Using the Kernel Security Module Interface

Greg shows how to create a simple kernel module that uses the LSM framework.

by Greg Kroah-Hartman

At the 2001 Linux Kernel Summit, NSA developers presented their work on Security-Enhanced Linux (SELinux) and emphasized the need for enhanced security support in the main Linux kernel. In the ensuing discussion, a consensus was reached that a general access-control framework for the Linux kernel was needed. This approach would allow different security models to work without modifying the main kernel code.

Out of this discussion grew the Linux Security Module Project (LSM). A number of developers worked together to create a framework of kernel hooks that would allow many security models to work as loadable kernel modules. A detailed description of the design of this general-purpose system was published in a paper at the 2002 USENIX Security Conference (lsm.immunix.org/docs/lsm-usenix-2002/html/) and a technical description of how the LSM interface works was presented in a paper at the 2002 Ottawa Linux Symposium (www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz).

During the 2002 Linux Kernel Summit, the technical description of the project was presented, and the first portion of the LSM framework appeared in the 2.5.29 kernel release. Further kernel releases contained more portions of the LSM framework, and hopefully the entire patch will be included by the time of the 2.6.0 release.

This article does not attempt to describe how the LSM framework works or the design decisions that were made in creating it; the previously mentioned references do an excellent job of that. Instead, this article shows how easy it is to create a simple kernel module that uses the LSM framework.

Root Plug

Our example uses the 2.5.31 kernel release, which contains enough of the LSM interface for us to create a useful module. In our module, we want to prevent any programs with the group ID of 0 (root) from running if a specific USB device is not plugged in to the machine at that moment. This provides us with a simple way of preventing root exploits from running on our machine, or for new users to log in when we are not present.

This example creates a kernel module called root_plug, which is available as a patch against a clean 2.5.31 kernel tree from the *Linux Journal* FTP site [ftp.ssc.com/pub/lj/listings/issue103/6279.tgz].

For a description of UNIX systems that handle the user and group ID values and how they interact with the setuid class of system calls, see the excellent paper by Hao Chen, David Wagner and Drew Dean entitled ``Setuid Demystified", which was presented at the 2002 USENIX Security conference (www.cs.berkeley.edu/~daw/papers/setuid-usenix02.ps).

The LSM interface is four simple functions:

```
int register_security
    (struct security_operations *ops);
int unregister_security
    (struct security_operations *ops);

int mod_reg_security (const char *name,
                  struct security_operations *ops);
```

```
int mod_unreg_security  (const char *name,
                  struct security_operations *ops);
```

A security module registers a set of security_operations function callbacks with the kernel by calling the function register_security(). If that fails, it means that some other security module probably has been loaded already, so the mod_reg_security() function is called in an attempt to register with this security module. This can be seen in the following code:

```
/* register ourselves with the security framework */
if (register_security (&rootplug_security_ops)) {
   printk (KERN_INFO
       "Failure registering Root Plug module "
       "with the kernel\n");
   /* try registering with primary module */
   if (mod_reg_security (MY_NAME,
                         &rootplug_security_ops)) {
       printk (KERN_INFO "Failure registering "
               "Root Plug module with primary "
               "security module.\n");
       return -EINVAL;
   }
   secondary = 1;
}
```

When the module wants to unload itself, the reverse process must happen. If we used mod_reg_security() to register ourselves, the mod_unreg_security() function should be called, otherwise the unregister_security() function is the proper thing to call. The following code shows this logic:

```
/* remove ourselves from the security framework */
if (secondary) {
   if (mod_unreg_security (MY_NAME,
                           &rootplug_security_ops))
      printk (KERN_INFO
              "Failure unregistering Root Plug "
              " module with primary module.\n");
} else {
   if (unregister_security (
       &rootplug_security_ops)) {
      printk (KERN_INFO "Failure unregistering "
              "Root Plug module with the kernel\n");
   }
}
```

The rootplug_security_ops is a large structure of function pointers that are called when various events happen in the kernel. This includes such things as whenever an inode is accessed, a module is loaded or a task is created. As of the 2.5.31 kernel, there were 88 different function pointers needed. The majority of these functions not needed by most security models, but they must be implemented, or the kernel will not work properly. If a security module does not need to do anything for a specific hook, a ``good'' value needs to be returned to the kernel. An example of this can be seen in the following function:

```
static int rootplug_file_permission
    (struct file *file, int mask)
{
    return 0;
}
```

This function is called whenever the kernel wants to determine if a specific file can be accessed at this moment in time. A security module can look at the file, check whether the current user has proper authority and possibly refuse to grant it.

What Hook to Use

For our example module, we want to be able to stop a new program from being run if our USB device is not present. This can be done by using the bprm_check_security hook. This function is called when the execve system call is made, right before the kernel tries to start up the task. If an error value is returned from this function, the task will not start. Here is our hook function:

```
static int rootplug_bprm_check_security
    (struct linux_binprm *bprm)
{
    if (bprm->e_gid == 0)
        if (find_usb_device() != 0)
            return -EPERM;
    return 0;
}
```

This function checks the value of the effective group ID at which the program is to be run. If it is zero, the function find_usb_device() is called. If the USB device is not found in the system, -EPERM is returned, which prevents the task from starting.

