# Experiences with Intrusion Detection in High Performance Computing

**Scott Campbell, Jim Mellander**
*Lawrence Berkeley National Laboratory*
*National Energy Research Scientific Computing Center*
*scottc@nersc.gov, jmellander@lbl.gov*

**ABSTRACT***: The application of Cybersecurity in HPC has historically been considered as counterproductive to research in Open Science. NERSC proposes a systematic way of determining where Cybersecurity-significant data should be sampled as well as an overview of our analysis methodology. To demonstrate this we look at the Bro Intrusion Detection system as well as the instrumented SSHD currently in use.*

**KEYWORDS:** Intrusion Detection, Security

## 1 Introduction

Successful High Performance Computing requires a combination of technical innovation as well as political and operational experience to balance out the many (sometimes contradictory) pressures encountered in this field. This is particularly true with respect to operational cybersecurity that, at best, is seen as a necessary evil, and considered as generally restrictive of performance and/or functionality. As a representative high-performance open-computing site, NERSC has decided to place as few roadblocks as possible for access to site computational and networking resources. The apparent tension between efficient system operation and cybersecurity is from our perspective a false dichotomy – in fact, we have come to see the role of cybersecurity as an enabling technology to facilitate maximum performance and functionality.

Rather than providing a new tool, method or schema for analysis, this paper presents a survey of our evaluation and design practices. The motivation for doing this is not merely to share ideas that we have found to work, but also to examine what can be done regarding recurring system security problems found within a representative HPC site as well as identifying potential bottlenecks we envision in near-future systems and network designs.

When an application, host or cluster comes under attack, the activity can typically be broken out into two parts - the initial attack and the followup hacking steps taken if local access is gained. Because NERSC's role in open-science means that we provide a rather porous face for the scientists using our facility, the initial attack can come from almost anywhere. There is no site-wide firewall as it would interfere with high performance networking. Firewalls are in place, however, as part of the infrastructure protecting staff and core internal resources. Once an attacker lands on a system - something that we *assume* as possible from the beginning – the attacker's behavior tends to become somewhat better defined and easier to identify. As an analogy, although it is simpler to look for ripples on a pond than for the rock that makes them, we strive to look for both.

Determining what to look for, and where to look for it is probably the most complicated cybersecurity problem in our environment. No HPC site, particularly within the domain of open science, has a simple task in identifying and reducing the risk associated with attacks against systems and infrastructure. Two fundamental issues - the rapid, changing pace in both networking and computational systems themselves as well as an increasing sophistication of attackers, exacerbate effectively addressing this problem.

## 2 Solution Methodology

In addressing the question of what, how and where to monitor in order to identify security incidents, NERSC follows a methodology stressing the culture of open science - data gathering and measurement, repeatable testing and careful analysis. This methodology assumes a significant understanding of systems, networking and computer science theory, so it can hardly be represented in a red light - green light style.

A summary of our methodology is as follows:

1. Gather as much raw data as possible, focusing on high yield areas, as defined below.

2. Filter and organize this data for efficient analysis.

3. Analyze data, comparing to expected and normal activity.

The *source* of data can be envisioned in a way analogous to protocol layering. Data types can interact with each other, and may be useful only in terms of the local system by which they are derived, or be wholly independent. This data can also be derived from sources outside individual systems like network traffic, to inter-system data like batch scheduler logs. A table providing examples of these data types can be found in Table. 1.

| | |
|---|---|
| **Per Host** | process accounting, application (ex Apache), sshd |
| **Inter-System** | batch scheduler, xcat logs |
| **Cross-Site** | network data, syslog, dns logs[1] |

**Table 1**, Sample data types and sources.

Note that Table 1 is not to be construed as a comprehensive list.

While each layer has different characteristics and data source(s), the same general approach and methodology can be applied to all of them. These are discussed in more detail in the next section.

### *2.1 Design Patterns in Data Gathering*

A *design pattern* is a general solution to some sort of commonly occurring problem. It is a description or template for how to solve a problem that can be applied in many different situations. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system. [DP11] For the class of problems we are addressing here, we have a pair of design patterns that help us identify optimal locations for data gathering by focusing detection efforts on data types on interfaces between zones of trust and control as well as data types that are more homogeneous across the site. Note that a design pattern does not describe the activities taken with respect to the data gathered.

