlban kijavítottam a Woody (vagy ha úgy tetszik: a stable) bejegyzéseket, a SID-re kiadtam az `apt-get update`, `apt-get install modconf` parancsot. Máris csorgott a hálózatról a program új változata, a telepítés után pedig egycsapásra megszűnt az előbb említett gond.
A másik érdekesség számomra az eszközök elnevezése volt, ezek közül is az PS/2-es egeremé. Az X nem tudott elindulni, mert nem találta a „Core Pointer”-t, vagyis az egeremet, ugyanis eredendően a ***/dev/psaux*** volt a beállításfájlban megadva. Ezt ***/dev/input/mouse0***-ra kellett kijavítanom.
A frissítés során talán a legfurcsább helyzet akkor áll elő, ha előtte nem `initrd`-s rendszermagunk volt. (Az `initrd` egy a rendszerindításhoz használt memórialemez- (ramdisk) fájl.) Ha viszont a Debian alatt a gyökérkönyvtárban ***initrd.img*** fájlt találunk, akkor az azt jelenti, hogy valószínűleg az előző rendszermagunk is `initrd`-s volt.

A rendszerbetöltőnkhöz hozzá kell adnunk az alábbi sorokat, különös tekintettel az `initrd`-bejegyzésre:

GRUB:

```
title Debian2.6
          root (hd0,2)
          kernel /vmlinuz root=/dev/hda3
          initrd /initrd.img
```

LILO:

```
image=/vmlinuz
label=Linux
read-only
initrd=/initrd.img
```

Természetesen mindenki a saját rendszeréhez igazítva módosítsa a fenti beállításokat .
A felhasználók biztosan találkozni fognak ilyen, vagy ehhez hasonló dolgokkal, bármelyik Linux-változatot futtassák is. Ha előfordul olyan érdekesség, ami a többi felhasználó számára is hasznos lehet, kérek midenkit, hogy írja meg nekem a csontos.gyula@linuxvilag.hu címre.

Csontos Gyula