Signed Kernel Modules

Now you can make the kernel check modules for a cryptographic signature before inserting them.

by Greg Kroah-Hartman

Signed kernel modules have been a feature of other operating systems for a number of years. Some people and companies like the idea of installing only modules (or drivers, as they are sometimes called) that are known to be blessed by some authority in their operating systems. Given the changes in how Linux loads kernel modules, signed kernel modules easily can be added to the Linux kernel. This article discusses how I have implemented this feature and details how to use it.

In a signed kernel module, someone has inserted a digital signature into the module stating they trust this specific module. I am not going to try to persuade anyone that Linux should have this ability, that it should be required or even that it provides increased security. I describe only how to do it and provide the method for its implementation, if anyone wants to use it.

Public key cryptography is used to make signed kernel modules work. For an overview of the RSA public key cryptographic algorithm—what it is and how it works—see the *Linux Journal* Web article at www.linuxjournal.com/article/6826. This article assumes readers are familiar with the basics of public-key cryptography and that they are able to patch, build and load a new Linux kernel onto their machines. For instructions on how to build and load a new kernel, see the very helpful Linux Kernel HOWTO located at www.tldp.org.

In the 2.5 kernel development series, Rusty Russell rewrote the way Linux kernel modules work. In previous kernels, the majority of the module loading logic was stored in user space. With Rusty's changes, all of that logic moved into the kernel, reducing the amount of architecture-independent logic and simplifying the user interface greatly. One nice side benefit of this is the kernel now has access to the entire module file in raw form. The kernel module simply is a file in ELF format. ELF stands for executable and linking format and is the format used for executable programs. The ELF specification can be found in text form at www.muppetlabs.com/~breadbox/software/ELF.txt.

ELF files are comprised of different sections. These sections can be seen by running the readelf program. For example:

```
$ 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
```

Because ELF files are made up of sections, it is easy to add a new section to the module file and have the kernel read it into memory when it tries to load the module. If we put an RSA-signed section into the module, the kernel can decrypt the signature and compare it to the signature of the file it just loaded. If it matches, the signature is valid and the module is inserted successfully into the kernel's memory. If the signature does not match, either something has been tampered with in the module or the module was not signed with a proper key. The module then can be rejected—that is what my patch does.

How the Kernel Code Works

When the kernel is told to load a module, the code in the file kernel/module.c is run. In that file, the function load_module does all of the work of breaking the module into the proper sections, checking memory locations, checking symbols and all the other tasks a linker generally does. The patch modifies this function and adds the following lines of code:

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

This new function, module_check_sig does all of the module signature-checking logic. If it returns an error, the error `Improper Permission` is returned to the user and module loading is aborted. If the function returns a 0, meaning no error occurred, the module load procedure continues on successfully.

The module_check_sig function is located in the file kernel/module-sig.c. The first thing the function does is check to see if a signature is located within the module. This is done with the following lines of code:

```
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;
```

This bit of code loops through all of the different ELF sections in the kernel module and looks for one called module_sig. If it does not find the signature, it returns an error and prevents this module from being loaded. If it does find the signature, the function continues.

Once the kernel has found the module signature, it needs to determine what the hash value is of the module it is being asked to load. To do this, it generates the SHA1 hash of the ELF section that contains executable code or data used by the kernel. The kernel already contains code to generate SHA1 hashes (along with other kinds of hashes, including MD5 and MD4), so most of the logic for this step is present already.

The function first allocates a crypto transformation structure by requesting the SHA1 algorithm. It then

initializes this structure with the following lines of code:

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

The sha1_tfm variable is used to create the SHA1 hash of the specific portions of the ELF file that we want, as shown in the following code:

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

    /* We only care about sections with "text" or
       "data" in their names */
    if ((strstr(name, "text") == NULL) &&
        (strstr(name, "data") == NULL))
        continue;
    /* avoid the ".rel.*" sections too. */
    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);
}
```

In this code, we care only about the ELF sections with the word text or data in their names but not ones that contain the characters .rel. After all of the sections have been found and fed to the SHA1 algorithm, the SHA1 hash is placed into the variable sha1_result with the following lines:

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

Now that the SHA1 hash is computed and the place with the signed hash has been found, all that is left to do is unencrypt the signed hash and compare it to the calculated one. This step is done in the last line of this function:

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

The rsa_check_sig function is located in the security/rsa/rsa.c file and uses the GnuPG code itself, which was ported to run in the kernel to unencrypt the signature and compare the values. The description of how this works is beyond the scope of this article.

How the User-Space Code Works

Now that we have seen how the kernel determines whether a module is signed properly, how do we get a signature into a module in the first place? Two user-space programs, extract_pkey and mod, and one small script, sign (in the security/rsa/userspace/ directory), can be found in the kernel patch. The two programs can be built by running the Makefile in this directory. The extract_pkey program is used to place a public key into the kernel, and the mod program is used by the sign script to sign a kernel module.

In order to sign a module, an RSA-signing key must be generated, which can be done by using the gnupg program. To generate an RSA-signing key, pass the --gen-key option to gpg:

