LBNL-******

Instrumented SSH

NERSC Center Division

Lawrence Berkeley National Laboratory

Scott Campbell
Craig Lant

May 27, 2009

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Introduction

NERSC recently undertook a project to access and analyze Secure Shell (SSH) related data. This includes authentication data such as user names and key fingerprints, interactive session data such as keystrokes and responses, and information about non-interactive sessions such as commands executed and files transferred. Historically, this data has been inaccessible with traditional network monitoring techniques, but with a modification to the SSH daemon, this data can be passed directly to intrusion detection systems for analysis. The instrumented version of SSH is now running on all NERSC production systems. This paper describes the project, details about how SSH was instrumented, and the initial results of putting this in production.

Motivation

In the past few years, access methods have changed from clear text protocols like telnet and rlogin to encrypted protocols like SSH. As a result, intrusion detection systems have lost access to most of the data associated with interactive login sessions. Records show that the most common security threats to high performance computing and mass storage facilities, like NERSC, occur over the encrypted SSH channel. This is why the NERSC Security team has been searching for a way to regain access to this data for analysis by intrusion detection systems.

Because science users need to login and execute arbitrary commands on NERSC systems, the center grants several thousand researchers "shell" access. Unfortunately, this kind of access results in significant security risk.

The most common threat to NERSC comes in the form of compromised authentication credentials. This occurs when an attacker discovers a user's account name and password or public/private key pair and uses this information to gain unauthorized access to NERSC resources. The attacker will then attempt to escalate their privileges, steal authentication credentials from other users of the system, and jump to other systems. These attacks are extremely hard to detect because all of this activity is done with legitimate login credentials and under the cover of SSH encryption.

An effective response to this threat model must include the ability to monitor and analyze login sessions to differentiate attack behavior from normal user behavior. In the past, intrusion detection systems typically accomplished this by monitoring network traffic. The keystrokes and response data associated with login sessions could then be analyzed in various ways to determine the likelihood that a system and/or account had been compromised.

Unfortunately, the widespread adoption of SSH has made the analysis of login session activity by monitoring network traffic impossible because SSH encrypts all data before

it's transmitted over a network. This is not to say that the use of SSH and similar encrypted protocols should be discouraged. On the contrary, NERSC requires the use of SSH as it helps to prevent compromises in the first place. However, there is a clear need for intrusion detection systems to regain access to login session data.

Therefore, with this project, NERSC chose to modify the SSH daemon itself. Since the daemon manages unencrypted session data, it makes an obvious place to gain access to that data and pass it along to intrusion detection systems.

Design Constraints and Decisions

Before a modified version of SSH could be put into production on NERSC systems, a number of design requirements had to be met. These requirements guided the decisions that were made throughout the design process. We had three fundamental requirements:

Avoid the introduction of new bugs or compromised security: A critical requirement was that we were able to demonstrate with high confidence that our modified version of SSH was just as stable and secure as the OpenSSH code base we started with.

The "user experience" must be unchanged: We worked very hard to ensure that the newly modified version of SSH did not affect the way users interacted with NERSC systems. At NERSC, login messages notify users of this security monitoring. However, the user is not required to run a special version of the SSH client nor have we removed or changed any existing capabilities of OpenSSH.

Ensure minimal impact on system resources: Like most computational facilities, NERSC system resources including CPU time, memory, and network bandwidth are at a premium. This is why any additional demands made by an instrumented SSH must be insignificant compared to an unmodified SSH.

With this in mind, the following design choices were made:

OpenSSH Code Base: There are several implementations of the SSH protocol. We decided to use OpenSSH as a starting point for this project because it was already in use on most NERSC systems and is a very popular, open source standard. This minimized any disruption to the user experience and built on the already high level of security provided by OpenSSH.

In addition, OpenSSH is a widely deployed and well-maintained implementation of SSH. A large community of developers support the code and it has been well tested on many different architectures and is under constant scrutiny for bugs and vulnerabilities. By adding the instrumentation to this code base, the instrumented version of SSH can be easily and quickly deployed on most systems without change. This reduces the amount

of code that must be tested and maintained, which in turn, reduces the likelihood of introducing new bugs or compromising the inherent security of SSH.

By starting with the OpenSSH code base, we were able to include the high performance networking code developed at the Pittsburgh Supercomputing Center. This enhancement to OpenSSH improves performance over long haul networks for NERSC users. This means that researchers can move large data sets over wide-area networks more quickly.

