The Law Enforcement and Forensic Examiner Introduction to Linux

A Beginner's Guide

Barry J. Grundy
Special Agent
NASA Office of Inspector General
Computer Crimes Division
Code 190 Greenbelt Rd.
Greenbelt, MD 20771
(301) 286-3358
bgrundy@imx.hq.nasa.gov

VER 2.0.5
January 2004

Legalities

Foreword

This purpose of this document is to provide an introduction to the GNU/Linux (Linux) operating system as a forensic tool for computer crime investigators. There are better books written on the subject of Linux (by better qualified professionals), but my hope here is to provide a single document that allows a user to sit at the shell prompt (command prompt) for the first time and not be overwhelmed by a 700-page book.

Tools available to investigators for forensic analysis are presented with practical exercises. **This is by no means meant to be the definitive "how-to" on forensic methods using Linux**. Rather, it is a ***starting point*** for those who are interested in pursuing the self-education needed to become proficient in the use of Linux as an investigative tool. Not all of the commands offered here will work in all situations, but by describing the ***basic*** commands available to an investigator I hope to "start the ball rolling". I will present the commands, the reader needs to follow-up on the more advanced options and uses. Knowing *how* these commands work is every bit as important as knowing what to type at the prompt. If you are even an intermediate Linux user, then much of what is contained in these pages will be review. Still, I hope you find some of it useful.

Over the past couple of years I have repeatedly heard from colleagues that have tried Linux by installing it, and then proceeded to sit back and wonder "what next?" You have a copy of this introduction. Now download the exercises and drive on.

As always, I am open to suggestions and critique. My contact information is on the front page. If you have ideas, questions, or comments, please don't hesitate to call or e-mail me. Any feedback is welcome.

This document is often updated. Check for newer versions (numbered on the front page) on the NASA Headquarters FTP site or in the "resources" section of the Ohio HTCIA website:

ftp://ftp.hq.nasa.gov/pub/ig/ccd/linuxintro/

http://www.ohiohtcia.org/resource.html

A word about the "GNU" in GNU/Linux

When we talk about the Linux operating system, we are actually talking about the GNU/Linux operating system (OS). Linux itself is *not* an OS. It is just a kernel. The OS is actually a combination of the Linux kernel and the GNU utilities that provide the tools allowing us to interact with the kernel. Which is why the proper name for the OS is "GNU/Linux". We (incorrectly) call it "Linux" for convenience.

Why Learn Linux?

One of the questions I hear most often is: "why should I use Linux when I already have [*insert Windows GUI forensic tool here*]?"

There are many reasons why Linux is quickly gaining ground as a forensic platform. I'm hoping this document will illustrate some of those attributes.

- Control – not just over your forensic software, but the whole OS and attached hardware.
- Flexibility – boot from a CD (to a complete OS), file system support, platform support, etc.
- Power – A Linux distribution *is* a forensic tool.

Another point to be made is that simply knowing *how* Linux works is becoming more and more important. While many of the Windows based forensic packages in use today are fully capable of examining Linux systems, the same cannot be said for the examiners.

As Linux becomes more and more popular, both in the commercial world and with desktop users, the chance that an examiner will encounter a Linux system in a case becomes more likely (especially in network investigations). Even if you elect to utilize a Windows forensic tool to conduct your analysis, you *must* at least be familiar with the OS you are examining. If you do not know what is normal, then how do you know what does not belong? This is true on so many levels, from the actual contents of various directories to strange entries in configuration files, all the way down to how files are stored. While this document is more about Linux as a forensic tool rather than analysis of Linux, you can still learn a lot about how the OS works by actually *using* it.

I. Installation

First and foremost, *know your hardware*. If your Linux machine is to be a dual boot system with Windows, then use the Windows Device Manager to record all your installed hardware and the settings used by Windows. If you are setting up a standalone Linux system, then gather as much documentation about your system as you can. **This has become much less important with the evolution of the Linux install routines**. Hardware compatibility and detection have been *greatly* improved over the past couple of years.

- Hard drive – knowing the size and geometry is helpful when planning your partitioning.
- SCSI adapters and devices (note the adapter chipset). SCSI is very well supported under Linux.
- Sound card (note the chipset).
- Video Card (important to know your chipset and memory, etc.).
- Monitor timings
  - Horizontal and vertical refresh rates.
- Network card settings (chipset).
- Network Parameters:
  - IP (if not DHCP)
  - Netmask
  - Broadcast address
  - DNS servers
  - Default gateway
- Modem
  - NO WINMODEMS. (Support is being worked on – check http://www.linmodems.org. Note that if you have an HSF modem, Conexant has released Linux drivers! Find them at http://www.conexant.com/customer/. I use them, and they work.)
- USB support is in kernel 2.4. USB is standard in current distributions.
- IEEE1394 (firewire) support is included in current distributions.

Most distributions have a plethora of documentation, including online help and documents in downloadable form. For example, Red Hat users can check for hardware compatibility and installation issues at: http://www.redhat.com/support/hardware/

If you cannot find your monitor documentation and need it for XFree86 (the Linux GUI) setup, then go to: http://www.monitorworld.com/monitors_home.html

Distributions

Linux comes in a number of different "flavors". These are most often referred to as "distributions" ("distro"). Default kernel configuration, tools that are included (system management and configuration, etc.) and the package format (i.e., the upgrade path) most commonly differentiate the various Linux distros.

It is common to hear users complain that device X works under Suse Linux, but not on Red Hat, etc. Or that device Y did not work under Red Hat version 7.3, but an upgrade to 8.0 "fixed it". Most often, the difference is in the version of the Linux *kernel* being used and therefore the updated drivers, or the patches applied by the distribution vendor, not the version of the distribution (or the distribution itself).

Red Hat

One of the most popular Linux distributions (right now). Red Hat works with companies like Dell, IBM and Intel to assist business in the adoption of Linux for enterprise use. Use of RPM and Kickstart began the first "real" user upgrade paths for Linux. Red Hat is an excellent choice for beginners because of the huge install base and the proliferation of online support. The install routine is well polished and hardware support is well documented. While Red Hat has elected to move into a more enterprise oriented business model, it is still a viable option for the desktop through the "Fedora Project" (http://fedora.redhat.com/).

If you are installing Linux specifically to accompany this document, then I'd suggest any of Red Hat 7.3, 8.0, or 9.0, depending on your required hardware support.

Debian

Not really for beginners. The installation routine is not as polished as some other distributions. Debian has always been a hacker favorite. It is also one of the most "non-commercial" Linux distributions, and true to the spirit of GNU/GPL.

SuSE

Another distribution with its own proprietary install program, *YaST2*. SuSE is German in origin. It is by far the largest software inclusive distribution, and comes with six (6!) CD's (or a DVD).

Slackware

The original commercial distribution. Slackware has been around for years. Installation is not as easy as others. Good standard Linux. Not over-encumbered by GUI config tools.

Mandrake Linux

Red Hat based and rapidly gaining on Red Hat's desktop market share, Mandrake is a favorite of many beginners and desktop users. It is heavy on GUI configuration tools, allowing for easy migration to a Linux desktop environment. Mandrake is also a good choice to use with this document's exercises.

Gentoo Linux

Source-centric distribution that is optimized during install – my personal favorite. Once through the complex installation routine, upgrading the system and adding software is made extremely easy through Gentoo's "Portage" system. *Not* for beginners, though. You are left to configure the system entirely on your own.

My suggestion for the absolute beginner would be either the newest version of Mandrake (currently 9.2) or Red Hat (currently 9.0). Mandrake is actually a Red Hat based distribution with numerous GUI enhancements that make the learning process easier for "newbies". Also keep in mind that Red Hat is discontinuing support for it's free distribution, which is being replaced by the Fedora Project. If you really want to "dive in" and bury yourself, go for Gentoo, Debian or Slackware. If you choose one of the latter distributions, be prepared to read…a lot.

One thing to keep in mind: If you are going to use Linux in a forensic capacity, then try not to rely on GUI tools too much. Almost all settings and configurations in Linux are maintained in text files (usually in either your home directory, or in */etc*). By learning to edit the files yourself, you avoid problems when either the X window system is not available, or when the specific GUI tool you rely on is not on a system you might come across. In addition, knowledge of the text configuration files will give you insight into what is "normal", and what might have been changed when you examine a subject system.

Installation Methods:

- Buy a book! (Most come with a distribution).
- Download the needed files, create a boot and root disk and read online! (See the "Linux Support" section at the end of this document).
- Get hold of a distribution CD and boot from it (change your bios to boot from the CD if needed).
- Use a bootable Linux distribution (covered later).

If you have access to a bootable installation CDROM (download an ISO image and burn it on a CDR, buy a book that includes a CD set, etc.), then this process will be easier. Much of the work is done for you, and relatively safe defaults are provided. As mentioned earlier, hardware detection has gone through some great improvements in the last year or two. I strongly believe that Red Hat or Mandrake Linux are far easier and faster to install than Windows 2000. Typical Linux installation is well documented online (check the "how-tos" at the Linux Documentation Project: http://www.tldp.org/). There are numerous books available on the subject, and as previously mentioned most of these are supplied with a Linux distribution ready for install.

Bootable ISO's can downloaded from http://www.linuxiso.org/ and burned to a CD. Familiarize yourself with Linux disk and partition naming conventions (covered in Chapter II of this document) and you should be ready to start.

Installation Overview

1) Decide on standalone Linux or dual boot.

    -Install Windows first in a dual boot system.
    -Determine how you want the Linux system to be partitioned.
    -Do NOT create any extra partitions with Windows **fdisk**. Just leave the space unallocated. The Linux install will create the partitions (or allow you to).

2) Boot the Linux Media

    -Hopefully you have a bootable CDROM (and booting from the CD is supported in your BIOS.)
    -In many cases you can use *boot.img* from the installation CD to create a bootable floppy for the install if booting from the CD is not possible.

3) Accepting most defaults works.

    -Your hardware will be detected and configured under most (if not all) circumstances. If the install freezes or breaks, try again in “text” mode or “expert” mode, if available. This is often caused by video card or SCSI card problems or conflicts. Support online is extensive if you have problems.

4) Partition and format for Linux

    -Use at least two partitions.
    -Root (/) as type “Linux Native”.
    -Swap as type “Linux Swap” (use 2x your system memory as a starting point for swap size).
    -You will hear a lot about using multiple partitions for different directories. Don’t let that confuse you. There are arguments both for and against using multiple partitions for a Linux file system. If you are just starting out, just use one large root (/) partition, and one swap partition. This will allow for the least confusion. You will see examples of other partitioning schemes later.

    If you are comfortable with the idea of multiple partitions for your Linux install, then it is recommended that you use at least a separate */boot* partition along with the root (/) and swap.

5) Package installation (system)

-When asked which packages to select for installation, it is usually safe for a beginner to select "everything" (as in Red Hat or Mandrake). This allows you to try all the packages, along with both KDE and Gnome (X Window GUIs). This can take as much as 2GB on some of the newer distributions, however it includes *all* the software you are likely to need for a long time (including "office" type applications). This is not really optimal or recommended, but for a learning box it will give you the most exposure to available software for experimentation.

6) Installation Configuration

-Sound

-Usually automatic. If not, search the Web. The answer is out there. If the sound is not configured automatically, try running *sndconfig* if it is available (i.e. most Redhat systems).

-Xfree86 (X Window system)

-Know your hardware.

-If you choose to configure X during the installation routine, **do not** click "yes" when asked if you want X to start automatically every time you system boots. This can make problem solving difficult and results in less control over the system. You can always start the GUI with "*startx*" from the command line.

7) Boot Method (the Boot loader…selects the OS to boot)

-LILO or GRUB.

-Some people find GRUB more flexible and secure. Usually select the option to install to the MBR. The presences of other boot loaders determine where to install LILO or GRUB.

-The boot loader contains the code that points to the kernel to be booted. Check www.tldp.org for "multiOS" and "multiboot" How-To documents.

-Bootdisk should be created for rescue.

8) Create a username for yourself

-Linux is a multiuser system. It is designed for use on networks (remember, it is based on Unix). The "root" user is the system administrator, and is created by default during installation. Exclusive use of the "root" login is DANGEROUS. Linux assumes root knows what he or she is doing and allows “root” to do anything he or she wants, including destroy the system. Create a new user. Don’t log in as “root” unless you must. Having said this, much of the work done for forensic analysis must be done as “root” to allow access to raw devices and system commands.

The New 2.6 Linux Kernel

In December of 2003, the Linux 2.6 kernel was released. While this is another milestone in the Linux saga, it would be wise to stay with the 2.4 kernel until tests are done on changes that affect our work.

Many of the changes in 2.6 are geared toward enterprise use and scalability. The new kernel release also has a number of infrastructure changes that could have a huge impact on Linux as a forensic platform. For example, there is enhanced support for USB and a myriad of other external devices. The kernel module and entire device sub-systems have been changed and improved, making them more robust. And we will soon have access to “user mode” Linux that could provide a whole new environment for us to work in.

As with all forensic tools, we need to have a clear view of how the new kernel will interact with our forensic platforms and subject hardware. This will take some time.

II. Linux Disks, Partitions and the Filesystem

Disks

Linux treats its devices as files. The special directory where these "files" are maintained is "*/dev*".

• Floppy (a:)	/dev/fd0
• Floppy (b:)	/dev/fd1
• 1st Hard disk (master, IDE-0)	/dev/hda
• Hard disk (slave, IDE-0)	/dev/hdb
• Hard disk (master, IDE-1)	/dev/hdc, etc.
• 1st SCSI hard disk	/dev/sda
• 2nd SCSI hard disk	/dev/sdb, etc.

Partitions

1st Hard disk (master, IDE-0)	/dev/hda
• 1st Primary partition	/dev/hda1
• 2nd Primary partition	/dev/hda2, etc.
• 1st Logical drive (on ext'd part)	/dev/hda5
• 2nd Logical drive	/dev/hda6, etc.
2nd Hard disk (slave, IDE-0)	/dev/hdb
• 1st Primary partition	/dev/hdb1, etc.
CDROM or 3rd disk (master, IDE-1)	/dev/hdc
CDROM (SCSI)	/dev/scd0
1st SCSI disk	/dev/sda
• 1st Primary partition	/dev/sda1, etc.

The pattern described above is fairly easy to follow. If you are using a standard IDE disk, it will be referred to as "hdx" where the "x" is replaced with an "a" if the disk is connected to the primary IDE controller as master and a "b" if the disk is connected to the primary IDE controller as a slave device. In the same way, the IDE disks connected to the secondary IDE controller as master and slave will be referred to as "hdc" and "hdd" respectively.[1]

[1] Some distributions support *devfs* which uses a different naming scheme. Don't let this confuse you. The pattern described above is still supported through "links" for compatibility. See http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html for more information.

This is an example of the output of **fdisk -l /dev/hda** on a dual boot system:

```
   Disk /dev/hda: 255 heads, 63 sectors, 1582 cylinders
   Units = cylinders of 16065 * 512 bytes
  Device     Boot   Start   End     Blocks      Id    System
/dev/hda1    *      1       255     2048256     b     Win95 FAT32
/dev/hda2           256     638     3076447+    83    Linux
/dev/hda3           639     649     88357+      82    Linux swap
/dev/hda4           650     1582    7494322+    f     Win95 Ext'd (LBA)
/dev/hda5           650     1453    6458098+    b     Win95 FAT32
/dev/hda6           1454    1582    1036161     b     Win95 FAT32
```

fdisk –l /dev/hd*x* gives you a list of all the partitions available on a particular drive. Each partition is identified by its Linux name. The "boot flag" is indicated, and the beginning and ending cylinders for each partition is given. The number of blocks per partition is displayed. Finally, the partition "Id" and file system type are displayed. To see a list of valid types, run **fdisk** and at the prompt type "l" (the letter “el”). Do not confuse Linux **fdisk** with DOS **fdisk**. They are very different. The Linux version of **fdisk** provides for much greater control over partitioning.

BEFORE FILESYSTEMS ON DEVICES CAN BE USED, THEY MUST BE MOUNTED! Any file systems on partitions you define during installation will be mounted automatically every time you boot. We will cover the mounting of file systems in the section that deals with Linux commands, after you have some navigation experience.

Keep in mind, that even what not mounted, *devices* can still be written to. Simply not mounting a file system does not protect it from being inadvertently changed through your actions.

Note that if you use a parallel ZIP drive or USB disk (thumb drive, memory stick, etc.), it will be accessed as */dev/sda* (assuming no other SCSI devices) or */dev/sdb*. Support must be compiled either into the kernel, or as a loadable module. Most new distros have USB support already included. Also, a Linux formatted ZIP disk is */dev/sda1* and a DOS (FAT) formatted ZIP disk is */dev/sda4* (no, I don't know why). It’s not unusual for Linux to recognize storage peripherals as SCSI devices. The same is true for

IEEE1394 "firewire" devices. In order mount file systems on these types of "external" devices, we may need to delve a little deeper into modules.

Using modules

It's difficult to decide when to introduce modules to a new user. The concept can be a little confusing, but "out of the box" Linux distributions rely heavily on modules for device and file system support. For this reason, we will make an effort to get familiar with the concept early on.

Modules are really just "drivers" that can be loaded and unloaded from the kernel dynamically. They are object files (*.o) that contain the required driver code for the supported device or option (file system support under Linux is often loaded as a module). The various modules available on your system are located in */lib/modules/<KERNEL-VERSION>/*. Note that the current kernel version running on your system can be found using the command **uname -r**.

Modules are installed and removed from the system "on the fly" using the following commands (as root):

insmod	-to insert the module
rmmod	-to remove the module
lsmod	-to get a list of currently installed modules

For example, to get USB support for a USB thumb drive on my system, I need to load a couple of modules. With the USB device plugged in, we install the needed modules (*usb-uhci.o* for many controllers, and *usb-storage.o* for the storage interface) with:

```
insmod usb-uhci   (depending on your USB controller)
insmod usb-storage
```

Note that while the module is named with a ".o" extension, we do not include that in the insertion command.

Another method we can use to install modules is with the command **modprobe**. This command installs the requested module after attempting to determine if there are any other modules that must be loaded to satisfy dependencies. Check out **man depmod** and **man modprobe**.

You can check to see if the module has been correctly loaded with:

```
lsmod
```

The output should be:

```
Module          Size        Used by
usb-storage   104384        0
usb-uhci       24356        0  (unused)
```

Note that the output of **lsmod** will include any other drivers loaded by the system as well (including those loaded at boot time).

Often, once you have loaded the required drivers, you will see activity lights on the drives kick in as the equipment becomes accessible. Again, USB and Firewire drives are usually picked up as SCSI devices once the modules are loaded, so check */dev/sda* or */dev/sdb* using **fdisk** to see if the device is recognized and which partitions are available.

The above commands will get USB working on many systems, for Firewire, try installing the modules *ieee1394.o*, *ohci1394.o*, and for Firewire hard drive support, *sbp2.o*. If these commands don't work for your system's USB or Firewire controller, then try searching Google for answers specific to your hardware.

Modules on Newer systems

On newer Linux systems (like the one you are probably using now) there is often an automatic *kernel module daemon* that handles the loading and unloading of modules automatically. When you issue a command that requires a module that is not yet loaded, the kernel will often detect your request and load the applicable module. The module "autoloader" is useful and ends the need to install modules by hand using **insmod** in ***some*** cases (like when mounting supported file systems). It is likely that your system follows this convention. I would not suggest relying on this feature, however. If you have the kernel sources on your system, check out */usr/src/linux/Documentation/modules.txt* for more detailed information.

The Filesystem

Like the Windows file system, the Linux file system is hierarchical. the "top" directory is referred to as "the root" directory and is represented by "/". Note that the following is not a complete list, but provides an introduction to some important directories.

```
/ (“root” not to be confused with “/root”)
        |_ bin
        |       |_ <files> ls, chmod, sort, date, cp, dd
        |_boot
        |       |_<files> vmlinuz, system.map
        |_ dev
        |       |_<devices> hd*, tty*, sd*, fd*, cdrom
        |_ etc
        |       |_X11
        |               |_ <files> XF86Config, X
        |       |_<files> lilo.conf, fstab, inittab, modules.conf
        |_ home
        |       |_barry (your user’s name is in here)
        |               |_<files> .bashrc, .bash_profile, personal files
        |       |_other users
        |_mnt
        |       |_cdrom
        |       |_floppy
        |       |_other external file system mount points
        |_root
        |       |_<root user's home directory>
        |_sbin
        |       |_<files> shutdown, cfdisk, fdisk, insmod
        |_usr
        |       |_local
        |       |_lib
        |       |_man
        |_var
        |       |_log
```

On most Linux distributions, the directory structure is organized in the same manner. Certain configuration files and programs are distribution dependant, but the basic layout is similar to this.

Directory contents can include:

- **/bin** -Common commands.
- **/boot** -Files needed at boot time, including the kernel images pointed to by LILO (the LInux LOader) or GRUB.
- **/dev** -Files that represent devices on the system. These are actually interface files to allow the kernel to interact with the hardware and the file system.
- **/etc** -Administrative configuration files and scripts.
- **/home** -Directories for each user on the system. Each user directory can be extended by the respective user and will contain their personal files as well as user specific configuration files (for X preferences, etc.).
- **/mnt** -Provides mount points for external, remote and removable file systems.
- **/root** -The root user's home directory.
- **/sbin** -Administrative commands and process control daemons.
- **/usr** -Contains local software, libraries, games, etc.
- **/var** -Logs and other variable file will be found here.

Another important concept when browsing the file system is that of *relative* versus *explicit* paths. While confusing at first, practice will make the idea second nature. Just remember that when you provide a pathname to a command or file, including a "/" in front means an *explicit* path, and will define the location *starting from the top level directory (root)*. Beginning a pathname **without** a *"/"* indicates that your path *starts in the current directory* and is referred to as a *relative* path. More on this later.

