Network Radar: Final Report

TR-2002-01

Todd Heberlein
Net Squared, Inc.
todd@NetSQ.com

Table of Contents

1 INTRODUCTION 1

2 NETWORK MONITORING FRAMEWORK (NMF) 2

3 PATTERN MATCHERS 4

3.1 INTRODUCTION 4
3.2 SIGNATURE PYRAMID 5
3.3 DATA SOURCE 5
3.3.1 Intractable Problems 5
3.3.2 Analyzing without Context 6
3.3.3 Inline Protocol Data 8
3.4 SIGNATURE MATCHING ENGINE 8
3.4.1 Boyer-Moore (BM) 9
3.4.2 Knuth-Morris-Pratt (KMP) 9
3.4.3 Regular Expression (RX) 10
3.4.4 Finite State Machine (FSM) 10
3.4.5 Signature Compression 10
3.5 SIGNATURES 13
3.6 PATTERN MATCHING SUMMARY 14

4 RESEARCH EXAMPLE: CLUSTER-BASED DETECTION 14

4.1 INTRODUCTION 14
4.2 THUMBPRINT CONCEPT 15
4.3 COMPARISONS TO TRADITIONAL DETECTION TECHNIQUES 17
4.4 EXAMPLE OPERATIONAL MODEL 18
4.5 EXAMPLE OF DETECTION IN ACTION 18

5 INSTALLING NETWORK RADAR 21

5.1 INSTALLING THIRD-PARTY LIBRARIES 21
5.1.1 Building Packet Capture Library 22
5.1.2 Building Regular Expression Library 22
5.2 INSTALLING NMF AND NETWORK RADAR 22
5.3 BUILDING C-BASED LIBRARIES AND APPLICATIONS 22
5.3.1 Building Network Radar Bridge Libraries 22
5.3.2 Building Sample C Applications to Connect to Nrat Sensor 22
5.4 COPY READ-ONLY TEMPLATE FILES 23

6 RUNNING NRAT 23

6.1 NRAT CONFIGURATIONS 24

List of Figures

Figure 1: Network Monitoring Framework 4
Figure 2: Signature Pyramid 5
Figure 3: Sendmail Commands and Content 7
Figure 4: Telnet Negotiation 8
Figure 5: Two Simple Patterns 11
Figure 6: Signature Cross Product 12
Figure 7: Table and Diagram for "aa | bb" 12
Figure 8: Eliminate Unreachable States 13
Figure 9: Numerical Representation 15
Figure 10: Traditional Checksums 16
Figure 11: Thumbprint Checksums 16
Figure 12: Operational Model 18
Figure 13: ILOVEYOU Worm Strikes Rome Labs 20

1 Introduction

The Network Radar project performed for the Air Forces Research Laboratory (AFRL), specifically Rome Labs, and the Defense Advanced Research Projects Agency (DARPA) developed innovative approaches to building network sensors and innovative network sensor technologies. The sensor technologies were bundled into a suite of applications called Network Radar and delivered to Rome Labs and DARPA's Technology Integration Center (TIC).

Several groups at Rome Labs and DARPA integrated portions of Network Radar into their own security management systems. Projects that integrated Network Radar into their systems include the Automated Intrusion Detection Environment (AIDE) Advanced Concept Technology Demonstration (ACTD), the Extensible Prototype For Information Command and Control ($EPIC^2$), the Air Force Enterprise Defense (AFED) system, and the DARPA Cyber Command System (CCS – actually the DARPA BAA 97-11 integration project went through numerous name changes through the years).

While the entire Network Radar effort included efforts to build novel architectures used to quickly assembling special purpose network sensors as well as a provide for a number of novel analysis techniques, most people tried to pigeonhole our work as simply a network-based intrusion detection system, and in the end, that is how most of the larger integration efforts used our software. The central piece these organizations used as an intrusion detection system was an application called nrat, so the technology behind that particular technology dominates this report.