As a result of this design decision and in support of the open software and OpenSSH communities, any changes made to the OpenSSH code base must be releasable under the same licensing. Thus, no libraries or other software elements could be introduced that would restrict the release of the instrumented version of OpenSSH.

Minimize changes to the code base: As part of a regiment of good coding practices, we chose to minimize changes to existing code as much as possible. We sought to take advantage of existing code and capabilities whenever it was possible. This reduced the amount of code we needed to support, reduced the likelihood of new bugs or vulnerabilities being introduced. In addition, this helped ensure that the user experience would remain unchanged. Rather than re-creating existing capabilities, we chose to rely on existing capabilities that were well tested, supported, and understood.

An example of this practice is our use of stunnel to transport data from a monitored system to an analysis system. Stunnel is a well-established and widely used utility for creating secure SSL-encrypted network communication channels that can be easily incorporated with other utilities. Rather than trying to recreate that capability and support it for our purposes, we simply pass data from the SSH daemon to a local stunnel instance which securely transports it to an stunnel listener on our analysis system.

Decoupled analysis: We tried to decouple the analysis of the SSH data from the SSH daemons and clients as much as possible. The normal functioning of SSH is not, dependent on the analysis subsystem. This has a number of significant advantages.

Analysis of this data can be somewhat demanding of system resources on large-scale systems, especially if that system that supports many users simultaneously. By offloading this processing to an entirely separate system, we minimized impact on user resources. Decoupling also allowed us to minimize the impact on our users should any part of the analysis fail. The SSH daemon can continue to function properly and gracefully recover should the system performing the collection and analysis of the data go completely off-line.

Decoupling also allows us to aggregate data from many systems onto a single system dedicated to managing and analyzing this data. This allows for a much more sophisticated analysis with a broader view of activity. Because this data is not stored on the system being monitored, it's far more difficult for an attacker to modify the data in an

attempt to "cover their tracks." The data can be stored on a system with far more restrictive access and security policies. This helps reduce the possibility of the instrumented version of SSH being subverted by an attacker.

Finally, offloading the analysis to a separate system allows for a fairly simple way to scale the capability. Multiple analysis systems can be deployed, each one performing analysis for a subset of the systems being monitored. Building on the communication capabilities built into the Bro intrusion detection system, noteworthy events can still be aggregated and correlated across multiple analysis systems.

Thorough Review and Testing: A critical system such as SSH cannot be placed into production without considerable review and testing. To ensure that all the necessary functionality and design criteria were met, we conducted formal design and code reviews. Knowledgeable reviewers were recruited within NERSC as well as outside of NERSC. By keeping the code open source, we are also encouraging critical analysis from the larger community of developers familiar with SSH. In addition, extensive testing is performed on each version of the code before it is put into production on NERSC systems.

System Architecture

A fundamental goal of this project is simply to provide a robust and secure communication conduit between SSH and a remote intrusion detection system. The architecture of the system can be divided into two parts. The "server side" includes the modified OpenSSH daemon(s) running on the systems to be monitored. The "analysis side" includes the intrusion detection system(s) as well as log maintenance running on a separate system. At NERSC, we use the Bro intrusion detection system which is easily modified to analyze SSH sessions. The conduit connecting the server and analysis sides is stunnel which is a widely available, general purpose, SSL implementation. No changes were needed to stunnel for this project. Only a very simple stunnel configuration file is provided as part of the distribution. Figure 1 shows the overall architecture.

