USB Boot Authentication

Sean Easton, Bobby Roy

July 31, 2012

Abstract

For our project, we wanted to edit the Linux kernel boot process to require that a user, as part of the authentication process, insert a uniquely-keyed USB device. If the user failed to provide the required device, the boot process would halt, preventing the machine from booting normally until the device was inserted. We wanted to develop a system that added an additional layer of security on top of the password authentication system already present in GNU Linux. The new layer of security would be keyed based on something that the user possessed (the USB drive) rather than just something the user knew (a password).

1 Introduction

For our final project, we decided to interact with the boot code contained within the Linux kernel. We wanted to better understand the boot code in the Linux kernel as well as better understand how to implement security mechanisms inside of the kernel. Instead of forcing users to exclusively authenticate themselves using a password, we decided to add functionality within the kernel to support authenticating a user using a USB drive. We decided that the USB-drive-based authentication would key the drive based on the serial number of the USB drive. Instead of completely replacing the password-based authentication system, the USB-authentication system would provide an additional layer of security, prompting the user for the USB drive before prompting them for a password. If the user failed to insert the correct USB drive, the system would halt the boot process, waiting until the user provided a USB drive with the correct serial number. Once the device was inserted into the machine, the user would then be prompted to enter his or her password as usual.

2 Setup

The machine we designed out project on had the following specifications:

<table>
<thead>
<tr>
<th>Optiplex 780</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core 2 Duo 2.93 GHz x 2</td>
</tr>
<tr>
<td>Ubuntu 12.04 64-bit</td>
</tr>
<tr>
<td>3.7 Gbs of Memory</td>
</tr>
<tr>
<td>120 Gbs of hard drive space</td>
</tr>
</tbody>
</table>
3 Unsuccessful Approaches

We began our project by attempting to outline the boot process of the Linux kernel. We wanted to understand exactly how the kernel booted to determine where to enter our changes. The code was rather dense, so we sought out guides that annotated the boot process and helped explain, in high-level terms, what was taking place at each step. However, even with the guides and code, we needed to introduce our own printk statements into the code and boot the kernel a number of times to really see what was going on in detail.

3.1 LUKS and DM-Crypt

The first solution we looked at involved a combination of LUKS and DM-Crypt as a means by which we could ensure the contents of the drive remained secure unless the kernel booted properly. We wanted the hard drive to remain encrypted in the event that the user failed to authenticate himself/herself properly. While we could’ve easily leveraged these two tools to solve our problem, it wouldn’t have required any tampering with the Linux kernel itself. By using LUKS and DM-Crypt, we could have encrypted the entire disk and placed an unencrypted, boot partition on a USB drive that would unencrypt the hard-drive at boot time. However, since the final project needed to involve writing code within the Linux kernel, we abandoned this approach.

3.2 The Bootloader

We also considered the possibility of either editing GRUB writing our own boot loader in order to accomplish our task. Our thinking was that we could have the boot loader jump to a specified point in the kernel boot code (specifically, jump to the code we were going to add), and perform the check there, loading whatever additional subsystems were necessary to allow for the mounting and reading of a USB device. We began reading about GRUB and attempting to learn how to perform the edits and leverage GRUB for these purposes. However, as in the first approach we explored, this approach didn’t necessitate the writing of Linux kernel code. We decided to abandon this approach as well.

3.3 Understanding the Boot Process

In the meantime, we were trying to insert our code in the correct point within the boot code. We wanted to make sure we had all of the appropriate subsystems and kernel resources at our disposal to properly mount the drive, but we wanted to perform the mount as early in the boot process as possible. We began inserting code into the init/main.c file. We peppered the file with printk statements in order to determine when and where certain functions were called. After outlining init/main.c, we began focusing our attention on the init_post() function. We determined that it was the final function to execute before the run_init_process() function attempted to load the actual init binary. It seemed like the latest possible place we could begin inserting our alterations, so we thought we’d begin by placing our code there. We were able to prototype code that successfully mounted and read from a USB drive inside of a kernel module. However, when that same code was placed in the kernel, it failed. We weren’t sure why our code worked inside of a module but failed during the boot process. We suspected it likely had something to do with important kernel subsystems that weren’t available to us until the kernel was fully booted, so we decided to begin working in the latest possible portion of the boot code before the init binary was executed.
3.4 Mounting the Root Filesystem

We also examined the init/do_mounts.c file, hoping to find a function that would allow us to mount the USB device. We added printk statements and outlined the order of the function calls, similar to our approach in init/main.c. We determined that prepare_namespace() is called from kernel_init(). kernel_init() then calls mount_root(). mount_root() is responsible for calling mount_block_root() which then calls do_mount_root() and finally, do_mount_root() calls sys_mount(). The whole process is necessary for mounting the root file system. We were hoping that we could exploit the mounting operations for mounting the root file system to somehow mount the USB drive and read the keyfile.