III. The Linux Boot Sequence (Simplified)

Booting the kernel

The first step in the (simplified) boot up sequence for Linux is loading the kernel. The kernel image is usually contained in the */boot* directory. It can go by several different names…

- bzImage
- vmlinuz

Sometimes the kernel image will specify the kernel version contained in the image, i.e. *bzImage-2.4.18*. Very often there is a soft link (like a shortcut) to the most current kernel image in the */boot* directory. It is normally this soft link that is referenced by the boot loader, LILO (or GRUB). The boot loader specifies the “root device” (boot drive), along with the kernel version to be booted. For LILO, this is all controlled by the file */etc/lilo.conf.* Each “image=” section represents a choice in the boot screen. GRUB is a more recent boot loader used by many Linux distributions. The concept is the same. You can look at the **man** or **info** pages for either boot loader for more information.

more /etc/lilo.conf

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz-2.4.17   ← Defines the Linux kernel to boot
    label=linux_old           ← Menu choice in LILO
    root=/dev/hda3            ← Where the root file system is found
    read-only
other=/dev/hda1               ← Defines alternate boot option
   label=Win2k                ← Menu choice in LILO
   table=/dev/hda
```

In the case of GRUB, each section beginning with “title” is a choice for booting and can include Linux as well as other operating systems, including Windows. Note again the reference to the kernel location, and the “root device” (where the root filesystem is located). GRUB starts it’s counting from 0, so where you see "*hd0,0*" it is referring to the first IDE disk, followed by the first partition. See the **info** or **man** page for GRUB.

In the following GRUB example, there will be three different Linux kernel choices offered in the boot menu. They all use the same root file system, but differ in the kernel image loaded from the */boot* partition.

more /etc/grub.conf

```
boot=/dev/hda
default=0
timeout=10
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
title Linux (2.4.20)
        root (hd0,0)
        kernel /boot/bzImage-2.4.20 ro root=/dev/hda1
title Linux-enh (2.4.20-xfs-enh)
        root (hd0,0)
        kernel /boot/bzImage-2.4.20-xfs-enh ro root=/dev/hda1
title Linux (2.4.19)
        root (hd0,0)
        kernel /boot/bzImage-2.4.19 ro root=/dev/hda1
```

Once the system has finished booting, you can see the kernel messages that "fly" past the screen during the booting process with the command **dmesg.** Often, this command can be used to find hardware problems, or to see how your suspect drive was detected (geometry, etc). The output can be piped through a paging viewer to make it easier to see:

dmesg | less

Initialization

The next step starts with the program */sbin/init.* This program really has two functions:

- initialize the runlevel and startup scripts
- terminal process control (respawn terminals)

In short, the init program is controlled by the file */etc/inittab*. It is this file that controls your runlevel and the global startup scripts for the system.

Runlevel

The runlevel is simply a description of the system state. For our purposes, it is easiest to say that (for *Red Hat*, at least – other systems, like *Suse* may differ):

- runlevel 0 = shutdown
- runlevel 1 = single user mode
- runlevel 3 = full multiuser mode / text login
- runlevel 5 = full multiuser / X11 / graphical login
- runlevel 6 = reboot

In the file */etc/inittab* you will see a line similar to:

```
id:3:initdefault:
```

It is here that the default runlevel for the system is set. If you want a text login (which I would strongly suggest), set the above value to "*3*". You can always use "*startx*" to get to the X Window GUI system. If you want a graphical login, you would edit the above line to contain a "5".

Global Startup Scripts

After the default runlevel has been set, *init* (via */etc/inittab*) then runs the following scripts:

- */etc/rc.d/rc.sysinit* - handles system initialization, file system mount and check, PNP devices, etc.
- */etc/rc.d/rc X* - where *X* is the runlevel passed as an argument by *init*. This script then calls the specified script for the runlevel that is being started.
- */etc/rc.d/rc.local* - called from within the specific runlevel scripts, *rc.local* is a general purpose script that can be edited to include commands that you want started at bootup (sort of like *autoexec.bat*).

Again, this is somewhat Red Hat / Mandrake specific. Other distributions can differ slightly (some differ greatly!), but the concept remains consistent. Once you become familiar with the process, it will make sense. The ability to manipulate startup scripts is an important step in your Linux learning process.

Bash

Bash *(Bourne Again Shell*) is the default command shell for Red Hat, Mandrake, and many other Linux distros. It is the program that sets the environment for your command line experience in Linux. The functional equivalent in DOS would be *command.com*. There are a number of shells available, but we will cover bash here.

There are actually quite a few files that can be used to customize a user's Linux experience. Here are two that will get you started. I am assuming here that you are using the bash shell.

- */home/$USER/.bashrc*[2] - This script is located in each user's home directory (*$USER*) and can be edited by the user, allowing him or her to customize their own environment. It is in this file that you can add aliases to change the way commands respond.
- */etc/bashrc* - This is the global bash initialization file. Edits made to this file will be applied to all bash shell users.

[2] In bash we define the contents of a variable with a dollar sign. $USER is a variable that represents the name of the current user. To see the contents of shell individual variables, use "echo $VARNAME".

The bash startup sequence is actually more complicated than this, but this should give you a starting point. In addition to the above files, check out */home/$USER/.bash_profile* and */etc/profile.* The **man** page for bash is an interesting (and long) read, and will describe some of the customization options. In addition, reading the **man** page will give a good introduction to the programming power provided by bash scripting.

IV. DOS / Linux Equivalent Commands

"DOS command" = Linux equivalent

"dir" = **ls** list files.
ls –F classifies files and directories.
ls –a show all files (including hidden).
ls –l detailed file list (long view).
ls –lh detailed list (long, with “human readable” file sizes).

Output of **ls -l**

```
total 11
drwx------    5 barry    501      1024 Feb  7 15:07 Desktop
drwxr-xr-x    2 barry    users    1024 Feb  9 08:19 Files
drwx------    2 barry    users    1024 Dec 18 15:58 Mail
-rw-r--r--    1 barry    users       0 Feb  9 09:21 Mydata.file.today
drwxr-xr-x   26 barry    users    1024 Jan 23 00:49 Office51
drwxr-xr-x    2 barry    users    1024 Nov  8 16:21 Prog
drwxr-xr-x    2 barry    users    1024 Dec 16 15:42 Programming
drwxr-xr-x    2 barry    users    1024 Feb  9 09:27 Sample
drwxr-xr-x    2 barry    users    1024 Feb  9 08:41 autosave
drwxr-xr-x    2 barry    users    1024 Feb  8 15:50 bin
drwx------    2 barry    501      1024 Oct  9 14:00 nsmail
drwxrwxr-x    3 barry    501      1024 Jan 23 00:42 software
-rw-r--r--    1 barry    users       0 Feb  9 09:20 textfile
```

"cd" = **cd <dir>** change directory to <dir>.
cd (by itself) shortcut back to your home directory.
cd .. up one directory (note the space between “cd” and “..”.
cd - back to the last directory you were in.
cd /*dirname* change to the specified directory. Note that the addition of the “/” in front of the directory implies an explicit (absolute) path, not a relative one. With practice, this will make more sense.
cd *dirname* change to the specified directory. The lack of a “/” in front of the directory name implies a relative path meaning *dirname* is a subfolder of our current directory.

"copy" = **cp**

cp *sourcefile destinationfile* copy a file.

“cls” = **clear**

clears the terminal screen of all text and returns a prompt.

"move" and "ren" = **mv**

mv *sourcefile destinationfile* move or rename a file.

"del" = **rm**

rm *filename*	deletes a file.
rm -r	recursively deletes all files in directories and subdirectories.

"help" or " /?" = **man**

man *command*	displays a "manual" page for the specified command. Use "q" to quit. VERY USEFUL.

If you want to find information about a command called **find**, including its usage, options, output, etc., then you would use the “man page” for the command **find**.

Output of **man find:**

```
FIND(1L)                                  FIND(1L)
     find - search for files in a directory hierarchy

SYNOPSIS
    find [path...] [expression]

DESCRIPTION
     This  manual page documents the GNU version of find.  find
     searches the directory tree rooted at each given file name
     by  evaluating  the  given  expression from left to right,
     according to the rules of precedence (see  section  OPERA-
     TORS),  until  the outcome is known (the left hand side is
     false for and operations, true for or ),  at  which  point
     find moves on to the next file name.
                              <CONTINUES WITH MORE INFO>
```

"md" = **mkdir**

mkdir *directoryname* creates a directory. Again, remember the difference between a relative and explicit path here.

"type" = **cat** or **more** or **less**

cat *filename* The simplest form of file display, **cat** streams the contents of a file to the standard output (usually the terminal). **cat** actually stands for "*concatenate*". This command can also be used to add files together (useful later on…). For example:

cat *file1 file2* > *file3*

Takes the contents of *file1* and *file2* and streams the output which is redirected to a single file, *file3*. This effectively adds the two files into one single file (the original files remain unchanged).

more *filename* displays the contents of a file one page at a time. Unlike its DOS counterpart, Linux **more** takes filenames as direct arguments.

less *filename* **less** is a better **more**. Supports scrolling in both directions, and a number of other powerful features. **less** is actually the GNU version of **more**, and on many systems you will find that **more** is actually a link to **less**.

Note that you can string together several options. For example:

ls -aF

will give you a list of all files (-a), including hidden files, and file/directory classification (-F, which shows "/" for directories, "*" for executables, and "@" for links).

Output of **ls –aF :**

```
./              .emacs            .kderc        .zshrc             Sample/
../             .gnome/           .kpackage/    Desktop/           autosave/
.Xauthority     .gnome_private/   .mailcap      Files/             bin/
.Xdefaults      .gqviewrc         .maxwellrc    Mail/              mylink@
.bash_history   .gxedit           .netscape/    Mydata.file.today  nsmail/
.bash_logout    .gxedit.apps      .sversionrc   Office51/          samp_script.sh*
.bash_profile   .kaudioserver     .user.rdb     Prog/              snapshot01.gif
.bashrc         .kde/             .vimrc        Programming/       software/
                                                                   textfile
```

Additional useful commands

grep - search for patterns.

g**rep** ***pattern filename***

Grep will look for occurrences of *pattern* within the file *filename.* **grep** is an extremely powerful tool. It has hundreds of uses given the large number of options it supports. Check the **man** page for more details.

find -allows you to search for a file (wild cards – actually “expressions” permitted). To look for your *XF86Config* file, you might try:

find / -name XF86Config -print

This means "find, starting in the root directory (/), by name, *XF86Config* and print the results to the screen". **find** also allows you to search by file type or even file times (actually *inode* times).

pwd -prints the present working directory to the screen.

```
pwd
/home/barry
```

file -categorizes files based on what they contain, regardless of the name (or extension, if one exists). Compares the file header to the "magic" file in an attempt to ID the file type. For example:

```
file snapshot01.gif
snapshot01.gif: GIF image data, version 87a, 800 x 600
```

ps -list of current processes. Gives the process ID number (PID), and the terminal on which the process is running.

ps -ax shows all processes (-**a**), and all processes without an associated terminal (-**x**).

Output (partial) of **ps -ax** on my system as it is running right now:

```
PID TTY    STAT          TIME   COMMAND
   1 ?        S           0:04 init
   2 ?        SW          0:00 [kflushd]
   3 ?        SW          0:00 [kupdate]
   4 ?        SW          0:00 [kpiod]
   5 ?        SW          0:00 [kswapd]
  191 ?       S           0:01 /sbin/pump -i eth0
  243 ?       S           0:00 klogd
328 tty1      SW          0:00 [login]
  329 tty2    S           0:00 login -- root
  330 tty3    S           0:00 /sbin/mingetty tty3
  331 tty4    S           0:00 /sbin/mingetty tty4
  332 tty5    S           0:00 /sbin/mingetty tty5
  333 tty6    S           0:00 /sbin/mingetty tty6
  340 tty1    SW          0:00 [bash]
  353 tty1    S           0:00 sh /usr/X11R6/bin/startx
  360 tty1    S           0:00 xinit /etc/X11/xinit/xinitrc -- :0 -auth /home/barry/
  361 ?       R           2:04 /etc/X11/X :0 -auth /home/barry/.Xauthority
  365 tty1    S           0:05 kwm
  368 tty1    S           0:00 kbgndwm
```

strings -prints out the readable characters from a file. Will print out strings that are at least four characters long (by default)from a file. Useful for looking at data files without the originating program, and searching executables for useful strings, etc.

chmod -changes the permissions on a file. (See the section in this document on permissions).

chown -changes the owner of a file in much the same way as chmod changes the permissions.

```
chown ralph filename

-rwxrwxr--   1 ralph    user       1643 Jan 19 23:23 filename
```

chgrp - changes a file's group attribute. Works the same as chown, but affects the group instead of the owner.

shutdown -this command MUST be used to shutdown the machine and cleanly exit the system. This is not DOS. Turning off the machine at the prompt is not allowed and can damage your file system (in some cases)[3]. You can run several different options here (check the man page for many more):

shutdown -r now	-will reboot the system now (change to runlevel 6).
shutdown -h now	-will halt the system. Ready for power down (change to runlevel 0).

File Permissions

Files in Linux have certain specified file permissions. These permissions can be viewed by running the ls -l command on a directory or on a particular file. For example:

```
ls –l filename

-rwxr-xr-x   1 barry    user       1643 Jan 19 23:23 filename
```

If you look close at the first 10 characters, you have a dash (-) followed by 9 more characters. The first character describes the type of file.

[3] This has become much less of an issue with the newer journaled file systems used by Linux.

A dash (-) indicates a regular file. A "d" would indicate a directory, and "b" a special block device, etc.

First character of **ls -l** output:

- = regular file
d = directory
b = block device
c = character device
l = link

The next 9 characters indicate the file permissions. These are given in groups of three:

Owner	Group	Others
rwx	rwx	rwx

The characters indicate

r = read
w = write
x = execute

So in the above *filename* we have

-rwx r-x r-x

This gives the file owner read, write and execute permissions (rwx), but restricts other members of the owner’s group and users outside that group to only read and execute the file (r-x).

Now back to the chmod command. There are a number of ways to use this command, including explicitly assigning r, w, or x to the file. We will cover the octal method here because the syntax is easiest to remember (and I find it most flexible). In this method, the syntax is as follows

chmod *octal filename*

octal is a three digit numerical value in which the first digit represents the owner, the second digit represents the group, and the third digit represents others outside the owner's group. Each digit is calculated by assigning a value to each permission:

read (r)	= 4
write (w)	= 2
execute (x)	= 1

For example, the file *filename* in our original example has an octal permission value of 755 (rwx =7, r-x =5, r-x=5). If you wanted to change the file so that the owner and the group had read, write and execute permissions, but others would only be allowed to read the file, you would issue the command:

chmod 774 filename

4(r)+2(w)+1(x)=7
4(r)+2(w)+1(x)=7
4(r)+0(-)+0(-) =4

A new long list of the file would show:

```
-rwxrwxr--   1 barry    user       1643 Jan 19 23:23 filename
                        (rwx=7, rwx=7, r--=4)
```

Metacharacters

Linux also supports wildcards (metacharacters)

- * for multiple characters (including ".").
- ? for single characters.
- [ ] for groups of characters or a range of characters or numbers.

This is a complicated and *very* powerful subject, and *will* require further reading… Refer to “regular expressions” in your favorite Linux text, along with “globbing” or “shell expansion”. There are important differences that can confuse a beginner, so don’t get discouraged by confusion over what “*” means in different situations.

Command Hints

1. Linux supports command line editing.
2. Linux has a history list of previously used commands (stored in *.bash_history* in your home directory).
   -use the keyboard arrows to scroll through commands
   you've already typed.
3. Linux commands and filenames are CASE SENSITIVE.
4. Learn output redirection for stdout and stderr “>” and “2>”.
5. Linux uses “/” for directories, DOS uses “\”.
6. Linux uses “-“ for command options, DOS uses “/”
7. To execute commands in the current directory (if the current directory is not in your PATH), use the syntax "*./command*". This tells Linux to look in the present directory for the command.

Pipes and Redirection

Like DOS, Linux allows you to redirect the output of a command from the standard output (usually the display or "console") to another device or file. This is useful for tasks like creating an output file that contains a list of files on a mounted volume, or in a directory. For example:

```
ls -al > filelist.txt
```

The above command would output a long list of all the files in the current directory. Instead of outputting the list to the console, a new file called "*filelist.txt*" will be created that will contain the list. If the file "*filelist.txt*" already existed, then it will be overwritten. Use the following command to append the output of the command to the existing file, instead of overwriting it:

```
ls -al >> filelist.txt
```

Another useful tool similar to that available on DOS is the *command pipe*. The command pipe takes the output of one command and "pipes" it straight to the input of another command. This is an extremely powerful tool for the command line. Look at the following process list:

```
ps -ax
```

```
PID TTY     STAT   TIME COMMAND
  1 ?        S       0:04 init
232 ?        S       0:00 syslogd -m 0
271 ?        S       0:00 inetd
328 tty1     SW      0:00 [login]
329 tty2     S       0:00 login -- root
330 tty3     SW      0:00 [mingetty]
  340 tty1   SW      0:00 [bash]
353 tty1     SW      0:00 [startx]
360 tty1     SW      0:00 [xinit]
361 ?        R       2:41 /etc/X11/X :0 -auth /home/barry/.Xauthority
519 pts/0    S       0:00 bash
2490 tty2    S       0:00 -bash
2727 pts/1   R       0:00 ps -ax
```

What if all you wanted to see were those processes ID's that indicated a bash shell? You could "pipe" the output of **ps** to the input of **grep**, specifying "bash" as the pattern for grep to search. The result would give you only those lines of the output from **ps** that contained the pattern "bash".

```
ps -ax | grep bash
```

```
340 tty1      SW      0:00 [bash]
519 pts/0     S       0:00 bash
2490 tty2     S       0:00 -bash
```

A little later on we will cover using pipes on the command line to help with analysis. Stringing multiple powerful commands together is one of using Linux for forensic analysis.

The SuperUser

If Linux gives you an error message "*Permission denied*", then in all likelihood you need to be "root" to execute the command or edit the file, etc. You don't have to log out and then log back in as "root" to do this. Just use the **su** command to give yourself root powers (assuming you know root's password).

```
su -
```

Then enter the password when prompted. You now have root privileges (the system prompt will reflect this). Note that the "-" after **su** allows Linux to apply root's environment (including root’s path) to your **su** login. So you don't have to enter the full path of a command. Actually, **su** is a “switch user” command, and can allow you to become any user (if you know the password), not just root.

When you are finished using your **su** login, return to your own self by typing **exit**.

A word of caution: Be VERY judicious in your use of the root login. It can be destructive. For simple tasks that require root permission, use **su** and use it sparingly.

V. Editing with Vi

There are a number of terminal mode (non-GUI) editors available in Linux, including ***emacs*** and ***vi***. You could always use one of the available GUI text editors in Xwindow, but what if you are unable to start X? The benefit of learning ***vi*** or ***emacs*** is your ability to use them from an xterm, a character terminal, or a **telnet** (use **ssh** instead!) session, etc. We will discuss ***vi*** here. (I don't do ***emacs*** :-)). ***vi*** in particular is useful, because you will find it on all versions of Unix. Learn ***vi*** and you should be able to edit a file on any Unix system.

Using Vi

You can start ***vi*** either by simply typing **vi** at the command prompt, or you can specify the file you want to edit with **vi *filename***. If the file does not already exist, it will be created for you.

vi consists of two operating modes, *command* mode and *edit* mode. When you first enter ***vi*** you will be in command mode. Command mode allows you to search for text, move around the file, and issue commands for saving, save-as, and exiting the editor. Edit mode is where you actually input and change text.

In order to switch to edit mode, type either **a** (for append),**i** (for insert), or one of the other insert options listed on the next page. When you do this you will see "--Insert--" appear at the bottom of your screen. You can now input text. When you want to exit the edit mode and return to command mode, hit the escape key.

You can use the arrow keys to move around the file in command mode. In addition, there are a number of other navigation keys described below.

If you lose track of which mode you are in, hit the escape key twice. You should hear your computer beep and you will know that you are in command mode.

In current Linux distributions, *vi* is usually a link to ***vim*** (*vi improved*). This newer version of ***vi*** comes with a nice online tutorial. **It is worth your time**. Try typing **vimtutor** at a command prompt.

Vi command summary

Entering Edit Mode:

a	=	append text (after the cursor)
i	=	insert text (directly under the cursor)
o (the letter “oh”)	=	open a new line under the current line
O (capital “oh”)	=	open a new line above the current line

Command Mode:

0 (zero)	=	Move cursor to beginning of current line.
$	=	Move cursor to the end of current line.
x	=	delete the character under the cursor
X	=	delete the character before the cursor
dd	=	delete the entire line the cursor is on
:w	=	save and continue editing
:wq	=	save and quit (can use ZZ as well)
:q!	=	quit and discard changes
:w *filename*	=	save a copy to *filename* (save as)

The best way to save yourself from a messed up edit is to hit **<ESC>** followed by **:q!** That command will quit without saving changes.

Another useful feature that can be used in command mode is the string search. To search for a particular string in a file, make sure you are in command mode and type

/string

Where *string* is your search target. After issuing the command, you can move on to the next hit by typing "**n**".

vi is an extremely powerful editor. There are a huge number of commands and capabilities that are outside the scope of this guide. See **man vi** for more details. Keep in mind there are chapters in books devoted to this editor. There are even a couple of books devoted to ***vi*** alone.

VI. Mounting File Systems on Disks

There is a long list of file system types that can be accessed through Linux. You do this by using the **mount** command. Linux has a special directory used to **mount** file systems to the existing Linux directory tree. This directory is called */mnt*. It is here that you can dynamically attach new file systems from external (or internal) storage devices that were not mounted at boot time. Actually you can **mount** file systems anywhere (not just on */mnt*), but it's better for organization. Here is a brief overview.

Any time you specify a mount point you must first make sure that that directory exists. For example to mount a floppy under */mnt/floppy* you must be sure that */mnt/floppy* exists. After all, suppose we want to have a CDROM and a floppy mounted at the same time? They can't both be mounted under */mnt* (you would be trying to access 2 file systems through one directory!). So we create directories for each device’s file system under the parent directory */mnt*. You decide what you want to call the directories, but make them easy to remember. Keep in mind that until you learn to manipulate the file */etc/fstab* (covered later), only root can mount and unmount file systems.

```
mkdir /mnt/floppy
mkdir /mnt/cdrom
```

Newer distributions usually create these mount points for you, but you might want to add others for yourself (mount points for subject disks or images, etc. like */mnt/data* or */mnt/analysis*)

<u>The Mount Command</u>

The "**mount**" command uses the following syntax:

mount -t <*filesystem*> -o <*options*> <device> <*mountpoint*>

Example: Reading a DOS / Windows floppy

- Insert the floppy and type:

```
mount -t vfat /dev/fd0 /mnt/floppy
```

- Now change to the newly mounted file system:

```
cd /mnt/floppy
```

- You should now be able to navigate the floppy as usual.
- When you are finished, EXIT OUT of the */mnt/floppy* directory, and unmount the file system with:

```
umount /mnt/floppy
```

- Note the proper command is ***u*mount**, not ***un*mount**. This cleanly unmounts the disk. DO NOT remove the disk OR SWAP the disk until it is unmounted.
- If you get an error message that says the file system cannot be unmounted because it is busy, then you most likely have a file open from that directory, or are using that directory from another terminal. Check all you xterms and virtual terminals and make sure you are no longer in the mounted directory.

Example: Reading a CDROM

- Insert the CDROM and type:

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
```

- Now change to the newly mounted file system:

```
cd /mnt/cdrom
```

- You should now be able to navigate the CD as usual.
- When you are finished, EXIT OUT of the */mnt/cdrom* directory, and unmount the file system with:

```
umount /mnt/cdrom
```

If you want to see a list of file systems that are currently mounted, just use the **mount** command without any arguments or parameters. It will list the mount point and file system type of each device on system, along with the mount options used (if any). This is actually read from a file called

/proc/mounts, part of a virtual file system that keeps an up to date “snapshot” of the current system configuration. Try the following two commands:

```
mount
cat /proc/mounts
```

The ability to mount and unmount file systems is an important skill in Linux. There are a large number of options that can be used with **mount** (some we will cover later), and a number of ways the mounting can be done easily and automatically. Refer to the **mount** info or man pages for more information.

The file system table (/etc/fstab)

It might seem like "**mount -t iso9660 /dev/cdrom /mnt/cdrom**" is a lot to type every time you want to mount a CD or a disk. One way around this is to edit the file */etc/fstab*. This file allows you to provide defaults for your mountable devices, thereby shortening the commands required to mount them. My */etc/fstab* looks like this:

```
/dev/hda2   /            ext2      defaults                 1 1
/dev/hda5   /mnt/apps    vfat      user,noauto,defaults     0 0
/dev/hda6   /mnt/data    vfat      user,noauto,defaults     0 0
/dev/hda3   swap         swap      defaults                 0 0
/dev/fd0    /mnt/floppy  vfat      user,noauto              0 0
/dev/hdc    /mnt/cdrom   iso9660   user,noauto,ro           0 0
/dev/sda4   /mnt/zip     vfat      user,noauto,defaults     0 0
none        /proc        proc      defaults                 0 0
```

The columns are:
<device>   <mount point>   <filesystem>   <default options>

With this */etc/fstab*, I can mount a floppy or CD by simply typing:

```
mount /mnt/floppy
mount /mnt/cdrom
```

The above **mount** commands look incomplete. When not enough information is given, the **mount** command will look to */etc/fstab* to fill in the blanks. If it finds the required info, it will go ahead with the mount.