> **DP1**: Information rich data is located at boundaries between trust layers within an organization. This concept scales from the inside/outside boundary all the way down to within individual systems. For example, we gather network data not only from the external border interface, but also at firewall boundaries and across the network attaching public facing web services to NERSC proper. Trying to get inter-system network data would not only be technically very difficult, but also be low in information value as the computational systems exist within the same area of trust.
>
> Within a system, this pattern can be applied to (for example) process accounting information where individual records are not nearly as interesting as those found from transitions between unprivileged to privileged state.
>
> **DP2**: Permeated data - this pattern embodies data types that exist throughout the site in a more or less homogeneous state. The most common example of this design pattern would be data derived from a syslog server. Data that matches this characteristic tends to have tremendous breadth of scope but is also unstructured and takes a fair amount of work to generate actionable results.

Each source layer has different characteristics, but the interesting feature from an analysis perspective is the location of the layer. The mechanics of gathering are strongly dependant on the type of data and the expected rate. Border traffic can go to network tapping equipment, application logs can be syslogged or accessed via a local script and process accounting information can be copied to a global file system. Note once again that this step is solely concerned with identifying and gathering data that may potentially be of forensic interest.

### *2.2 Filter and Normalize*

Once the data is gathered, it becomes necessary to process and reduce the data volume to a level where it can be efficiently accessed and analyzed. A useful way to conceptualize this step is to identify it with the idea of data abstraction. As an example, network packets can be bunched together and abstracted as connections. Web traffic can be broken down into a small number of fundamental components such as URI, return code, header values, etc. For SSH logins, corresponding abstractions would be items such as account names, failure/success of authentication, source address, etc. Much of the detail in the data is abstracted without changing the basic information content.

A principal tool NERSC uses to performs this abstraction is the Bro Intrusion Detection System [PA98] - the architecture (detailed in §3.1) is broken into two parts with the event generation component doing the mechanical part of the abstraction via event generation.

[1] Syslog and DNS data are included in this section since they are gathered collectively for the entire site.

Reduction is not done by deleting data (since we may need to go back and look at something new), but by filtering and abstracting *from* the data stream and (whenever possible) recording the raw data for as long a time as practical. A useful example of this concept is our analysis of SSH login data. SSH login success and failure messages are extracted from the stream, parsed into a regular form and passed to the analysis framework detailed later in §2.3 .

The final step is to take the abstracted data and canonicalize it so that it is in normal form and can be machine processed. Without this last step it becomes quite difficult to process and compare results. Even something as seemingly straightforward as ssh login results as delivered by a syslog process require that the various formats provided by individual vendors and software versions be parsed to provide the same data types in a predictable and automated manner.

### *2.3 Analysis and Presentation*

Up until this point, the information that we have gathered has not been tainted with any notion about it being benign or hostile – all that has been done is that we have abstracted and normalized the data into a standardized format. This purposeful decision reflects our desire to provide reproducible evidence-based decision-making.

In the Analysis step, local site security policy is used to evaluate the standardized data. In a practical sense, this means that network data is analyzed for (among other things) scanning activity, and HTTP semantics and content are analyzed for hostile activity. Non-network traffic such as syslog or ssh keystroke data is digested in much the same way. As this is the point of monitoring, there are fairly complex details at this stage. Notwithstanding, this is ultimately a reasonably simple task. At this stage, we can also analyze based on divergences from historical or statistical norms.

The Presentation process is tied into the analysis step – we aggregate the information created during the analysis process and present it for human consumption. This stage tends to be one of the most important given the volume of data and data types that need to be looked at. As expected, we are constantly evolving presentation capabilities in an attempt to access the increasing information volumes.

The interface between the raw agnostic information and the policy defines what we see as either a known problem or interesting enough to warrant reporting on.

## 3 Solution Examples

In order to more completely explain what we mean by the generalities described in §2, we will provide two fleshed-out examples. In each case, we expect to provide enough background information to fully understand both the tool and the threat that the tool was designed to address.

Our first example is the Bro Intrusion Detection System. We will describe both its use as a general analysis and reporting tool and the architectural changes made for high bandwidth situations. The second presents the instrumented Secure Shell daemon, which provides real time analysis of user keystrokes, command execution and ssh metadata information like TCP port forwarding.

Given our space limitations only general descriptions will be provided, but adequate information is available in the reference section to help answer most typical questions.