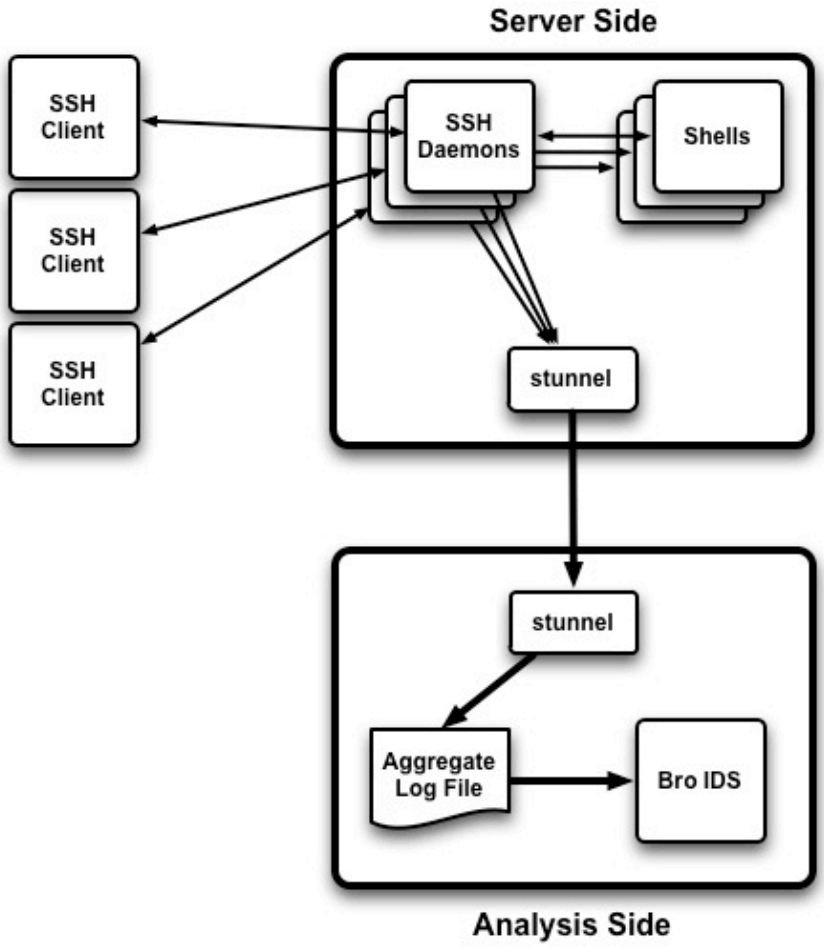

Figure 1: Overall Architecture

Server Side Subsystem

The server side subsystem is perhaps the most significant part of this system. This is where the majority of new code resides and where a mistake is most likely to impact users. For this reason, we spent a lot of time carefully crafting and testing this part of the system.

OpenSSH operates by running a "master daemon" that listens for new TCP connections and forks a separate copy of itself to handle each new connection. These individual connection or "session" daemons then handle authentication, communication, and command execution for the user. Once a user is authenticated, an interactive shell is executed and the daemon manages the standard input, output, and error data streams for the user.

Each SSH session supports multiple data streams or "channels" by multiplexing them over a single TCP connection. In the example of an interactive command shell, at least

three channels (standard input, standard output, and standard error) are multiplexed over the single TCP connection.

Our goal was to modify the daemon such that it would also forward copies of the data from each channel to our intrusion detection system while meeting our design constraints of minimizing the impact on the user as well as the system being monitored. To achieve this, we built, on existing infrastructure as much as possible. We used the existing buffer management capability in OpenSSH to implement separate buffers for each communication channel and used stunnel as our network transport mechanism. So ultimately, a target system looks as in figure 2.

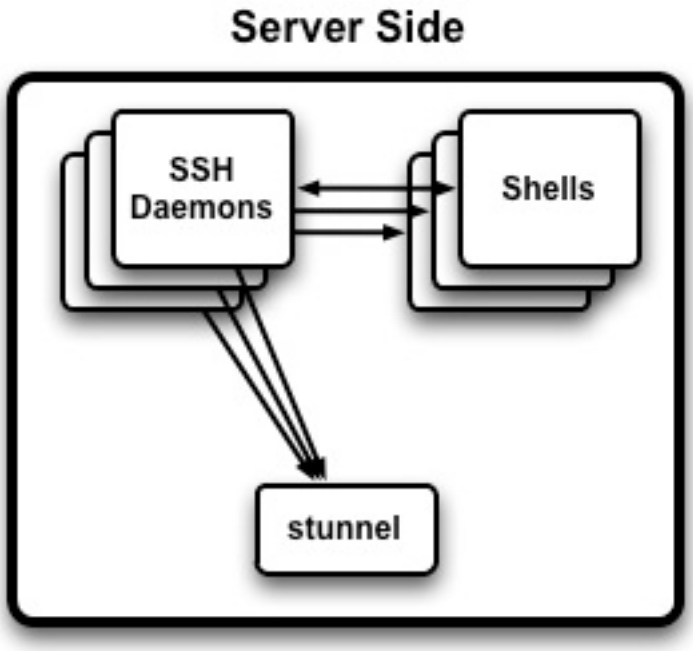

Figure 2: Server Side Architecture

Buffering

OpenSSH already provides very robust and efficient buffer management. We took advantage of this by calling for an additional buffer for each data stream and periodically flush each of these buffers to the listening stunnel as in figure 3. In general, these buffers are flushed upon receiving a new-line character. However, there are configurable limits on how much data will be copied to the stunnel.

