Local System Security via SSHD Instrumentation

Scott Campbell
National Energy Research Scientific
Computing Center,
Lawrence Berkeley National Lab

scampbell@lbl.gov

ABSTRACT

In this paper we describe a method for near real-time identification of attack behavior and local security policy violations taking place over SSH. A rational is provided for the placement of instrumentation points within SSHD based on the analysis of data flow within the OpenSSH application as well as our overall architectural design and design principles. Sample attack and performance analysis examples are also provided.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information Flow Controls –

General Terms

Measurement, Security.

Keywords

SSH, keystroke logging, Bro IDS, Intrusion Detection, policy enforcement.

1. INTRODUCTION

The adoption of SSH as the defacto protocol for interactive shell access has proven to be extremely successful in terms of avoiding shared media credential theft and man in the middle attacks. At the same time it has also created difficulty for attack detection and forensic analysis for the computer security community. The SSH protocol and it's implementations such as OpenSSH [9] provide tremendous power and flexibility. Examples of this flexibility include authentication and encryption options, shell access, remote application execution and X11 and SOCKS forwarding. While the benefits gained vastly exceed the difficulties introduced by this protocol, the loss of visibility into user activity created problems for the security groups tasked with monitoring network based logins and activity.

The National Energy Research Scientific Computing Center (NERSC) is the primary open science computing facility for the Office of Science in the U.S. Department of Energy. It is one of the largest facilities in the world devoted to providing computational resources and expertise for basic scientific research, and has on the average 4000 users across seven primary computational platforms. The significant majority of user interaction involves interactive ssh logins. To address this lack of visibility into user activity on our high performance computing (HPC) infrastructure, we introduced an instrumentation layer into the OpenSSH application and feed the output into a real time analyzer based on the Bro IDS. This instrumentation provides application data such as user keystrokes and login details, as well as *metadata* from the SSHD such as session and channel creation details. This data is fed to an analyzer where local site security policy is applied to it, allowing decisions to be made regarding hostile activity. The data analyzer is based on the Bro intrusion detection system (IDS) [10] which provides a native scripting language to handle data structures, tables, timers to express local security policy. In this capacity Bro is being used as a flexible data interpreter. A key differentiator between the instrumented SSHD (iSSHD) and many other security tools and research projects is that iSSHD is not designed to detect and act on single anomalous events (like unexpected command sequences), but rather to enforce local security policy on data provided by the running SSHD instances.

A key idea is that the generation of data is completely decoupled from its analysis. The iSSHD instance generates data and the analyzer applies local policy to it. By using the Broccoli library [4], we convert the structured text data output by iSSHD into native bro events that are processed by the analyzer system [3]. Events, as their name implies, are single actions or decisions made by a user that are agnostic from a security analysis perspective. Bro processes these events in the same way as network traffic events, applying local security policy to interpret them as desired.

Local security policy can be thought of as sets of heuristics that describe (in this context) what behaviors are considered unacceptable or suspect. This behavior might be a command like "mkdir …", application usage like remotely executing a login shell, or tunneling traffic to avoid blocked ports. iSSHD was designed so that the installed SSHD instance would not need to be modified with every new threat. Instead, changes are made on the analysis/policy side as new problems are identified. This not only simplifies administration, but also allows experiments to be run on previous logs without significant work.

While NERSC has no explicit legal or privacy issues with intercepting communications on local systems, we recognize the importance of an informed user and staff population. To help address this we chose a policy of complete transparency. Each major group at NERSC was allowed representation in the design process and code review. As well, the entire user community was alerted to the changes by making announcements at User Group meetings and email notices. The complete source code is available to anyone interested and can be secured through the LBNL Technology Transfer Office.

The iSSHD project has been used in production capacity at NERSC for nearly three years on approximately 350 hosts. There are around 4000 user accounts with a daily average of 52,000 logins per day on the collective set of multi-user systems. In addition to the obvious security functionality, there are a number of other non-security purposes like debugging user problems or job analysis where having access to historical keystroke data has been quite beneficial in tracking down systems problems.

The reminder of the paper is structured as follows. In related work similar coding projects and tools are presented. Next the execution flow within an unmodified OpenSSH 5.8p1 instance is mapped out. This flow provides a way to determine the most effective points for instrumentation. In section four, the overall architecture and design goals are detailed including the integration of Bro into the process. Section five provides implementation

details describing the inherent tradeoffs between complete monitoring and resource limitations. Section six has examples of attacks and some rudimentary analysis. Finally future work and references are provided.

2. RELATED WORK

Related work can be generalized into several groups. These are research projects relating to SSH data access, hacker activities, and more generalized detection of SSH credential theft detection in the HPC environment.

The work most similar to our own involves the hacker community's use of backdoored SSHD instances to steal authentication credentials. In principle there is little difference between this behavior and the functionality provided by the iSSHD except in terms of the *breadth* of data provided. Statically backdoored OpenSSH code has been around since at least 1999 [14], and more recent versions are trivial to locate - see [15] for example.

Besides directly replacing the existing SSHD binary, there are at least three additional ways to access session data. The first is via direct access to a user's terminal devices by a privileged user. This can be achieved by one of dozens of small applications or as part of a larger kernel rootkit [18]. A more subtle approach is to interfere with kernel level behavior, thereby preventing a user space analysis of the terminals from giving away the access. Typically rather than just looking at terminal IO, input and output system calls are intercepted via a hidden kernel module. This information is transmitted to an analysis tool or recorded. There are innumerable examples of this approach within the rootkit community [11] as well as Honeypot implementations such as Sebek [13]. Finally you can interact with the running SSHD process by injecting code into it [16] or using process debugging to "jump" from their stolen user account to a potentially privileged session on another machine [17] [1]. These last two cases are somewhat subtle in that no changes to the actual static (non-running) binary are made.

There is a general class of SSH related security work focusing on user account theft via anomaly detection, both in terms of command sets as well as process accounting data. These include Yurcik [21] [22] and Joohan Lee et al. [5] who look for account compromises within the HPC domain via accounting and command analysis. Historically, there is a rich collection of research relating to account masquerading, with a nice write-up by Malek et al. [6]. This last class of ideas can be fed by or used with the iSSHD and incorporated into the sites overall intrusion detection design since they are orthogonal to the actual iSSHD.