### *3.1 The Bro Intrusion Detection System*

Intrusion Detection Systems (IDS) are fundamental security tools for any large publicly accessible network. This is particularly true when running a large multi-user system with thousands of remote accounts and a tremendous diversity of running software.

The Bro intrusion detection system is fairly complex, but can be described in general terms without much difficulty. From the network analysis perspective, traffic is received via a standard *pcap* interface, and processed into a series of *events* by an *event engine* integral to Bro. An *event* is a basic functional unit within bro and is a major mechanism Bro uses to communicate internally and externally. These events are not assigned any sort of value in terms of security bias (and are often referred to as *agnostic*), but instead are passed over to the policy side of the application via an *event-handler* (essentially a function that is called when an event is generated). For instance, when a SYN packet is seen by Bro's network engine, generally a *new_connection* event is triggered which handles setting up state for that connection via a new_connection event handler.

Event-handlers are written in a domain-specific scripting language designed for (near) real time network traffic analysis. The scripting language provides a huge advantage over pattern matching schema - constructs such as data structures, timers, tables and asynchronous events are all built in and the Bro distribution contains thousands of lines of policy-script that cover most typical configurations. Network state, including domains, IP addresses and counts of significant actions can also be maintained. This scripting language is used to translate a local site's security policy into an actionable mechanism that maps directly to the ideas presented in §2.3.

Recent changes have allowed bro to offer a useful mechanism for interaction with external applications via the event mechanism. Events can be registered with Bro

as external, and the *broccoli* library, along with appropriate language bindings, is used to allow Bro to send data to, and trigger an action of, an external program. Similarly, external programs can use *broccoli* to communicate data and trigger an *event* within Bro using any well structured information (such as normalized syslog data). This allows bro to be used as a general state engine so that in addition to, or in place of, network traffic, *any* sort of event can be processed and analyzed.

Such a capability also allows for asynchronous processing of data, while allowing Bro to simultaneously perform real time analysis. Data can be assembled in an event for further processing, and sent to an external process, with Bro then continuing until the event returns. When a result is available (perhaps by performing a database query or name resolution), it can signal bro via an event, and bro will pick up the results, and can act on them appropriately. The ability of a Bro instance to share state information and operate asynchronously is key to the success of the Bro Cluster, described next.

### *3.1.1 The Bro Cluster*

HPC sites are often on the bleeding edge of network bandwidth usage, due to the user-base's increasingly voracious appetite for data. This presents a substantial challenge to IDS operations, as it is important to effectively monitor this ever-increasing bandwidth without impeding traffic flow or missing cybersecurity-significant data. Although impressive hardware advances have, to some extent, allowed ever-increased monitoring functionality, IDS hardware is increasingly less able to fully monitor the high-bandwidth traffic patterns now common in the HPC community. To address this problem, LBNL partnered with cPacket [CP] to create an intelligent load-balancing hardware front-end which would allow traffic to be distributed amongst a series of worker nodes in order to allow for continued analysis at high bandwidths.

Although high-speed interconnects are common in the HPC community, IDS operations have generally depended on off-the-shelf hardware to monitor and communicate. The key concepts guiding the architectural decisions for the clustering of Bro systems on commodity hardware are:

1. Intelligently split the traffic in real time, so that each individual monitoring node only sees a portion of the traffic, but sees sufficient traffic to independently operate.

2. The data transferred between the nodes ought to be the analysis results of the individual nodes, and not raw traffic or any significant subset of it.

Decision 1 precludes the naive round robin approach to the traffic-distribution problem, since if System A sees the initial SYN packet of a TCP connection, and System B sees the response SYN/ACK, inter-worker node communication must necessarily take place to have a full view of the session, and thus performance suffers. A sensible approach, therefore, is to ensure that at least all the traffic matching a 5-tuple [protocol, source host, source port, destination host, destination port] is mapped to a single analysis node (we temporarily put aside the possibility of a large flow between two systems taken an inordinate amount of bandwidth, and thus overwhelming the analysis node). The Bro Cluster approach is, in practice, to send all traffic matching each 2-tuple [source host, destination host] to a designated worker node for three reasons. First the operation is quite cheap as a simple hash is appropriate. Second, operational results have shown that there are few hot spots (spikes in CPU utilization) within the worker nodes, and when they exist it is not typically pathological. Finally the operation is symmetric for most hashing operations so that hash(src,dst) = hash(dst,src) - this fulfills the design requirement to keep all connection state on the same worker node.