Note the "user" entry in the options column for some devices. This allows non-root users to mount the devices. Very useful. To find out more about available options for */etc/fstab*, enter **info fstab** at the command prompt.

Also keep in mind that default Linux installations will often create */mnt/floppy* and */mnt/cdrom* for you already. After installing a new Linux system, have a look at */etc/fstab* to see what is available for you. If what you need isn't there, add it.

VII. Linux and Forensics

Included Forensic Tools

Linux comes with a number of simple utilities that make imaging and basic analysis of suspect disks and drives comparitively easy. These tools include:

- **dd -**command used to copy from an input file or device to an output file or device. Simple bitstream imaging.
- **sfdisk and fdisk -**used to determine the disk structure.
- **grep -**search files (or multiple files) for instances of an expression or pattern.
- **The loop device -**allows you to mount an image without having to rewrite the image to a disk.
- **md5sum and sha1sum -**create and store an MD5 or SHA hash of a file or list of files (including devices).
- **file -**reads a file's header information in an attempt to ascertain its type, regardless of name or extension.
- **xxd -** command line hexdump tool. For viewing a file in hex mode.
- **ghex and khexedit -**the Gnome and KDE (X Window interfaces) hex editors. Both have primitive search and byte selection capabilities.

Following is a *very* simple series of steps to allow you to perform an easy practice analysis using the simple Linux tools mentioned above. All of the commands can be further explored with "**man *command***". For simplicity we are going to use a floppy from a DOS machine. Again, this is just an introduction to the basic commands. These steps can be far more powerful with some command line tweaking.

Analysis organization

Having already said that this is just an introduction, most of the work you will do here can be applied to actual casework. The tools are standard Linux tools, and although the example shown here is *very* simple, it can be extended with some practice and a little (ok, a lot) of reading. The practice floppy (in raw image format from a simple **dd**) for the following exercise is available at:

ftp://ftp.hq.nasa.gov/pub/ig/ccd/linuxintro/practical.floppy.dd

Once you download the floppy image, put a floppy disk in your drive and create the practice floppy with the following command (covered in detail later):

```
dd if=practical.floppy.dd of=/dev/fd0
```

The output of various commands and the amount of searching we will do here is limited by the scope of this example and the amount of data on a floppy. When you actually do an analysis on larger media, you will want to have it organized. Note that when you issue a command that results in an output file, that file will end up in your current directory, unless you specify a path for it in the command.

One way of organizing your data would be to create a directory in your "home" directory for evidence and then a subdirectory for different cases. Since we will be executing these commands as root, the home directory is */root*

```
mkdir ~/evidence
```

The tilde (~) in front of the directory name is shorthand for "home directory", so when I type **~/evidence**, Linux interprets it **$HOME/evidence**. If I am logged in as root, the directory will be created as **/root/evidence**. Note that if you are already in your home directory, then you don't need to type ~/. Simply using **mkdir evidence** will work just fine. We are being explicit for instructional purposes. Directing all of our analysis output to this directory will keep our output files separated from everything else and maintain case organization.

For the purposes of this exercise, we will be logged in as “root”. I have mentioned already that this is generally a bad idea, and that you can make a mess of your system if you are not careful. Many of the commands we are utilizing here require root access (permissions on devices that you might want to access **should not** be changed to allow otherwise, and doing so would be far more complex than you think). So the output files that we create and the images we make will be found under **/root/evidence/**

An additional step you might want to take is to create a special mount point for all physical subject disk analysis (not that we normally mount subject disks…). This is another way of separating common system use with evidence processing.

```
mkdir /mnt/analysis
```

Determining the structure of the disk

There are two simple tools available for determining the structure of a disk attached to your system. The first, **fdisk**, we discussed eariler using the **-l** option. Replace the “x” with the letter of the drive that corresponds to the subject drive.

```
fdisk –l /dev/hdx
```

```
Disk /dev/hda: 255 heads, 63 sectors, 1582 cylinders
Units = cylinders of 16065 * 512 bytes
  Device Boot  Start    End   Blocks    Id   System
/dev/hda1         1     255   2048256   b    Win95 FAT32
/dev/hda2   *   256     638   3076447+  83   Linux
/dev/hda3       639     649     88357+  82   Linux swap
/dev/hda4       650    1582   7494322+  f    Win95 Ext'd (LBA)
/dev/hda5       650    1453   6458098+  b    Win95 FAT32
/dev/hda6      1454    1582   1036161   b    Win95 FAT
```

We can redirect the output of this command to a file for later use by issuing the command as:

```
fdisk –l /dev/hdx > fdisk.disk1
```

A couple of things to note here: The name of the output file (*fdisk.disk1*) is completely arbitrary. There are no rules for extensions. Name the file anything you want. I would suggest you stick to a convention and make it descriptive. Also note that since we did not define an explicit path for the file name, *fdisk.disk1* will be created in our current directory (for instance, */root/evidence/*).

Also note that you can expect to see strange output if you use **fdisk** on a floppy disk. Be aware of that if you attempt **fdisk** on the practice floppy. Try it on your harddrive instead to see sample output. Don't use **fdisk** on the practice floppy. The output won't make sense.

Creating a forensic image of the suspect disk

Make an image of the practice disk. This is your standard forensic image of a suspect disk. Execute the command from within the */root/evidence/* directory:

```
dd if=/dev/fd0 of=image.disk1 bs=512
```

This takes your floppy device *(/dev/fd0*) as the input file (*if*) and writes the output file (*of*) called *image.disk1* in the current directory (*/root/evidence/*). The **bs** option specifies the block size. This is really not needed for most block devices (hard drives, etc.) as the Linux kernel handles the actual block size. It's added here for illustration

For the sake of safety and practice, change the read-write permissions of your image to read-only.

```
chmod 444 image.disk1
```

The **444** gives all users read-only access. If you are real picky, you could use **400.** Note that the owner of the file is the user that created it.

Now that you have created an image file, you can restore the image to another disk for analysis and viewing. Put another (blank) floppy in and type:

```
dd if=image.disk1 of=/dev/fd0 bs=512
```

This is the same as the first **dd** command, only in reverse. Now you are taking your image (the input file "*if*") and writing it to another disk (the output file "*of*") to be used as a backup or as a working copy for the actual analysis.

Note that using **dd** creates an exact duplicate of the physical device. This includes all the file slack and unallocated space. We are not simply copying the logical file structure. Unlike many other forensic imaging tools, **dd** does not fill the image with any proprietary data or information. It is a simple bit stream copy from start to end. This (in my ever-so-humble opinion) has a number of advantages, as we will see later.

Mounting a restored image

Mount the restored (cloned) working copy and view the contents. Remember, we are assuming this is a DOS formatted disk from a Win 98/95 machine.

```
mount -t vfat -o ro,noexec /dev/fd0 /mnt/analysis
```

This will mount your working copy (the new floppy you created from the forensic image) on "*/mnt/analysis*". The "**–o ro,noexec**" specifies the options **ro** (read-only) and **noexec** (prevents the execution of binaries from the mount point) in order to protect the disk from you, and your system (and mountpoint) from the contents of the disk. There are other useful mount options as well, such as **noatime**. See the man page for more details.

Now **cd** to the mount point (*/mnt/analysis*) and browse the contents. Be sure to unmount the disk when you finish.

```
umount /mnt/analysis
```

Mounting the image using the loopback device

Another way to view the contents of the image without having to restore it to another disk is to mount using the *loop* interface. Basically, this allows you to "mount" a file system within an image file (instead of a disk) to a mount point and browse the contents. Your Linux kernel must have *loop* either compiled as a module or compiled into the kernel for this to work.

We use the same mount command and the same options, but this time we include the option "**loop**" to indicate that we want to use the *loop* device to mount the file system within the image file, and we specify a disk (partition) image rather than a disk device. Change to the directory where you created the image and type:

```
mount -t vfat -o ro,noexec,loop image.disk1 /mnt/analysis
```

Now you can change to */mnt/analysis* and browse the image as if it were a mounted disk! Use the **mount** command by itself to double check the mounted options. When you are finished browsing, unmount the image file.

```
umount /mnt/analysis
```

File Hash

One important step in any analysis is verifying the integrity of your data both before after the analysis is complete. You can get a hash (CRC, MD5, or SHA) of each file in a number of different ways. We will use the SHA hash. SHA is a hash signature generator that supplies a 160-bit "fingerprint" of a file or disk. It is not feasible for someone to computationally recreate a file based on the SHA hash. This means that matching SHA signatures mean identical files.

We can get an SHA sum of a disk by changing to our evidence directory (i.e. */root/evidence*) and doing (note that the following commands can be replaced with **md5sum** if you prefer to use MD5 sums):

```
sha1sum /dev/fd0
```

or

```
sha1sum /dev/fd0 > SHA.disk1
```

The redirection in the second command allows us to store the signature in a file and use it for verification later on. To get a hash of a raw disk (*/dev/hda, /dev/fd0*, etc.) the disk does NOT have to be mounted. We are hashing the device (the disk) not the file system. As we discussed earlier, Linux treats all objects, including physical disks, as *files*. So whether you are hashing a file of a hard drive, the command is the same.

We can get a hash of each file on the disk using the **find** command and an option that allows us to execute a command on each file found. We can get a very useful list of SHA hashes for every file on a disk (once it is mounted, as in the loop mount command on the previous page) by changing to the */mnt/analysis* directory:

```
mount -t vfat -o ro,noexec,loop image.disk1 /mnt/analysis

cd /mnt/analysis
```

and issuing the command:

```
find . -type f -exec sha1sum {} \;  > /root/evidence/SHA.filelist
```

This command says "**find**, starting in the *current* directory (signified by the "**.**"), any regular file (**-type f**) and execute (**-exec**) the command **sha1sum** on all files found (**{}**). Redirect the output to *SHA.filelist* in the /root/evidence directory (where we are storing all of our evidence files). The "**\;**" is an escape sequence that ends the **–exec** command.

You can also use Linux to do your verification for you. To verify that nothing has been changed on the original floppy, you can use the -c option with **sha1sum**. If the disk was not altered, the command will return "ok". Make sure the floppy is in the drive and type:

```
sha1sum -c /root/evidence/SHA.disk1
```

If the SHA hashes match from the floppy and the original SHA output file, then the command will return "OK" for */dev/fd0*. The same can be done with the list of file SHAs. Mount the floppy on */mnt/analysis,* change to that directory and issue the command:

```
sha1sum -c /root/evidence/SHA.filelist
```

Again, the SHA hashes in the file will be compared with SHA sums taken from the floppy (at the mount point). If anything has changed, the program will give a "failed" message. Unchanged files will be marked "OK".

The analysis

You can now view the contents of the read-only mounted or restored disk or loop-mounted image. If you are running the X window system, then you can use your favorite file browser to look through the disk. In most (if not all) cases, you will find the command line more useful and powerful in order to allow file redirection and permanent record of your analysis. We will use the command line here.

We are also assuming that you are issuing the following commands from the proper mount point (*/mnt/analysis/*). If you want to save a copy of each command's output, be sure to direct the output file to your evidence directory (*/root/evidence/)*

Navigate through the directories and see what you can find. Use the **ls** command to view the contents of the disk. The command in the following form might be useful:

```
ls –al
```

This will show all the hidden files (**-a),** give the list in long format to identify permission, date, etc. (**-l**). You can also use the **–R** option to list recursively through directories. You might want to pipe that through **less**.

```
ls –alR  | less
```

<u>Making a list of all files</u>

Get creative. Take the above command and redirect the output to your evidence directory. With that you will have a list of all the files and their owners and permissions on the suspect disk. This is a very important command. Check the **man** page for various uses and options. For example, you could use the **–i** option to include the inode in the list, the **–u** option can be used so that the output will include and sort by access time (when used with the **–t** option).

```
ls –laiRtu > /root/evidence/file.list
```

You could also get a list of the files, one per line, using the find command and redirecting the output to another list file:

```
find . -type f -print > /root/evidence/filelist.list.2
```

Have a look at the above commands, and compare their output. Which do you like better? Remember the syntax assumes you are issuing the command from the */mnt/analysis* directory (use **pwd** if you don't know where you are).

Now use the **grep** command on either of the file lists for whatever strings or extensions you want to look for.

```
grep -i jpg filelist.list
```

This command looks for the pattern “jpg” in the list of files, using the filename extension to alert us to a JPEG file.

Making a list of file types

What if you are looking for JPEG’s but the name of the file has been changed, or the extension is wrong? You can also run the command **file** on each file and see what it might contain.

```
file filename
```

The **file** command compares each file’s header (the first few bytes of a raw file) with the contents of the “magic” file (usually found in */usr/share/magic*). It then outputs a description of the file.

If there are a large number of files without extensions, or where the extensions have changed, you might want to run the **file** command on *all* the files on a disk (or in a directory, etc.). Remember our use of the **find** command’s **-exec** option with **sha1sum**? Let’s do the same thing with **file**:

```
find . -type f -exec file {} \; > /root/evidence/filetype.list
```

View the list with the more command, and if you are looking for images in particular, then use grep to specify that:

```
cat /root/evidence/filetype.list | grep image
```

This command would stream the contents of our *filetype.list* file using the **cat** command and pipe the output through **grep**, looking for instances of the string “image”.

Viewing files

For text files and data files, you might want to use **cat**, **more** or **less** to view the contents.

```
cat filename
more filename
```

```
less filename
```

Be aware that if the output is not standard text, then you might corrupt the terminal output (type "*reset*" at the prompt and it should clear up). It is best to run these commands in a terminal window in X so that you can simply close out a corrupted terminal and start another. Using the **file** command will give you a good idea of which files will be viewable.

Perhaps a better alternative for viewing unknown files would be to use the **strings** command. This command can be used to parse regular ASCII text out of any file. It's good for formatted documents (MS Word or Star Office), data files (Excel, etc.) and even binaries (i.e. unidentified executables) which might have interesting text strings hidden in them. It might be best to pipe the output through **less.**

```
strings filename | less
```

Have a look at the contents of the practice disk on */mnt/analysis*. There is a file called *arp.exe.* What does this file do? We can't execute it, and from using the **file** command we know that it's an i386 executable. Run the following command (again, assuming you are in the */mnt/analysis* directory) and scroll through the output. Do you find anything of interest (hint: like a usage message)?

```
strings arp.exe | less
```

If you are currently running the X window system, you can use any of the graphics tools that come standard with whichever Linux distribution you are using. **gqview** is one graphics tool for the **Gnome** desktop that will display thumbnails of all the recognized graphic files in a directory. Experiment a little. **konqueror** from the **KDE** desktop has a feature that will create a very nice html image gallery for you from all images in a directory. There are *g-scripts* that will do the same for the **Nautilus** file manager under **Gnome**.

Once you are finished exploring, be sure to unmount the floppy (or loop mounted disk image). Again, make sure you are not anywhere in the mount point when you try to unmount, or you will get the "busy" error.

```
umount /mnt/analysis
```

Searching unallocated and slack space for text

Now let's go back to the original image. The restored disk (or loop mounted disk image) allowed you to check all the files and directories (logical view). What about unallocated and slack space (physical view)? We will now analyze the image itself, since it was a byte for byte copy and includes data in the unallocated areas of the disk, as well as file slack space.

Let's assume that we have seized this disk from a former employee of a large corporation. The would-be cracker sent a letter to the corporation threatening to unleash a virus in their network. The suspect denies sending the letter. This is a simple matter of finding the text from a deleted file (unallocated space).

First, change back to the directory in which you created the image, whether it was the root's home directory, or a special one you created.

```
cd /root/evidence
```

Now we will use the **grep** command to search the image for any instance of an expression or pattern. We will use a number of options to make the output of **grep** more useful. The syntax of **grep** is normally:

```
grep –options <pattern> <search_range>
```

The first thing we will do is create a list of keywords to search for. It's rare we ever want to search evidence for a single keyword, after all. For our example, lets use "ransom", "$50,000" (the ransom amount), and "unleash a virus". These are some keywords and a phrase that we have decided to use from the original letter received by the corporation. Make the list of keywords (using **vi**) and save it as */root/evidence/searchlist.txt*. Ensure that each string you want to search for is on a different line.

```
$50,000
ransom
unleash a virus
```

Make sure there are NO BLANK LINES IN THE LIST OR AT THE END OF THE LIST!! Now we run the **grep** command on our image:

```
grep –aibf  searchlist.txt image.disk1 > hits.txt
```

Looking at the **grep** command we see that we are asking **grep** to use the list we created in "*searchlist.txt*" for the patterns we are looking for. This is specified with the "**-f *listfile***" option. We are telling **grep** to search *image.disk1* for these patterns, and we are redirecting the output to a file called *hits.txt*, so we can record the output and view them at our leisure. The **–a** option tells **grep** to process the file as if it were text, even if it's binary. The option **-i** tells **grep** to ignore upper and lower case. And the **-b** option tells **grep** to give us the byte offset of each hit so we can find the line in **xxd** or one of the graphical hex editors, like GHex.

Once you run the command above, you should have a new file in your current directory called *hits.txt.* View this file with **less** or **more** or any text viewer. Keep in mind that **strings** might be best for the job. Again, if you use **more** or **less**, you run the risk of corrupting your terminal if there are non-ASCII characters. We will simply use **cat** to stream the entire contents of the file to the standard output. The file *hits.txt* should give you a list of lines that contain the words in your *searchlist.txt* file. In front of each line is a number that represents the byte offset for that "hit" in the image file.

```
cat hits.txt
```

```
75441:you and your entire business ransom.
75500:I have had enough of your mindless corporate piracy and will no
longer stand for it. (...)
75767:Don't try anything, and dont contact the cops.  If you do, I will
unleash a virus that will bring down your whole network and destroy your
consumer's confidence.
```

In keeping with our command line philosophy, we will use **xxd** to display the data found at each byte offset. **xxd** is a command line hexdump tool, useful for examining files. Do this for each offset in the list of hits. This should yield some interesting results.

```
xxd -s offset image.disk1 | less
```

If you want to cheat a little, and use a GUI, try **GHex**. Find it on the **KDE** or **Gnome** menus, or simply type **ghex &** in a terminal window. It is a standard hex editor. Open the image file, and click on <Edit> and then <Goto Byte>. Type in the byte offset given in your hits.txt file and it should take you to that byte in the hex screen. The ASCII equivalent is displayed on the right.

VIII. Common Forensic Issues

Handling large disks

The example used in this text utilizes a file system on a floppy disk. What happens when you are dealing with larger hard disks? When you create an image of a disk drive with the **dd** command there are a number of components to the image. These components can include a boot sector, partition table, and the various partitions (if defined).

When you attempt to mount a larger image with the loop device, you find that the **mount** command is unable to find the file system on the disk. This is because **mount** does not know how to "recognize" the partition table. The easy way around this (although it is not very efficient for large disks) would be to create separate images for each disk partition that you want to analyze. For a simple hard drive with a single large partition, you would create two images.

Assuming your suspect disk is attached as the master device on the secondary IDE channel:

```
dd if=/dev/hdc of=image.disk  bs=4096      (gets the entire disk)
dd if=/dev/hdc1 of=image.part1 bs=4096   (gets the first partition)
```

The first command gets you a full image of the entire disk for backup purposes, including the boot record and partition table. The second command gets you the partition. The resulting image from the second command can be mounted via the loop device.

Note that although both of the above disks will contain the same file system with the same data, the sha1sums *will obviously not match.*

One method for handling larger disks (mounting the image with the loop device) is to send the **mount** command a message to skip trying to mount the first 63 sectors of the image. These sectors are used to contain information (like the MBR) that is not part of a normal data partition. We know that each sector is 512 bytes, and that there are 63 of them. This gives us an offset of 32256 bytes from the start of our image to the first partition we want to mount. This is then passed to the **mount** command as an option:

```
mount –t vfat –o ro,noexec,loop,offset=32256 image.disk /mnt/analysis
```

This effectively “jumps over” the first 63 sectors of the image and goes straight to the “boot sector” of the first partition, allowing the **mount** command to work properly.

You could also use NASA’s enhanced loopback driver, which we will discuss a little later.

When you are dealing with larger disks (over 2GB), you must also concern yourself with the size of your image files. If your Linux distribution relies on the 2.2.x kernel then you will encounter a file size limit of 2GB (on x86 systems). The Linux 2.4.x kernel solves this problem. You can either compile the 2.4.x kernel on your current system, or use a distribution that includes the 2.4.x kernel in its default installation. Just about any distribution from anytime this century (!) will have the 2.4 kernel.

Now that we know about the issues surrounding creating large images from whole disks, what do we do if we run into an error? Suppose you are creating a disk image with **dd** and the command exits halfway through the process with a read error? We can instruct **dd** to *attempt* to read past the errors using the **conv=noerror** option. In basic terms, this is telling the **dd** command to ignore the errors that it finds, and attempt to read past them. When we specify the **noerror** option it is a good idea to include the **sync** option along with it. This will “pad” the **dd** output wherever errors are found and ensure that the output will be “synchronized” with the original disk. This may allow file system access and file recovery where errors are not fatal. The command will look something like:

```
dd if=/dev/hdx of=image.disk1 conv=noerror,sync
```

In addition to the structure of the images and the issues of image sizes, we also have to be concerned with memory usage and our tools. You might find that **grep**, when used as illustrated in our floppy analysis example, might not work as expected with larger images and could exit with an error similar to:

```
grep: memory exhausted
```

The most apparent cause for this is that **grep** does its searches line by line. When you are “grepping” a large disk image, you might find that you have a huge number of bytes to read through before **grep** comes across a newline character. What if **grep** had to read 200MB of data before coming across a newline? It would "exhaust" itself (the input buffer fills up).

What if we could force-feed **grep** some newlines? In our example analysis we are "grepping" for text. We are not concerned with non-text characters at all. If we could take the input stream to **grep** and change the non-text characters to newlines, grep would have no problem. Note that changing the input stream to **grep** does *not* change the image itself. Also, remember that we are still looking for a byte offset. Luckily, the character sizes remain the same, and so the offset does not change as we feed newlines into the stream (simply replacing one "character" with another).

Let's say we want to take all of the control characters streaming into **grep** from the disk image and change them to newlines. We can use the *translate* command, **tr**, to accomplish this. Check out **man tr** for more information about this powerful command:

```
tr '[:cntrl:]' '\n' < image.disk1 | grep -abif searchlist.txt > hits.txt
```

This command would read: "Translate all the characters contained in the set of *control characters ([:cntrl:])* to *newlines (\n)*. Take the input to **tr** from *image.disk1* and pipe the output to **grep,** sending the results to *hits.txt*. This effectivley changes the stream before it gets to **grep.**

This is only one of many possible problems you could come across. My point here is that when issues such as these arise, you need to be familiar enough with the tools Linux provides to be able to understand ***why*** such errors might have been produced, and how you can get around them. Remember, the shell tools and the GNU software that accompany a Linux distribution are extrememly powerful, and are capable of tackling nearly any task. Where the standard shell fails, you might look at *perl* or *python* as options. These subjects are outside of the scope of the current presentation, but are introduced as fodder for further experimentation.

Preparing a disk for the suspect image

One common practice in forensic disk analysis is to "wipe" a disk prior to restoring a forensic image to it. This ensures that any data found on the restored disk is *from* the image and not from "residual" data. That is, data left behind from a previous case or image.

We can use a special device as a source of zeros. This can be used to create empty files and wipe portions of disks. You can write zeros to an entire disk using the following command:

```
dd if=/dev/zero of=/dev/hdx bs=4096
```

This starts at the beginning of the drive and writes zeros in every sector in 4096 byte chunks. Specifying larger block sizes can speed the writing process. Experiment with different block sizes and see what effect it has on the writing speed (i.e. 32k, 64k, etc.). I've wiped 60GB disks in under an hour on a fast IDE controller with the proper drive parameters. Specific drive parameters can be set using the **hdparm** command. Check **hdparm's** man page for available options**.** For instance, setting *dma* on a drive can dramatically speed things up.

