A Network Audit System for Host-based Intrusion Detection (NASHID)

CERIAS*
Purdue University
West Lafayette, IN 47907–1315
{daniels,spaf}@cerias.purdue.edu
CERIAS Technical Report 99/10

Thomas E. Daniels, Eugene H. Spafford

Abstract

Recent work has shown that conventional operating system audit trails are insufficient to detect low-level network attacks. Because audit trails are typically based upon system calls or application sources, operations in the network protocol stack go unaudited. Earlier work has determined the audit data needed to detect low-level network attacks. In this paper we describe an implementation of an audit system which collects this data and analyze the issues that guided the implementation. Finally, we report the performance impact on the system and the rate of audit data accumulation in a test network.

1 Introduction

Conventional audit trails do not contain enough information to detect network attacks. In Price's survey of conventional operation system audit trails [Pri97], only one of the operating systems collected data from the IP protocol stack. This audit system was part of Sun Microsystem's Basic Security Module (BSM), and the only protocol-level network data that it collects are Internet Protocol (IP) packet headers [Sun95]. One way of detecting these attacks is using network intrusion detection systems (NIDS), but these have been shown to have some major problems [PN98]. In this paper, we concentrate on a host-based approach to auditing the network data needed to detect network attacks.

1.1 Audit Data Needed

Daniels and Spafford described the audit data necessary to detect many low-level TCP/IP attacks [DS99]. This data consists primarily of valid packet headers received by the host and transitions in the TCP state machine. The notion of packet header validity is discussed further in Section 2.1. TCP state transitions are audited because of the overhead and complexity of simulating the TCP state machine. Also, different state machine implementations may behave differently on the same input, and we would therefore require a slightly different simulation to detect protocol stack-specific attacks. State transition audits allows us to write more platform independent attack signatures. Figure 1 shows the audit collected by the system and their corresponding sources in the protocol stack.

*Portions of this work were supported by sponsors of Center for Education and Research in Information Assurance and Security.