### *3.1.2 Performance Characteristics*

The most complete published performance characteristics for the Bro Cluster can be found in the Vallentin et.al NIDS Clustering paper [2]. This paper covers both the decision making process in creating the Bro cluster as well as performance evaluations on 10 Gbps internet links.

Figure 1 provides two perspectives on the cluster scaling problem by monitoring the amount of user CPU time used per second. Figure 1 (left) shows that nine of the ten backends (all except node 8) show very similar distributions, indicating quite similar CPU loads. Across these nine backends, the largest mean CPU utilization was 10.0%, and the largest standard deviation $\sigma$ = 4.8%, reflecting that both the loads and the load fluctuations leave ample headroom for increases in traffic. However, backend node 8 shows a notably different density shape (mean 10.7%, $\sigma$ = 5.7%). Upon examining the trace processed by node 8, the slice contained a single TCP connection which makes up 86% of the trace's total bytes (33 GB of 38 GB!). Just by being assigned this one connection, node 8 receives a significantly larger share of the overall traffic (other nodes on average received 6.5 GB). Note, though, that pretty much any flow-based traffic distribution scheme will wind up introducing this disparity, since it manifests at even the finest flow-based granularity. However, even so, node 8's CPU load stayed well within a manageable range (below 30% for 99.5% of the time).

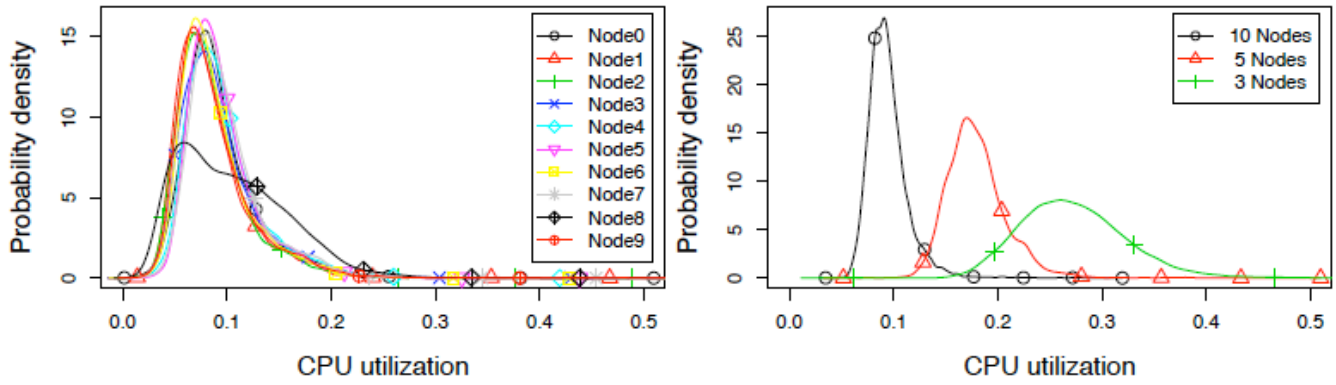


**Fig. 1**. Probability densities of backend CPU load (left), and probability densities for varying numbers of backends (right). Figure taken directly from [2].

Figure 1 (right) plots the CPU utilization for setups with 3, 5, and 10 worker nodes. For each run, we first averaged the one-second CPU samples across all nodes. We then plotted the probability density of these mean CPU loads. In the plot we see that the load indeed scales nearly linearly with the number of nodes: the mean load for 3 nodes is 27.4%, for 5 nodes it is 18.0%, and for 10 nodes it is 9.4%, with the corresponding values of σ being 5.5%, 3.0%, and 2.0%.

Note that the symmetric distribution of loads indicates a reasonably effective hashing mechanism for traffic distribution.

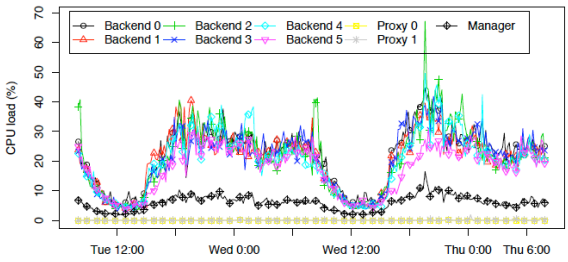


**Fig. 2**. CPU Load on U.C. Berkeley cluster. Taken directly from [VA07].

### *3.1.3 Results*

The clustered version of the Bro IDS can provide deep (and normally expensive) analysis of high volume traffic without significant packet loss or exceptional expense. There are already installations at 10 Gbps, and plans exist for moving to 100 Gbps late this year.

