Experience with Engineering a Network Forensics System

Ahmad Almulhem and Issa Traore

ISOT Research Lab
University of Victoria, Canada
{almulhem, itraore}@ece.uvic.ca

Abstract. *Network Forensics* is an important extension to the model of network security where emphasis is traditionally put on prevention and to a lesser extent on detection. It focuses on the *capture*, *recording*, and *analysis* of network packets and events for investigative purposes. It is a young field for which very limited resources are available. In this paper, we briefly survey the state of the art in network forensics and report our experience with building and testing a network forensics system.

1 Introduction

Most organizations fight computer attacks using a mixture of various technologies such as firewalls and intrusion detection systems [1]. Conceptually, those technologies address security from three perspectives; namely *prevention*, *detection*, and *reaction*. We, however, believe that a very important piece is missing from this model. Specifically, current technologies lack any *investigative* features. In the event of attacks, it is extremely hard to tie the ends and come up with a thorough analysis of how the attack happened and what the steps were. Serious attackers are skillful at covering their tracks. Firewall logs and intrusion detection alerts are unlikely to be adequate for a serious investigation. We believe the solution is in the realm of *Network Forensics* [2]; a dedicated investigation technology that allows for the capture, recording and analysis of network packets and events for investigative purposes. It is the network equivalent of a video camera in a local convenience store.

In this paper, we report our experience with designing, implementing and deploying a network forensics system. First, we review the topic of network forensics in section 2. In section 3, we review some related work. In section 4, a network forensics system will be proposed. In section 5, we will discuss our implementation of the proposed architecture and some interesting results. Finally, we conclude and discuss our future work in section 6.

2 Network Forensics

In 1997, security expert Marcus Ranum coined the term *network forensics* [2]. He also introduced a network forensic system called *Network Flight Recorder* [3].

Marcus, however, did not provide a definition for the new term. Therefore, we adopt the following one from [4]:

> *Network forensics* is the *capture*, *recording*, and *analysis* of network packets and events for investigative purposes.

When designing such a system, there are several challenges which include:

1. *Data Capture*:
   (a) Where should the data be captured?
   (b) How much data should be captured?
   (c) How do we insure the integrity of the collected data?
2. *Detection Efficiency*: The system should *detect* attacks efficiently in order to trigger the forensics process. Therefore, it should accommodate for different detection approaches.
3. *Data Analysis*: After collecting the data, the system has to *correlate* them in order to reconstruct an attacker's actions.
4. *Attacker Profiling*: The system has to maintain information about the attacker himself. For instance, it can identify the attacker's operating system through passive OS fingerprinting.
5. *Privacy*: Depending on the application domain, privacy issues can be a major concern.
6. *Data as Legal Evidences*: For the collected data to qualify as evidences in a court of law, they have to be correctly collected and preserved in order to pass *admissibility* tests [5, 6].

3 Related Work

Unlike the traditional *computer forensics* field, network forensics emerged in response to network hacking activities [7]. Typically, it is conducted by experienced system administrators rather than by law enforcement agencies [8].

The current practice in investigating such incidents is generally a manual brute-force approach. Typically, an investigation proceeds by examining various types of logs which are located in a number of places. For instance, a unix network is usually equipped with a dedicated auditing facility, such as *Syslogd*. Also, applications like web servers and network devices like routers, maintain their own logs. Various tools and homemade scripts are typically used to process these logs.

Brute force investigation, however, is a time consuming and error-prone process. It can also be challenging because the mentioned logs are usually scattered everywhere over the network. Also, these logs are not meant for thorough investigation. They may lack enough details or contrarily have lots of unrelated details. They also come in different incompatible formats and levels of abstractions.

On the high-end, there are commercial tools known as *network forensic analysis tools* which can be used for investigations in addition to varieties of tasks like network performance analysis [3, 9]. Generally, these tools are combined hardware/software solutions which continuously record network traffic. They also provide convenient GUI front-ends to analyse the recorded data.

The main problem with these commercial tools is dealing with encrypted traffic. Currently, the general approach is to install modified (trojaned) encrypted services like *ssh*. So if an attacker uses these services, his sessions can be decrypted. This, however, can be defeated, if the attacker installs his own encrypted service.

4 A Network Forensics System

In this section, we propose an architecture of a network forensics system that records data at the host-level and network-level. It also manages to circumvent encryption if an attacker chooses to use it. At first, we will provide an overview of the system, then discuss its main components in more details. Implementation and results will be postponed to the next section.