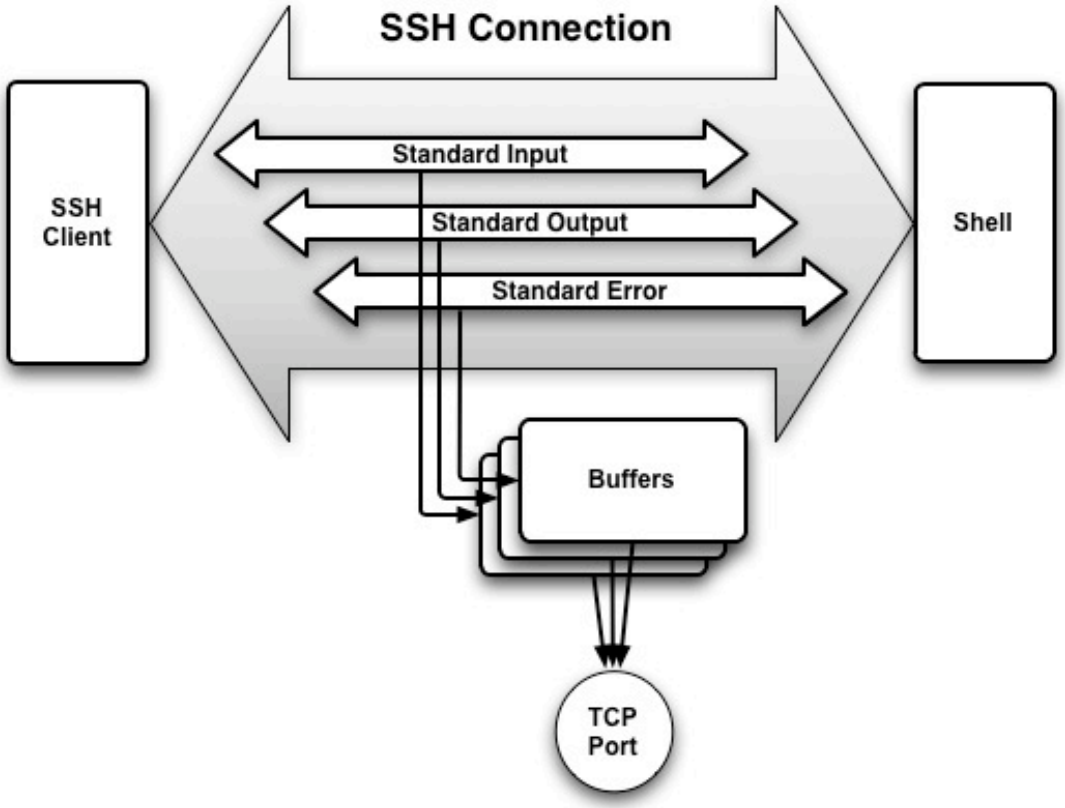

Figure 3: An SSH Connection

Some of the channels created in an SSH session are associated with a terminal or "tty device." These are typically the input, output, and error channels of an interactive login and tend to be of great interest for intrusion detection. Other channels that do not have an associated tty device are typically used for transporting files or "tunneling" other protocols like X windows or HTTP. However, sometimes these non-tty channels are used interactively. In fact, we've found that hackers will frequently do this to avoid certain kinds of logging. Because these two different types of channels (tty vs. non-tty) are used differently, we manage their buffers differently.

For buffers that have an associated tty device, two configurable limits are defined. First there is a line length limit (typically 1024 bytes). If this limit is reached before a new-line is seen, the buffer stops collecting new bytes, though count is still kept of how much data is ignored. The second limit is on the number of lines from the server without input from the client (typically 15). In other words, if 15 lines of text are sent from the server to the client without any input from the client, buffering is stopped until input is seen from the client again. This helps avoid clogging up the IDS with copious output from commands like "make world".