3. SSH Application and Protocol

In order to identify the best places to place instrumentation within the SSH application, it is necessary to understand the code path taken by typical behavior as well as subtleties within the protocol.

From a historical perspective there are two individual (and incompatible) versions of the SSH protocol available. Tatu Ylönen created version 1 in 1995 as a replacement for the then ubiquitous telnet and rlogin protocols. OpenSSH emerged with the OpenBSD group taking up development after a number of organizational changes including the splitting of the Ylönen code base at one of it's last open source implementations. The SecSH IETF working group developed version 2 originally published in 1998 and in 2006 a revised version of the protocol was adopted as a standard in RFC 4250 (Protocol Assigned Numbers) [23], 4251 (Protocol Architecture) [24], 4252 (Authentication Protocol) [25], 4253 (Transport Layer Protocol) [26], 4254 (Connection Protocol) [27].

In terms of this analysis, all paths and descriptions assume the use of version 2 protocol since version 1 has suffered a number of pathological security defects [19] which reduce it's use to older and unusual cases. In the case of the actual code instrumentation, this assumption is not made and both version 1 and 2 provide nearly identical logging. Section 3.1 represents a general overview and relationship between RFC and OpenSSH structure. Section 3.2 takes this high level design and fleshes it out, providing a code path and rational for instrumentation locations.

3.1 SSH Application and Protocol Layering

For this initial description we avoid taking into consideration a number of details in order to focus on the overall flow of information and data. For a generic shell interaction a simplified diagram of the data flow might look something like Figure 1.

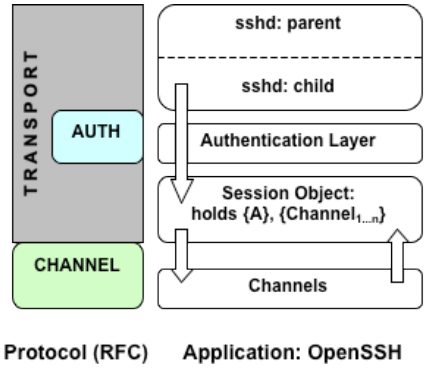

Figure 1: Application vs. Protocol design for typical SSHD session

Here Figure 1 is broken out into two columns – on the left there is the protocol layering as defined in RFC 4250-4254. The right side describes the application implementation of those layers. It is worth noting that the layers do not map 1 to 1 - in particular the role of the session object within the application, which according to RFC 4253 should be rolled into the transport layer. Here each application layer is a functional layer within the application, with the parent SSHD is represented as the top block. After a successful network connection is made, the process forks, and an *authentication context A* is created. This context is used for the lifetime of the login and is used to track a number of authentication based data values.

During the next step Key Exchange occurs, where the actual negotiation for a cipher, MAC and compression take place. First server authentication takes place via server/host key pairs. This authentication is transparent to the user if they have visited that SSHD server in the past. Assuming the server authentication is successful, algorithm negotiation for cipher and MAC takes place. Finally the short-lived session key is generated which is used to provide symmetric encryption for the data stream. This key is periodically re-negotiated after a given time or data volume passes. Since this is a reasonably well studied and logged area of

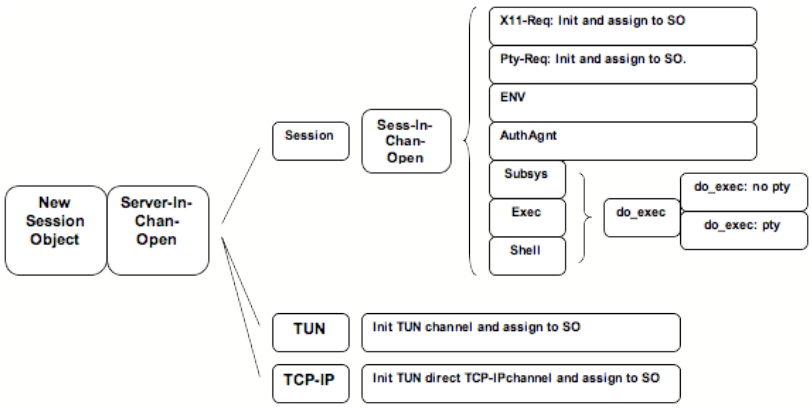

Figure 2: Internal SSHD Data Flow

the application, none of the exchange is recorded in the iSSHD besides what the system logs already do. If a strong reason to log the session crypto data could be come up with, there is no reason why it could not be done.

The Authentication Layer (unsurprisingly) provides the actual user authentication process. This process is extremely flexible with a number of options natively defined by the application as well as any generic PAM infrastructure. During the authentication process more than one type of authentication type can be examined so multiple fail and postpone events can be generated even for a successful login. Since we are less interested in the details of the authentication process than the outcome, there is little or no detailed logging from iSSHD except for the success/failure declaration as well as the authentication type being used. We apply the same rational to the key exchange process since in both cases relevant data can be preserved in regular system logs.

If the authentication process proves successful, a Session Object is created. This will be the primary container for not only the authentication context, but tty, X11 and channel data as well. The Session layer code also controls the mechanics of user login such as the login process, remote command execution, pty allocation and X11 forwarding.

The session object can create, use and destroy Channels. A channel can be thought of as a connection within the Session Object that has well defined semantics for data movement, windowing information, file descriptors and multiplexing capacity. Typically for a shell, you would allocate a single channel that holds the file descriptors for stdin, stdout and stderr. It is not unusual though to have many additional channels in use for X-windows, SOCKS forwarding and authentication agents. Data within a channel is not encrypted since it is contained within a session which already is. This is a critical point for monitoring which we will use to our advantage.

3.2 Common Code Paths During Execution

Now that the behavior of OpenSSH for a typical login has been described, we can more closely examine code paths for strategic places to insert instrumentation. Identifying those paths involved reading the source code as well as experimenting with sessions running in debug mode. Since the most common service for SSH to provide is remote shell login access, it was the initial target for both analysis and instrumentation. The execution path for this is identical to that shown in Figure 1, except for some additional details found in the session section. A location is considered a good candidate for auditing if (1) there exists a decision making branch where most or all connections traverse or (2) a final state is arrived at which contains security relevant information.