So how do we verify that the write (of zero's) was a success? You could check random sectors with a hex editor, but that's not realistic for a large drive. One of the best methods would be to use the **xxd** command (command line hexdump) with the "autoskip" option (works if a drive is wiped with *0x00*). The output of this command on a zero'd drive would give just three lines. The first line, starting at offset zero with a row of zeros in the data area, followed by an asterisk (*) to indicate identical lines, and finally the last line, with the final offset followed by the remaining zeros in the data area. Here's and example of the command on a zero'd drive (floppy) and its output.

```
xxd -a /dev/fd0
0000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
0167ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

IX. Advanced (Beginner) Forensics

The following sections are more advanced and detailed. New tools are introduced to help round out some of your knowledge and provide a more solid footing on the capabilities of the Linux command line. The topics are still at the beginner level, but you should be at least somewhat comfortable with the command line before tackling the exercises. Although I've included the commands and much of the output for those who are reading this without the benefit of a Linux box nearby, it is important that you follow along on your own system as we go through the practical exercises. Typing at the keyboard and experimentation is the only way to learn.

The Command Line on Steroids

Let's dig a little deeper into the command line. Often there are arguments made about the usefulness of the command line interface (CLI) versus a GUI tool for analysis. I would argue that in the case of large sets of regimented data, the CLI can sometimes be faster and more flexible than many GUI tools available today.

As an example, we will look at a set of log files from a single Unix system. We are not going to analyze them for any sort of smoking gun. The point here is to illustrate the ability of the CLI to organize and parse through data by using pipes to string a series of commands together, obtaining the desired output. Follow along with the example, and keep in mind that to get anywhere near proficient with this will require a great deal of reading and practice. The payoff is enormous.

Create a directory called "logs" and download the file *logs.tar.gz* into that directory:

ftp://ftp.hq.nasa.gov/pub/ig/ccd/linuxintro/logs.tar.gz

As always, have a look at the contents of the archive before haphazardly writing the contents to your drive:

```
tar tzvf logs.tar.gz
-rw------- root/root     8296 2003-10-29 16:14:49 messages
-rw------- root/root     8302 2003-10-29 16:17:38 messages.1
-rw------- root/root     8293 2003-10-29 16:19:32 messages.2
-rw------- root/root     4694 2003-10-29 16:23:18 messages.3
-rw------- root/root     1215 2003-10-29 16:23:33 messages.4
```

The archive contains 5 log files from a Unix system. The *messages* logs contain entries from a variety of sources, including the kernel and other applications. The numbered files result from log rotation. As the logs are filled, they are rotated and eventually deleted. On most Unix systems, the logs are found in */var/log/* or */var/adm*.

Untar the file:

```
tar xzvf logs.tar.gz
```

Let's have a look at one log entry:

```
cat messages | head -1
Nov 17 04:02:14 localhost123 syslogd 1.4.1: restart.
```

Each line in the log files begin with a date and time stamp. Next comes the hostname followed by the name of the application that generated the log message. Finally, the actual message is printed.

Let's assume these logs are from a victim system, and we want to analyze them and parse out the useful information. We are not going to worry about what we are actually seeing here, our object is to understand how to boil the information down to something useful.

First of all, rather than parsing each file individually, let's try and analyze all the logs at one time. They are all in the same format, and essentially they comprise one large log. We can use the **cat** command to add all the files together and send them to standard output. If we work on that data stream, then we are essentially making one large log out of all five logs. Can you see a potential problem with this?

```
cat messages* | less
```

If you look at the output, you will see that the dates ascend and then jump to an earlier date and then start to ascend again. This is because the later log entries are added to the *bottom* of each file, so as the files are added together, the dates appear to be out of order. What we really want to do is stream each file *backwards* so that they get added together with the most recent date in each file *at the top* instead of at the bottom. In this way, when the files are added together they are in order. In order to accomplish this, we use **tac** (yes, that's **cat** backwards).

```
tac messages* | less
```

Beautiful. The dates are now in order. We can now work on the stream of log entries as if they were one large (in order) file.

We will introduce a new command, **awk,** to help us view specific fields from the log entries. In this case, the dates. **awk** is an extremely powerful command. The version most often found on Linux systems is **gawk** (GNU **awk**). While we are going to use it as a stand alone command, **awk** is actually a programming language on its own, and can be used to write scripts for organizing data. Our concentration will be centered on **awk's** "print" function. See **man awk** for more details.

Sets of repetitive data can often be divided into columns or "fields", depending on the structure of the file. In this case, the fields in the log files are separated by simple white space (**awk's** default field separator). The date is comprised of the first two fields (month and day).

```
tac messages * | awk '{print $1 " " $2}' | less
```

```
Feb 8
Feb 8
Feb 8
…
```

This command will stream all the log files (each one from bottom to top) and send the output to **awk** which will print the first field (month - $1), followed by a space (" "), followed by the second field (day - $2). This shows the month and day for every entry. Suppose I just want to see one of each date when an entry was made. I don't need to see repeating dates. I ask to see one of each unique line of output with **uniq**:

```
tac messages* | awk ‘{print $1 “ “ $2}’ | uniq | less
```

```
Feb 8
Nov 22
Nov 21
Nov 20
…
```

This removes repeated dates, and shows me just those dates with log activity. If a particular date is of interest, I can **grep** the logs for that particular date (note there are ***2*** spaces between “Nov” and “4”, one space will not work):

```
tac messages* | grep “Nov  4”
```

```
Nov  4 17:41:27 localhost123 sshd(pam_unix)[27630]: session closed for
user root
Nov  4 17:41:27 localhost123 sshd[27630]: Received disconnect from
1xx.183.221.214: 11: Disconnect requested by Windows SSH Client.
…
```

Of course, we have to keep in mind that this would give us any lines where the string “Nov 4” resided, not just in the date field. To be more explicit, we could say that we only want lines that *start* with “Nov 4”, using the “^”:

```
tac messages* | grep “^Nov  4”
```

Also, if we don’t *know* that there are *two* spaces between “Nov” and “4”, we can tell **grep** to look for any number of spaces between the two:

```
tac messages* | grep “^Nov[ ]*4”
```

The above **grep** expression translates to “Lines starting (^) with the string “Nov” followed by zero or more (*) of the preceding characters ([/space/]) followed by a 4”. Obviously, this is a complex issue. Knowing how to use regular expression will give you huge flexibility in sorting through and organizing large sets of data.

As we look through the log files, we may come across entries that appear suspect. Perhaps we need to gather all the entries that we see containing the string “Did not receive identification string from <*IP*>” for further analysis.

```
tac messages* | grep “identification string” | less
```

```
Nov 17 19:26:43 localhost123 sshd[2019]: Did not receive identification
string from 200.92.72.129
Nov 18 18:55:06 localhost123 sshd[11204]: Did not receive identification
string from 62.66.248.243
…
```

Now we just want the date (fields 1 and 2), the time (field 3) and the remote IP address that generated the log entry. The IP address is the last field. Rather than count each word in the entry to get to the field number of the IP, we can simply use the variable “$NF”, which means “number of fields”. Since the IP is the last field, its field number is equal to the number of fields:

```
tac messages* | grep “identification string” |
        awk ‘{print $1” “$2” “$3” “$NF}’ | less
```

```
Nov 17 19:26:43 200.92.72.129
Nov 18 18:55:06 62.66.248.243
Nov 20 14:13:11 200.83.114.131
…
```

We can add some tabs (**“\t”**) in place of spaces to make it more readable:

```
tac messages* | grep “identification string” |
        awk ‘{print $1” “$2”\t“$3”\t“$NF}’ | less
```

```
Nov 17  19:26:43       200.92.72.129
Nov 18  18:55:06       62.66.248.243
Nov 20  14:13:11       200.83.114.131
…
```

This can all be redirected to an analysis log or text file for easy addition to a report (note that “**> report.txt**” *creates* the report file, “**>> report.txt**” *appends* to it):

```
echo “Localhost123:  Log entries from /var/log/messages” > report.txt
```

```
echo “\”Did not receive identification string\”:” >> report.txt
```

```
tac messages* | grep “identification string” |
              awk ‘{print $1” “$2”\t“$3”\t“$NF}’ >> report.txt
```

We can also get a sorted (**sort**) list of the unique (**-u**) IP addresses involved in the same way:

```
echo “Unique IP addresses:” >> report.txt
```

```
tac messages* | grep “identification string” |
              awk ‘{print $NF}’ |  sort -u >> report.txt
```

```
less report.txt
```

The resulting list of IP addresses can also be fed to a script that does **nslookup** or **whois** database queries.

As with all the exercises in this document, we have just sampled the abilities of the Linux command line. It all seems somewhat convoluted to the beginner. After some practice and experience with different sets of data, you will find that you can glance at a file and say “I want that information”, and be able to write a quick piped command to get what you want in a readable format *in a matter of seconds*. As with all language skills, the Linux command line “language” is perishable. Keep a good reference handy and remember that you might have to look up syntax a few times before it becomes second nature.

Fun with DD

We've already done some simple imaging and wiping using **dd**, let's explore some other uses for this flexible tool. **dd** is sort of like a little forensic Swiss army knife (talk about over-used clichés!). It has lots of applications, limited only by your imagination.

Splitting files and images

One function we might find useful would be the ability to split images up into usable chunks, either for archiving or for use in another program. We will first discuss using **split** on its own, then in conjunction with **dd** for "on the fly" splitting.

For example, you might have a 10GB image that you want to split into 640MB parts so they can be written to CD-R media. Or, if you use a program such as *Ilook Investigator* and need files no larger than 2GB (for a fat32 partition), you might want to split the image into 2GB pieces. For this we use the **split** command.

split normally works on lines of input (i.e. from a text file). But if we use the **–b** option, we force split to treat the file as *binary* input and lines are ignored. We can specify the size of the files we want along with the prefix we want for the output files. The command looks like:

```
split –b XXm <file to be split> <prefix of output files>
```

where **XX** is the size of the resulting files. For example, if we have a 6GB image called *image.disk1.dd*, we can split it into 2GB files using the following command:

```
split –b 2000m image.disk1.dd image.split.
```

This would result in 3 files (2GB in size) each named with the prefix "image.split." as specified in the command, followed by "aa", "ab", "ac", and so on:

```
image.split.aa
image.split.ab
image.split.ac
```

The process can be reversed. If we want to reassemble the image from the split parts (from CD-R, etc.), we can use the **cat** command and redirect the output to a new file. Remember **cat** simply streams the specified files to standard output. If you redirect this output, the files are assembled into one.

```
cat image.split.aa image.split.ab image.split.ac > image.new
```

or

```
cat image.split.a* > image.new
```

Another way of accomplishing this would be to split the image as we create it (i.e. from a **dd** command). This is essentially the “on the fly” splitting we mentioned earlier. We do this by piping the output of the **dd** command straight to **split**.

```
dd if=/dev/hdx | split –b 2000m – image.split.
```

In this case, instead of giving the name of the file to be split in the **split** command, we give a simple “**-**“. The single dash is a descriptor that means “standard input”. In other words, the command is taking its input from the data pipe provided by the standard output of **dd**.

Once we have the image, the same technique using **cat** will allow us to reassemble it for hashing or analysis.

For practice, let’s take the practical exercise floppy disk we used earlier and try this method on that disk, splitting it into 360k pieces:

```
sha1sum /dev/fd0
        f5ee9cf56f23e5f5773e2a4854360404a62015cf  /dev/fd0
```

```
dd if=/dev/fd0 | split –b 360k – floppy.split.
        2880+0 records in
        2880+0 records out
```

- remember, the “records” are 512 byte blocks (times 2880 = 1.44Mb)

```
ls –lh
        -rw-r--r--   1 root   root      360k Aug 14 08:13 floppy.split.aa
        -rw-r--r--   1 root   root      360k Aug 14 08:13 floppy.split.ab
        -rw-r--r--   1 root   root      360k Aug 14 08:13 floppy.split.ac
        -rw-r--r--   1 root   root      360k Aug 14 08:13 floppy.split.ad
```

```
cat floppy.split.a* | sha1sum
        f5ee9cf56f23e5f5773e2a4854360404a62015cf  -
```

(The out put of this command shows a "-" in place of the filename. This represents the fact that the hash was calculated from "standard input" to **sha1sum**, not a file or device)

```
cat floppy.split.a* > new.floppy.image
ls -lh
        -rw-r--r--   1 root   root    360k Aug 14 08:13 floppy.split.aa
        -rw-r--r--   1 root   root    360k Aug 14 08:13 floppy.split.ab
        -rw-r--r--   1 root   root    360k Aug 14 08:13 floppy.split.ac
        -rw-r--r--   1 root   root    360k Aug 14 08:13 floppy.split.ad
        -rw-r--r--   1 root   root   1.4M Aug 14 08:14  new.floppy.image
```

```
sha1sum new.floppy.image
        f5ee9cf56f23e5f5773e2a4854360404a62015cf  new.floppy.image
```

Looking at the output of the above commands, we see that all the sha1sum's match. We find the same hash for the disk, for the split images "cat-ed" together, and for the newly reassembled image.

Data Carving with dd

In this next example, we will use **dd** to carve a JPEG image from a chunk of raw data. By itself, this is not a real useful exercise. There are lots of tools out there that will "carve" files from forensic images, including a simple cut and paste from a hex editor. However, the purpose of this exercise is to help you become more familiar with **dd**. In addition, you will get a chance to use a number of other tools in preparation for the "carving". This will help familiarize you further with the Linux toolbox. First you will need to download the raw data chunk from:

ftp://ftp.hq.nasa.gov/pub/ig/ccd/linuxintro/image_carve.raw

Have a brief look at the file *image_carve.raw* with your wonderful command line hexdump tool, **xxd**:

```
xxd image_carve.raw | less
```

It's really just a file full of random characters. Somewhere inside there is a standard JPEG image. Let's go through the steps we need to take to "recover" the picture file using **dd** and other Linux tools. We are going to stick with command line tools available in most default installations.

First we need a plan. How would we go about recovering the file? What are the things we need to know to get the image (picture) out, and only the image? Imagine **dd** as a pair of scissors. We need to know where to put the scissors to start cutting, and we need to know where to stop cutting. Finding the start of the JPEG and the end of the JPEG can tell us this. Once we know where we will start and stop, we can calculate the *size* of the JPEG. We can then tell **dd** where to start cutting, and how much to cut. The output file will be our JPEG image. Easy, right? So here's our plan, and the tools we'll use:

1) Find the start of the JPEG (**xxd** and **grep**)
2) Find the end of the JPEG (**xxd** and **grep**)
3) Calculate the size of the JPEG (in bytes using **bc**)
4) Cut from the start to the end and output to a file (using **dd**)

This exercise starts with the assumption that we are familiar with standard file headers. Since we will be searching for a standard JPEG image within the data chunk, we will start with the stipulation that the JPEG header begins with hex *ffd8* with a six-byte offset to the string "*JFIF*". The end of the standard JPEG is marked by hex *ffd9*.

Let's go ahead with step 1: Using **xxd**, we pipe the output of our *image_carve.raw* file to grep and look for the start of the JPEG[4]:

```
xxd image_carve.raw | grep ffd8
00052a0: b4f1 559c ffd8 ffe0 0010 4a46 4946 0001  ..U…….JFIF..
```

[4] The perceptive among you will notice that this is a "perfect world" situation. There are a number of variables that can make this operation more difficult. The grep command can be adjusted for many situations using a complex regular expression (outside the scope of this document).

As the output shows, we've found the pattern "*ffd8*" near the string "*JFIF*". The start of a standard JPEG file header has been found. The offset (in hex) for the *beginning of this line of **xxd** output* is *00052a0.* Now we can calculate the byte offset in decimal. For this we will use the **bc** command. **bc** is a command line "calculator", useful for conversions and calculations. It can be used either interactively or take piped input. In this case we will echo the hex offset to **bc**, first telling it that the value is in base 16. **bc** will return the decimal value.

```
echo "ibase=16; 00052A0" | bc
21152
```

It's important that you use *uppercase letters* in the hex value. Note that this is NOT the start of the JPEG, just the start of the line in **xxd's** output. The "*ffd8*" string is actually located another 4 bytes farther into that line of output. So we add 4 to the start of the line. Our offset is now *21156*. We have found and calculated the start of the JPEG image in our data chunk.

Now it's time to find the end of the file.

Since we already know where the JPEG starts, we will start our search for the end of the file *from that point*. Again using **xxd** and **grep** we search for the string:

```
        xxd –s 21156 image_carve.raw | grep ffd9
0006c74: ffd9 d175 650b ce68 4543 0bf5 6705 a73c  …ue..hEC..g..<
```

The **–s 21156** specifies where to start searching (since we know this is the front of the JPEG, there's no reason to search before it and we eliminate false hits from that region). The output shows the first "*ffd9*" at hex offset *0006c74*. Let's convert that to decimal:

```
echo "ibase=16; 0006C74" | bc
27764
```

Because that is the offset for the *start* of the line, we need to add 2 to the value to include the ffd9 (giving us *27766*). Now that we know the start and the end of the file, we can calculate the size:

```
echo "27766 - 21156" | bc
6610
```

We now know the file is 6610 bytes in size, and it starts at byte offset 21156. The carving is the easy part! We will use **dd** with three options:

skip= how far into the data chuck we begin "cutting".
bs= (block size) the number of bytes we include as a "block".
count = the number of blocks we will be "cutting".

The input file for the **dd** command is *image_carve.raw.* Obviously, the value of **skip** will be the offset to the start of the JPEG. The easiest way to handle the block size is to specify it as **bs=1** (meaning one byte) and then setting **count** to the size of the file. The name of the output file is arbitrary.

```
dd if=image_carve.raw of=carv.jpg skip=21156 bs=1 count=6610
6610+0 records in
6610+0 records out
```

You should now have a file in your current directory called *carv.jpg.* If you are in X, simply use the **xview** command to view the file (or any other image viewer) and see what you've got.

```
xview carv.jpg
```

Carving partitions with dd

Now we can try a more useful exercise in carving with **dd**. Often, you will obtain or be given a **dd** image of a full disk. At times you might find it desirable to have each separate partition within the disk available to search or mount. Remember, you cannot simply mount an entire disk image, only the partitions.

This can be accomplished in a number of ways. One theoretically complex method would be to use the Enhanced Loopback driver (provided by NASA's Computer Crimes Division) to associate an entire disk image with a single loop device that is "partition aware". We will talk more about this method a little later.

The method we will use in this exercise entails identifying the partitions within a **dd** image using the standard loopback device along with **fdisk** or **sfdisk.** We will then use **dd** to carve the partitions out of the image.

First, let's grab the practice disk image that we will be working on. This is a **dd** image of a 330MB disk from a Linux system that was compromised ("hacked").

ftp://ftp.hq.nasa.gov/pub/ig/ccd/linuxintro/able2.tar.gz

The **tar** archive contains the disk image, the MD5 digest values, and the imaging log file with information collected during the imaging process.

Create a directory called "*able2*" in your */root* directory. This will be the working directory for the following exercise. Again, the vast majority of steps taken in preparation for, and execution of a forensic analysis require root access to commands and devices.

Once you have downloaded the file, check the md5sum (it should match the output below):

```
md5sum able2.tar.gz
a0cef6c441ae84ef931c541d73ee619f  able2.tar.gz
```

The file name is derived from the original *hostname* of the machine that was compromised ("hacked"). Very often we name our cases and evidence with the original hostname of the machine we are investigating.

If the MD5 matches, then we can continue…We now need to check the contents of the **tar** archive, then extract and decompress the archive.

```
tar tzvf able2.tar.gz
able2.dd  (disk image)
able2.log (log of the collection)
md5.dd    (md5 hash of the image)
md5.hdd  (md5 hash of the original disk)
```

```
tar xzvf able2.tar.gz
```

This executes the **tar** command with the options **–x** to extract the files, **-z** to decompress the files, **-v** for verbose output, and **–f** to specify the file.

Have a look at the files that result:

```
ls -lh
total 464M
-rwxrwxr-x    1 root  root     330M Aug 10 21:16 able2.dd
-rwxrwxr-x    1 root  root     3.6k Aug 11 07:56 able2.log
-rwxrwxr-x    1 root  root     134M Aug 11 14:42 able2.tar.gz
-rwxrwxr-x    1 root  root       43 Aug 10 21:16 md5.dd
-rwxrwxr-x    1 root  root       43 Aug 10 21:04 md5.hdd
```

The output of **ls –lh** (the **–lh** is for "long list with human readable sizes") shows the 330MB **dd** image, the logfile and two files that record the original MD5 hashes, one for the image (*md5.dd*) and one for the original disk (*md5.hdd*). At this point you can check the hash of the *able2.dd* and compare it to the value stored in *md5.dd* to be sure the image is intact.

```
cat md5.dd
02b2d6fc742895fa4af9fa566240b880  able2.dd
```

```
md5sum able2.dd
02b2d6fc742895fa4af9fa566240b880  able2.dd
```

Okay, now we have our image, and we have verified that it is an accurate copy. We now want to know a little bit about the contents of the image and what it represents. During the evidence acquisition process, it is essential that information about the disk be recorded. Standard operating procedures should include collection of disk and system information, and not just the **dd** image itself.

The file *able2.log* was created from the output of various commands used **during the evidence collection process**. The log includes information about the investigator that gathered the evidence, information about the system, and the output of commands including **hdparm**, **fdisk**, **sfdisk** and hashing functions. We create the log file by appending (">>") the output of the commands, in sequence, to the log:

command **>> logfile.txt**

Look at the log file, *able2.log,* using **less** and scroll down to the section that shows the structure of the disk (the output of **fdisk –l /dev/hdd** and **sfdisk –l –uS /dev/hdd**):

```
fdisk output for SUBJECT disk:

Disk /dev/hdd: 345 MB, 345830400 bytes
15 heads, 57 sectors/track, 790 cylinders
Units = cylinders of 855 * 512 = 437760 bytes

  Device Boot    Start     End      Blocks   Id  System
/dev/hdd1            1      12       5101+   83   Linux
/dev/hdd2           13     132       51300   83  Linux
/dev/hdd3          133     209      32917+   82  Linux swap
/dev/hdd4          210     790     248377+   83  Linux

##########################################################
sfdisk output for SUBJECT disk:

Disk /dev/hdd: 790 cylinders, 15 heads, 57 sectors/track
Units = sectors of 512 bytes, counting from 0

 Device   Boot   Start       End     #sectors   Id   System
/dev/hdd1           57     10259       10203    83   Linux
/dev/hdd2        10260    112859      102600    83   Linux
/dev/hdd3       112860    178694       65835    82   Linux swap
/dev/hdd4       178695   675449       496755    83   Linux
```

The output shown above is directly from the victim hard drive (the machine *able2*), recorded prior to obtaining the **dd** image. It shows that there are 4 partitions on the drive. The data partitions are *hdd1, hdd2* and *hdd4*. The *hdd3* partition is actually a *swap* partition (for virtual memory). Remember that the designation *hdd* indicates that the victim hard drive was attached to our forensic workstation as the slave drive on the secondary IDE controller during the imaging process, NOT how it was attached in the original machine.

The command **sfdisk –l –uS /dev/hdd** gave us the second listing above and shows the partition sizes in units of "sectors" (**-uS)**. The output also gives us the start of the partition. For our partition carving exercise (as with the raw data carving), all we need is the starting offset, and the size.

Let's go ahead and **dd** out each partition. If you have the output of **sfdisk –l –uS /dev/hdx,** the job is easy.