4.1 System Overview

In a typical network with multiple hosts, the proposed system consists of three main modules which are arranged as shown in Fig. 1.

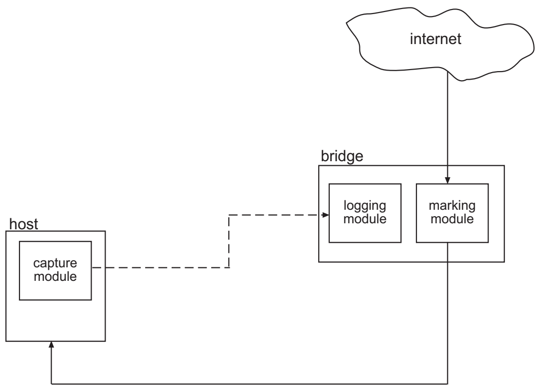


Fig. 1. The overall architecture of the system

The modules are

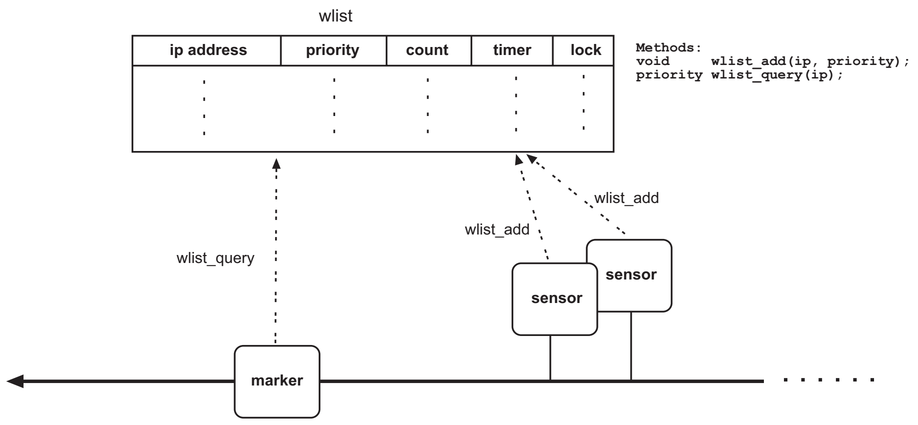
1. a *marking module*; a network-based module for identifying and marking suspicious packets as they enter the network,
2. *capture modules*; host-based modules which are installed in all the hosts in order to gather marked packets and post them to a logging facility, and

3. a *logging module*; a network-based logging facility for archiving data.

Together, these modules form a kind of closed circuit. An incoming packet first passes through the marking module which marks "suspicious" packets. Subsequently, when a host receives a marked packet, it posts the packet to the logging module. Each module will now be explained in further details.

4.2 Marking Module

This module is the entry point to our system. It is in charge of deciding whether a passing-by packet should be considered friendly or malicious. Then, it marks the packet accordingly. In nutshell, this module relies on a group of sensors to maintain a list of suspicious IP addresses. Then, it marks a passing-by packet if it's source IP address is in the list.



```
Methods:
void     wlist_add(ip, priority);
priority wlist_query(ip);
```

Fig. 2. The marking module

Figure 2 depicts the architecture of this module, which consists of the following three components:

1. *Sensors*: One or more sensor(s) to report suspicious IP addresses to a watch list (*wlist*). It is important to note that a sensor is not limited to a network-based IDS. It can be any process that can report suspicious IP addresses. This is essential to increase the system's detection efficiency.
2. *A Watch List (wlist)*: A list of suspicious IP addresses. We will explain it in more details shortly.
3. *A Marker*: A process to mark packets. Before sending a packet to its way, it queries the watch list to check whether the packet's source IP address is in the list. It accordingly modifies the *type of service field (TOS)* in the IP header.

The *watch list (wlist)* is basically a data structure which maintains a list of the current system's offenders. One may think of it as a cache memory of suspicious IP addresses. Each row corresponds to a unique IP address that has been reported by at least one of the sensors. For every suspicious IP address, the list also maintains the following information:

1. *priority*: A measure which indicates the *current* offence level of the corresponding IP address. Three levels are defined; namely HIGH, MEDIUM and LOW. A sensor must be able to classify an attack into one of these three levels. When different priorities are reported for a given IP address, the list only keeps the highest.
2. *count*: A counter which is incremented every time the corresponding IP address is reported.
3. *timer*: A count-down timer which is automatically decremented every second. If it reaches zero, the corresponding row is removed from the list. This field is set to a certain value when an IP address is first added to the list. It is also reset to that value every time the IP address is reported. One may think of this field as a sliding time window. If an IP address was not seen for a long time (say 1 week), we may remove it from the list.
4. *lock*: This field is to synchronize accesses. It is needed because the list is asynchronically accessed by a number of processes.