### *3.2 Instrumented SSHD*

While the adoption of SSH as the standard form of communication between users and HPC services has proven to be extremely successful in terms of avoiding traditional keystroke logging and man in the middle attacks, it has also created problems in terms of attack detection and forensic analysis for the computer security community. While the benefits gained vastly exceed the difficulties introduced by this protocol, the loss of visibility into user activity created problems with the security groups tasked with monitoring network based logins and activity.

To address the lack of visibility into activity happening on our multi-user HPC infrastructure, we introduced an instrumentation layer into the OpenSSH application and tied the resultant data set into a real time analysis using the Bro IDS. This instrumentation provides both application layer data like keystrokes and login details, as well as metadata from the sshd such as session and channel creation details. This data is then fed to an analyzer, where it is interpreted based on local site security policy. A key differentiator between the instrumented sshd (iSSHD) and many other security tools and research projects is that the iSSHD is not designed to detect and act on single anomalous events (like unexpected command sequences), but rather it is designed to enforce local security policy on data provided by the running sshd instances.

The data analyzer is based on the Bro intrusion detection system [PA98]. This IDS normally takes network traffic, turns it into agnostic events and processes it via local policy script. By using the *broccoli* library, it is possible to convert structured data into serialized bro events that can be handed to the actual analyzer system [HD05]. This separation of policy and data generation mechanism provides the ability to take remotely generated events and

use the native scripting language to handle data structures, tables, timers and local security policy. In this capacity, we are principally using Bro as a powerful state engine which is being fed raw/agnostic events from the iSSHD application.

It is worth mentioning that we make no attempt to hide the fact that iSSHD is installed. An announcement was made to the user community, and an opportunity to provide feedback was provided. In addition the version string clearly provides indication of a non-standard installation.

### *3.2.1 Architecture and Design*

The design for the iSSHD was driven by a series of principles that focused more on not degrading the user experience than on any sort of security directive. These principles were:

1. **Avoid instability or security problems from our code**: 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 must 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.

Some results from this seem quite intuitive such as the use of OpenSSH [3] as the code base. Others, like decoupling analysis from data collection, were somewhat more involved and required testing and experimentation to reach our goals. The final design incorporated a three-part strategy that completely separates the data collection, transfer and analysis from one another. This is quite similar to the design of Bro described in §3.1 that also decouples the creation of agnostic (typically network) events from the analysis that enforces local security policy.

In addition to adding our desired auditing functionality, we also added the Pittsburgh Supercomputing Center's high performance OpenSSH patch set [RB08]. These patches provide significant gains in terms of bulk data transfer performance, which was seen as an additional win.

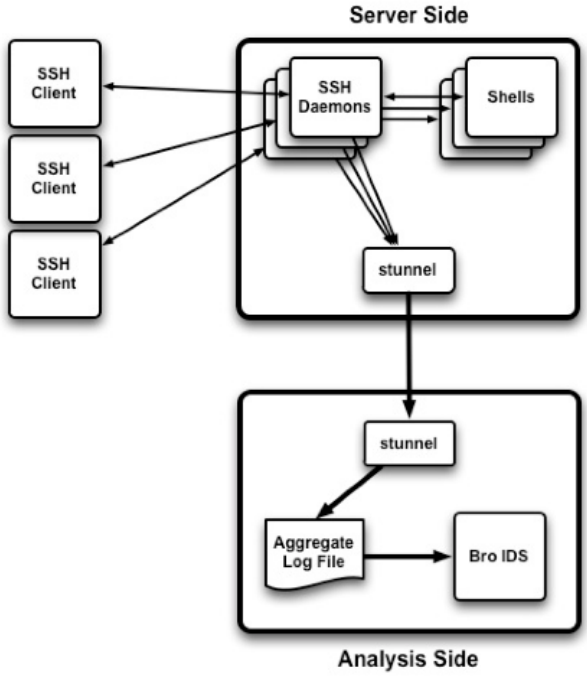


**Figure 3**, Architecture of iSSHD.

As we see in Fig. 3, there are three components of the iSSHD solution. First, we see the server engine which exists within the iSSHD process space and generates the raw event data; secondly, stunnel [4] is used to transport data from the iSSHD process to the analyser in a non-blocking manner; and thirdly, the analysis procedure – note that no changes are needed to the SSH clients, and operationally, the entire process is transparent to the user.