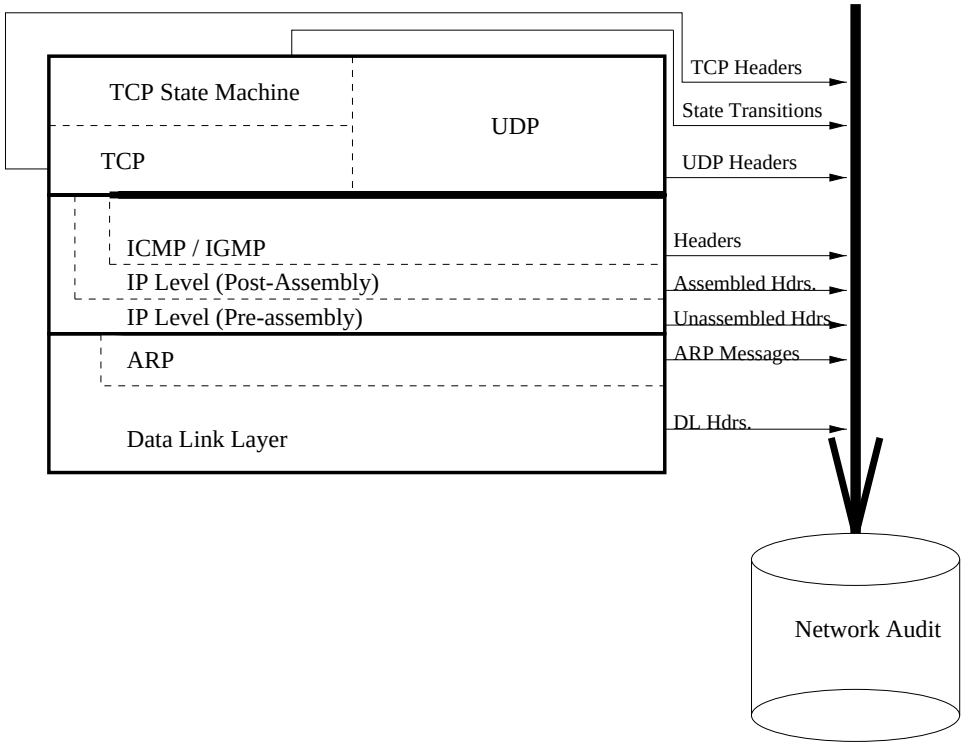

Figure 1. Network audit data is collected from several points within the protocol stack. It consists of protocol headers and transitions in the TCP state machine.

1.2 Previous Network Audit Work

A common network audit source used in the past is the operating system's low-level network monitoring system. One such interface is the Data Link Provider Interface (DLPI) in Solaris [Mic91]. DLPI and similar interfaces on other operating systems allow a user level process to monitor data link traffic received by the system. Using this information source for intrusion detection has some of the problems of collecting network audit data from an external network monitor as described in [PN98]. Although problems associated with simulating network behavior are no longer an issue, the IDS still must simulate the local protocol stack in order to detect attacks that exist at layers above the data link level. Packages such as `tcpdump` [JLM], use these interfaces for data collection

Another existing network audit source is the tcpwrappers package [Ven]. Tcpwrappers is a software package that allows the administrator to collect some information about incoming TCP and UDP requests. When a service request is made to `inetd`, a small wrapper program is launched that collects information about the service. The wrapper then launches the real daemon for which the connection was bound and passes the connection to it. TCP wrappers allow collection of data about the source address and ports of the service request along with time of initiation, but they do not provide any packet header or content information. Furthermore, TCP wrappers do not work for daemons that are not launched by `inetd`.

2 Goals of NASHID

Our goal in developing NASHID is to demonstrate a system that can efficiently audit network information necessary for detecting low-level IP attacks against a given host. To do this, we collect only valid network data and do our data collection within the protocol stack of the host. We must collect data within the protocol stack so that we do not have to simulate the protocol stack to determine the validity of packets.

2.1 Validity of Packets

We consider a valid packet as one that passes the acceptance checks placed upon it after it has been passed to a given protocol layer. A trivial example of an invalid packet is a packet with an incorrect checksum. A less obvious example of an invalid packet is a TCP packet with an illegal set of option bits set. Some implementations of TCP may accept the packet normally while others may reject the packet. Because of this, NASHID must be protocol stack specific. Capturing valid packets without performing other checks on them also implies that our audit probes must be executed after the appropriate protocol layer (or sublayer) checks have been made on the packet.

2.2 Locality of Audit

Because a packet header may be valid or invalid at several different levels, we must audit packet headers at each level of the protocol stack. An example of this would be an invalid UDP packet encapsulated by a valid IP packet and valid Ethernet frame. While the UDP packet will be discarded at the UDP layer, the IP packet header will be processed and might represent an attack. Furthermore, if we wait until the packet has reached the top of the stack to audit its encapsulating headers, a lower-level header may already have exploited a vulnerability and crashed the system. In this case, the audit mechanism would miss the attack, and we would not know what caused the crash.

3 Description of NASHID

NASHID consists of a Linux kernel modified to collect the network audit data and two user level utilities to process the audit data output by the kernel. We chose Linux because the source code is readily available and easy to modify. When we began implementation, we decided to use the latest stable release of the Linux kernel (2.0.34)

[To98], but our performance results are based on a later port to version 2.0.36. The user space utilities convert the raw audit output of the kernel into a text format and also provide parsing and record association capabilities.

3.1 The Existing Kernel Audit System

The Linux kernel has a simple audit mechanism that we call klog. Klog includes an internal kernel function called `printk()` so that kernel code can output debugging and other informative messages. The `printk()` function takes arguments similar to the `printf()` function from the standard input/output package, but it puts the message into a special buffer that is treated as a circular buffer instead of standard output. `Printk()` also outputs the message to the Linux console if the message's priority is greater than a hard coded threshold.