To interact with *wlist*, two methods are provided:

1. *wlist_add(ip, priority)*: A method to add an attacker's IP address to the list.
2. *wlist_query(ip)*: A method which returns the priority of a given IP address.

Finally, since the list is limited in size, one may wonder what happens if the list becomes full and a newcomer needs to be accommodated. Obviously, we need to decide which row should be replaced. Specifically, we should replace the *least important* row. A row with a low *timer* value indicates that the corresponding IP address was not seen for a long time. On the other hand, a high *count* value suggests that the corresponding IP address is suspicious. Thus, finding the least important row is a function of the three fields; namely *priority*, *count* and *timer*. Formally, let the priority, count, and timer be p_i, c_i and t_i respectively for a given row i. Then, the least important row (l) is

$$l = f(p_i, c_i, t_i)$$

The exact definition of this function is implementation specific. We will show an example definition when we discuss our implementation.

4.3 Capture Module

The second major component in our architecture is a collection of lightweight capture modules, which reside silently in the hosts waiting for marked packets. They, then, arrange to reliably transport them to the logging module for safe archival. This transportation is necessary because we cannot store the data locally. Once a system has been compromised, it cannot be trusted.

Installing capture modules in hosts is essential for two reasons. First, there is no guarantee that a suspicious packet will actually compromise or damage a host. In fact, the packet may be directed to a nonexistent service or even a nonexistent host. Installing capture modules in the hosts insures logging only relevant packets.

The second and more important reason is the fact that attackers increasingly use *encryption* to hide their activities. As a result, sniffing their traffic or trying to break-it is either useless or impractical. We may choose to install trojaned encrypted services; say *ssh*. However, careful attackers usually avoid these services and use their own encrypted channels. Therefore, only at the host, we can circumvent encryption and fully record an attacker's actions. This can be done by intercepting certain system calls [10].

4.4 Logging Module

The logging module is our system's repository where attack data are being stored. Ideally, one would turn to this module for reliable answers and documentation about any attack.

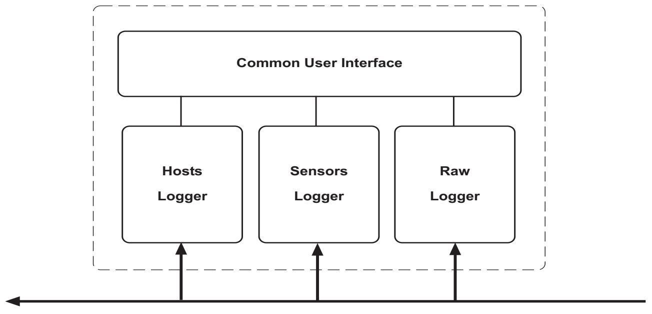


Fig. 3. The logging module

Figure 3 shows the architecture of a network-based logging module. We propose to use the following three loggers:

1. Hosts Logger: This logger is responsible for storing data sent by the capture modules. It is expected to log detailed data pertaining to real attacks. Therefore, storage requirements should be low.
2. Sensors Logger: This logger stores the sensors' alerts. Although, a typical alert is only a one-line text message, it provides a quick diagnosis about an attack. This logger is also expected to require low storage requirement.

3. Raw Logger: This is an *optional* logger which provides a last resort solution when other loggers fail. It archives raw packets straight off the line. In busy networks, however, this logger is expected to require an excessive amount of storage.

The last part in this module's architecture is a layer that should provide a common user interface to access these loggers.

5 Implementation and Results

5.1 Implementation

To test our approach, we built a prototype of the proposed architecture using two PCs; a host and a bridge configured as shown in Fig. 1. The host is a PC with a 400MHz Pentium II processor, 132MB RAM and 6GB hard drive. To allow break-in, we installed a relatively old Linux distribution; namely RedHat 7.1 Also, we enabled a vulnerable service; namely FTP (wu-ftpd 2.6.1-16). We also installed a *capture* module called *sebek* [10] from the *Honeynet* project [11]. It is a kernel-based data capture tool which circumvent encryption by intercepting the *read* system call.