```
dd if=able2.dd of=able2.part1.dd bs=512 skip=57 count=10203
dd if=able2.dd of=able2.part2.dd bs=512 skip=10260 count=102600
dd if=able2.dd of=able2.part3.dd bs=512 skip=112860 count=65835
dd if=able2.dd of=able2.part4.dd bs=512 skip=178695 count=496755
```

Examine these commands closely. The input file (**if=able2.dd**) is the full disk image. The output files (**of=able2.part#.dd**) will contain each of the partitions. The block size that we are using is the sector size (**bs=512)**, which matches the output of the **sfdisk** command. Each **dd** section needs to start where each partition begins (**skip=X),** and cut as far as the partition goes (**count=Y).** We also obtained partition number three, the swap partition. This can also be searched with **grep** and **strings** (or carving utilities) for evidence.

This will leave you with four *able2.part*.dd* files in your current directory that can now be loop mounted.

What if you have a **dd** image of the full disk, but no log file or access to the original disk, and therefore no info from **sfdisk** or **fdisk**?

We can use the standard loopback device to associate the **dd** image with a device, and then run our commands against that:

```
losetup /dev/loop0 able2.dd
```

As we discussed earlier (during our floppy exercise), the *loop* device allows us to associate a regular file (forensic image) with a device. In this case we have a **dd** image that we want to view as a disk. We use **losetup** to tell the system to associate the file *able2.dd* to the device file */dev/loop0.* Essentially, this makes */dev/loop0* look like the original disk.

```
sfdisk –l –uS /dev/loop0
Disk /dev/loop0: cannot get geometry

Disk /dev/loop0: 0 cylinders, 0 heads, 0 sectors/track
Warning: The first partition looks like it was made
 for C/H/S=*/15/57 (instead of 0/0/0).
For this listing I'll assume that geometry.
Units = sectors of 512 bytes, counting from 0

 Device         Boot   Start      End     #sectors    Id     System
/dev/loop0p1              57      10259     10203     83     Linux
/dev/loop0p2           10260     112859    102600     83     Linux
/dev/loop0p3          112860     178694     65835     82     Linux swap
/dev/loop0p4          178695     675449    496755     83     Linux
```

Aside from the error messages at the beginning of the output, notice that the actual disk geometry (in sectors) matches that taken from the original disk! The partitions are now noted as */dev/loop0p**, indicating, “loop device zero, partitions 1 through 4”. In a pinch, we could use this to gather information from the standard loop device to determine the disk partitioning[5].

When you are finished with the *loop0* association, be sure to remove it:

```
losetup –d /dev/loop0
```

Unfortunately, you cannot mount the partitions associated with */dev/loop0p**. The block devices don’t actually exist. This is where the NASA Enhanced Loopback Driver comes in.

The NASA Enhanced Loopback Driver

In the preceding section we discussed using **dd** to carve partitions out of a bit-stream forensic image. The reason we do this is to allow us to *mount* those partitions for analysis. We found that we could use the standard loopback driver to assist us in determining the partitions contained in the

[5] The purpose of this section is to explore the concept of the *loop* devices in more detail. In actuality, the original **dd** image is a file just like */dev/loop0* or */dev/hdx*. Try the **fdisk** or **sfdisk** commands on the **dd** image itself to see what I mean.

image if the original drive is not available. We also learned that we couldn't use the standard loopback driver to actually mount the partitions.

The NASA Computer Crimes Division has developed an Enhanced Loopback Driver that takes steps toward solving these issues. It is available in several forms from the following site:

ftp://ftp.hq.nasa.gov/pub/ig/ccd/enhanced_loopback/

The documentation (located at the same place) is clear, and the installation is fairly straightforward. Although you should *always* keep in mind that when you are messing with the system kernel, you are acting as a brain surgeon on your computer. It's always possible that it won't wake up again (won't boot). For the most part, this is recoverable with a boot disk, but the process can be frustrating and time consuming.

The Enhanced Loopback driver is actually a kernel module. It is made available as a part of a full kernel pre-compiled binary, or as a full kernel source package that you can customize and compile yourself. If you wish to try the NASA Loopback Kernel, consider using the pre-compiled binary. If you are interested in learning to compile your own kernel for forensic use, then I would suggest reading Thomas Rude's (Farmerdude) thorough paper on the subject at:

http://www.crazytrain.com/monkeyboy/FSK.pdf

Farmerdude offers some detailed information on what the kernel is, how it works, and options available to you to make your kernel fit your needs. Do not underestimate the importance of this subject to your continued Linux education. The benefits and dangers of compiling a custom kernel are outside the scope of this beginner's guide, but I would strongly suggest you read Farmerdude's paper if you have any desire to learn more nuts and bolts Linux. There is a plethora of information available at Farmerdude's website regarding Linux and its application as a forensic platform at

http://www.crazytrain.com

Determining the Subject Disk Filesystem Structure

Going back to our *able2* case **dd** images, we now have the original image along with the partition images that we carved out.

able2.dd	(original image)
able2.part1.dd	(1st Partition)
able2.part2.dd	(2nd Partition)
able2.part4.dd	(4th Partition)

The next trick is to mount the partitions in such a way that we reconstruct the original filesystem.

One of the benefits of Linux/Unix systems is the ability to separate the filesystem across partitions. This can be done for any number of reasons, allowing for flexibility where there are concerns about disk space or security, etc.

For example, a System Administrator may decide to keep the directory */var/log* on its own separate partition. This might be done in an attempt to prevent rampant log files from filling the root ("/" not "*/root*") partition and bringing the system down. It is common to see */boot* in its own partition as well. This allows the kernel image to be placed near "the front" (in terms of cylinders) of a hard drive, an issue in older versions of the Linux boot loader LILO. There are also a variety of security implications addressed by this setup.

So when you have a disk with multiple partitions, how do you find out the structure of the file system? Earlier in this paper we discussed the */etc/fstab* file. This file maintains the mounting information for each file system, including the physical partition; mount point, file system type, and options. Once we find this file, reconstructing the system is easy. With experience, you will start to get a feel for how partitions are setup, and where to look for the *fstab.* To make things simple here, just mount each partition (loopback, read only) and have a look around.

One thing we might like to know is what sort of file system is on each partition before we try and mount them. We can use the **file** command to do this[6]. Remember from our earlier exercise that the **file** command determines the type of file by looking for "header" information.

```
file able2.part*
        able2.part1.dd: Linux rev 1.0 ext2 filesystem data…
        able2.part2.dd: Linux rev 1.0 ext2 filesystem data…
        able2.part3.dd: Linux/i386 swap file (new style)…
        able2.part4.dd: Linux rev 1.0 ext2 filesystem data…
```

Previously, we were able to determine that the partitions were "Linux" partitions from the output of **fdisk** and **sfdisk.** Now **file** informs us that the file system type is *ext2*[7]. We can use this information to mount the partitions.

```
mount -t ext2 -o ro,loop able2.part1.dd /mnt/analysis/
```

Do this for each partition (either unmounting between partitions, or mounting to a different mountpoint) and you will find the */etc* directory containing the *fstab* file in *able2.part2.dd* with the following entries:

```
/dev/hda2       /           ext2        defaults      1 1
/dev/hda1       /boot       ext2        defaults      1 2
/dev/hda4       /usr        ext2        defaults      1 2
/dev/hda3       swap        Swap        defaults      0 0
```

So now we see that the logical file system was constructed from three separate partitions (note that */dev/hda* here refers to the disk when it is mounted in the original system):

```
"/" (root)      ← mounted from /dev/hda2 (data on hda2)
   |_ bin/            (data on hda2)
   |_boot/      ← mounted from /dev/hda1 (data on hda1)
   |_dev/             (data on hda2)
   |_etc/             (data on hda2)
```

[6] Keep in mind that the **file** command relies on the contents of the *magic* file to determine a file type. If this command does not work for you in the following example, then it is most likely because the magic file on your system does not inlcude headers for filesytem types.

[7] You can also use the *auto* filesystem type under the mount command, but I prefer to be explicit. Check **man mount** for more information.

```
        |_home/              (data on hda2)
        |_lib/               (data on hda2)
        |_opt/               (data on hda2)
        |_proc/              (data on hda2)
        |_usr/        ← mounted from/dev/hda4 (data on hda4)
        |_root/              (data on hda2)
        |_sbin/              (data on hda2)
        |_tmp/               (data on hda2)
        |_var/               (data on hda2)
```

Now we can create the original file system at our analysis mount point by creating separate directories for each partition. The mount point */mnt/analysis* already exists. We create the other two with:

```
mkdir /mnt/analysis/boot
mkdir /mnt/analysis/usr
```

Now we mount each partition image at its respective mountpoint:

```
mount -t ext2 -o ro,loop able2.part2.dd /mnt/analysis/
mount -t ext2 -o ro,loop able2.part1.dd /mnt/analysis/boot
mount -t ext2 -o ro,loop able2.part4.dd /mnt/analysis/usr
```

We now have the recreated original file system under */mnt/analysis*:

```
"/" (root)            ← mounted on /mnt/analysis
        |_ bin/
        |_boot/       ← mounted on /mnt/analysis/boot
        |_dev/
        |_etc/
        |_home/
        |_lib/
        |_opt/
        |_proc/
        |_usr/        ← mounted on /mnt/analysis/usr
        |_root/
        |_sbin/
        |_tmp/
        |_var/
```

At this point we can run all of our searches and commands just as we did for the previous floppy disk exercise on a complete file system "rooted" at */mnt/analysis*.

As always, you should know what you are doing when you mount a complete file system on your forensic workstation. Be aware of options to the **mount** command that you might want to use (check **man mount** for options like "*nodev*" and "*nosuid*", "*noatime*" etc.). Take note of where links point to from the subject file system. Note that we have mounted the partitions "read only" (**ro**). Remember to unmount each partition when you are finished exploring.

X. Advanced Forensic Tools

So now you have some experience with using the Linux command line and the powerful tools that are provided with a Linux installation. However, as forensic examiners, we soon come to find out that time is a valuable commodity. While learning to use the command line tools native to a Linux install is useful for a myriad of tasks in the "real world", it can also be tedious. After all, there are Windows based tools out there that allow you to do much of what we have discussed here in a simple point and click GUI. Well, the same can be said for Linux.

The popularity of Linux is growing at a fantastic rate. Not only do we see it in an enterprise environment and in big media, but we are also starting to see its widening use in the field of computer forensics. In recent years we've seen the list of available forensic tools for Linux grow with the rest of the industry.

In this section we will cover a number of forensic tools available to make your analysis easier and more efficient. We will cover both free tools and commercial tools.

> AUTHOR'S NOTE: Inclusion of tools and packages in this section in no way constitutes an endorsement of those tools. Please test them yourself to ensure that they meet your needs. The tools here were chosen because it was suggested by a large number of readers of the original Introduction document that I provide information on forensic packages for Linux.
>
> Since this is a Linux document, I am covering available Linux tools. This does not mean that the common tools available for other platforms cannot be used to accomplish many of the same results. On a personal note, I do maintain that analysis of a Unix system is best accomplished with a Unix (like) toolset.

Sleuthkit

The first of the tools we will cover here is actually not a GUI tool at all, but rather a collection of command line tools. We cover them here because we will soon introduce a tool (Autopsy) that provides a nice GUI wrapper.

The Sleuthkit is written by Brian Carrier and maintained at http://www.sleuthkit.org. It is partially based on The Coroner's Toolkit (TCT) originally written by Dan Farmer and Wietse Venema. The Sleuthkit adds additional file system support (FAT and NTFS). Additionally, the Sleuthkit allows you to analyze various file system types regardless of the platform you are currently working on. The current version, as of this writing is sleuthkit-1.66. Go to the "downloads" section of the Sleuthkit website and grab a copy. For the sake of simplicity, let's just download the file to our */root* (root user's home) directory.

Installation is easy. You can simply un-tar the file then change in to the resulting directory:

```
tar xzvf sleuthkit-1.66.tar.gz
cd sleuthkit-1.66
```

Take a moment to read the included documentation. We will continue with a short description here, but most of what you need to know is right there.

Compiling the tools is as simple as typing **make** right in that directory. If you run into any problems, read the *INSTALL* document. When the compiling is finished, you will find the Sleuthkit tools located in the *sleuthkit-1.66/bin* directory. The **man** pages for each command are located in the *sleuthkit/man* directory. Normally you would move or link the executable files to a common directory somewhere in your path. We are going to leave the tools where they are and call them explicitly.

The Sleuthkit's tools are organized by what the author calls a "layer" approach.

- File system layer – **fsstat,**
- File name layer – **fls, ffind**
- Content (data) layer – **dcalc, dcat, dls, dstat**
- Meta data (inode) layer – **icat, ils, ifind, istat**

Notice that the commands that correspond to the analysis of a given layer begin with a common letter. For example, the file system command starts with "**fs**", and the inode layer commands start with "**i**".

We are going to do a quick sample analysis using just a few of the Sleuthkit command line tools. Like all of the other exercises in this document, I'd suggest you follow along if you can. Using these commands on your own is the only way to really learn the techniques.

Let's have a look at a couple of the file system and file name layer tools, **fsstat** and **fls**. We will run them against our *able2* partition images[8]. For the sake of this analysis, the information we are looking for is located on the root partition.

Remember from our previous analysis of the able2 **dd** images that the root ("/") file system is located on the second partition (*able2.part2.dd*).

Make sure you are in */root/sleuthkit-1.66/bin/* (or wherever you installed the Sleuthkit) and run the following command:

```
./fsstat /root/able2/able2.part2.dd
```

[8] The Sleuthkit works on partition images, not on whole disk images. This is one reason why it might be useful to learn how to carve partitions (or take partition dd images).

You should see the following output (partial):

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: EXT2FS
Volume Name:
Last Mount: Thu Feb 13 02:33:02 1997
Last Write: Sun Aug 10 14:50:03 2003
Last Check: Tue Feb 11 00:20:09 1997
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 - 12880
Root Directory: 2

CONTENT-DATA INFORMATION
--------------------------------------------
Fragment Range: 0 - 51299
Block Size: 1024
Fragment Size: 1024
<CONTINUES>
```

The **fsstat** command provides file system specific information about the file system located in a device or partition image.

We can get more information using the **fls** command. **fls** lists the file names and directories contained in a file system, with a number of options. If you type "**./fls**" on its own, you will see the available options (view the **man** page for a more complete explanation).

```
./fls –f linux-ext2 –Frd /root/able2/able2.part2.dd
```

```
r/r * 11120(realloc):   var/lib/slocate/slocate.db.tmp
r/r * 10063:    var/log/xferlog.5
r/r * 10063:    var/lock/makewhatis.lock
r/r * 6613:     var/run/shutdown.pid
r/r * 1046:     var/tmp/rpm-tmp.64655
r/r * 6609(realloc):    var/catman/cat1/rdate.1.gz
r/r * 6613:     var/catman/cat1/rdate.1.gz
r/r * 6616:     tmp/logrot2V6Q1J
r/r * 2139:     dev/ttYZ0/lrkn.tgz
d/r * 10071(realloc):   dev/ttYZ0/lrk3
r/r * 6572(realloc):    etc/X11/fs/config-
l/r * 1041(realloc):    etc/rc.d/rc0.d/K83ypbind
l/r * 1042(realloc):    etc/rc.d/rc1.d/K83ypbind
l/r * 6583(realloc):    etc/rc.d/rc2.d/K83ypbind
l/r * 6584(realloc):    etc/rc.d/rc4.d/K83ypbind
l/r * 1044:     etc/rc.d/rc5.d/K83ypbind
l/r * 6585(realloc):    etc/rc.d/rc6.d/K83ypbind
r/r * 1044:     etc/rc.d/rc.firewall~
r/r * 6544(realloc):    etc/pam.d/passwd-
r/r * 10055(realloc):   etc/mtab.tmp
r/r * 10047(realloc):   etc/mtab~
r/- * 0:        etc/.inetd.conf.swx
r/r * 2138(realloc):    root/lolit_pics.tar.gz
r/r * 2139:     root/lrkn.tgz
```

In this case, we are running the **fls** command against an Ext2 file system (**-f linux-ext2**), showing only file entries (**-F**), descending into directories (**-r**), and displaying deleted entries (**-d**). The output gives us the file name and the *inode* to which that file is associated.

Now let’s use a couple of Metadata (inode) layer tools included with the Sleuthkit. An inode has unique number and is assigned to a file. The number corresponds to the *inode table*, allocated when a partition is formatted. The inode contains all the metadata available for a file, including the modified/accessed/changed times and all the data blocks allocated to that file.

First we are going to use **istat** from the Sleuthkit. Remember that **fsstat** took a *file system* as an argument and reported statistics about that file system. Well, **istat** does the same thing; only it works on a specified *inode*.

If you look at the output of our **fls** command, you will see a file called *lrkn.tgz* located in the */root* directory (the last file in the output of our **fls** command). The inode displayed by **fls** for this file is *2139*. Note that this same inode also points to a file in */dev* (same file, different location). We are going to use **istat** to gather some information about inode *2139*. Remember, we are in the *sleuthkit/bin* directory, so we use "**./**" to indicate that the command (not in our path) is run from the current directory:

```
./istat -f linux-ext2 /root/able2/able2.part2.dd 2139 | less
```

```
inode: 2139
Not Allocated
Group: 1
uid / gid: 0 / 0
mode: -rw-r--r--
size: 3639016
num of links: 0

Inode Times:
Accessed:       Sun Aug 10 00:18:38 2003
File Modified:  Sun Aug 10 00:08:32 2003
Inode Modified: Sun Aug 10 00:29:58 2003
Deleted:        Sun Aug 10 00:29:58 2003

Direct Blocks:
22811 22812 22813 22814 22815 22816 22817 22818
22819 22820 22821 22822 22824 22825 22826 22827
...<snipped>
32225 32226 32227 32228 32229 32230 32231 32232
32233 32234

Indirect Blocks:
22823 23080 23081 23338 23595 23852 24109 24366
30478 30735 30992 31249 31506 31763 32020
```

This reads the inode statistics (**./istat**), on an ext2 (**-f linux-ext2**) partition (**/root/able2/able2.part2.dd**) from inode **2139.** There is a large amount of output here, showing all the inode information and the direct and indirect blocks[9] that contain all of the file's data. We can either pipe the output to a file for logging or evidence results, or we can send it to **less** for viewing.

We now have the name of the deleted file (from **fls**) and the inode information, including where the data is stored (from **istat**). Now we are going to use the **icat** command from the Sleuthkit to grab the actual data contained in the data blocks referenced from the inode. **icat** also takes the "inode" as an argument and reads the content of the data blocks that are assigned to that inode, sending it to standard output.

We are going to send the contents of the data blocks assigned to that inode to a file for closer examination. Again, we issue the following command from the *sleuthkit/bin* directory:

```
./icat –f linux-ext2 /root/able2/able2.part2.dd 2139 > /root/lrkn.tgz.2139
```

This runs the **icat** command on our ext2 (**-f linux-ext2**) partition (**able2.part2.dd**) and streams the contents of the data blocks associated with inode **2139** to the file **/root/lrkn.tgz.2139**. The filename is arbitrary; I simply took the name of the file from **fls** and appended the inode number to indicate that it was recovered. Normally this output should be directed to some results directory.

Now that we have what we hope is a recovered file, what do we do with it? Look at the resulting file with the **file** command:

```
file lrkn.tgz.2139
lrkn.tgz.2139: gzip compressed data, was "lrkn.tar", from Unix
```

Have a look at the contents of the recovered archive (pipe the output through **less**…it's long). Remember that the "**t**" option lists the contents of the archive.

[9] For a detailed description of "direct" and "indirect" blocks, see http://e2fsprogs.sourceforge.net/ext2intro.html.

Don't just haphazardly extract an archive without knowing what it will write, or especially where[10]:

```
tar tzvf lrkn.tgz.2139 | less
```

```
drwxr-xr-x  lp/lp          0  1998-10-01 18:48:18 lrk3/
-rwxr-xr-x  lp/lp        742 1998-06-27 11:30:45 lrk3/1
-rw-r--r--  lp/lp        716 1996-11-02 16:38:43 lrk3/MCONFIG
-rw-r--r--  lp/lp       6833 1998-10-03 05:02:15 lrk3/Makefile
-rw-r--r--  lp/lp       6364 1996-12-27 22:01:43 lrk3/README
<CONTINUES>
```

Notice that there is a *README* file included in the archive. If we are curious about the contents of the archive, perhaps reading the *README* file would be a good idea, yes? Rather that extract the entire contents of the archive, we will go for just the *README* using the following **tar** command:

```
tar xzvfO lrkn.tgz.2139 lrk3/README > /root/README.2139
```

The difference with this **tar** command is that we specify that we want the output sent to *stdout* ("**O**") so we can redirect it. We also specify the name of the file that we want extracted from the archive (**lrk3/README**). This is all redirected to a new file called */root/README.2139*.

If you read that file, you will find that we have uncovered a "rootkit", full of trojanized programs used to hide a hacker's activity.

What we have seen here is a simple (and in many ways incomplete) example of the Sleuthkit's command line tools for forensic examination. If you are left a little confused, just go through the steps one at a time. If you don't understand the commands or options, check the usage and read the **man** pages and Sleuthkit documentation. Run through the exercise a couple of times, and the purpose and outcome will make more sense. Take your time and experiment a little with the options.

[10] Let's face it, it would be BAD to have an archive that contains a bunch of trojans and other nasties (evil kernel source or libraries, etc.) overwrite those on your system. Be extremely careful with archives.

We are now going to look at an easier way to "point and click" your way through a Sleuthkit exam, organizing your investigation as you go using Autopsy.

Autopsy

Autopsy is another great program by Brian Carrier that provides a nice html based front end to the Sleuthkit. In addition to allowing you easy access to the functions provided by the Sleuthkit tools, Autopsy provides a vehicle for organizing your cases and the images and hosts associated with those cases. Download Autopsy at the Sleuthkit website.

We begin by installing Autopsy (version 1.75 will be used with Sleuthkit 1.66). As with all software, you should read the included documentation thoroughly before you get started. The documents included in the Autopsy (and Sleuthkit, for that matter) are well written and explain everything you need to know. Some of the high points to take note of before beginning the installation process:

1. We have to create an "Evidence Locker" for Autopsy to store generated results. Do this with the **mkdir** command in a suitable directory with enough space:

   ```
   mkdir /root/autopsy_evid/
   ```

2. Know the path to your Sleuthkit directory. You have to enter it during the Autopsy install.

3. You can also use a hash database for data reduction purposes (known good files and/or known bad files). While we are not going to use such a database here, be aware of the capability. Proper use of hash databases can often be a huge time saver.

Begin the installation by untarring the *autopsy-1.75.tar.gz* archive in the root directory with.

```
tar xzvf autopsy-1.75.tar.gz
```

Change into the resulting *autopsy-1.75* directory and type **make.** The program will search for several files, and then prompt you for the location of your Sleuthkit install. Enter the path. When you are asked about the location of your NSRL database, just hit enter (unless you installed it). Finally, enter the path of our Evidence Locker (*/root/autopsy_evid*).

All we need to do is start the Autopsy process. Obviously, we will need to be in the X Window environment to use the tool. In a terminal window, enter:

./autopsy

Once the Autopsy process starts, you will want to open your browser and copy the resulting URL into your browser window.

```
==============================================

               Autopsy Forensic Browser
            http://www.sleuthkit.org/autopsy/
                       ver 1.75

==============================================

Evidence Locker: /root/autopsy_evid/
Start Time: Sun Aug 17 16:13:56 2003
Remote Host: localhost
Local Port: 9999

Open an HTML browser on the remote host and paste this URL in it:

    http://localhost:9999/3098252907250697l042/autopsy

Keep this process running and use <ctrl-c> to exit
```

Be careful not to close the terminal window. The Autopsy process starts as a “child” of the terminal in which it is started. If the terminal window is killed, so is Autopsy.