The Linux kernel provides two mechanisms for passing messages from its klog ring buffer to user level processes. These two interfaces are the `/proc/kmsg` file and `syslog()` system call. [Red95b, Red95a] A process uses the `/proc/kmsg` interface by simply reading from it as if it were a sequential file. The `syslog()` system call, not to be confused with the `syslogd` daemon, allows a process to request a specified number of bytes of the circular buffer be written to a buffer in the memory space of the process.

Linux uses the klog facility to report kernel error and status messages. These include the messages written to the console during system startup, kernel panics, kernel error messages, and loadable module startup messages. These messages are ASCII text and many of them are displayed to the console.

3.2 Extending the Existing Kernel Audit System

NASHID's primary requirement was to efficiently audit packet header data. In order to minimize the time spent in the kernel, we chose to pass binary images of packet headers instead of converting them to some other encoding such as text. Because klog is designed to work with text messages, it is not capable of handling binary images of packet headers. Additionally, we did not want other kernel text messages interspersed with the network audit data. To accomplish this, we created a new audit facility that resembles klog, but supports arbitrary binary data,

Our approach was to create another klog mechanism by copying and modifying appropriate portions of the Linux source code. This new mechanism creates its own distinct ring buffer in static kernel storage. NASHID's ring buffer differs from that of klog in that NASHID's is larger. We increased the buffer size to 131,072 bytes, nearly the largest array that can be statically allocated in the Linux kernel. First, we added the `audpacket()` function that replaces `printk()`. To audit a packet header, the programmer calls `audpacket()` with the size of the header and a pointer to it. Other arguments specify the `skbuff` buffer object from which the packet comes, a rough time stamp, and an additional message that the programmer may include in the audit record. `Audpacket()` forms an audit record from this information and puts it into the NASHID ring buffer.

```
<4>,rid=3,length=20,jiffies=129396,track_no=1980,ftn(0)=1980,
ftn(1)=1979,ftn(2)=1978,ftn(3)=1977

20 byte data payload
```

Figure 2. An audit record header as it is stored in the ring buffer. Following this header would be 20 bytes of an IP packet header.

Audit Record Type	Record Identifier
ARP Reply	1
Ethernet Frame	2
Assembled IP header	3
IP Fragment Header	4
ICMP Header	5
IGMP Header	6
TCP Header	7
UDP Header	8
TCP State Machine Transition	9
Loopback Pseudoheader	12

Figure 3. The valid audit record identifiers (rids) in NASHID.

Audit records stored in the ring buffer consist of a simple text header followed by a binary data payload as shown in Figure 2. The text header is a comma delimited list of attribute-value pairs preceded by a priority marker and succeeded by a newline character. An example of an audit record header is shown in Figure 2. The <4> is a holdover from the klog facility and indicates a priority value for the record. NASHID ignores this. The rid value indicates the type of the current record. The values possible for rid are shown in Figure 3. The length attribute specifies the number of payload bytes that follow the header.

A rough timestamp is provided by the jiffies attribute. When the system boots, the jiffies value in the kernel is initialized to zero. During servicing of the timer interrupt, which defaults to a frequency of 100 Hz, jiffies is incremented. The jiffies value in the audit record is passed to `audpacket()` because an audit system may wish to hold an audit message for some amount of time before submitting it. This allows the programmer to maintain the jiffies value manually and then submit the time that the event occurred instead of when it was audited.

The track_no attribute specifies the tracking number for the packet. When a packet arrives via a data link interface such as Ethernet or the loopback device, it is assigned a unique tracking number. This tracking number is associated with the packet's buffer structure that is then passed up the protocol stack as the packet is processed. Each time a packet header is audited, the tracking number is included in the header so that the headers can later be associated with each other for analysis.