Figure 2 provides a more detailed set of code paths for nearly any use of OpenSSH. Here every box represents a transition between user privilege or application function and ultimately represents an event sent to the iSSHD analyzer. The creation of the Session Object (SO) begins on the left side and the path moves to the right till the users objective is reached. In it, a number of common paths that immediately stand out. The horizontal split between session and tunnel driven services is an obvious candidate for instrumentation. As a reminder, the session code tends to be more execution oriented – i.e. involved with the invocation of services, commands and shells. Since it is not unusual for an attacker to use a known tool or service in a way which is unusual, *how* we instrument the path is extremely important. Decision branches such as “session-in-channel-open” provide the path of what was asked for, and logging details at the end of the code path provide information regarding what was actually done. In any case, policy can be written to provide notice if the local site finds any part of the execution path objectionable.

Using the same rational, the lower half of Figure 2 provides the same opportunity to audit this behavior in some detail for tunneling and port forwarding activity. While not implemented in this design, it should be at least possible (though perhaps not practical) to access the forwarded data instead of just identifying the static forwarding requests.

The level of logging may seem excessive, but such detail can prove to be quite powerful for forensic analysis when combined with local site policy. Local site policy - described later in some detail - can act on specific session events like tunneling which may not be allowed by a centers usage policy. There is a huge benefit to be had in identifying the exact execution path of an attacker. Since it is not unusual for a tool like ssh to be used in a way which was not foreseen by the security community we tend to error on the side of caution.

4. SYSTEM ARCHITECTURE

For the iSSHD architecture, we selected three principles fundamental to the design and implementation process. If at any time one of these principles was in contradiction with the design, something was wrong with the architecture. The principles are:

1. **Avoid introducing stability or security problems**: We need to demonstrate with high confidence that our modified version of SSH is just as stable and secure as the original code base.
2. **Unchanged user experience**: The modified version of SSH can not affect the way users interact with NERSC systems, require a special version of the SSH client or application, nor remove any existing capabilities.
3. **Minimal impact on system resources**: System resources including CPU time, memory, and network bandwidth are at a premium. Additional demands made by the instrumented SSH must be insignificant compared to an unmodified SSH instance.

Based on these requirements, the following choices were made in the architecture and development plan:

1. **Use OpenSSH as the code base**. OpenSSH has an exceptionally good reputation and is already used on the multi-user production systems. In addition, we were able to add on the Pittsburgh Supercomputing Center's high performance OpenSSH patch set [12]. This provides significant gains in terms of bulk data transfer performance.
2. **Minimizing changes to the code base**. As part of the project we made an active attempt to minimize the number of changes to the original code. In addition, we chose to use other tools and capabilities rather than write them ourselves. An example of this would be the use of stunnel [20] rather than attempting to write an add on to ssh for our own data encryption.
3. **Decoupled Analysis**: Taking our experience from the Bro IDS, we chose to fully decouple the analysis from the generation of the ssh instrumentation data. To do this it was necessary to remove any dependencies between the running iSSHD and the back end analysis. This is done by making all writes to the back end non-blocking stressing that a failure of the analysis infrastructure should result in the loss of security data before an interrupted user experience.

The overall design of the iSSHD can be broken out into two sections – the event generation within the running iSSHD process, and the logging and analysis that compares those events against local policy. Much of §3 was involved with the thought process that took place before the coding started. With that in mind, we turn to the actual design and implementation of the system itself.

It should be noted that the core of the analysis side currently exists as a log repository with scripts feeding live data to the Bro IDS. The use of Bro is not technically required since the file exists as structured text, which provides the ability to feed the information to any another tool. We will assume for the remainder of the paper that Bro will be used.

4.1 Server Side

The iSSHD server is modified OpenSSH code that provides events for further logging and analysis. Within the SSHD application (as described in §3.2) there are ideal locations where we extract information about user activity. Such information includes login and authentication data, session and channel creation, port forwarding, and keystroke/application data. This data is normalized in terms of data types as well as being formed into structured text. This text is then written to a local socket (provided by stunnel) using a non-blocking descriptor. Details of this process follow.

For events, a number of data types are defined. Not unexpectedly these types map approximately with the native data types defined by Bro. This includes the usual integer, string and count as well as more network specific types like address and subnet. In order to encapsulate arbitrary data, both unstructured string and binary data is URL encoded using the stringcoders library [8]. This mechanism is used in reproducing user activity since even simple terminal sessions include Unicode characters and colors. An additional benefit of URL encoding is to safely encapsulate traffic that might be directed toward either the analysis system or the terminal session of the individual doing the analysis. Original versions of the instrumentation attempted to remove non-printing characters from the recorded data, but information loss and textual confusion ultimately pointed toward the URL encoding solution as a better option.

As has been already described, the most basic unit of information provided by iSSHD is called an *event*. Events, as their name implies, are single actions or decisions made by a user that are agnostic from a security analysis perspective. Lines typed by the user as well as logins and channel creations are all examples of events.

For event creation, all activity points to a single function. This reduces confusion and creates a single point for information gathering. A sample function call looks something like:

```
s_audit("channel_new", "count=%d count=%i
        uristring=%s", found, type, t1buf);
```

The function s_audit is the general event handling operation within iSSHD. There are three sets of arguments that it takes – the first is just the event name (in this case "channel_new"). The second defines data typing for the Broccoli interpreter and has printf() type structure. Any additional arguments define the data associated with the event type. Here, 'found' is the index for the free channel slot, 'type' defines the type/state of the channel (ie: SSH_CHANNEL_LARVAL, SSH_CHANNEL_AUTH_SOCKET), and 't1buf' is the URL encoded channel name such as *server-session* or *auth socket*. After passing through the Broccoli interpreter, an event named "channel_new" will be created with three arguments. Note that there is no indication that the channel creation is considered a good or bad thing – such a determination will be left to the analysis side of the iSSHD.