Copying the URL shown above (yours might differ) and pasting it into your browser results in the Autopsy HTML interface being displayed. Take note of the "Help" button in the lower right. The HTML help pages are detailed and extremely useful if you get stuck or curious:

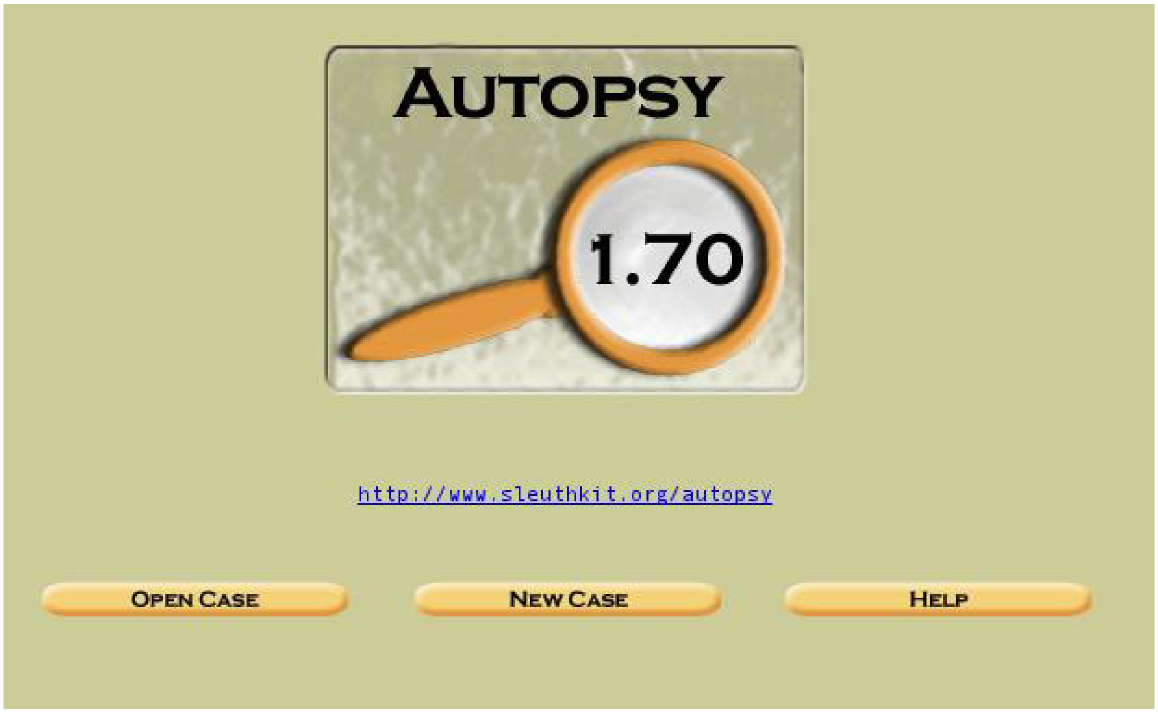
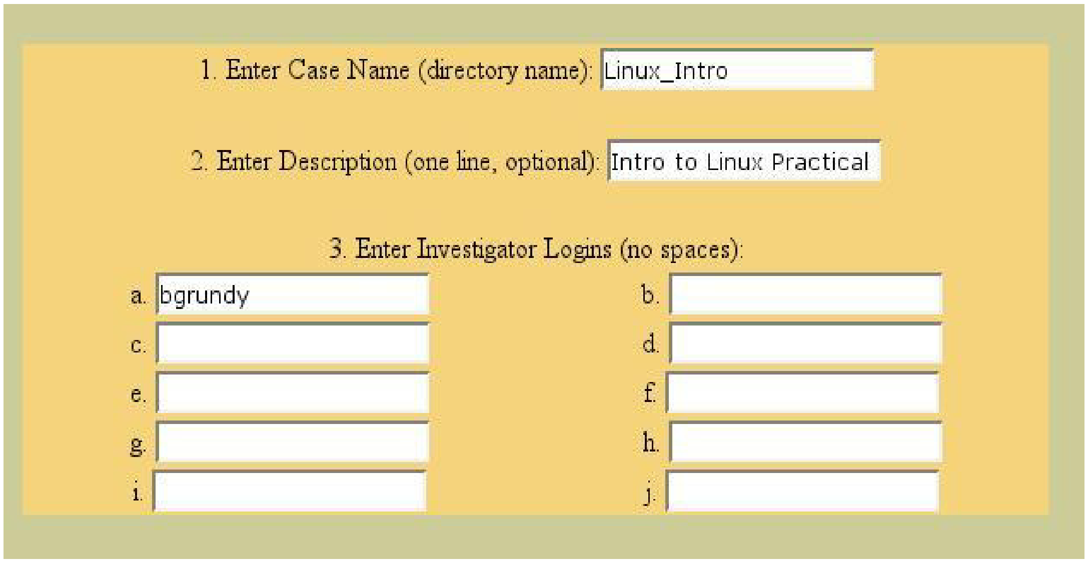


Figure 1. Autopsy opening screen

Click on "New Case" and fill in the required information:



1. Enter Case Name (directory name): Linux_Intro

2. Enter Description (one line, optional): Intro to Linux Practical

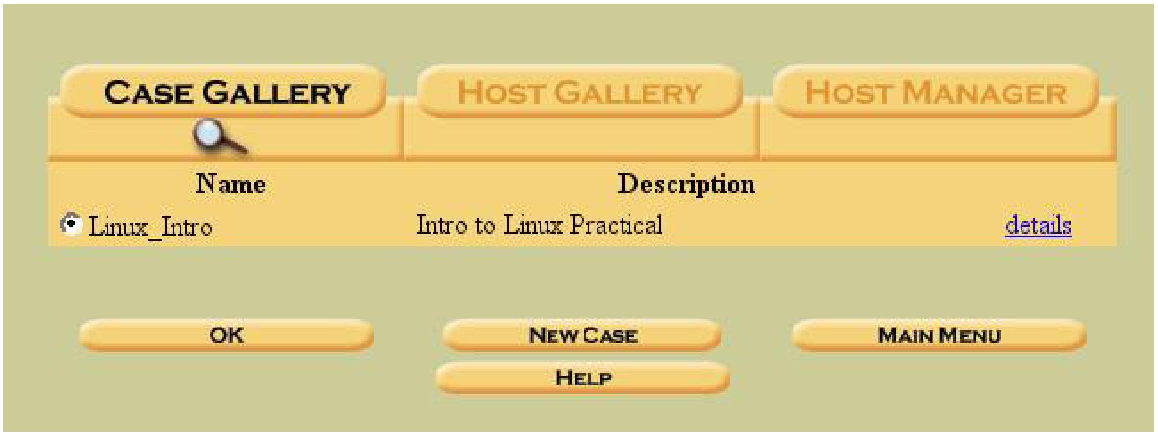
3. Enter Investigator Logins (no spaces):

a. bgrundy
b.
c.
d.
e.
f.
g.
h.
i.
j.

Figure 2. Entering new case information

Click on “New Case” again, and then read the information that Autopsy provides on the steps it has taken (creating new directories, etc.).

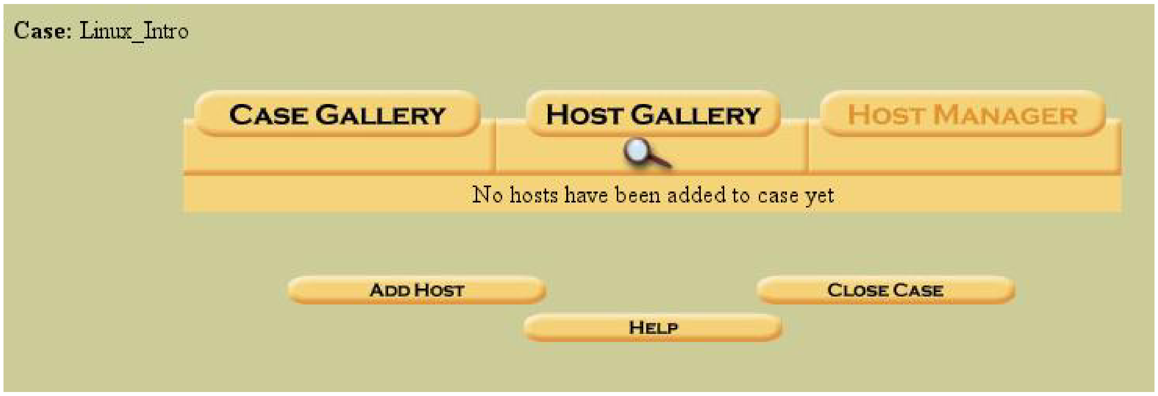
You are then taken to the “Case Gallery”. When you have more than one case started, they will all be displayed here. As you start Autopsy, you will select the case with the corresponding radio button and then click “ok” to go to that case.



Name	Description	
● Linux_Intro	Intro to Linux Practical	details

Figure 3. The "Case Gallery"

After entering the case that we just created, we are presented with a screen that tells us that we need to add a “host” in the “Host Gallery”. The case name is displayed in the top left hand corner:

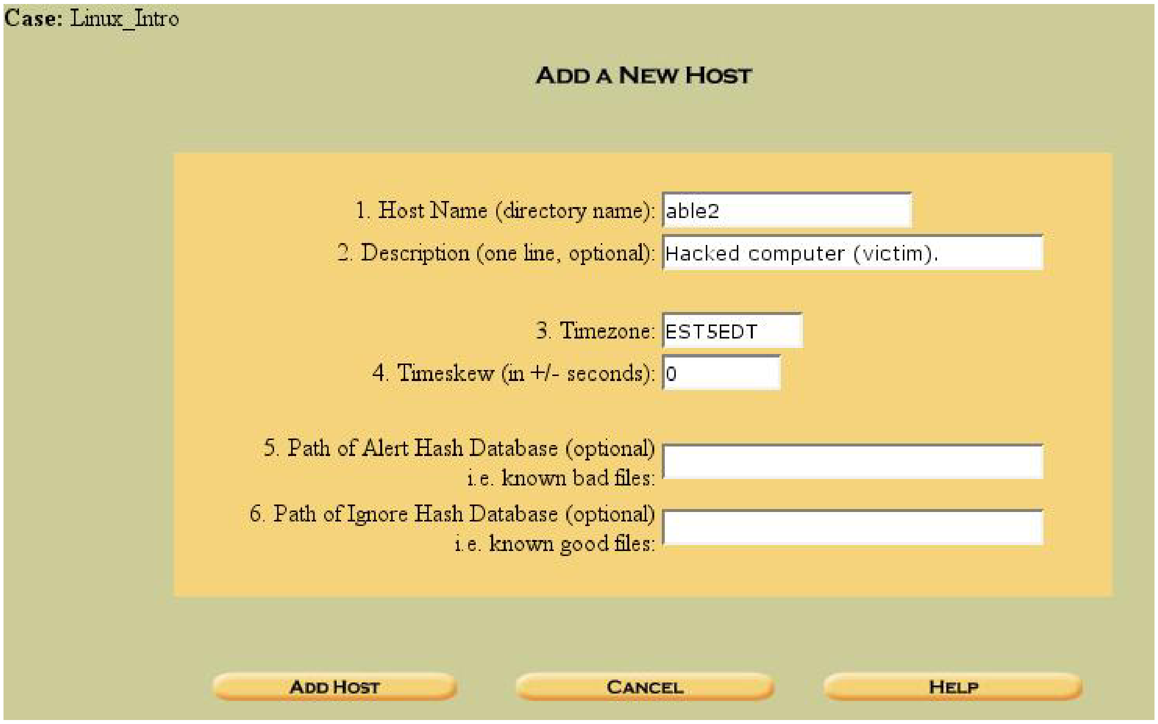

Case: Linux_Intro

No hosts have been added to case yet

Figure 4. An empty "Host Gallery"

At this point, we need to provide a little explanation that might be obvious to some, but not to those of you who do not commonly handle network intrusion cases. The term “host” refers to any computer, identified by name or IP address, connected to the network (local or wide area). A host computer can be either a victim or a "hostile". Both require forensic examination in most cases. Details on network intrusion investigations are outside the scope of this document.

Enter the "Add Host" section and fill in the required information:



Case: Linux_Intro

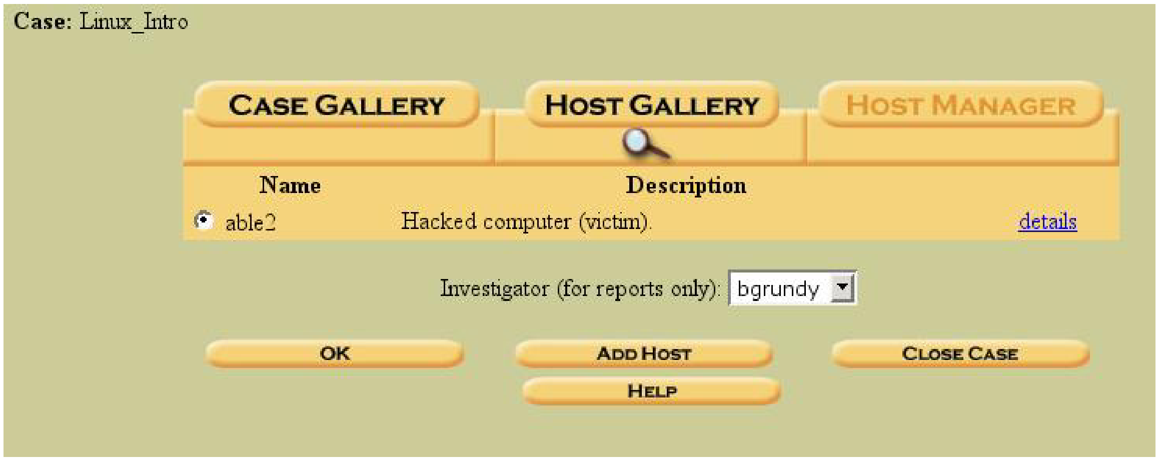
ADD A NEW HOST

1. Host Name (directory name): able2
2. Description (one line, optional): Hacked computer (victim).
3. Timezone: EST5EDT
4. Timeskew (in +/- seconds): 0
5. Path of Alert Hash Database (optional) i.e. known bad files:
6. Path of Ignore Hash Database (optional) i.e. known good files:

Figure 5. Adding a new host

The information we provide above includes the Host Name (name or IP of the computer), which will be used to name the directory in our evidence locker, a description, the time zone of the computer, and the time skew.

The available time zones can be found in your */usr/share/zoneinfo* directory. In this case, I used EST5EDT (Eastern Standard Time, 5 hours from GMT, Eastern Daylight Time). The time skew describes the number of seconds the host's clock was off from a standard clock. This is important for reconciling log entries and file times.



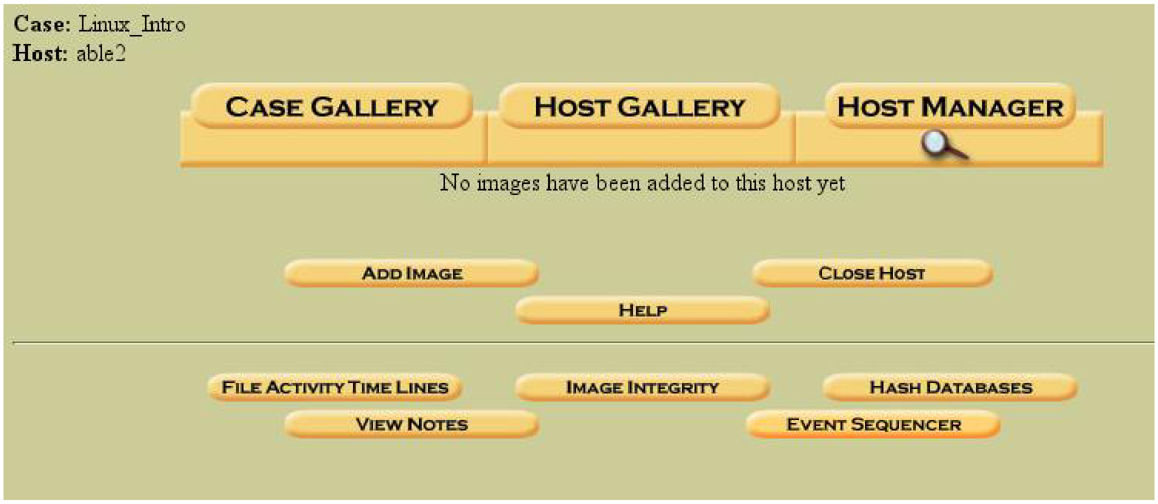
Case: Linux_Intro

Name	Description	
● able2	Hacked computer (victim).	details

Investigator (for reports only): bgrundy

Figure 6. A single host entered in the "Host Gallery"

Now that we have the case defined, and a host within that case defined, we need to tell the host about the different forensic images that make up our evidence gathered from *able2.* Click "ok" (above) to enter the *able2* "Host Manger" area. Again, if there were multiple hosts defined for this case, we would select the host we want to manage with the appropriate radio button.

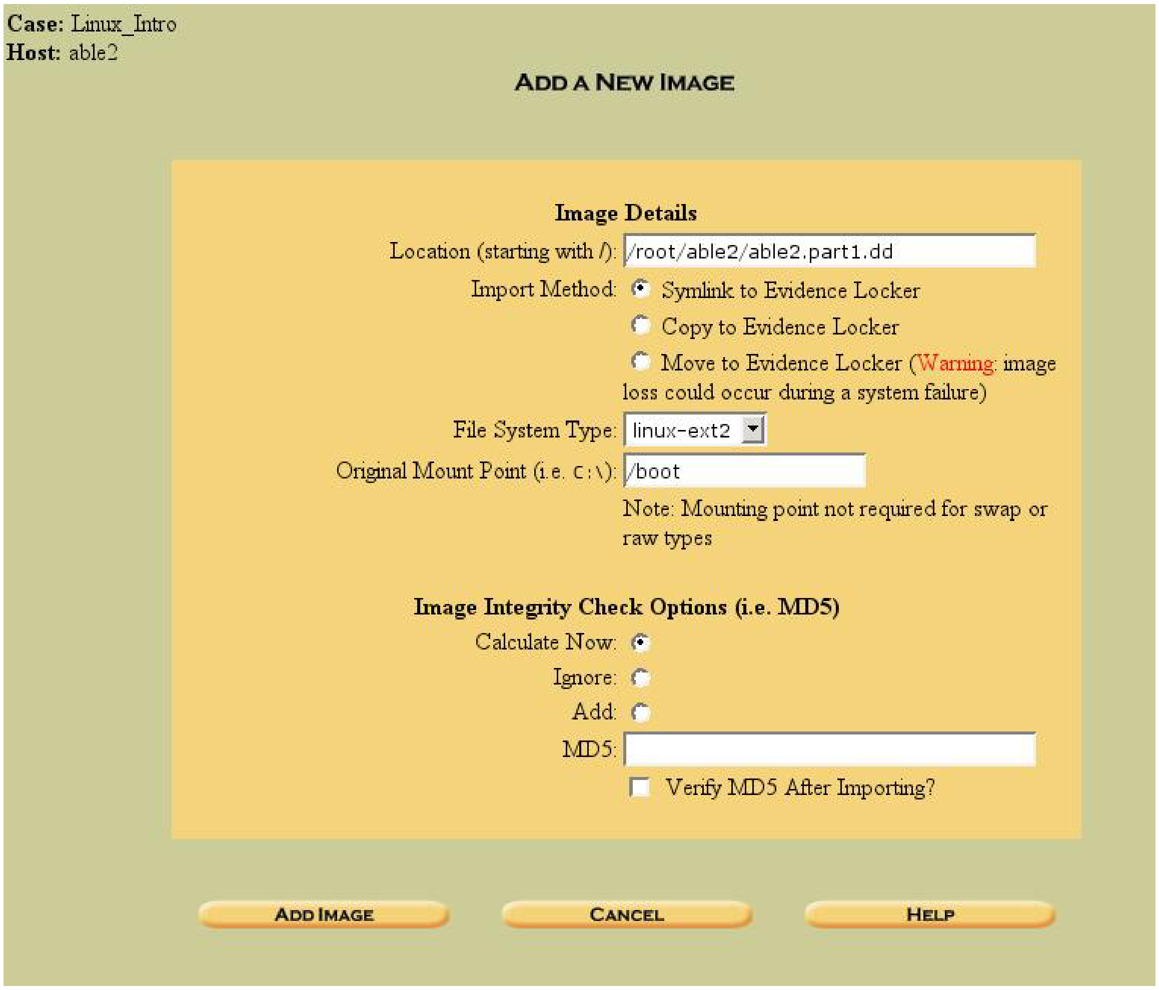


Case: Linux_Intro
Host: able2

No images have been added to this host yet

Figure 7. The "Host Manager" prior to adding forensic images

We are now in the Host Manager (above). Notice that "No images have been added" is telling us that there is not yet any evidence associated with this host. We need to add the appropriate images, so click on "Add Image".



Case: Linux_Intro
Host: able2

ADD A NEW IMAGE

Image Details

Location (starting with /): /root/able2/able2.part1.dd

Import Method: ● Symlink to Evidence Locker
○ Copy to Evidence Locker
○ Move to Evidence Locker (Warning: image loss could occur during a system failure)

File System Type: linux-ext2

Original Mount Point (i.e. C:\): /boot

Note: Mounting point not required for swap or raw types

Image Integrity Check Options (i.e. MD5)

Calculate Now: ●

Ignore: ○

Add: ○

MD5:

☐ Verify MD5 After Importing?

Figure 8. Adding the partition images to the "Host Manager" for our host

Much of the information we need to input in the dialog box shown above was calculated earlier in our exercises. We already mentioned that Sleuthkit (and therefore Autopsy) works (for now) on partition images, not on disk images. Earlier, we calculated the sizes and locations of the partitions and carved them out of the original *able2.dd* image. It is these images that we will use in this case.

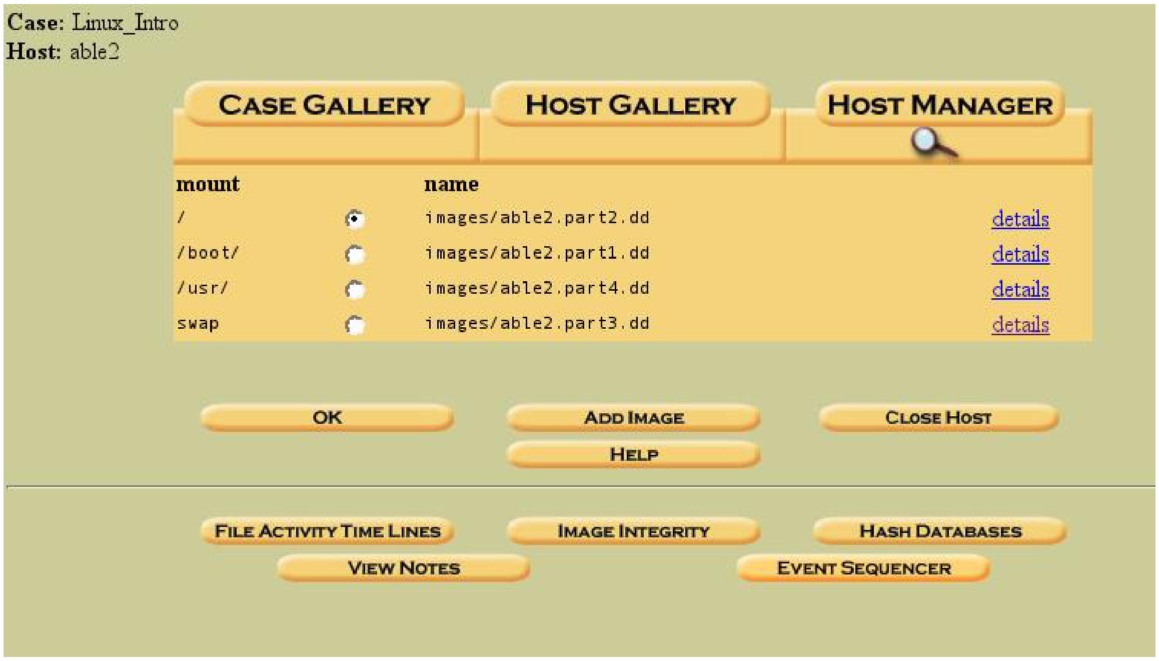
Autopsy gives you the choice of “linking” (creating a shortcut) from the original location of the partition image, copying the image, or moving the image to the evidence locker (*/root/autopsy_evid/…*). I prefer to symlink the image. Do whichever you prefer, but make sure you have enough room.