An example of the use of the track_no attribute is shown in Figure 4. All three audit headers have the same tracking number and are therefore derived from the Ethernet header audited in Line 1. Line 2 audits the IP header carried by the Ethernet frame. Finally, Line 3 audits the TCP header encapsulated by the IP packet of Line 2.

The ftn attributes in Figure 2 are specified only when IP fragmentation occurs. The ftn values specify the tracking numbers of the IP fragments from which the given packet was assembled This is done in the kernel by adding nodes to a linked list of fragment tracking numbers in the `skbuff` buffer structure. When `audpacket()` forms the audit record, it creates the ftn values in the audit headers. The number in parentheses is added to make the ftn attribute names unique.

User level access to the NASHID ring buffer is provided via a file in the virtual `proc` file system called `/proc/audk`. The program reading from `/proc/audk` is responsible for parsing the audit records from the

```
1. <4>,rid=2,length=14,jiffies=41861685,track_no=50876
(14 bytes of Ethernet Header omitted)

2. <4>,rid=3,length=20,jiffies=41861685,track_no=50876
(20 bytes of IP Header omitted)

3. <4>,rid=7,length=20,jiffies=41861685,track_no=50876
(20 bytes of TCP Header omitted)
```

Figure 4. NASHID Audit records of the reception of a TCP packet using IP over Ethernet.

binary audit stream. This is done by the audit interpretation tool described below.

3.3 Network Audit Interpretation Tool

NASHID includes two user level tools for interpreting and using the audit data read from the ring buffer. The first tool, called the Network Audit Interpretation Tool (NAIT), reads from `/proc/audk` and generates an ASCII text representation of the audit data. NAIT does this by overlaying the appropriate `struct` from the system C language header files onto the buffer that was read. It then outputs an attribute-value pairs for every field in the structure.

An example of the NAIT output format is shown in Figure 5. The word following "begin record" declares the type of the audit record. The first attributes are copied from the header of the ring buffer audit record. The attributes that follow the it are the values of the TCP header fields.

NAIT output format is a convenient human-readable representation of the trail. The format is easily described by a grammar that can be parsed by an LL(k) parser. We need to be able to easily parse the format so that we can compress the audit data using some techniques developed by others in our organization.

3.4 Network Audit Correlator Tool

Evaluating a typical signature for detecting network level attacks requires values from several different protocol headers of the same packet. NASHID supports this using the tracking number mechanism, but it would be significant repeated effort for each signature evaluator to have to aggregate the various protocol headers for a given packet. To prevent this, we developed the Network Audit Correlator Tool (NACT). NACT provides a subscription style interface so that a signature evaluator can register with NACT to receive specific types of network audit.

NACT reads the output of NAIT and maintains a list of lists of audit records. Each sublist contains audit records with the same tracking number. Audit records that are derived from IP fragments also are linked to the audit records that represent their constituent fragments. When an audit record is received with a given rid, the subscription list for that rid is checked. If there are any subscribers for that record type, the record along with its associated audit records from the list of lists are sent to the subscriber.

As an example of this process, we will refer to Figure 4. As NACT receives the audit record on line 2, it will be added to the same sublist as the record on line 1, which we assume has already been received. If a subscriber has requested assembled IP headers (rid=3), that subscriber will then sent record 2 and its ancestor, record 1. Similarly

```
begin_record TCP
rid=7,length=20,jiffies=95223220,track_no=111537,
tcp_sourceport = 65283, tcp_destport = 5632,tcp_seq = 2628222749,
tcp_ack_seq = 506415025,tcp_hlength = 5, tcp_reserved1 = 0,
tcp_reserved2 = 0, tcp_urg = 0, tcp_ack = 1, tcp_psh = 0,
tcp_rst = 0, tcp_syn = 0, tcp_fin = 0,tcp_window = 14370,
tcp_check = 38857, tcp_urg_ptr = 0
end_record
```

Figure 5. An example of a TCP packet header audit record output by NAIT. The record includes the audit header output by the kernel.

if an evaluator has subscribed for TCP Headers, record 3 will be sent along with record 2 and record 1. As a kind of garbage collection, the system clears a sublist when the highest possible header type is received and processed. For instance, when a UDP audit record is received, it is assumed that the IP and lower audit records with the same tracking number have been received so the list is cleared after sending the data to subscribers.