When a user logs in using iSSHD, a session is created on the analysis side and information about the activity is logged. If behaviour is indicative of known suspicious activity such as running a known bad command)\, remotely executing a shell, or performing some action like connection proxying, an alarm is propagated, and appropriate action taken. In addition, the entire session is logged and available for forensic analysis in the event that the data requires further review.

Table 2 provides a subset of the total available information types available from iSSHD.

**Connection**

| | |
|---|---|
| SSHD_CONNECTION_START | Log connection 4-tuple and local interface addresses. Create session id. |
| SSHD_CONNECTION_END | Close session id. |

**Authentication**

| | |
|---|---|
| AUTH_INFO | General authentication event - type, result id etc |
| AUTH_INVALID_USER | Log id, source IP |
| AUTH_KEY_FINGERPRINT | Log RDA/DSA and fingerprint. Can test against known bad values. |
| AUTH_PASS_ATTEMPT | Attempted password, SSH v. 1 |

**Session/Channel**

| | |
|---|---|
| CHANNEL_DATA_CLIENT<br>CHANNEL_DATA_SERVER<br>CHANNEL_DATA_SERVER_SUM | Data created by or returned to the ssh client. There are non-tty versions of these as well. The final event happens when data is skipped. |
| CHANNEL_NEW<br>CHANNEL_END | Creation or destruction of channel within the ssh session. |
| CHANNEL_PORTFWD_REQ<br>CHANNEL_SOCKS4/5 | Sample port forward and socks request events. |
| SESSION_REMOTE_DO_EXEC<br>SESSION_REMOTE_EXEC_PTY<br>SESSION_REMOTE_EXEC_NO_PTY | Events tied to remote command execution |
| SESSION_REQUEST_DIRECT_TCPIP<br>SESSION_TUN_INIT<br>SESSION_X11FWD | Session events related to the directed TCP/IP, tunneling and X11 forwarding of traffic. |

**Misc**

| | |
|---|---|
| SSHD_START<br>SSHD_EXIT | Start/exit of the sshd process. |
| SSHD_SERVER_HEARTBEAT | Periodic message sent from running iSSHD to identify that (1) it is alive and (2) it has not been replaced. |

**Table 2**, iSSHD sample event set.

### *3.2.2 Performance Data*

Based on points (2) and (3) of the Architecture and Design principles, it was expected that there would be little impact on performance for running iSSHD. Table 3 confirms our expectations, as far as non-interactive sessions are concerned. We are still gathering data for interactive shell access since it is difficult to take a "real" user session and run it on two sshd instances at the same time.

| | Remote Exec | SCP Binary | SCP ASCII | SFTP ASCII |
|---|---|---|---|---|
| 5.8p1 NoMod | 3.45 [0.10] | 9.85 [0.11] | 0.70 [0.01] | 1.01 [0.39] |
| 5.8p1 NERSC | 3.31 [0.12] | 9.85 [0.15] | 0.69 [0.02] | 1.56 [0.34] |

**Table 3**, iSSHD performance measurements.

The data provided by keystroke logging presents an interesting problem in that the content can be of arbitrary length, and will probably contain non-printing ASCII characters. As a performance enhancement, 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 exceeded. In situations where too much data is generated (such as large compile runs), the volume of data is huge and the value of the data is almost zero. To address this we adopted the same idea as used in the network Time Machine [MS08]: 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 (dozens or hundreds of server data events per client data event), we can avoid transmitting excess data from the iSSHD. This was one situation where it was necessary to build logic into the code running within the sshd process. For both normal tty channels as well as channels not bound to a tty cutoff values are put in place to avoid excessive data copying. For the situation of non-tty communications (which can include file transfers), the ratio of printing to non-printing characters is looked at to avoid needlessly copying binary files.

The differentiation between binary and ASCII results for both SCP and SFTP file transfers is understandable based on how non-tty data channel is examined. ASCII files will have significantly more data copied to the analyzer so there will be somewhat higher overhead.

### *3.2.3 Results*

The most useful thing to show here is an example session which includes many if the alert and auditing functions. Given space and column limitations, we have included a typical login session and other sample logs in Appendix I.