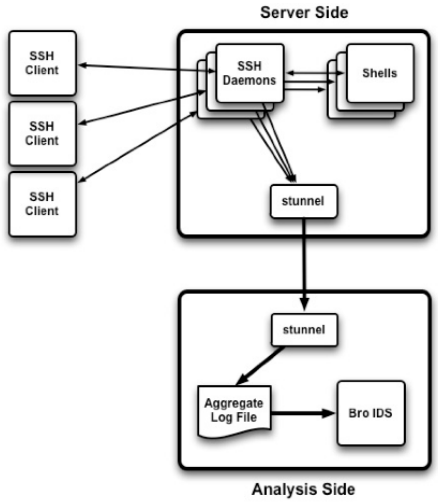

Figure 3: Overall iSSHD Architecture

Data provided by keystroke logging presents an interesting problem in that the content can be of arbitrary length, and will contain non-printing ASCII characters. To avoid inefficiencies, we cache keystroke data in a channel buffer queue using the native channel buffer types until a new line character is seen or data volume is exceeds a threshold. In situations where too much data is generated on the server side (such as large compile runs), the value of this additional data is almost zero. To address this, we adopted the same idea as used in the network Time Machine [7]: specifically that most security sensitive data and events tend to cluster them selves to the beginning of interactive sessions. By making the distinction between interactive sessions (where there are roughly the same order of magnitude of client initiated data events as server) and highly asymmetric connections (with dozens or hundreds of server data events per client data event), we can avoid excess resource consumption by the iSSHD. This is one situation where it was necessary to build logic into the code running in the iSSHD. Table 1 provides cutoff values for both normal tty channels as well as channels not bound to a tty. For the situation of non-tty communications, the ratio of printing to non-printing characters is also looked at to avoid needlessly copying binary files.

Table 1: Default cutoff values for user and server data.

TTY	Details	Default Value
Yes	Max line length or line count for client input between server inputs.	15 lines, 64k bytes
Yes	Max line length or line count for server input between client inputs.	15 lines, 64k bytes
No	Initial sample value (ISV) before determining binary data.	1024 bytes
No	Maximum data in total for either client or server inputs.	.5M bytes
No	Percentage of non ascii-printing characters, after ISV, allowed for continued sampling.	30%

For example if a user (client side) types 'ls -l' in a normal tty based login, the iSSHD would provide the server echo of 'ls -l' as well as the next 14 lines or 64k bytes of server side output (whichever is exceeded first). The line/byte count is reset every time client data is processed. The cutoff values are modifiable at compile time and are set somewhat conservatively since the assumption is that there is a large number of iSSHDs feeding into a single analysis system.

4.2 Data Analysis

Data analysis consists of any component except for the iSSHD itself. Practically it can be thought of as the stunnel as well as the bro instance and related policy.

The stunnel is not particularly interesting in that we are using it to transport data from an open file descriptor on the iSSHD side, to the analyzer host. Since this is just a simple implementation of a well-known application, we will focus on the details provided by the policy.

The bro policy is designed to track individual sessions and whatever activity is contained within them - normal shell sessions, remote code execution or subsystem invocation. Each session is defined by the start of the ssh connection and continues through any activity until that connection ends. The series of events for a routine login looks something like Figure 4 when printed directly from the iSSHD.

Each of these lines represents an event and the data associated with it. Policy can be written to trigger on specific events, their data, or both. Of obvious interest is a users keystroke data and the systems response. Since we have direct access to near real time keystroke information, we look for extremely unlikely - and highly suspicious - character sequences. These might include known toolkit signatures, abnormal root shell prompts for /bin/sh, or any other unexpected commands. Sets of commands that individually do not represent a significant interest, but which are suspicious in total represent the second type of alarm. These two categories are defined by two sets of signatures – the first for commands or strings worthy of immediate notification, and the second for sets of these commands or strings present in the user session.

In order to circumvent logging from the system login() facility, it is not unusual for attackers to remotely execute a shell via 'ssh host sh -i' . This style of reconnaissance has become so common during hostile activity that we made sure that it could be simply alarmed and all interactive data recorded. To address this, traffic on non-tty channels had to be tracked and analyzed since the tty invocation is part of the standard unix login() facility. Since data on these channels can include binary streams, the ratio of ASCII to non-ASCII packets is monitored. If after a pre-defined sampling window this ratio exceeds a threshold, further monitoring on that channel is dropped. We have experienced tremendous success in logging both the remote execution of shell binaries as well as monitoring commands to and from such occurrences.

```
SSHD_CONNECTION_START

AUTH_KEY_FINGERPRINT uristring=0x.. uristring=DSA
AUTH_INFO uristring=Accepted uristring=scottc
uristring=publickey

SESSION_NEW uristring=SSH2
CHANNEL_NEW  count=0 count=SSH_CHANNEL_LARVAL
uristring=server-session
SERVER_INPUT_CHANNEL_OPEN uristring=session
CHANNEL_NEW count=1 count=SSH_CHANNEL_AUTH_SOCKET
          uristring=auth+socket
SESSION_INPUT_CHANNEL_REQ count=0
               uristring=auth-agent-req@openssh.com
SESSION_INPUT_CHANNEL_REQ count=0 uristring=pty-req
SESSION_INPUT_CHANNEL_REQ count=0 uristring=shell

CHANNEL_DATA_SERVER count=0
 uristring=%0ALast+login:+Sat+Jan++8+14:45:31+2011
CHANNEL_DATA_CLIENT count=0 uristring=exit
CHANNEL_DATA_SERVER count=0 uristring=exit
CHANNEL_DATA_SERVER count=0 uristring=%0Alogout

SESSION_EXIT count=0 count=28221 count=0
CHANNEL_FREE count=0 uristring=server-session
CHANNEL_FREE count=1 uristring=auth+socket

SSHD_CONNECTION_END
```

Figure 4: Event series for a shell login.