Channels that do not have associated tty devices are treated somewhat differently. Buffers are still flushed on new-lines and there is still a maximum line size (typically 1024). However, after a total of 1024 bytes have been seen, the ratio of non-printing characters to printing characters is computed. At a configurable threshold that ratio determines if the channel is believed to be carrying binary or readable data. If it appears to be binary data, then buffering is turned off to avoid slowing down large data transfers. If it looks like readable data, collection continues to a configurable maximum (typically 0.5 Megabytes). Table 1 is a list of limits for both types of channels.

tty?	Limit	Typical Value
yes	max line length	1024 bytes
yes	max server lines without client data	20 lines
no	max line length	1024 bytes
no	max data buffered if it's mostly printable text	0.5 MB
no	max data buffered if it's mostly non-printing text	1024 bytes

Table 1

Data Transport

A second issue is getting the data from the SSH daemon onto a security system for archive and analysis. This must be an entirely non-blocking process to ensure that the user experience is unaffected by downstream failures. To do this, we chose to use stunnel which is readily available on all of our systems. This means the data from each channel across multiple sessions and multiple hosts is multiplexed through stunnel onto the analysis system. Then, each channel must be reassembled on the analysis system and attributed to the appropriate host, session, and user.

A single stunnel process runs on each target system and they each connect with a single stunnel process on the analysis system. The SSH daemon communicates with the local stunnel process via a local TCP port and is entirely non-blocking. Whenever a write to the local TCP port fails, an error is logged, and an attempt is made to reestablish communication with the local stunnel process. If that also fails, the data is discarded and the SSH daemon simple continues normally. This ensures that each SSH daemon will both gracefully fail and gracefully recover from any problems with the stunnel process.

To facilitate reassembly of these data streams, each message is tagged with a unique identifier. These identifiers are a combination of the hostname, the port number that the master daemon is listening on, a random number assigned to the session daemon, and the channel number. The hostname allows multiple hosts to report to a single analysis

system. The port number allows for multiple SSH daemons on a single host, listening on different ports. The random session number identifies individual sessions and avoids problems with changing process identifiers (PIDs). Finally, each channel within a session is assigned a unique identifier by SSH.

In addition to the data from each channel, a number of other SSH events are sent through the same mechanism. For example, SSH daemon start and stop events, authorization events, session start and stop events, and sftp events are also logged. Each SSH master daemon also sends regular "heartbeat" events. Appendix B is a partial list of events.

Analysis Side Subsystem

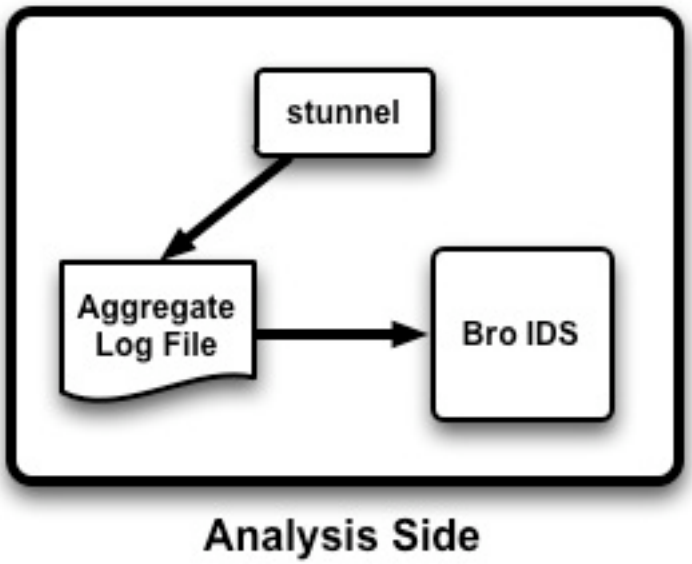

Figure 4: Analysis Side Architecture

The analysis side subsystem consists of the receiving stunnel process, a simple script to dump the received data into a flat log file, an existing tool that monitors the log file and generates Bro events, and Bro itself. Other analysis tools or intrusion detection systems could be easily employed since the data to be analyzed is in a fairly simple format.

At NERSC all monitored systems are connected to a single analysis subsystem. However, should the need arise, the monitored systems can be split among several analysis subsystems. This provides the ability to scale the system as needed. With most NERSC systems being monitored one analysis subsystem is easily handling all of NERSC.

Bro requires little change to take advantage of this capability. It already has a set of signatures for analyzing telnet/rlogin sessions. However, as experience is gained working with this new data, enhancements are being made to Bro to improve its ability to detect illegitimate and malicious activity while reducing the frequency of false positives.

Because of the independence designed into the two sides of the system (server side and analysis side), active development on the analysis side has no impact to SSH running on NERSC systems. So research can and does continue in new ways of analyzing this data.