From a more pragmatic perspective the auditing and analysis functionality has allowed NERSC to quickly

identify dozens of compromised user credentials as well as the knowledge that entire generations of attack tools will alarm on their use. Besides attack detection, the iSSHD provides considerable insight into the tactics 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. This provides an important window into attacker activities that can be a sobering reminder that not all the attackers we see are naive or unskilled.

## 4. Future Work

There are a number of areas of future work including 100 Gbps border traffic, integration with on system data sources like process accounting as well as the initial analysis of code categorization based on inter-computational node behaviour.

## 5. Summary

Intrusion detection in the HPC realm is a reasonably young field and subject to considerable change in short time. We present our methodology for data source selection and two sample tools – the Bro Cluster IDS and Instrumented SSHD – as examples of this design strategy in action.

## 6. 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.

## 7. References

[DP11] http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

[MS08] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson and F. Schneider, Enriching Network Security Analysis with Time Travel, Proc. ACM SIGCOMM,

[PA98] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time. Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998

[VA07] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and Brian Tierney, **The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware**, Proc. RAID 2007.

Appendix 1 – Example of user login auditing log

CONNECTION

```
#1 - host 16 SSHD_CONNECTION_START 127.0.0.1:62186/tcp -> 0.0.0.0:2222/tcp
#1 - host 16 SSHD_CONNECTION_START 127.0.0.1_128.105.18.134_10.37.129.2_10.211.55.2
```

AUTHENTICATION

```
#1 - host 16 AUTH_KEY_FINGERPRINT 05:b1:...:16:45 type DSA
#1 - host 16 AUTH Postponed scottc publickey 127.0.0.1:62186/tcp > 0.0.0.0:2222/tcp
#1 - host 16 AUTH_KEY_FINGERPRINT 05:b1:...:16:45 type DSA
#1 - host 16 AUTH Accepted scottc publickey 127.0.0.1:62186/tcp > 0.0.0.0:2222/tcp
```

SESSION

```
#1 - host 16 SESSION_NEW SSH2
#1 - host 16 CHANNEL_NEW [0] server-session server-session
#1 - host 16 CHANNEL_NEW [1] auth socket auth socket
#1 0-server-session host 16 SESSION_INPUT_CHAN_REQUEST AUTH-AGENT-REQ@OPENSSH.COM
#1 0-server-session host 16 SESSION_INPUT_CHAN_REQUEST PTY-REQ
#1 0-server-session host 16 SESSION_INPUT_CHAN_REQUEST SHELL
```

USER DATA

```
#1 0-server-session host 16 DATA_SERVER Last login: Thu May  5 16:25:33 2011
#1 0-server-session host 16 DATA_SERVER
#1 0-server-session host 16 DATA_SERVER

#1 0-server-session host 16 DATA_SERVER_SUM_SKIP: 1067
#1 0-server-session host 16 DATA_SERVER
#1 0-server-session host 16 DATA_CLIENT pwd
#1 0-server-session host 16 DATA_SERVER cisco-wifi-134:~ scottc$ pwd
#1 0-server-session host 16 DATA_SERVER /Users/scottc
#1 0-server-session host 16 DATA_CLIENT unset HISTFILE
#1 0-server-session host 16 DATA_SERVER cisco-wifi-134:~ scottc$ unset HISTFILE
#1 0-server-session host 16 DATA_CLIENT exit
#1 0-server-session host 16 DATA_SERVER cisco-wifi-134:~ scottc$ exit
#1 0-server-session host 16 DATA_SERVER logout
```

EXIT

```
#1 - host SESSION_EXIT
#1 0-server-session host 16 CHANNEL_FREE
#1 1-auth socket host 16 CHANNEL_FREE
```

```
#1 - host 16 SSHD_CONNECTION_END 127.0.0.1:62186/tcp -> 0.0.0.0:2222/tcp
```

Instance of ‘unset HISTFILE’ triggers an alarm which is logged and can be attached to an email or pager.

```
SSHD_Hostile #1 server-session host:2222 16
       scottc @ 127.0.0.1 -> 0.0.0.0:2222/tcp
       unset HISTFILE [ ]
```

**Field Values**

```
#1 0-server-session host 16 SESSION_INPUT_CHAN_REQUEST SHELL
```

1- Session identification
2- Channel number and type, ‘-‘ means no channel
3 - Host identifier
4 - Client session id. Random 32 bit number
5 - Event identifier
6 - Event information