NACT was written in Java 1.1 using JavaCC as the parser generator. This was chosen for ease of development. A drawback of using Java for NACT is its slow performance, but NACT provides a simple proof of concept. The current implementation provides an `AssemblyArea` class that performs the aggregation of the various audit records. Objects that subclass the `Subscriber` class can subscribe to the `AssemblyArea` to receive audit records. One example of a simple subscriber is the `LandDetector` class. `LandDetector` subscribes to TCP Header Audit Records in its constructor. `LandDetector` is required to have a `receive` method that is called by `AssemblyArea` when a subscribed audit record is processed. It is then the responsibility of `LandDetector` to detect the attack and report if one is found. Other subscribers that we have implemented include one to detect SYN floods and the Ping of Death attack.

4 Performance

In this section we present a few measurements of the impact that NASHID has on the system. As a benchmark, we compare NASHID's performance impact on the system with that of tcpdump [JLM]. Since tcpdump is based on the commonly used network traffic capture library, libpcap [NRG], it is reasonable to consider it a representative capture mechanism for network data. Additionally, we present some measurements of NASHID's rate of audit data generation in a test network. We do not consider the impact of NAIT or NACT because these are unoptimized proof of concept tools, and a performance oriented host-based intrusion detection system has no need to convert all network data to text as NAIT does.

4.1 System Impact

We developed an experiment to measure the impact that NASHID has on the performance of system processes. We used the real world running time of a simple program called `perf`. `Perf` contains a tight `for` loop of 100,000,000 iterations. By running `perf` in varying system conditions and comparing its running time, we

evaluate NASHID's performance impact on the system. We use this approach because we need a way to measure the impact of kernel and user tools on the rest of the host system.

4.1.1 Experiment Setup and Procedure

Our experiments were run on a switched Ethernet LAN with 3 300 Mhz Pentium II hosts. Each host has 128 megabytes of RAM and runs Redhat Linux 5.1. The hosts use Intel EtherExpress Pro 100 network interface cards and are connected to a Cisco Catalyst 2900 Series XL 100 megabit Ethernet switch with 12 ports. The host machines are xanadu (an NFS and NIS server), isher (a client of xanadu running NASHID), and greaves (another client of xanadu but not running NASHID). No other hosts are connected to the switch.

The goal of experiment one was to measure the effect of the NASHID kernel probes on the performance of a system under nominal network load. Performance measurements were taken on isher while running a base Linux kernel without NASHID compiled into it. A second set of measurements were performed while running the same kernel with NASHID compiled into it. In both cases, the network cable was disconnected from isher.

The goal of experiment two was to measure the effect of NASHID usage on the performance of a system under high network load in a 10 Megabit network, and in these conditions, compare NASHID's performance with that of Tcpdump. Four sets of measurements were taken on isher—each during high network load as described in Section 4.1.2. The first measurements were taken while running the base Linux kernel without NASHID compiled into it. The second set of measurements were taken with NASHID compiled into the kernel but with nothing reading from `/proc/audk`. The third set of measurements were taken while saving the output of `/proc/audk` to a local file using `cat`. Finally, using the non-NASHID kernel, measurements were taken while tcpdump was used to save a binary image of the first 68 bytes of each received frame using the command `tcpdump -w` *filename*.

Experiment three was conducted in the same manner as experiment two except that it was done in a 100 Megabit network.

4.1.2 Experimental Details

Five measurements were taken in each measurement set, and the mean was taken as the final result. Each measurement consisted of one run of `perf` and was timed using the `time` command. The "real world" running time output by `time` was the measurement recorded for each run.