The final area to explicitly mention is the ability of iSSHD to intercept authentication data. When considering our options for recording passwords during authentication, we ended up having to carefully balance the utility and risk of retaining the data. In the context of a forensic analysis, a password might be tremendously valuable if used in a legally sanctioned criminal investigation. On the other hand having such valuable credential information in the logs represents a huge risk in and of itself, even without taking into consideration passwords recorded for other institutions by users transiting local systems. Ultimately the decision to record passwords is left to the local site as a configure time option so that it cannot be adjusted without recompiling the iSSHD. Since it is not unusual for sites to share lists of known compromised keys via their fingerprints, public keys presented for authentication can be compared to a list of known bad keys and alarms raised when a suspicious key is seen.

4.3 Event Details

As previously suggested, events generated by the iSSHD are without any sort of predefined notions of good or bad since it is the role of the analyzer to interpret these events. These events can be roughly grouped by function, with types auth, channel, session, server and sshd. In addition to these, the sftp subsystem also has a number of events associated with it.

The example presented in Figure 4 shows the series of events seen in a "normal" login. Two of the most important in terms of monitoring and analysis are CHANNEL_DATA_CLIENT and CHANNEL_DATA_SERVER. These events provide unfiltered client keystroke and server echo/response data. If a user types "lz<backspace>s<enter>" you would see "lz%7Fs" from the client side and "lz%08+%08s" from the server side in the URI encoded data. The characters '%7F' and '%08' are the control characters delete and backspace respectively which can be seen from standard ascii definitions. Since we assume all user-generated data is potentially hostile, we reduce the possibility of accidentally interpreting control characters in the process of reading and interpreting the data by storing it in an encoded form.

Each event also includes timestamp, server id (process ID + server hostname + listening port), client id (32 bit random number) and interface address list. This information is tracked by the analyzer bro policy as a locally unique session identifier - for example #12345. This session id will remain constant for any activity attached to that users session. This event data is missing from figure 4 (and the other session figures) to allow for better clarity. Additionally, data is maintained for the channel id so session #12 might contain channel 0 and channel 1. Since the session object holds channel objects, the session id (ex #12) is the same and the channel identifier will be different. A small number of events, mostly connected to the running sshd daemon itself, do not have all these fields since there is no notion of client session to be had when the daemon is starting or emitting a heartbeat event.

5. RESULTS AND PERFORMANCE DATA

Presenting quantifiable results for the iSSHD is somewhat complicated since there is no control data to base comparisons against. Since the number of incidents is not large, checked against a control group or varied across sites, it presents more of an anecdotal story than an effective hypothesis test. Using iSSHD we have identified approximately three-dozen instances of stolen credentials. Most of them are not particularly interesting, but at the same time we can catch this class of attacker *before* anything can get interesting. Because of this, we will present an unusually qualitative analysis for the security and policy enforcement capabilities. For performance data we will look at a number of measurements comparing iSSHD to an unmodified version running on the same hardware. In addition we will also provide a simple analysis of aggregate user events that would be extremely difficult (or impossible) without the data set.

Besides detection, the iSSHD provides considerable insight into the tactics, skill levels and motivations for many of the attackers on our systems. In many cases the forensic logs quickly provide a clear indication of the success, skill level and threat presented by an intruder.

5.1 Sample 1: Remote Shell Invocation

Figure 5 provides a textbook example of a "classic" stolen credential and local exploit attack. This user (resu) made the mistake of having the same password for at least two sites - NERSC and the remote site that was compromised. Here the attacker remotely executes a shell to log in, then attempts a local linux exploit. Note that because of the shell invocation, communications are not via the normal tty interface - a technique detailed in §4.2 .

Details follow with some of the data fields removed for clarity.

```
1  AUTH_OK resu keyboard-interactive/pam
   1.1.1.1:52073/tcp > 0.0.0.0:22/tcp
2  NEW_SESSION SSH2
3  NEW_CHANNEL_SESSION exec
4  SESSION_REMOTE_DO_EXEC sh -i
5  SESSION_REMOTE_EXEC_NO_PTY sh -i
6  NOTTY_DATA_CLIENT uname -a
7  NOTTY_DATA_SERVER Linux comp05 2.6.18-…GNU/Linux
8  NOTTY_DATA_CLIENT unset HISTFILE
9  NOTTY_DATA_CLIENT cd /dev/shm
10 NOTTY_DATA_CLIENT mkdir ...
11 NOTTY_DATA_CLIENT cd ...
```

```
12  NOTTY_DATA_CLIENT wget
                  http://host.example.com:23/ab.c
13  NOTTY_DATA_CLIENT gcc ab.c -o ab -m32
14  NOTTY_DATA_CLIENT ./ab
15  NOTTY_DATA_SERVER [32mAc1dB1tCh3z [0mVS Linux
                    kernel 2.6 kernel 0d4y
16  NOTTY_DATA_SERVER $$$ Kallsyms +r
17  NOTTY_DATA_SERVER $$$ K3rn3l r3l3as3:
                    2.6.18-194.11.3.el5n-perf
18  NOTTY_DATA_SERVER ??? Trying the
                    F0PPPPpppppp__m3th34d
19  NOTTY_DATA_SERVER $$$ L00k1ng f0r kn0wn
                    t4rg3tz..
20  NOTTY_DATA_SERVER $$$ c0mput3r 1z aqu1r1ng n3w
                    t4rg3t...
21  NOTTY_DATA_SERVER !!! u4bl3 t0 f1nd t4rg3t!?
                    W3'll s33 ab0ut th4t!
22  NOTTY_DATA_CLIENT rm -rf ab ab.c
23  NOTTY_DATA_CLIENT kill -9 $$
24  SSH_CONNECTION_END 1.1.1.1:52073/tcp >
                  0.0.0.0:22/tcp
```

Figure 5: Remote shell invocation example.

We can see a number of clear indicators that something is going on which is not normal user activity. First is the interactive session on a non-tty channel created by remotely executing a shell (line 3-5). Second, the unset HISTFILE command and the creation of a directory called "..." under /dev/shm (line 8-10). Finally the exploit is downloaded, compiled and (unsuccessfully) run (line 12-21). Highlighted text represents commands and output that as part of the default policy distribution are considered sufficiently unusual or dangerous to warrant alarming on.