The bridge is a PC with a 1.7GHz Celeron processor, 512MB RAM, 40GB hard drive and 2 NICs. We installed a custom Linux operating system and a collection of tools and homemade programs which reflect the proposed architecture. Figure 4 shows the internal architecture of this bridge. It hosts both the *marking* and *logging* modules.

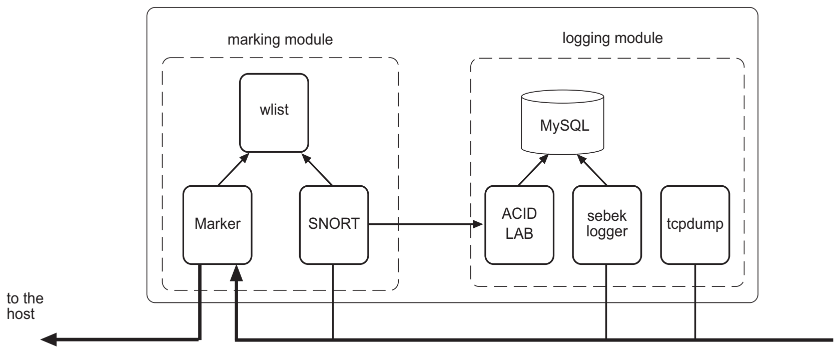


Fig. 4. The Bridge Internal

The marking module follows the architecture described earlier. Only one sensor was used; namely SNORT [12]. Both the watch list *(wlist)* and the marker were implemented in C language. When *wlist* becomes full, we used the following function: $l = min(\{t_i \mid t_i$ is the *timer* value of the i^{th} row in *wlist* $\})$ where *min* is the minimum function.

The logging module also follows the proposed architecture. It consists of three loggers and MySQL [13] as a backbone database. The first logger records packets captured by the host. Since *sebek* was used to capture packets there, we used its corresponding server-side tools. The second logger is for the sensor; i.e. SNORT. We used SNORT's *barnyard* tool to log alerts in MYSQL and *ACID Lab* [14] for analysis. Finally, we chose *tcpdump* [15] as a raw logger just in case we miss something.

5.2 Results

The prototype was connected to the Internet for 12 days from March 17th until March 28th of 2004. Its IP address was not advertised. It was, however, given to members of our research lab who were interested in participating in the experiment. During the experiment, the host was compromised three times using a known FTP exploit.

General Statistics: Once the prototype was connected to the Internet, the host started receiving traffic. Table 1 lists the number of received packets grouped by protocol type.

Table 1. Number of friendly and strange packets directed to the host

	friendly packets	strange packets
tcp	70130	133216
udp	8928	9581
icmp	5150	6986
total	84208	149783
	36%	64%

Table 2. Storage requirement for each logger

	count	size
SNORT	3482 alerts	111KB
sebek	336132 packets	38MB
tcpdump	734500 packets	69MB

The first column lists the number of *friendly* packets; i.e. packets generated by participating members of our research lab. The second column lists the number of *strange* packets; i.e. packets coming from uninvited strangers. Overall, 64% of the traffic was not friendly. The table also shows that TCP is more frequent than other protocols. For the strangers' traffic, *TCP* packets are about 10 times (20 times) more than *UDP* (*ICMP*).

Table 2 sorts the storage requirement for the three used loggers in ascending order. SNORT requires the least amount, while tcpdump requires the most. Although, sebek is a powerful tool in honeypot settings, it actually did not fit our need. It captures far more data than we need. In the future, we are planning on developing our own capture module.

A Detailed Attack: We now discuss an attack by some stranger who successfully compromised the host and installed a rootkit. Overall, he generated about 1100 packets and caused a 158 *SNORT* alerts: 2 high priority, 154 medium priority and 2 low priority. Using the collected data, we were able to reconstruct his attack. Figure 5 shows a time-line diagram of his attack's steps.