Finding a USB Device

The find_usb_device() function simply goes through all of the USB devices in the system and sees if the device specified by the user is present. The USB devices are kept in a tree, starting at the root hub device. The different root hubs are kept in a list of buses. These buses are checked in order in the find_usb_device() function:

```
static int find_usb_device (void)
{
    struct list_head *buslist;
    struct usb_bus *bus;
    int retval = -ENODEV;

    down (&usb_bus_list_lock);
    for (buslist = usb_bus_list.next;
         buslist != &usb_bus_list;
         buslist = buslist->next) {
        bus = container_of (buslist,
                            struct usb_bus,
                            bus_list);
        retval = match_device(bus->root_hub);
        if (retval == 0)
            goto exit;
    }
exit:
    up (&usb_bus_list_lock);
    return retval;
}
```

The match_device() function looks at the device passed to it. If it matches the expected device, then it returns success. Otherwise, it looks at the children of this device, calling itself recursively:

```
static int match_device (struct usb_device *dev)
{
   int retval = -ENODEV;
   int child;

   /* see if this device matches */
   if ((dev->descriptor.idVendor == vendor_id) &&
       (dev->descriptor.idProduct == product_id)) {
       /* we found the device! */
       retval = 0;
       goto exit;
   }
```

```
   /* look at all of the children of this device */
   for (child = 0; child < dev->maxchild; ++child) {
       if (dev->children[child]) {
           retval =
               match_device (dev->children[child]);
           if (retval == 0)
               goto exit;
       }
   }
exit:
   return retval;
}
```

Specifying a USB Device

Because every user has different types of USB devices, specifying the device to look for must be done in a simple manner. All USB devices have a specific vendor and product ID. You can see these values by using the lsusb or usbview program when there are some USB devices plugged in to your system. This information also is shown in the /proc/bus/usb/devices file, in the lines starting with ``P:''. See the Documentation/usb/proc_usb_info.txt file for more information on how the data in this file is presented.

The match_device() function looks to see if the value of the specific device matches the vendor_id and product_id variables. These variables are defined in the code as:

```
static int vendor_id = 0x0557;
static int product_id = 0x2008;

MODULE_PARM(vendor_id, "h");
MODULE_PARM_DESC(vendor_id,
            "USB Vendor ID of device to look for");

MODULE_PARM(product_id, "h");
MODULE_PARM_DESC(product_id,
           "USB Product ID of device to look for");
```

This allows the module to be loaded with the vendor and product ID specified on the command line. For example, if you want to specify a USB mouse with vendor ID of 0x04b4 and product ID of 0x0001, the module would be loaded with:

```
modprobe root_plug vendor_id=0x04b4 \
product_id=0x0001
```

If no vendor or product ID is specified on the module load command line, the code defaults to looking for a generic USB to serial converter with a vendor ID of 0x0557 and a product ID of 0x2008.

Building the Module

Finally, we need to add our module to the kernel build process. This is done by adding the following line to the security/Config.in file:

```
tristate 'Root Plug Support'
CONFIG_SECURITY_ROOTPLUG
```

And the following line to the security/Makefile file:

```
obj-$(CONFIG_SECURITY_ROOTPLUG)    += root_plug.o
```

These changes allow the user to select this kernel module either to be built into the kernel directly or as a module. Run your favorite *config option to select the ``Root Plug Support'' (`make oldconfig` will work

nicely here, as only the new option will be asked about if you already have a working .config file set up for your kernel). Then build the kernel as usual.

After your kernel is built and running, load the root_plug module by typing (as root):

```
modprobe root_plug vendor_id=<YOUR_VENDOR_ID> \
product_id=<YOUR_PRODUCT_ID>
```

Now try to run a program as root with your specified USB device plugged in to your system, and then try it without. With the module loaded, and the device removed, the following error happens on my machine:

```
$ sudo ls
sudo: unable to exec
/bin/ls: Operation not permitted
```

Plug the device back in, and things should work just fine.

But Is It Secure?

This example shows how powerful and simple the LSM interface can be. With one hook, any program with the root group ID is prevented from running unless a device is physically present in the system.

Using this code, if the device is not present, users are not allowed to log in to the console, as mingetty traditionally runs as root. But users can log in through SSH as normal users, as sshd already was running before the device was removed. Web pages also can be served, and other services that do not run as root (your mail server, database server, etc.) also will function properly. If one of these server programs were broken into, and they tried to spawn a root shell, that root shell would not be allowed to run.

This module does not prevent any program already running as root from cloning itself, or keep a program from trying to change the privileges that are currently assigned to it. To check for these things, the task_* functions in the security_operations structure should be used. The implementation of these functions will be much like the bprm_check_security function, but the parameters passed to the function will be different, so the egid will need to be determined differently.

There are probably other methods of taking an existing running program and spawning a root process that this module does not catch. Please do not use it in a production environment, but rather as a learning exercise for how to create other LSM example code.

Acknowledgements

I would like to thank Chris Wright, Stephen Smalley, James Morris and all of the other programmers who helped create the LSM interface and get it accepted into the main kernel tree. Due to their hard work, Linux now has a flexible security model that will give everyday users the ability to have access to different security models with little effort. I also would like to thank Alan Cox for the initial idea that spawned this example.

For more information about the LSM Project, the development mailing list, documentation and patches for different kernel versions, please see the web site at lsm.immunix.org.



Greg Kroah-Hartman *is currently the Linux USB and PCI Hot Plug kernel maintainer. He works for IBM, doing various Linux kernel-related things, and can be reached at greg@kroah.com.*