5.2 Sample 2: Cluster Reconnaissance

This example is one of the more complex and educational that we have captured, providing a clear snapshot of the methodology and tactics taken by a pair of hackers looking into our systems. Since they are sharing a common login via the GNU screen utility we can see the interaction between them and get an understanding of their *methods and communication,* something quite difficult under normal conditions. While there are several thousand lines of interaction from the event, space limitations force us to only include a small chunk of the most interesting (and amusing) lines.

```
1   DATA_CLIENT /sbin/arp -a
2   DATA_SERVER b@n:~> /sbin/arp -a
3   DATA_SERVER comp05 (192.168.49.94) at
    00:00:30:FB:00:00 [ether] PERM on ss
4   DATA_SERVER b@n:~>
5   DATA_CLIENT oh wow
6   DATA_SERVER b@n:~> oh wow
7   DATA_SERVER b@n:~> /sbin/arp -an |wc -l
8   DATA_SERVER 9787
9   DATA_CLIENT rofl hax it hacker
10  DATA_SERVER b@n:/u0> sorry, im gonna s roll
                a cigarette and smoke it, y
11  DATA_SERVER b@n:/u0> then im gonna come back
                and try to hack ok ?
12  DATA_SERVER b@n:/u0> i am gonna go for one
13  DATA_SERVER b@n:/u0> you cant smoke inside?
                terrible
14  DATA_SERVER b@n:/u0> its f cold as f***
```

Figure 6a: Initial communication and Note: removal additional server fields, time and session id

The text from the screen session is marked in blue, and event names are once again bolded. The overall behavior can be broken out into several sections. In Figure 6a, lines 1-10, arp tables are used to identify locally attached systems. In this case the large number of them (9787) seems to cause the need for a few moments thinking about how to proceed. This is one of the initial indicators that the attackers are not just blindly running tools. It also indicates that they are probably in the western hemisphere.

```
1   DATA_CLIENT hmm cd .. ;ssh-keygen -t
2   DATA_SERVER b@n:~/.ssh> hmm
3   DATA_SERVER b@n:~/.ssh> cd ..
4   DATA_SERVER b@n:~/.ssh> ssh-keygen -t dsa
5   DATA_SERVER Gen pub/private dsa key pair.
    ...
6   DATA_CLIENT ls
7   DATA_SERVER b@n:~/.ssh> ls
8   DATA_SERVER id_dsa id_dsa.pub known_hosts
9   DATA_CLIENT cat id_dsa.pub > authorized_keys
10  DATA_SERVER b@n:~/.ssh> cat id_dsa.pub >
                authorized_keys
11  DATA_CLIENT ssh -oHashKnownHosts=yes
                192.168.0.1
12  DATA_SERVER b@n:~/.ssh> ssh
                -oHashKnownHosts=yes 192.168.0.1
13  DATA_CLIENT cat > ssh_cn010onf
14  DATA_SERVER b@n:~/.ssh> cat > ssh_config
15  DATA_CLIENT cat known_hosts | grep -v
                192.168.0.1
16  DATA_SERVER b@n:~/.ssh> cat known_hosts |
                grep -v 192.168.0.1 > tmp
    ...
17  DATA_SERVER b@n:/tmp> what are you trying to
    do get ride of t pressing yes?
18  DATA_SERVER b@n:/tmp> clearly
19  DATA_SERVER b@n:/tmp> lol set known_hosts to
    dev null n00b
20  DATA_SERVER b@n:/tmp> that is such a hack
    and completely improper
21  DATA_SERVER b@n:/tmp> and a good way to lose
    a box if you forget to remove it
22  DATA_SERVER b@n:/tmp> nononosec phrack.org
    done? wn? its in issue 64
```

Figure 6b: Generate local key pair and populate across NFS share, attempt generic NFS type attacks via suid 0 program.

```
1  DATA_CLIENT ps axuw |grep snort
2  DATA_SERVER ps axuw |grep snort
3  DATA_SERVER b 36684 0.0 0.0 2740 564 pts/10
   S+ 20:39 0:00 grep snort
```

Figure 6c: Looking for IDS processes.

By Figure 6b discussion has indicated a familiarity with insecure multi-host NFS file systems - interestingly, they did not attempt to use NFSShell. From here (lines 4-5) the pair generate a pass-phraseless ssh key to use across the systems sharing the home file system, once again indicating a familiarity with shared file systems and how they can be used. They grapple a bit with configuration issues and interestingly use the HashKnownHosts option to obscure records left in the known_hosts file. Figure 6c provides an example of IDS detection.

Ultimately this pair logged in to 19 local systems and never managed to get root access. The dialog here is as long as it is in order to convey the relative sophistication and interesting method of the attackers.

5.3 Performance Data

There are numerous points of reference in comparing the performance of the iSSHD with an unmodified OpenSSH. In this case we will be looking at aggregate remote command execution time, time to copy binary and ascii files, cpu usage for general activity, and memory usage for the child process.

This command set is run remotely via remote execution with the system time command providing information about total execution time, system and user cpu usage. We recognize the differences between remotely executing a script containing commands and manually running them. Ultimately we chose to run via the script for repeatability and ease of use since tools such as Expect do not provide additional functionality.

	Remote Exec	SCP Binary	SCP ASCII
SSHD	42.78 [0.05]	9.85 [0.11]	0.70 [0.01]
iSSHD	43.03 [0.18]	9.85 [0.15]	0.69 [0.02]

Table 2: Run time values for three tests, values in seconds, standard deviation in brackets. Average remote command execution time increases by 0.6%.

For Table 2 column 1, "Remote Exec" is a set of 13 remotely executed commands including normal user activity like ls, touch configure and make. From a simple ratio test, the iSSHD takes in total about 0.25 seconds more to run or about 0.6%. This indicates that the *average* behavior of interactive shell commands should not be adversely affected, but limited variations in keystroke responsiveness could be lost. Given the way that large volume logging is done (as described in §4.1), this is not at all surprising. For the additional columns in Table 2, we have the time to completion values for using scp to transfer a medium size ASCII file as well as a medium size binary file. In this case, medium size is on the order of 100MB. In each case the additional overhead caused by the memory copy and transmit did not provide a significant (or measurable) difference in the measured time. In this measurement, the same file was moved from one directory on the local system to another 40 times in a row. The task was then repeated with the iSSHD to reduce the influence of variable overhead and caching.