Passwords and Sensitive Data

Perhaps the most controversial aspect of this project is the potential for exposure of sensitive data including passwords. We are very sensitive to this and have taken every precaution to protect the integrity OpenSSH and our users data. One area of concern is with passwords, as it is trivial for this instrumentation to capture passwords as they are typed. This is true both for the passwords used to login to our systems as well as any passwords entered after login. For example, if a user logs in to one of our systems and from there logs into another system, whatever credentials they used will be captured. Because some organizations want to avoid capturing that data, we have included an option that attempts to do just that.

While we did try to avoid putting too much analysis into the SSH daemon itself, this particular bit of data scrubbing seemed most effectively placed in the daemon. With this option turned on, if the word "Password:" or "Passphrase:" is detected on an output channel associated with a tty, immediately followed by something on the input channel associated with the same tty, then the string on the input channel is probably a password and will be discarded.

This of course, is not “fool proof.” There are applications and systems with different password prompts. However, the intent is to catch and ignore the majority of passwords and, for that, this heuristic is fairly effective.

Lessons Learned

The instrumented version of SSH has been running on production systems at NERSC for several months and we’ve been testing a second version that corrects some problems we discovered with the first version. Among the features added were capturing key fingerprints and improved handling of channels not associated with tty devices.

Recording SSH key fingerprints is very useful in several ways. Periodically, we, at NERSC, are given lists of key fingerprints that are either known to have been compromised and/or are known to have been used by attackers. It becomes a trivial matter to monitor for their use on our systems with this feature. In addition, observing the fact that a user has begun using a new key to access our systems is worth noting in Bro and possibly worth further investigation if there are any other signs of a compromised account. Finally, whenever we suspect that an account has been compromised, we require that the user destroy their old key(s) and generate a new one.

Looking at the finger prints of the keys they're using gives us a simple verification that the user has done this correctly.

Something we had not anticipated was the widespread use by attackers of "`ssh user@host sh -i`". This gives the attacker an interactive shell while bypassing some system logging, such as the history file. In our original version, it also bypassed our logging. We've fixed that as described earlier. While the use of a command like this is suspicious, it's not proof of an attack in and of itself. By capturing the keystrokes from the session that follows, our intrusion detection system is able to make a much more definitive assessment.

Results

Even without the improvements described above, our use of the instrumented SSH has been very helpful in detecting attacks as well as forensic analysis. A number of compromises would have surely gone completely undetected without the ability to look inside of SSH sessions. Many of the signatures used by Bro to detect hacker activity doesn't show up in any other logs.

Another powerful tool has turned out to be the forensic analysis of compromises. By examining what attackers do on our systems we're able to improve our intrusion detection system to be more sensitive to attacks with fewer false positives.

Appendix A shows an example of an attack that would likely not have been detected by any other means. Furthermore, we learned a great deal about how the attackers in this case operated since we were able to monitor communication between attackers.

Appendix A: An Example

This is an incident that took place in early March of 2009. The logs have been sanitized and reduced to show the interesting aspects of the attack. In this case, attackers were able to gain access to one of our user accounts. With that, they logged into one of our production systems and attempted to gain privileged access as well as a foothold across several systems within a cluster. Fortunately, they were detected, and did not gain privileged access on any of our systems.