The same switch was used for both experiment two and three, but in experiment two, the ports of the switch were forced to 10 Megabit full duplex mode. In experiment three, the switch was used in its default 100 Megabit full duplex mode.

High network load at isher was created by having xanadu and greaves "ping flood" isher. The command executed on greaves and xanadu to send the ping flood was `ping -f -s 10000 isher`. The "-s 10000" options causes the host to send ICMP echo requests with 10,000 bytes of data. The "-f" option, for flood, instructs the host to repeatedly send ICMP packets as quickly as the host can manage. Because the 10,000 byte data payload will cause the IP packets to exceed the maximum transmission unit (MTU) of the Ethernet, the flood consists of IP fragments that assemble to ICMP echo requests.

4.1.3 Performance Results

In experiment one, no difference in system performance between a kernel with NASHID probes and one without was observed during nominal network load. This is to be expected since the probes would have executed very few times if at all. This simply confirms that NASHID has not added any non-network related performance overhead to the system.

Experiment two measured performance degradation in a 10 Mb Ethernet network. The results are summarized in Figure 6. We observe that the presence of the kernel probes alone only slows the system by 6.7% However,

by simply saving this data to a file, we additionally degrade the performance of the system by nearly as much as collecting the data in the first place. Finally, we see that in this case, using tcpdump to save the data is about 2% more costly than NASHID.

Experiment three measured performance degradation in a 100 Mb Ethernet network, and the results are summarized in Figure 7. In this case, the NASHID probes cause a 36% degradation relative to an unprobed kernel in the same situation. The combination of the overhead of the kernel probes and saving the data degraded system performance by 113%. Finally, the performance impact of using tcpdump to save the data instead is just less than that of NASHID at 110%.

4.1.4 Analysis of Performance Results

A simple analysis of our performance results shows that the NASHID approach to collecting network audit data is very similar in cost to the use of conventional mechanisms like tcpdump. In the 10 Megabit network, tcpdump caused a 2% more severe degradation of performance than NASHID. In the 100 Megabit case, NASHID was only 3% more costly than tcpdump. We are not certain why NASHID performs better than tcpdump in one case and not the other, but we suspect that it may be caused by larger disk write buffers in use by tcpdump. This might only become apparent when the data rate becomes high.

It is important to remember that NASHID is for host-based intrusion detection, and therefore, in real life we would not expect network throughput as high as in the 10 Megabit experiment other than in extreme situations. Since legitimate network traffic bound for any given host is unlikely to maintain consistent high data rates as in these experiments, one may take them as a worst case scenario. In this light, we feel a 12% performance hit in the 10 Megabit case and a 113.51% performance hit in the 100 Megabit case is reasonable for collection and saving of network audit data. Realistically though, it is unlikely that a practical host-based intrusion detection system would save all of the network data. It is more likely that the system would evaluate this data in real time and only record suspect traffic for later analysis.

4.2 Amount of Audit Data

The amount of audit data generated by NASHID is dependent on the amount and type of network traffic that the NASHID host receives. We designed a simple experiment to measure the worst case rate of audit data collected by NASHID. Using the same network described above in the 10 Megabit configuration, we collected the output of `/proc/audk` on isher while xanadu and greaves both ping flooded it. We first started the ping floods and recorded the data for one minute. We did this for three trials each during a new set of ping floods and no other network activity.

The results of the above experiment were that isher collected 4.33 Megabytes of audit data per minute on average. While considerable amount of audit data is generated by NASHID in this experiment, it is important to remember that these are near worst case figures. The NASHID host was being attacked by multiple hosts on the same 10 Megabit LAN. The hosts were saturating the network with traffic destined for the NASHID host. We look forward to running NASHID on systems in production mode in the future to get a better idea of the amount of network audit collected by a typical system. As before, it is also important to remember that a practical host-based intrusion detection system would not be likely to save all network audit data.