In most of our Sleuthkit commands from the previous exercise, we had to specify the file system type with a command option (**-f linux-ext2**). In the above dialog box, we specify it in a dropdown box. Any operation taken on that image will apply that file system type.

Finally, we fill in the "Original Mount Point". Again, we found this out earlier in our exercises by accessing the */etc/fstab* file from the *able2.part2.dd* image. In much the same way that we did it manually earlier[11], Autopsy is using the information we are providing to rebuild the original file system from the *able2* host.

able2.part1.dd	*/boot*	*ext2*	*/dev/hda1*
able2.part2.dd	*/*	*ext2*	*/dev/hda2*
able2.part4.dd	*/usr*	*ext2*	*/dev/hda4*
able2.part3.dd	*swap*	*Swap*	*/dev/hda3*

We need to go through the "Add Image" process once for each partition image that we carved from the original *able2.dd*. Note that for the file system type for *able2.part3.dd* (the swap partition), you can just put **none**. When you are finished, the Host Manager page should look like this:



Case: Linux_Intro
Host: able2

mount		name	
/	●	images/able2.part2.dd	details
/boot/	○	images/able2.part1.dd	details
/usr/	○	images/able2.part4.dd	details
swap	○	images/able2.part3.dd	details

Figure 9. Completed "Host Manager" for our host, *able2*

[11] See the section Determining the Subject Disk Filesystem Structure on page 76.

The image above shows the completed Host Manager entries for our case. We are now ready to explore Autopsy, and the powerful forensic capabilities it gives us.

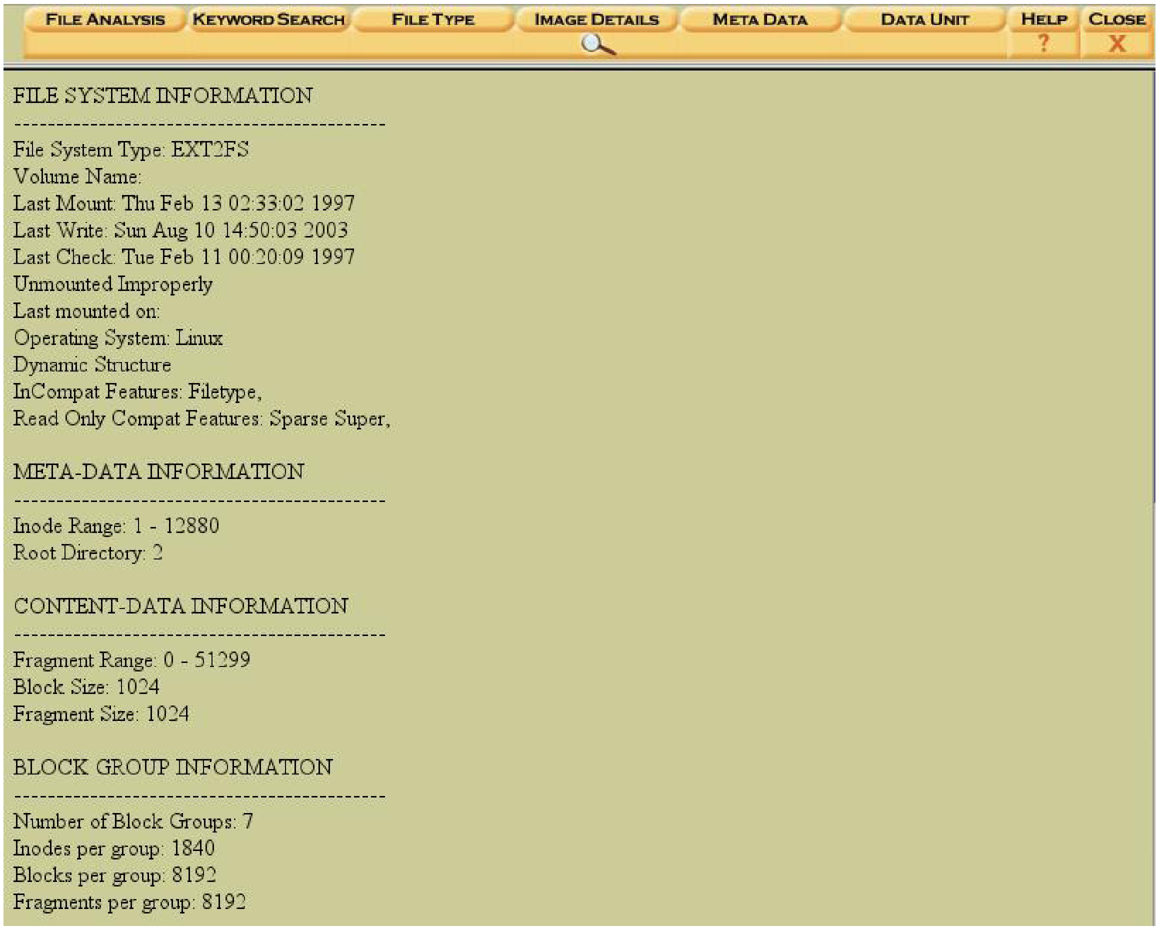
In addition to giving us a "point and click" environment for accessing the commands available with the Sleuthkit, Autopsy has also done a good job of organizing our case based on the information we filled in.

```
/root (root user's home directory)
        |_ autopsy_evid               ←our evidence locker
               |_ Linux_Intro         ←our case name
                      |_able2         ←our first host
                             |_images
                             |_logs
                             |_mnt
                             |_output
                             |_reports
                      |_host 2        ←if we add another computer
                      |_host3
               |_Second Case
                      |_host
               ...etc.
```

Most of the exploration of this powerful software package I will leave up to the reader. This is a beginner's document, aimed only at getting you started. We will go through a couple of steps, and leave the rest to your curiosity…

Let's have a look at some of the commands and steps we took using the Sleuthkit's command line tools on this same evidence, this time using Autopsy.

First, in the Host Manager make sure the radio button for *able2.part2.dd* (the "/" partition) is selected, and click "okay". In the resulting page, there is a frame at the top with a row of buttons for various functions. Click on the button "Image Details" and look at the output. Does it look familiar?

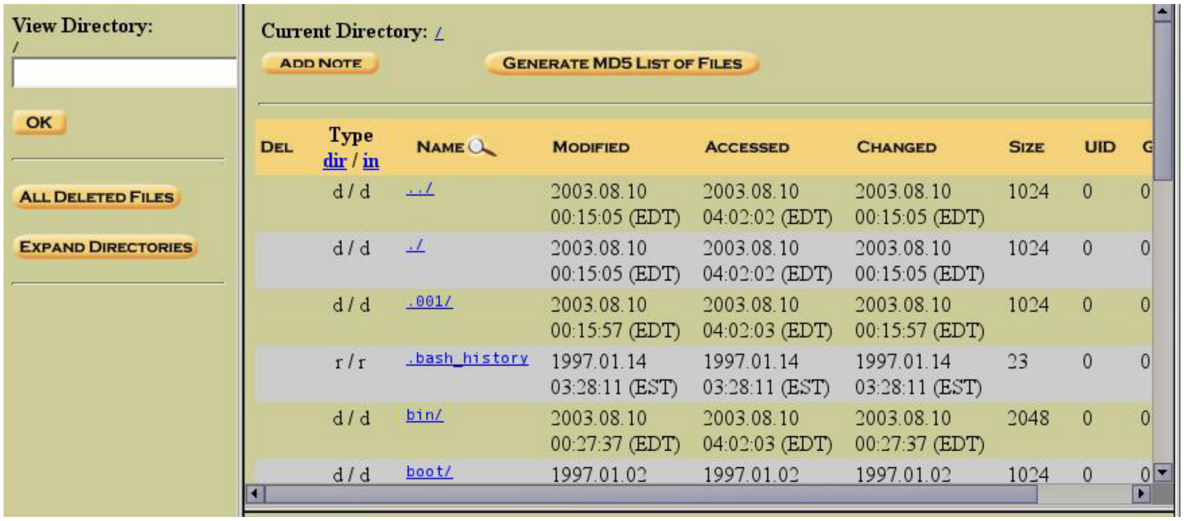


```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: EXT2FS
Volume Name:
Last Mount: Thu Feb 13 02:33:02 1997
Last Write: Sun Aug 10 14:50:03 2003
Last Check: Tue Feb 11 00:20:09 1997
Unmounted Improperly
Last mounted on:
Operating System: Linux
Dynamic Structure
InCompat Features: Filetype,
Read Only Compat Features: Sparse Super,

META-DATA INFORMATION
--------------------------------------------
Inode Range: 1 - 12880
Root Directory: 2

CONTENT-DATA INFORMATION
--------------------------------------------
Fragment Range: 0 - 51299
Block Size: 1024
Fragment Size: 1024

BLOCK GROUP INFORMATION
--------------------------------------------
Number of Block Groups: 7
Inodes per group: 1840
Blocks per group: 8192
Fragments per group: 8192
```

Figure 10. "Image Details" provides similar output to "fsstat"

This is the same output we saw from our **fsstat** command from the Sleuthkit! Let's go a little deeper and see if we can reproduce more of our Sleuthkit output.

In the top frame row of buttons, click "File Analysis" (we are still in the *able2.part2.dd* image, the "/" file system of *able2*). We are now given a "tree" view of the contents of the selected image.

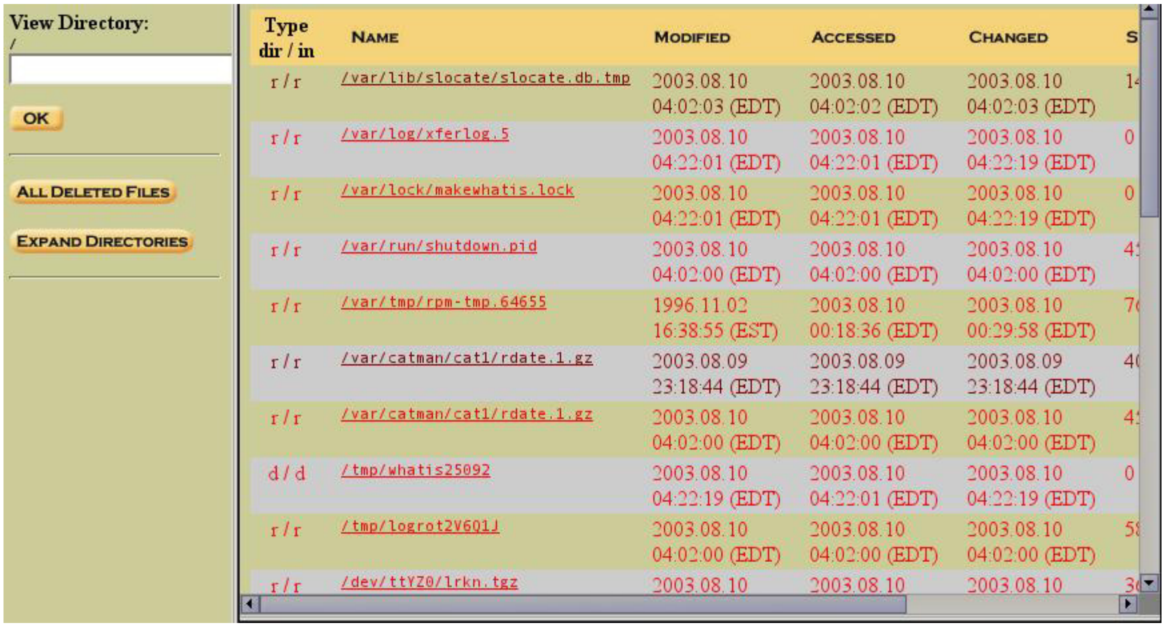


Current Directory: /

Del	Type dir / in	Name	Modified	Accessed	Changed	Size	UID	G
	d / d	../	2003.08.10 00:15:05 (EDT)	2003.08.10 04:02:02 (EDT)	2003.08.10 00:15:05 (EDT)	1024	0	0
	d / d	./	2003.08.10 00:15:05 (EDT)	2003.08.10 04:02:02 (EDT)	2003.08.10 00:15:05 (EDT)	1024	0	0
	d / d	.001/	2003.08.10 00:15:57 (EDT)	2003.08.10 04:02:03 (EDT)	2003.08.10 00:15:57 (EDT)	1024	0	0
	r / r	.bash_history	1997.01.14 03:28:11 (EST)	1997.01.14 03:28:11 (EST)	1997.01.14 03:28:11 (EST)	23	0	0
	d / d	bin/	2003.08.10 00:27:37 (EDT)	2003.08.10 04:02:03 (EDT)	2003.08.10 00:27:37 (EDT)	2048	0	0
	d / d	boot/	1997.01.02	1997.01.02	1997.01.02	1024	0	0

Figure 11. "File Analysis" display

In the left hand pane, there is a button labeled "All Deleted Files". Click on this button…

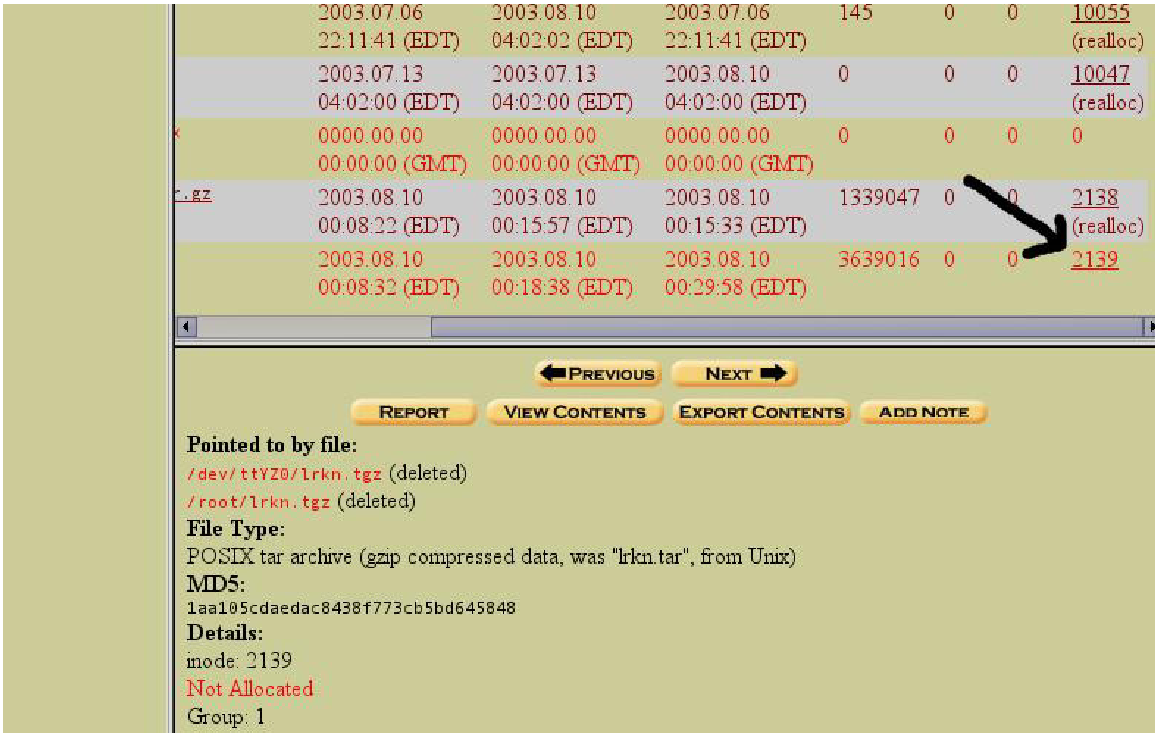


Type dir / in	Name	Modified	Accessed	Changed	S
r / r	/var/lib/slocate/slocate.db.tmp	2003.08.10 04:02:03 (EDT)	2003.08.10 04:02:02 (EDT)	2003.08.10 04:02:03 (EDT)	14
r / r	/var/log/xferlog.5	2003.08.10 04:22:01 (EDT)	2003.08.10 04:22:01 (EDT)	2003.08.10 04:22:19 (EDT)	0
r / r	/var/lock/makewhatis.lock	2003.08.10 04:22:01 (EDT)	2003.08.10 04:22:01 (EDT)	2003.08.10 04:22:19 (EDT)	0
r / r	/var/run/shutdown.pid	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	4
r / r	/var/tmp/rpm-tmp.64655	1996.11.02 16:38:55 (EST)	2003.08.10 00:18:36 (EDT)	2003.08.10 00:29:58 (EDT)	7
r / r	/var/catman/cat1/rdate.1.gz	2003.08.09 23:18:44 (EDT)	2003.08.09 23:18:44 (EDT)	2003.08.09 23:18:44 (EDT)	4
r / r	/var/catman/cat1/rdate.1.gz	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	4
d / d	/tmp/whatis25092	2003.08.10 04:22:19 (EDT)	2003.08.10 04:22:01 (EDT)	2003.08.10 04:22:19 (EDT)	0
r / r	/tmp/logrot2V601J	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	5
r / r	/dev/ttYZ0/lrkn.tgz	2003.08.10	2003.08.10	2003.08.10	3

Figure 12. "All Deleted Files" provides similar output to fls

This generates a list of all the deleted files found on this partition. Again, do you recognize the output? It provides similar output to our **fls** command from the Sleuthkit. Scroll to the bottom of the list, and you will see our */root/lrkn.tgz,* recovered by the Sleuthkit from inode 2139. Once you scroll to the bottom of the list and see */root/lrkn.tgz*, you can scroll all the way to the right and see that the file is indeed associated with inode 2139.

If you click ON THE INODE NUMBER (2139), you will see the following:



	2003.07.06 22:11:41 (EDT)	2003.08.10 04:02:02 (EDT)	2003.07.06 22:11:41 (EDT)	145	0	0	10055 (realloc)
	2003.07.13 04:02:00 (EDT)	2003.07.13 04:02:00 (EDT)	2003.08.10 04:02:00 (EDT)	0	0	0	10047 (realloc)
	0000.00.00 00:00:00 (GMT)	0000.00.00 00:00:00 (GMT)	0000.00.00 00:00:00 (GMT)	0	0	0	0
.gz	2003.08.10 00:08:22 (EDT)	2003.08.10 00:15:57 (EDT)	2003.08.10 00:15:33 (EDT)	1339047	0	0	2138 (realloc)
	2003.08.10 00:08:32 (EDT)	2003.08.10 00:18:38 (EDT)	2003.08.10 00:29:58 (EDT)	3639016	0	0	2139

Pointed to by file:
/dev/ttYZ0/lrkn.tgz (deleted)
/root/lrkn.tgz (deleted)
File Type:
POSIX tar archive (gzip compressed data, was "lrkn.tar", from Unix)
MD5:
1aa105cdaedac8438f773cb5bd645848
Details:
inode: 2139
Not Allocated
Group: 1

Figure 13. Click ON THE INODE number (all the way to the right, see the arrow)

Again, you see the output is similar to out output from the Sleuthkit command **istat** (under "Details:"). Clicking on "Export Contents" produces a dialog box that allows us to save the file to our local file system. If you retrace the same process we followed with the previous exported contents of inode 2139 using Sleuthkit[12], you will find the same results.

This is as far as we will go with Sleuthkit/Autopsy in this beginner environment. Hopefully you will continue to explore some of the useful features of Autopsy on your own (the timeline feature is especially useful for intrusion investigations).

[12] See Page 86.

SMART for Linux

SMART, by ASR Data, is a commercial (not free) GUI based forensic tool for Linux that has a great interface allowing access to a full set of forensic analysis capabilities.

http://www.asrdata.com/SMART/

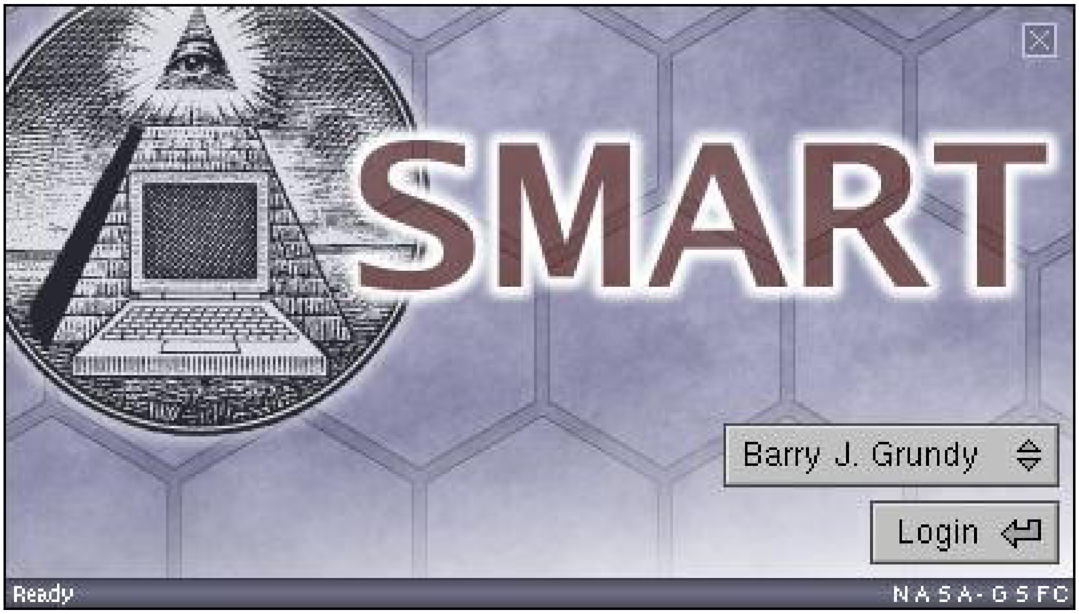
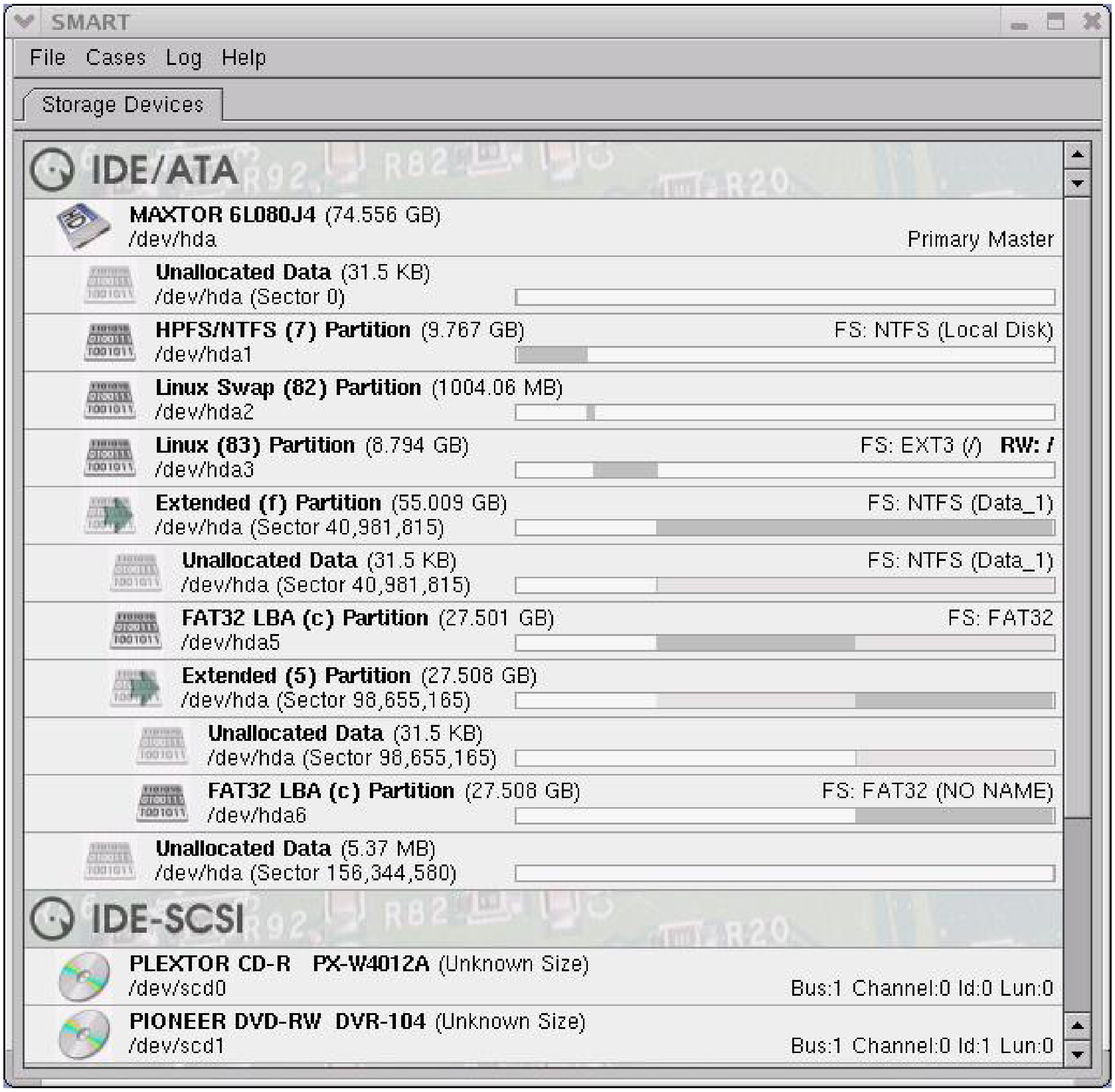


Figure 14. Smart splash screen and login

We are not going to do a full practical exercise with SMART, since it is a commercial tool, and not many beginner readers will have it available. Following is a small tour to give you a taste of the SMART interface.