Looking at CPU usage for the same two data sets demonstrates differences in application behavior. First, the system CPU dominated the total time by ~ 4:1 for total CPU time per transaction. This is not surprising given that the majority of this activity is driven by read() and write() calls as well as polling during periods of inactivity.

Figure 7 shows the relationship between execution time and CPU time for both sets of test runs. One thing to notice is the slope of the linear regression curve. Total CPU usage decreases since the faster you move a constant set of data, the harder the data must be pushed during the (shorter) time window. The product of the two terms as a histogram we see a very tight set of values (s^2), implying this relationship.

The final metric is memory use, which ends up being quite consistent both in terms of native and iSSHD when looking at results from the data generation scripts. Within SSHD, there are a limited number of ways that memory becomes allocated once a session completes initialization – the most common being internal data buffering and channel creation. In both of these cases the size growth is minimal for the modifications made since data buffering from interactive sessions are cleared once they are written to the stunnel socket.

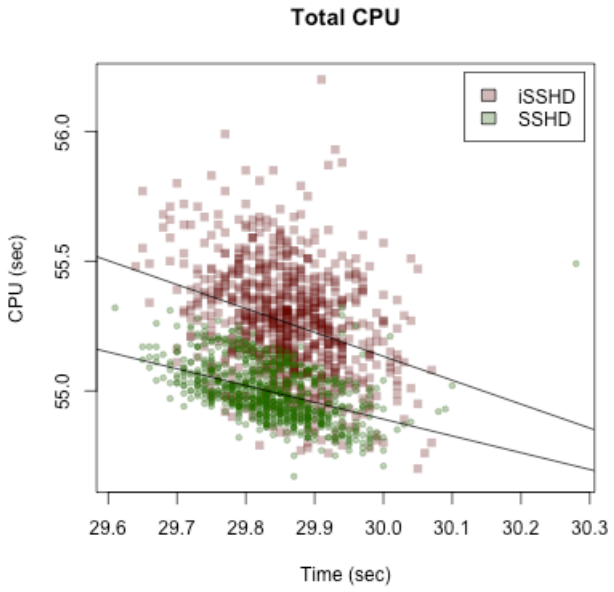

Figure 7: Total CPU time vs. length of transaction time for test data runs against iSSHD and native SSHD.

The overall conclusion is that the changes made to introduce instrumentation into iSSHD do not have a significant impact on performance or usability.

5.4 Overall Observations

Overall the iSSHD project has provided insight into probably three-dozen compromised user accounts since 2009. In each of these cases it was possible to not only quickly determine the success of the attack, but also get exploit tools and code used.

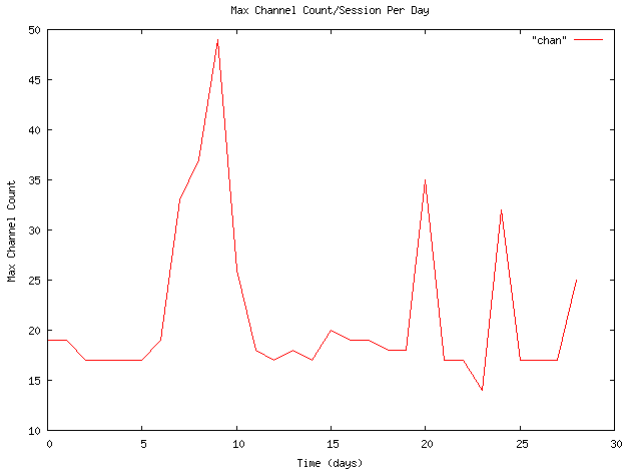

Figure 8: Distribution of maximum channels/session for November 2010.

As suggested in the introduction, the iSSHD also provides a tremendous source of measurement data as well. We have not yet

begun to fully explore this avenue, but there is no technical reason why we could not use this to identify needs for the user community. An example of this would be to systematically explore port-forwarding behaviors to see if we could deliver network services differently. Besides problem solving, the measurement data can also provide an interesting repository of pure research data. Figure 8 provides an example of the maximum channel count per session per day during November 2010). It is interesting to note that some users are exceeding 50 channels per session – in this case the majority of this is web browsing. This might be done (for example) to visit social networking sites blacklisted by a users local institution. This has interesting security repercussions to be sure.

6. FUTURE WORK

Since the iSSHD is relatively new, there is a great deal of learning going on with regard to what information is useful as well as available. There are several areas that we are actively looking into for future releases. The first is the detection of local terminal session hijacking as described in §2 by [17][18]. The second is the extraction of keystroke data from the X11 x-terminal data-stream, which is currently opaque. There is currently some prototype work completed for the session hijacking (detailed below), while tapping into the X11 stream represents a possible way to look into the protocols being tunneled over the ssh channel.

6.1 Local Session Hijacking

In the available literature and toolkits, there are a number of ways that a local attacker can tap into a running session and "reach across" the network to access further systems and resources. In particular this can be done to elevate privilege if the user has gained root access on the external system, or to hop over one time password authentication. We are familiar with examples of the later.

In the SSH-Jack application [17], ptrace is attached to the ssh client process, finds the channel setup code, then patches the memory to request a remote shell attached to a local TCP socket. The user running the ssh client is completely unaware that this is happening since they are running under a different set of channels in the same user session. We are hoping to look for an unusual ssh_session2_open() call and match it to the expected state for a normal session to help identify this attack. Regardless of this, the entire communications from the new channel will be logged and analyzed in the same way that normal user activity is.

A more common attack involves a local root user looking to jump off the compromised host through some sort of multi-factor authentication. In many cases this involves the opening of the victim users terminal descriptors for standard in, out and error then writing data directly into the sockets. The running ssh is not even aware that anything is amiss since it is just transiting data normally. We are looking to use the Linux *inotify* interface [2] to monitor and log additional file open events on the terminals file descriptors. This is still in its prototype phase.

7. CONCLUSION

We have presented an instrumented version of the OpenSSH application that allows for a local site to log and analyze user activities on local HPC resources. This analysis can be used to enforce local security policy with respect to SSH usage, which would otherwise be difficult or impossible with normal tools.