Figure 5 is a partial listing of the accounting records associated with this attack. Of note here is the absence of anything that would indicate an attack. The commands issued by these attackers didn't deviate significantly from what we would expect of the legitimate user of this account. Based on the accounting logs and other logs from the systems under attack, nothing out of the ordinary would have been detected.

```
ACCOUNTING RECORDS FROM:  Wed Mar  4 19:05:01 PST 2009
COMMAND                      START    END          REAL      CPU        MEAN
NAME       USER     TTYNAME  TIME     TIME       (SECS)   (SECS)     SIZE(K)
mkscrdir   ---      pts/4   20:07:03 20:07:03     0.22     0.00        0.00
sed        ---      pts/4   20:07:03 20:07:03     0.00     0.00        0.00
awk        ---      pts/4   20:07:03 20:07:03     0.00     0.00        0.00
modulecmd  ---      pts/4   20:07:03 20:07:03     0.28     0.00        0.00
modulecmd  ---      pts/4   20:07:08 20:07:08     0.00     0.00        0.00
bash       ---      pts/4   20:07:08 20:07:08     0.02     0.00        0.00
w64        ---      pts/4   20:07:12 20:07:12     0.00     0.00        0.00
Uname      ---      pts/4   20:07:13 20:07:13     0.00     0.00        0.00
df         ---      pts/4   20:07:18 20:07:18     0.00     0.00        0.00
Ls         ---      pts/4   20:07:21 20:07:21     0.02     0.00        0.00
gcc        ---      pts/4   20:07:24 20:07:24     0.00     0.00        0.00
Bash       ---      pts/4   20:07:33 20:07:33     0.00     0.00        0.00
pico       ---      pts/4   20:07:35 20:08:28    53.94     0.14      175.00
Xlcentry   ---      pts/4   20:08:31 20:08:31     0.03     0.03     4408.00
cc         ---      pts/4   20:08:31 20:08:31     0.05     0.00        0.00
make       ---      pts/4   20:08:31 20:08:31     0.05     0.00        0.00
bash       ---      pts/4   20:08:47 20:08:47     0.00     0.00        0.00
pico       ---      pts/4   20:08:51 20:09:11    20.89     0.00      804.00
…
csh        ---      ?       20:07:03 20:13:20   377.12     0.00     3248.00
```

Figure 5: Excerpt from Accounting Logs

Figure 6 shows a partial listing of the alerts we received from our intrusion detection system (Bro). The first resulted from a shell command ("`unset HISTFILE`") which is very rarely issued by legitimate users. Because this is a shell command, it doesn't show up in any accounting or other log files. The next three alerts resulted from the attacker opening the source code of a known hacker tool in an editor. Various telltale strings were detected during editing session. Again, non of this would show up in any log files. These are all examples of leveraging what we learned from the "clear text era" when we could capture telnet and rlogin sessions by simply monitoring the network. Now, we use more sophisticated means to capture the data. But, we're looking for the same things.

```
Mar  4 19:55:44 SSHD_Hostile #5068 0 53183_host_22 6529
     user @ 0.0.0.0 -> 0.0.0.0:22/tcp
     command: unset HISTFILE
Mar  4 20:10:23 SSHD_Hostile #5068 0 53183_host_22 6529
     user @ 0.0.0.0 -> 0.0.0.0:22/tcp
     command: shellcode=( # by intropy <at> caughq.org
Mar  4 20:10:23 SSHD_Hostile #5068 0 53183_host_22 6529
     user @ 0.0.0.0 -> 0.0.0.0:22/tcp
     command: "x40x82xffxfd" # bnel <shellcode>
Mar  4 20:10:23 SSHD_Hostile #5068 0 53183_host_22 6529
     user @ 0.0.0.0 -> 0.0.0.0:22/tcp
     command: execve("/usr/bin/passwd",],{"EGG":egg+shellcode,"LC_TIME":bof})
```

Figure 6: Bro Alerts

Figure 7 shows an excerpt from one of the attackers' login session where they downloaded and attempted to compile a new hacker tool. We were able to capture the source code of this tool. Fortunately, the tool wouldn't compile, nor would it have worked if they had figured out how to get it to compile. We're now able to monitor for this tool and generate alerts if it's downloaded or used again.

```
1236230278.781065 #5449 0 77772_host_22 73661 data_server user@host:/tmp/.tmp>
     rcp lp@0.0.0.0:forker.c .
user@host:/tmp/.tmp> gcc -o f forker.c
forker.c: In function 'main':
forker.c:19: warning: incompatible implicit declaration of built-in function
     'exit'
forker.c:27: warning: incompatible implicit declaration of built-in function
     'exit'
forker.c:39: warning: incompatible implicit declaration of built-in function
     'exit'
```