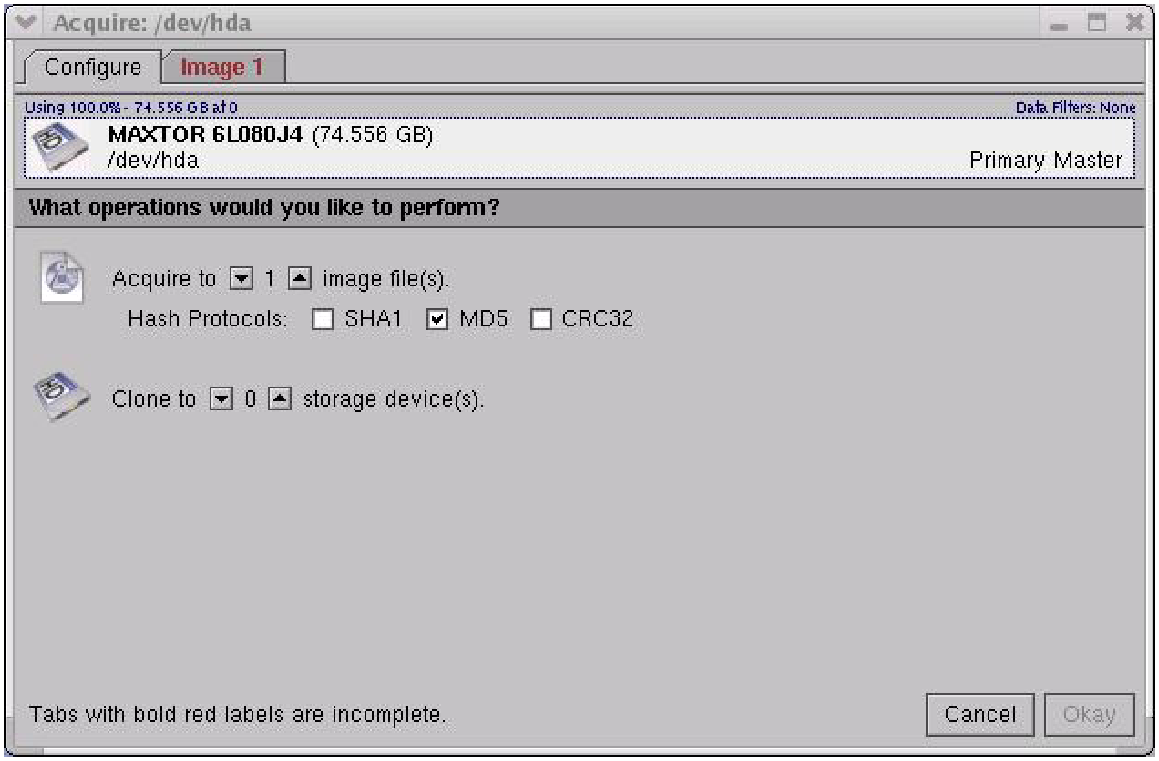
Opening SMART provides the user with a view of the physical layout of all the devices recognized on the system, including internal and external drives. This gives the examiner an overall picture of what file systems reside on each drive, the sizes of each partition, and the amount of unallocated space on the drive.



```
IDE/ATA
MAXTOR 6L080J4 (74.556 GB)
/dev/hda                                          Primary Master
Unallocated Data (31.5 KB)
/dev/hda (Sector 0)
HPFS/NTFS (7) Partition (9.767 GB)                FS: NTFS (Local Disk)
/dev/hda1
Linux Swap (82) Partition (1004.06 MB)
/dev/hda2
Linux (83) Partition (8.794 GB)                   FS: EXT3 (/)  RW: /
/dev/hda3
Extended (f) Partition (55.009 GB)                FS: NTFS (Data_1)
/dev/hda (Sector 40,981,815)
  Unallocated Data (31.5 KB)                      FS: NTFS (Data_1)
  /dev/hda (Sector 40,981,815)
  FAT32 LBA (c) Partition (27.501 GB)             FS: FAT32
  /dev/hda5
  Extended (5) Partition (27.508 GB)
  /dev/hda (Sector 98,655,165)
    Unallocated Data (31.5 KB)
    /dev/hda (Sector 98,655,165)
    FAT32 LBA (c) Partition (27.508 GB)           FS: FAT32 (NO NAME)
    /dev/hda6
Unallocated Data (5.37 MB)
/dev/hda (Sector 156,344,580)
IDE-SCSI
PLEXTOR CD-R   PX-W4012A (Unknown Size)
/dev/scd0                                         Bus:1 Channel:0 Id:0 Lun:0
PIONEER DVD-RW  DVR-104 (Unknown Size)
/dev/scd1                                         Bus:1 Channel:0 Id:1 Lun:0
```

Figure 15. Smart's opening window with drive identification

SMART is a "right click" driven program. Most functions available to an examiner for a given object are accessed through a mouse driven menu system. For instance, right clicking on a physical device (disk or partition) provides a menu that includes "Acquire". Selection of this item provides a dialog box to allow forensic imaging.



Using 100.0% - 74.556 GB at 0

Data Filters: None

MAXTOR 6L080J4 (74.556 GB)
/dev/hda
Primary Master

What operations would you like to perform?

Acquire to 1 image file(s).

Hash Protocols: □ SHA1 ☑ MD5 □ CRC32

Clone to 0 storage device(s).

Tabs with bold red labels are incomplete.

Figure 16. Forensic image acquisition dialog box. Red text indicates incomplete items…

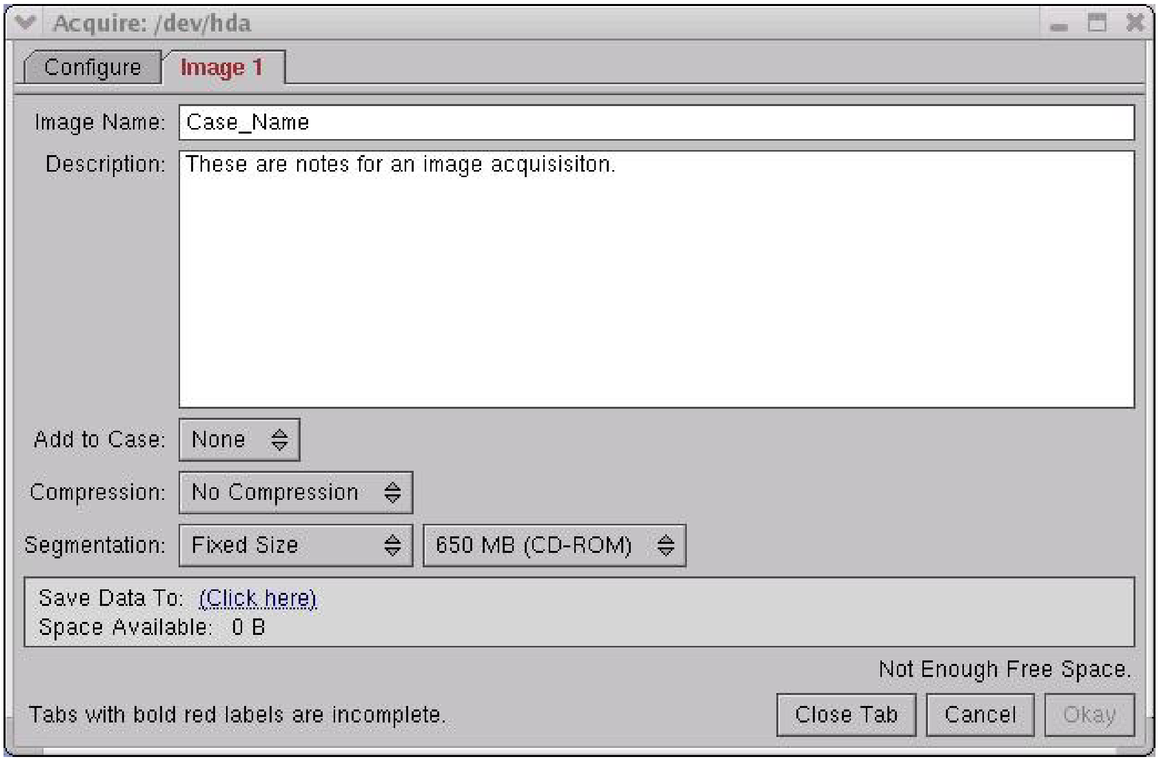


Image Name: Case_Name

Description: These are notes for an image acquisisiton.

Add to Case: None

Compression: No Compression

Segmentation: Fixed Size 650 MB (CD-ROM)

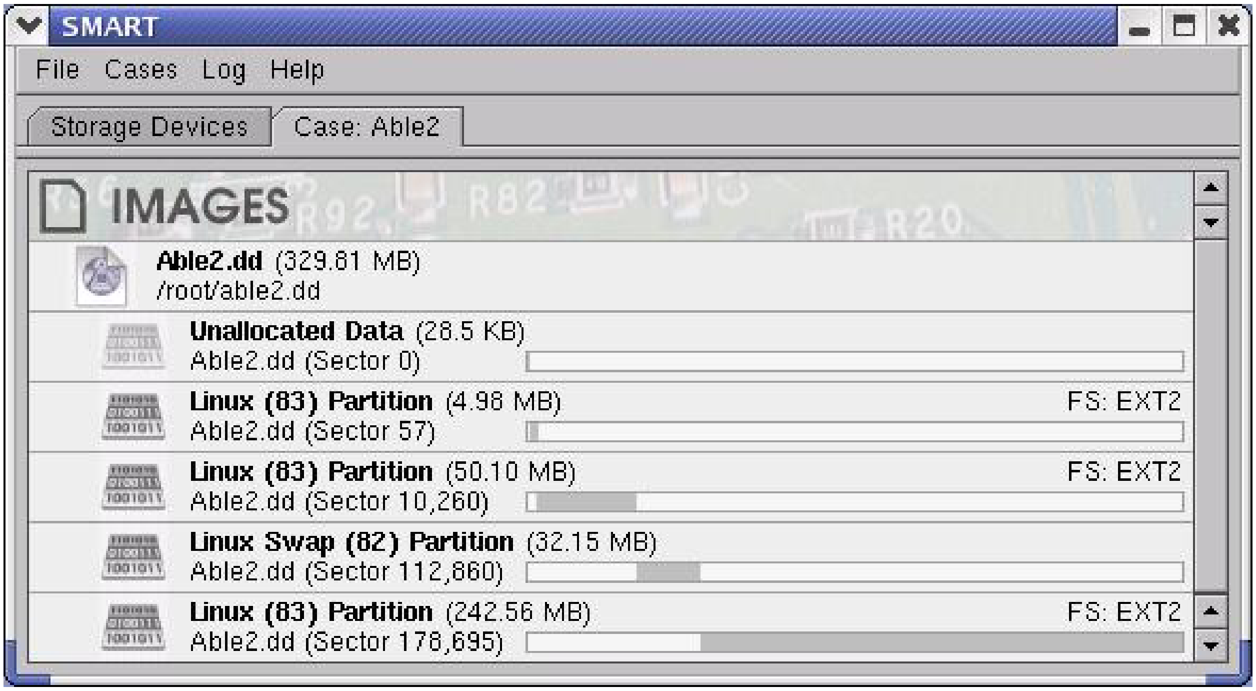
Save Data To: (Click here)
Space Available: 0 B

Not Enough Free Space.

Tabs with bold red labels are incomplete.

Figure 17. The "Image" tab under "Acquire".

Case management under SMART is straightforward. Once a forensic image (or multiple images) is added as evidence to a case, SMART will parse the image and provide details on the contents. Here we've added our *able2.dd* image to a case:



Able2.dd (329.81 MB)
/root/able2.dd

Unallocated Data (28.5 KB)
Able2.dd (Sector 0)

Linux (83) Partition (4.98 MB) FS: EXT2
Able2.dd (Sector 57)

Linux (83) Partition (50.10 MB) FS: EXT2
Able2.dd (Sector 10,260)

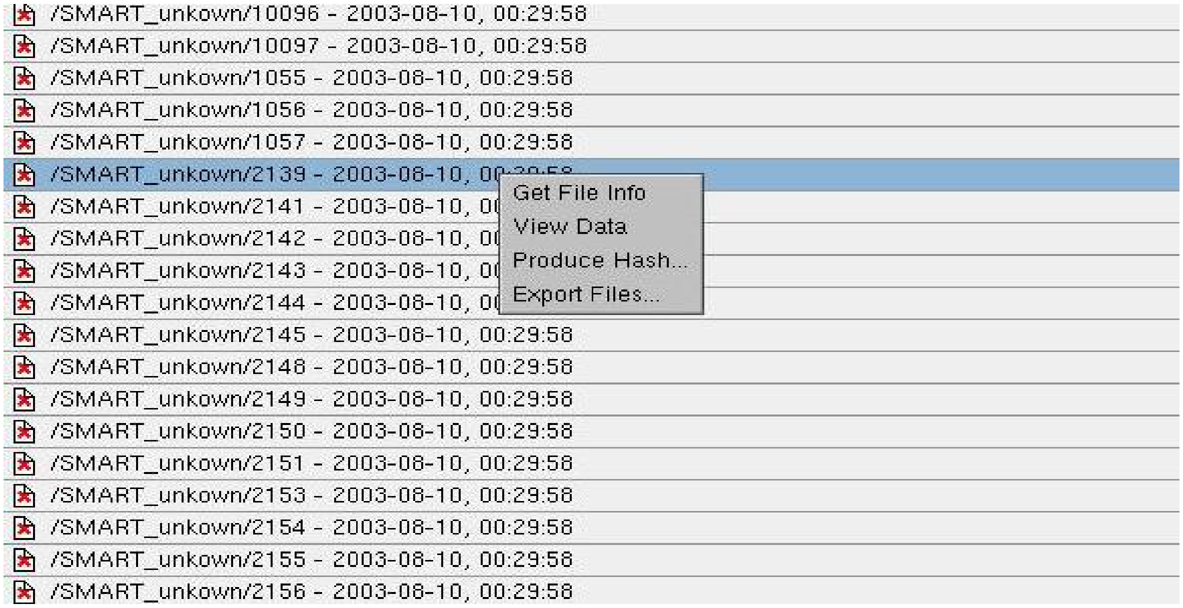
Linux Swap (82) Partition (32.15 MB)
Able2.dd (Sector 112,860)

Linux (83) Partition (242.56 MB) FS: EXT2
Able2.dd (Sector 178,695)

Figure 18. A SMART look at our evidence image

We see each of the partitions as a graphical representation of the same sort of information we might gather using **fdisk –l** on a physical disk.

Right clicking on a partition allows you to "Study" it and obtain information and a file listing (including deleted files). Additional right clicks on the files will access menus that allow us to export the file(s) or view them as raw data for closer study.



```
/SMART_unkown/10096 - 2003-08-10, 00:29:58
/SMART_unkown/10097 - 2003-08-10, 00:29:58
/SMART_unkown/1055 - 2003-08-10, 00:29:58
/SMART_unkown/1056 - 2003-08-10, 00:29:58
/SMART_unkown/1057 - 2003-08-10, 00:29:58
/SMART_unkown/2139 - 2003-08-10, 00
/SMART_unkown/2141 - 2003-08-10, 00
/SMART_unkown/2142 - 2003-08-10, 00
/SMART_unkown/2143 - 2003-08-10, 00
/SMART_unkown/2144 - 2003-08-10, 00
/SMART_unkown/2145 - 2003-08-10, 00:29:58
/SMART_unkown/2148 - 2003-08-10, 00:29:58
/SMART_unkown/2149 - 2003-08-10, 00:29:58
/SMART_unkown/2150 - 2003-08-10, 00:29:58
/SMART_unkown/2151 - 2003-08-10, 00:29:58
/SMART_unkown/2153 - 2003-08-10, 00:29:58
/SMART_unkown/2154 - 2003-08-10, 00:29:58
/SMART_unkown/2155 - 2003-08-10, 00:29:58
/SMART_unkown/2156 - 2003-08-10, 00:29:58
```

Figure 19. Right click on a deleted file

The right click menu displayed for a file in a file listing allows you to perform a number of tasks. In the above screenshot, we see that we have the ability to export the contents of the deleted file at inode 2139, leading us to the same steps as we took with Sleuthkit/Autopsy on the same data.

Additionally, we can use SMART to loop mount the partitions with a simple click and browse the file system in either a terminal or in the file manager of your choice. This provides us the ability to use all our favorite Linux tools to search the logical file system and display the information we need for our analysis.

As with all advanced forensic tools, SMART provides excellent session and Case logging functions.

Other Advanced Linux Forensic Tools

There are too many tools to include in this document. For a sample of some other tools that might be of interest to a forensic examiner using Linux, check http://www.opensourceforensics.org/tools/unix.html .

XI. Bootable Linux Distributions

For so many people, this is the meat and potatoes of what makes Linux such a flexible operating system. Access to a bootable CD drive and the ability to reboot the machine can now give us the power to run a full-fledged Linux box without the need to install. For those who have not seen this in action, the power you can get from a CDROM, or even a floppy disk is amazing. This is not a complete list, but the following bootable distributions can give you some idea of what's available to you. There are many MANY more bootable distributions out there. Just do a Google search on "Linux bootable CD" for a sample.

Tomsrtbt - boot from a floppy

This small distribution is the definition of minimalist, and it fits on one floppy. You get a decent set of drivers for NICs and file systems (including FAT and NTFS). There's a basic set of common Linux tools, including **dd** and **rsh** or **nc** for imaging over net connections and more. The installation (to a floppy) can be done in Windows with an included batch file. The floppy holds a surprising number of programs, and actually formats your 1.44 Mb floppy to 1.722 Mb. Find it at http://www.toms.net/rb/

Knoppix - Full Linux without the install

This is a CDROM distribution for people who want to try a full-featured Linux distribution, but don't feel like installing Linux. It includes a full Linux environment and a huge compliment of software. The CD actually holds 2GB of software, including a full office suite, common network tools and just about anything else you're likely to need all compressed to a CD sized image. http://www.knoppix.net

Penguin Sleuth - Knoppix with a forensic flavor

Re-mastered by Ernie Bacca, this Knoppix based distribution maintains its user-friendliness and adds a good deal of forensic software as well. The complete list of software (including Sleuthkit/Autopsy) can be found at http://www.linux-forensics.com/forensics/pensleuth.html

White Glove Linux - Dr. Fred Cohen

White Glove Linux is a small, pocket-sized distribution developed by Dr. Fred Cohen (http://www.all.net). The CD runs a minimalist but usable Blackbox X interface and has a number of useful network audit tools that allow you to check the integrity of, and audit systems. Everything from file system checks, executable file validation and log checking, to finding graphical images on the target system are addressed by White Glove.

One of the important aspects of White Glove, especially with regard to the network tools, is that the CD is a "known source" distribution. This means that there are discrete sets of tools on the disk that have been reviewed and are tested and supported by Dr. Cohen. Larger public distributions (Knoppix, for example) attempt to get as much software on one disk as possible. We cannot always be certain that what each program advertises is the limit of what it does unless we are willing to test each tool or read the source code for ourselves. On the other hand, White Glove is a thoroughly tested and known source of network tools.

SMART for Linux - It's bootable!

The installation CDROM for SMART doubles as a boot disk as well, providing an excellent platform with an independently verified forensic tool for acquiring and analyzing physical media. The hardware detection is excellent (I've never had an issue with the CD). Once you start the system, you can run "**configx**" and then "**startx**" and get a choice of GUI's, including a very clean Fluxbox interface with easy access to the SMART User's Guide and the program itself. SMART's bootable CD provides a bootable environment that you can be sure is forensically sound. We've already had a glimpse of SMART's capabilities.
http://www.asrdata.com/SMART/

Conclusion

The examples and practical exercises presented to you here are very simple. There are quicker and more powerful ways of accomplishing what we have done in the scope of this document. The steps taken in these pages allow you to use common Linux tools and utilities that are helpful to the beginner.

Once you become comfortable with Linux, you can extend the commands to encompass many more options. Practice will allow you to get more and more comfortable with piping commands together to accomplish tasks you never thought possible with a default OS load (and on the command line to boot!);

- Compress an image as you create it.
- Image using **dd** over a network connection.
- Start learning how to automate these tasks with shell scripts (shell scripts are your friend!).

I hope that your time spent working with this guide was well spent. At the very least, I'm hoping it gave you something to do, rather than stare at Linux for the first time and wonder "what now?"

XI. Linux Support

Web sites to check for support:

Look here first: The Linux Documentation Project (LDP):
http://www.tldp.org

Linux Forensics:
http://linux-forensics.com

Open Source Forensic Software:
http://www.openforensics.org
http://www.opensourceforensics.org

Software:
http://sourceforge.net/

Thomas Rude's (Farmerdude) Website - Linux Forensics Guru.
http://www.crazytrain.com

Linux.com Sponsored by VA Linux
http://www.linux.com

The Official page of the Linux Kernel
http://www.kernel.org

Slashdot. News for Nerds. A must read, at least twice a day...
http://www.slashdot.org