5 Conclusions

We have presented a new audit mechanism for auditing network level information. By adding probes to the kernel, we have developed a system that efficiently audits protocol headers and other protocol level information for detection of network attacks. The system only collects network data that is valid at a given layer thereby

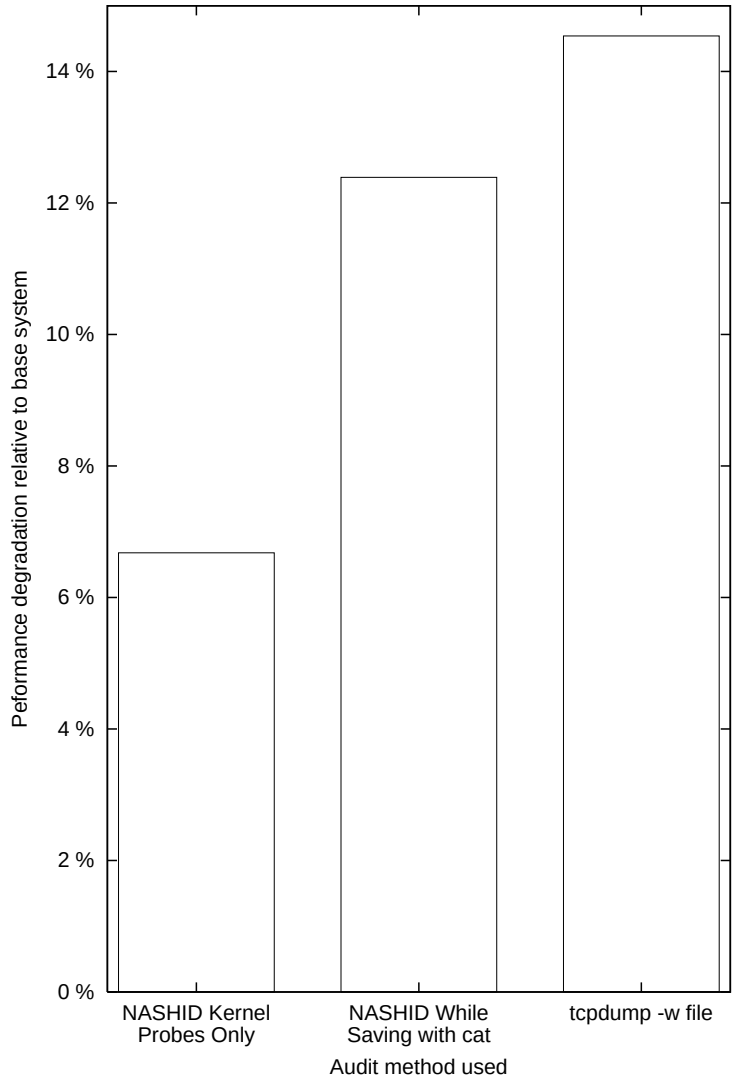

Figure 6. Experiment 2 shows system performance degradation caused by a ping flood in three different configurations. The numbers are percentage of degradation relative to measurements on the same system without NASHID probes compiled into it.