Figure 7: A New Tool is Downloaded

A very interesting aspect of this attack was that it was carried out by two attackers that were logged in at the same time. They were apparently using a feature of the GNU Screen utility that allows two people to share a single login session and communicate with one another. We were able to capture that communication. Figure 8 shows one of the attackers generating a new SSH key pair and attempting to automate the process of populating the known_hosts file with the other systems in the cluster. Following that is a rather humorous discussion between the attackers about the proper way to do that without getting caught. Of course, their discussion moot since they hadn't anticipated our use of an instrumented SSH.

```
user@host:~/.ssh> ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_dsa.
Your public key has been saved in /home/user/.ssh/id_dsa.pub.
The key fingerprint is:
a9:e1:69:8b:b3:c0:78:da:8e:dc:c9:7e:52:c3:76:6e user@host
user@host:~/.ssh> ls
id_dsa id_dsa.pub known_hosts
user@host:~/.ssh> cat id_dsa.pub > authorized_keys
user@host:~/.ssh> rm -rf id_dsa.pub
user@host:~/.ssh> ssh -oHashKnownHosts= =yes 192.168.0.1
...
user@host:/tmp> what are you trying to do get ride of t pressing yes?
user@host:/tmp> clearly
user@host:/tmp> lol set known_hosts to dev null n00b
user@host:/tmp> that is such a hack and completely improper
user@host:/tmp> and a good way to lose a box if you forget to remove it
user@host:/tmp> nononosec phrack.org done? wn? its in issue 64
```

Figure 8: Attackers Discuss Evading Detection

Appendix B: Partial Event List

Event Name	Example of Event
sshd_exit	`sshd_exit string=15065_127.0.0.1_22 addr=127.0.0.1 port=22`
sshd_restart	`sshd_restart string=15065_127.0.0.1_22 time=1191884518.290978 addr=127.0.0.1 port=22`
sshd_start	`sshd_start time=1191884494.121733 string=15065_127.0.0.1_22 addr=127.0.0.1 port=22`
ssh_remote_do_exec	`ssh_remote_do_exec time=1191792262.11193 string=6823_127.0.0.1_22 count=6828 string=/home/scottc/development/ instrumentedSSHD/TEST/sftp-server` `ssh_remote_do_exec time=1191885521.640921 string=15172_127.0.0.1_22 count=15251 string=scp -t /tmp/t.c.2`
auth_fail	`auth_fail time=1192226229.309228 string=7079_127.0.0.1_2222 string=scottc string=publickey address=127.0.0.1 port=34475 address=127.0.0.1 port=2222 count=7081`
auth_ok	`auth_ok time=1192226229.309228 string=7079_127.0.0.1_2222 string=scottc string=publickey address=127.0.0.1 port=34475 address=127.0.0.1 port=2222 count=7081`
invalid_user	`nvalid_user time=1192168424.338979 string=21612_127.0.0.1_2222 string=four count=21732`
channel_exit	`channel_exit time=1191738571.917689 string=7710_127.0.0.1_22 ch=0 status=0`
data_client	`data_client time=1191700436.559975 string=5852_127.0.0.1_22 count=5854 ch=0 line=ls`
data_server	`data_server time=1192168686.518089 string=21612_127.0.0.1_2222 count=21818 ch=0 line=analyofChatSystems.ps foo.doc`

Event Name	Example of Event
data_server_sum	data_server_sum time=1191701255.308138 string=5852_127.0.0.1_22 count=5874 ch=0 31736 additional bytes not logged
new_channel_session	new_channel_session time=1191734989.401384 string=7404_127.0.0.1_22 count=0 string=subsystem count=7424
new_session	new_session time=1191735487.33513 string=7710_127.0.0.1_22 type=SSH2 count=7713
sftp_process_close	sftp_process_close time=1191886155.155476 string=15357_127.0.0.1_22 int=9 int=0
sftp_process_do_stat	sftp_process_do_stat time=1191886153.768947 string=15357_127.0.0.1_22 string=/home/scottc
sftp_process_fsetstat	-
sftp_process_fstat	-
sftp_process_init	sftp_process_init time=1191886143.344560 string=15357_127.0.0.1_22 string=scottc addr=127.0.0.1
sftp_process_mkdir	sftp_process_mkdir time=1191792377.270115 string=6829_127.0.0.1_22 string=/home/scottc/test
sftp_process_opendir	sftp_process_opendir time=1191886153.784007 string=15357_127.0.0.1_22 string=/home/scottc
sftp_process_open	sftp_process_open time=1191886167.559053 string=15357_127.0.0.1_22 string=/home/scottc/nessus.tar.gz
sftp_process_readdir	sftp_process_readdir time=1191886154.696938 string=15357_127.0.0.1_22 string=/home/scottc
sftp_process_readlink	-

Event Name	Example of Event
sftp_process_realpath	`sftp_process_realpath time=1191886143.406256 string=15357_127.0.0.1_22 string=.`
sftp_process_remove	-
sftp_process_rename	-
sftp_process_rmdir	-
sftp_process_setstat	-
sftp_process_symlink	-
sftp_process_unknown	-