3.5 Reading a File in Kernelspace

Within init/main.c, we began by attempting to mount USB drive which failed. We tried to invoke the traditional system calls used to mount and read a file (mount, open, and read), only to discover that opening and reading a file in kernel space, while possible, is heavily discouraged. The Linux kernel development community avoids foisting policy on users and programmers when developing the kernel. Allowing the kernel to open and read files may tempt programmers into writing configuration files that are read by the kernel, requiring that a configuration file follow a particular naming scheme and be placed in a specific directory. As a result, opening a file in the kernel is discouraged and designed to be somewhat convoluted. To get around this, we called the following function: set_fs(KERNEL_DS). This function is responsible for changing the "data segment" or the space from which the kernel is allowed to accept system calls. In this case, it tells the kernel to accept system calls from kernel space. Unfortunately, while this worked when prototype in a module, it did not work at boot time. Likely, the sub-systems necessary to execute this operation were unavailable to us at that particular location in the kernel.

4 Successful Approaches

4.1 Detecting a Second Device

We then focused on getting the dev_t value for /dev/sdb (which would the USB drive). Doing this, we were able to detect when any second hard-drive was present on boot. If a second drive

```c
err = sys_mount("/dev/sdb1", "/key", "ext4", MS_MGC_VAL | MS_RDONLY | MS_SILENT, "mode =0755");
err = do_mount("/dev/sdb1", "/media/usb/", "ext4", MS_MGC_VAL, NULL);
if (err)
{
    printk(KERN_CRIT "NO MOUNT\n");
}
else
{
    printk(KERN_CRIT "YES MOUNT\n");
}
```
wasn’t detected, the boot process would halt. To implement the halt, we created a loop that would
infinitely wait for the second device.

```c
static void __init wait_for_dev()
{
    dev_t dev;
    if (dev == 0) {
        printk(KERN_INFO "Waiting for usb device ...\n");
        while ([driver_probe_done] != 0 ||
        {dev = name_to_dev_t("/dev/sdb")} == 0)
            msleep(100);
        async_synchronize_full();
    }
}
```

### 4.2 Uniquely Identifying the USB Drive

We then tried to focus on uniquely identifying the USB device. We tried to get the dev_t value
using the UUID of our USB drive, but we were unsuccessful. We tried passing in the UUID of
various partitions on the USB drive, but the function continually failed, likely due to the fact that
we still weren’t operating with a fully-functional Linux kernel at this point in the boot process. We
read the source for the function in question and attempted to use every permutation of the accepted
strings we could think of, none of which worked. Again, while our test code worked perfectly in a
kernel module, it did not work in the kernel.

We then tried getting the device serial number or device name from the struct device used in
the driver/base/dd.c file. These values were blank or null. We tried to get the driver info from the
struct but this was also blank or null. We then converted the device struct to an hd_device and
attempted to repeat the last process. This did not work. We tried to do a SCSI inquiry ioctl, but
were once again unsuccessful, even though our code functioned fine in a kernel module.

We were finally able to track down the location of the code that announces the presence of a
USB during the boot process. We had tried unsuccessfully using grep. Using cscope, we were able
to locate the point where the messages in syslog announcing the mounting of the USB devices are
located. We found several function/macros to check if a device struct is a USB device and convert
a device struct to a USB device struct. This finally allowed us to identify when the USB drive
was inserted and to grab the serial number from the device and compare it with a hardcoded value
that we included in the Linux kernel code. If the serial numbers matched, the system would boot
correctly.

```c
if([udev->serial] != NULL)
{
    if ([strcmp([udev->serial], "3513001D97827E69")] == 0) /* Hard coded usb device serial here*/
    {
        key_dev_found = 1;
    }
}
```
5 Conclusion

We were ultimately able to implement functionality within the Linux kernel to provide the user of our modified kernel with an added layer of security. The boot process now requires that a user insert a specific USB drive to boot the machine.

6 Future Work

For the future, we’d like to design an interface that allows a user of our modified kernel to easily alter the value that the USB drive’s serial is checked against in order to make the new features more convenient for the user. Additionally, we’d like to find a way to change the key value of the USB drive to something other than the serial number of the USB drive. Keying the USB drive using the serial provides us with a proof-of-concept example of a working form of our authentication scheme, but an alternative value that the user has more control over and is more difficult for an adversary to discover (the UUID of a partition, a keyfile stored on the USB drive, etc) is more desirable.