8. ACKNOWLEDGEMENTS

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

I would very much like to thank Tom Limoncelli for his help in the paper shepherding process.

9. REFERENCES

[1] "Trust Transience: Post Intrusion SSH Hijacking" to Blackhat Las Vegas, Adam Boileau

[2] Dow, Eli M., *Monitor Linux file system events with inotify*, IBM Linux Test and Integration Center, http://www-28.ibm.com/developerworks/linux/library/l-inotify.html?ca=dgr-lnxw07Inotify, 2005.

[3] H. Dreger, C. Kreibich, V. Paxson and R. Sommer, Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context, Proc. Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) 2005.

[4] C. Kreibich and R. Sommer. Policy-controlled Event Management for Distributed Intrusion Detection. 4th International Workshop on Distributed Event-Based Systems (DEBS'05), 2005, Columbus/Ohio, USA

[5] Joohan Lee, Muazzam Siddiqui, "High Performance Data Mining for Network Intrusion Detection Using Cluster Computing", International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), MIT Cambridge, November 2004

[6] Malek Ben Salem, Shlomo Hershkop, Salvatore J. Stolfo. "A Survey of Insider Attack Detection Research" in Insider Attack and Cyber Security: Beyond the Hacker, Springer, 2008

[7] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson and F. Schneider, Enriching Network Security Analysis with Time Travel, Proc. ACM SIGCOMM, August 2008.

[8] Nick Galbreath, stringencoders: A collection of high performance c-string transformations, http://code.google.com/p/stringencoders/

[9] Open SSH Project, http://www.openssh.org

[10] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time. Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998

[11] http://packetstormsecurity.org/files/view/42556/phalanx-b6.tar.bz2

[12] Chris Rapier and Benjamin Bennett. 2008. High speed bulk data transfer using the SSH protocol. In *Proceedings of the 15th ACM Mardi Gras conference,* (MG '08). ACM, New York, NY, USA, , Article 11 , 7 pages. DOI=10.1145/1341811.1341824 http://doi.acm.org/10.1145/1341811.1341824

[13] Know Your Enemy: Sebek. The Honeynet Project, November 2003 https://projects.honeynet.org/sebek

[14] SSHD Backdoor Homepage, http://emsi.it.pl/ssh/

[15] http://packetstormsecurity.org/files/author/5480/ : Backdoored version of OpenSSH 4.5p1 that logs passwords to /var/tmp/sshbug.txt.

[16] http://packetstormsecurity.org/files/view/45228/ssheater-1.1.tar.gz : SSHeater is a program that infects the OpenSSH daemon in run-time in order to log all future sessions and implement a backdoor where a single password, chosen by the user, can log into all accounts in the system. There's a log parser included in the package that can display authentication information about sessions as well as play the session just like TTYrec/play.

[17] http://www.storm.net.nz/projects/7

[18] http://datenterrorist.wordpress.com/2007/07/06/tty-sniffer-fur-linux-24/

[19] SSH CRC32 attack detection code contains remote integer overflow, Vulnerability Note VU#945216, United States Computer Emergency Readiness Team, http://www.kb.cert.org/vuls/id/945216

[20] W. Wong: Stunnel: SSLing Internet Services Easily. SANS Institute, November 2001.

[21] W. Yurcik, X. Meng, and N. Kiyanclar. NVisionCC: A visualization framework for high performance cluster security. In ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC), 2004.

[22] W. Yurcik, Chao Liu, “A first step toward detecting SSH identity theft in HPC cluster environments: discriminating masqueraders based on command behavior”, CCGRID '05 Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01

[23] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Protocol Assigned Numbers”, RFC 4250, January 2006

[24] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Protocol Architecture”, RFC 4251, January 2006

[25] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Authentication Protocol”, RFC 4252, January 2006

[26] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Transport Layer Protocol”, RFC 4253, January 2006

[27] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Connection Protocol”, RFC 4254, January 2006

Appendix 1

This is an abbreviated list of iSSHD events current as of January 2011. The event name is in the left column and a summary of returned data types is on the right. All events are processed in the current public release of the policy set. The description in the Returned Data column does not include all the default fields described in §4.3.

Authentication Events	Returned Data
auth_info	userid, auth type, success, source IP, dest IP
auth_invalid_user	userid
auth_key_fingerprint	fingerprint of pub key
auth_pass_attempt	userid, password

Channel Events	Returned Data
channel_data_client	URI encoded client data
channel_data_server	URI encoded server response
channel_data_server_sum	Data skipped by heuristics
channel_free	id of closed channel
channel_new	id, type, remote name
channel_notty_analysis_disable	printable/non-printable ratio for non-tty channel exceeds set ratio
channel_notty_client_data	URI encoded non-tty client data
channel_notty_server_data	URI encoded non-tty server data
channel_pass_skip	id of channel where pass skip happened
channel_port_open	type, listening port, path/hostname, remote ip, remote port
channel_portfwd_req	hostname, listening port
	type, listening port, path/hostname, remote ip, remote port
channel_post_fwd_listener	listen port, path/hostname, host port, type
channel_set_fwd_listener	type, wildcard bind, host, port to connect, listen port
channel_socks4	id, path/hostname, host port, s4 command, username
channel_socks5	id, path/hostname, host port, s5 command

Session Events	Returned Data
server_input_channel_open	chan_type, channel, window size
session_do_auth	session type, state
session_exit	chanid, parent pid, status
session_in_channel_req	chanid, chan type, session id
session_remote_do_exec	parent pid, command
session_remote_exec_no_pty	parent pid, command
session_remote_exec_pty	parent pid, command
session_request_direct_tcpip	orig host, orig port, dest host, dest port, session id
session_tun_init	tun type, can id
session_x11fwd	display as string

SSHD Events	Returned Data
sshd_connection_end	remote ip, remore port, local ip, local port, client id
sshd_connection_start	remote ip, remote port, local ip, local port, parent pid
sshd_exit	local ip, local port
sshd_restart	local ip, local port
sshd_server_heartbeat	select value
sshd_start	local ip, local port