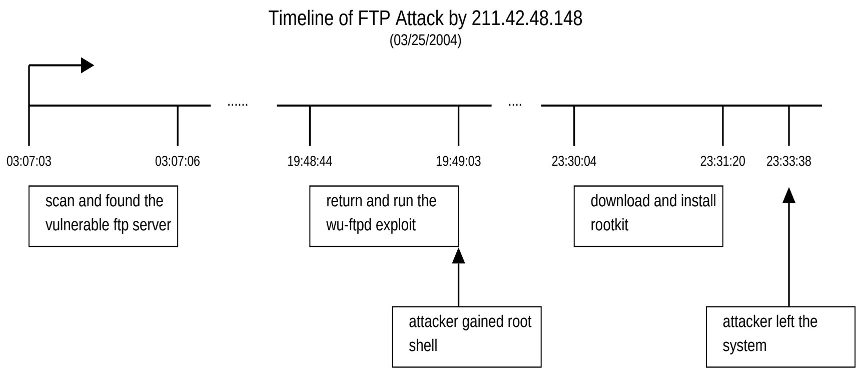

Timeline of FTP Attack by 211.42.48.148
(03/25/2004)
03:07:03
03:07:06
19:48:44
19:49:03
23:30:04
23:31:20
23:33:38
scan and found the vulnerable ftp server
return and run the wu-ftpd exploit
download and install rootkit
attacker gained root shell
attacker left the system

Fig. 5. Time Analysis of one of the attacks on our ftp server

At first, he scanned and found the vulnerable ftp server. After about 16 hours, he returned back with an effective exploit. He run the exploit and immediately gained a root shell. He then left the connection open for about 4 hours. When returned, he typed a number of commands and then exited. The following is a recreation of those commands.

```
[23:28:52] w
[23:29:54] wget
[23:30:04] wget 65.113.119.148/l1tere/l1tere.tgz
[23:30:19] ls
[23:30:24] tar xzvf l1tere.tgz
[23:31:20] ./setup
```

Those commands discloses the attacker's steps to downloading and installing a rootkit. Further analysis of this rootkit revealed the following main impacts:

1. creates directories and files under */lib/security/www/*.
2. removes other known rootkits.
3. replace some binaries with trojaned ones; many to mention!
4. installed a sniffer and a fake *SSHD* backdoor.
5. disable the anonymous vulnerable ftp server.
6. send an email to l1tere@yahoo.com with detailed information about the host.
7. cleans up and delete downloaded files.

Assessing the Results: Assessing the results is informal at this stage. We, however, can safely argue that we were able to detect and reconstruct all the compromises of the host. The proof pertains to using *sebek* at the host which was setup not to be accessed remotely. In particular, *sebek* can capture keystrokes. Therefore, seeing any keystrokes means a compromise. Also, *SNORT* (our sensor) is aware of the relatively old vulnerable *ftp* service. This gave us another indication of an ongoing attack.

6 Concluding Remarks

A network forensics system can prove to be a valuable investigative tool to cope with computer attacks. In this paper, we explored the topic of network forensics and proposed an architecture of network forensics system. We then discussed our implementation and obtained results. The proposed system manages to collect attack data at hosts and network. It is also capable of circumventing encryption if used by a hacker.

In the future, we plan to extend our system architecture with a fourth module named it expert module. The *expert module*, to be implemented as an expert system, will analyze the logged data, assess and reconstruct key steps of attacks. There are several facts that can be used to systematically characterize ongoing attacks and thereby may serve to construct the knowledge base of such expert system. For instance, the fact that some keystrokes are detected while only remote access is possible not only shows that a target has been compromised, but can also be used to partially reconstruct the attack.

References

1. Richardson, R.: 2003 csi/fbi computer crime and security survey (2003)
2. Ranum, M.: Network forensics: Network traffic monitoring. NFR Inc. (1997)
3. Ranum, M., et al.: Implementing a generalized tool for network monitoring. Proceedings of the Eleventh Systems Administration Conference (LISA '97) (1997)
4. searchSecurity.com: Definitions. (searchsecurity.techtarget.com)
5. Sommer, P.: Intrusion detection systems as evidence. Computer Net. **31** (1999)
6. Brezinski, D., Killalea, T.: Guidelines for evidence collection and archiving. BCP 55, RFC 3227 (2002)
7. Fennelly, C.: Analysis: The forensics of internet security. SunWorld (2000)
8. Berghel, H.: The discipline of internet forensics. Comm. of the ACM (2003)
9. King, N., Weiss, E.: Analyze this! Information Security Magazine (2002)
10. Balas, E.: Know Your Enemy: Sebek. Honeynet Project. (2003)
11. Spitzner, L.: Honeynetproject. (www.honeynet.org)
12. Roesch, M., Green, C.: Snort Users Manual. (2003)
13. MySQL. (www.mysql.com)
14. Danyliw, R.: Analysis console for intrusion databases. (acidlab.sourceforge.net)
15. tcpdump/libpcap. (www.tcpdump.org)