```
$ 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?
```

We want to create an RSA key, so we select option 5 and then choose the default key size of 1024:

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

Continue answering the rest of the questions, and eventually your RSA key is generated. But in order to use this key, we must create an encrypting version of it. To do that, run gpg again and edit the key you just created (in the text below, I have named my key testkey):

```
$ 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>
```

We want to add a new key, so type `addkey` at the prompt:

```
Command> addkey
Please select what kind of key you want:
   (2) DSA (sign only)
   (3) ElGamal (encrypt only)
   (5) RSA (sign only)
   (6) RSA (encrypt only)
Your selection?
```

Again, we want an RSA key, so choose option 6 and answer the rest of the questions. After the key is

generated, type `quit` at the prompt:

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

Now that we have a key, we can use it to sign a kernel module.

To sign a module, use the sign script, which is a simple shell script:

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

# strip out only the sections that we care about
./mod $module $module.out

# sha1 the sections
sha1sum $module.out | awk "{print \$1}" > \
$module.sha1

# encrypt the sections
gpg --no-greeting -e -o - -r $key $module.sha1 > \
$module.crypt

# add the encrypted data to the module
objcopy --add-section module_sig=$module.crypt \
$module

# remove the temporary files
rm $module.out $module.sha1 $module.crypt
```

The first thing the script does is run the program mod on the kernel module. This program strips out only the sections that we care about in the ELF file and outputs them to a temporary file. The mod program is described in more detail later.

After we have an ELF file that contains only the sections we want, we generate a SHA1 hash of the file using the sha1sum program. This SHA1 hash then is encrypted using GPG, the key is passed to it and this encrypted file is written out to a temporary file. The encrypted file is added to the original module as a new ELF section with the name module-sig. This is done with the program objcopy. And that is it. Using common programs already present on a Linux machine, it is easy to create a SHA1 hash, encrypt it and add it to an ELF file.

The mod program also is quite simple. It takes advantage of the fact that the libbfd library knows how to handle ELF files and manipulates them in different ways; it is based on the binutils program objdump. Because the libbfd library handles all of the heavy ELF logic, the mod program simply can iterate through all the sections of the ELF file it wants to with the following code:

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

        /* We only care about sections with "text"
           or "data" in their names */
        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);
    }
}
```

Now that we can sign a kernel module and the kernel knows how to detect this signature, the only remaining piece is to put our public key into the kernel so it can decrypt the signature successfully. A lot of discussion on the linux-kernel mailing list recently has centered on how to handle keys within the kernel properly. That discussion has produced some good proposals for how this aspect will be handled in the 2.7 kernel series. But for now, we do not worry about properly handling keys in flexible ways, so we compile it in directly.

First we need to get a copy of our public key. To do this, tell GPG to extract the key to a file called public_key:

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

To help manipulate GPG public keys, some developers at Ericsson created a simple program called extract_pkey to help dissect the keys into their different pieces. I have modified that program to generate C code for the public key.

Run the extract_pkey program and point it at the public_key file you generated previously. Have it send the output to a file called rsa_key.c:

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

After this step is finished, move that rsa_key.c on top of the file in the security/rsa/ directory, replacing my public key with yours:

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

Now you have generated a public and private RSA key pair and placed your public key into the kernel directory. Build the patched kernel, making sure to select the Module signature checking option, and then install it. If you boot in to this kernel, you will be allowed to load only the modules you have signed with your key, so be careful and test this only on a development machine.

What Is Left to Do?

As shown in this article, a number of different steps are required to generate a key, sign a kernel module and place the public key into the kernel image. This still is a rough development project. In order to make it more acceptable to the kernel developers and to the Linux community in general, these steps need to be automated,

making it easier to sign all kernel modules and handle the public key.

Besides the obvious need to simplify the use of this feature, some other future goals of this project include:

- Move the RSA code into the generic crypto framework, allowing other kernel features to use it.
- Allow more than one public key to be present in the kernel, letting multiple sources of signed kernel modules run in a single machine.
- Simplify the signing logic to allow GPG's native signing functionality or possibly the functionality provided in the bsign program to be used, instead of the custom mod program.

Acknowledgements

I would like to thank the developers at Ericsson, who have created a kernel patch and program called digsig, for allowing me to use their port of GPG to the kernel. I previously had done this, but the implementation was horrible; thankfully, they released their port and were very helpful. The digsig kernel patch allows users to sign programs and prevents the kernel from running any program not signed. More information about this project can be found at sourceforge.net/projects/disec.

I also would like to thank my employer, IBM, for allowing me to work on this project, and Don Marti, for prodding me to finish it and write this article.

Greg Kroah-Hartman currently is the Linux kernel maintainer for a variety of different driver subsystems. He works for IBM, doing Linux kernel-related things, and can be reached at greg@kroah.com.