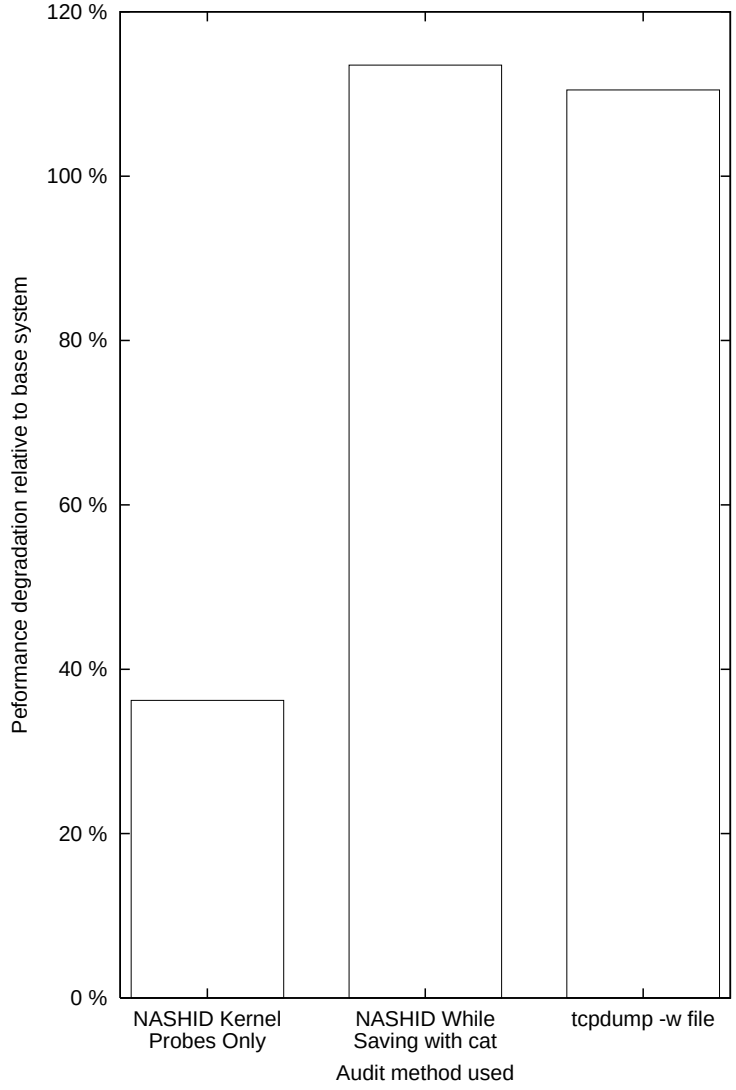

Figure 7. Experiment 3 shows system performance degradation caused by a ping flood in three different auditing configurations. The numbers are percentage of degradation relative to measurements on the same system without NASHID probes compiled into it.

reducing the likelihood of false positives. Using an identification number mechanism, we can easily tie together different protocol headers from the same packet. Additionally, the mechanism provides for correlation of IP fragment headers with the higher level headers to which they assemble. Using NASHID to collect raw audit data degrades system performance in a manner similar to other network capture methods.

References

[DS99] Thomas E. Daniels and Eugene H. Spafford. Identification of host audit data to detect attacks on low-level IP vulnerabilities. *Journal of Computer Security*, 7(1):3–35, 1999.

[JLM] V. Jacobson, C. Leres, and S. McCanne. tcpdump. available via anonymous ftp to `ftp://ftp.ee.lbl.gov`.

[Mic91] Sun Microsystems. DLPI man page. DLPI manual page from Solaris 2.5.1, 1991.

[NRG] Lawrence Berkeley National Laboratory Network Research Group. LIBPCAP 0.4. `ftp://ftp.ee.lbl.gov/libpcap.tar.Z`.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc, January 1998.

[Pri97] Katherine E. Price. Host-based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, December 1997.

[Red95a] Redhat 5.1 linux kernel log daemon (8) man page. Redhat 5.1 Linux manual page: man 8 klogd, November 1995.

[Red95b] Redhat 5.1 linux syslog(2) man page. Redhat 5.1 Linux manual page: man 2 syslog, June 1995.

[Sun95] Sun Microsystems, Inc., Mountain View, California. *SunSHIELD Basic Security Module Guide*, November 1995.

[To98] Linus Torvalds and other. Linux kernel. Available at `http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.34.tar.gz`, June 1998.

[Ven] Wietse Venema. TCP wrappers. available at `ftp://ftp.win.tue.nl/pub/security/tcp_wrappers_7.6.tar.gz`.