Nrat, which stands for Network Radar Audit Trail, began life as a simple program to generate an audit trail of network activity. The audit trail was written to a file. Later, other tools would analyze this audit file. For example, if at some point someone notices that a classified document was available on a public FTP server, the security administrator could search through nrat's network audit logs generated to determine who put the file there and who may have downloaded the document.

Over the years, prodded by users at Rome Labs and DARPA, nrat evolved into a highly configurable and interactive multi-purpose sensor that also includes intrusion detection capabilities.

Nrat's intrusion detection capabilities includes detecting known patterns in data streams (string-based detection), access to tagged ports or tagged hosts, access to tagged remote procedure call (RPC) functions, access to certain CORBA operations, and basic vertical and horizontal scanning. Vertical scanning is accessing multiple ports on a single machine, and horizontal scanning is accessing multiple machines. These were the capabilities DARPA used.

In addition to the basic intrusion detection capabilities listed above, nrat generates detailed information about network activity including user names logging into services and commands used in FTP and remote shells, detects users hopping across networks via telnet and rlogin, and taps into and potentially terminates interactive network sessions such as telnet and FTP. Also, Network Radar includes additional programs that add value to nrat's results, including the ability to generate transcripts of login sessions and send them to a remote security management console. Rome Labs' development groups exploited these features as well as the basic intrusion detection features.

The information from all these capabilities is provided to remote systems via several network servers. The network servers generate data in an ASCII format as well as a more compact and versatile binary format. The ASCII-oriented servers can help in debugging nrat as well as help developers learn the type and rates of information that might be sent to their security

management stations.. To make sure nrat is generating what you think it should be, simply telnet to the server port and watch the results in readable ASCII stream across your screen. Rome Labs and DARPA primarily used the ASCII output. Later, with the help of C-based bridging software we wrote, Rome Labs switched to the more efficient binary servers.

Nrat also serves as a research platform, allowing us to test new analysis approaches. One example is that we added a stream analyzer to search for similar content crossing the network in multiple connections. This technique could be useful for detecting a new attack tried against multiple servers or a worm spreading across the network. In Section 4 we discuss this in detail and describe how an early version of this technique tracked the Love Bug worm spreading through Rome Labs.

Finally, all these features and more (e.g., what protocols are analyzed) are configurable through over 70 switches that can be set in a file or overridden on the command line. A summary of all these switches is provided in Section 6.1.

The following is a roadmap to the rest of this report. Section 2 introduces the reader the Network Monitoring Framework, an extensible object-oriented toolkit for rapidly building custom network monitoring applications. Section 3 goes into the details of pattern matching systems (read: signature-based intrusion detection systems). It looks at some of the reported problems with them, and then it shows several approaches we took with Network Radar to address these problems. Section 4 introduces cluster-based detection. Network Radar also serves as a research platform, and this section discusses one approach we took to detect (and potentially thwart) fast spreading new attacks such as worms. Finally, Section 5 shows how to install Network Radar, and Section 6 shows how to run the primary Network Radar application, nrat.

2 Network Monitoring Framework (NMF)

The Network Monitoring Framework (NMF) is a toolkit of C++ objects that can be assembled into custom network analysis applications. Developers can also extend the set of objects by subclassing any of the existing object classes, thus providing their own custom functionality without the need to write an entire application.

The NMF was inspired by our earlier work developing the original Air Force Automated Security Incident Measurement (ASIM) sensor, DISA's Joint Intrusion Detection System (JIDS), and the Lawrence Livermore National Laboratory's Network Intrusion Detection System (NIDS). These were originally all the same system, the UC Davis Network Security Monitor (NSM).

As the system started to be deployed by various groups, we found that different people needed it to accomplish different tasks depending on their missions. Some people (site administrators) deployed the sensor as originally designed, a long-term intrusion detection sensor for an organization. These people would start the sensor running in one location, let the anomaly detector build of profiles of common data paths, and identify and manage attacks as they occurred over time. Another group (ASIM folks at Kelly AFB) deployed the sensor at a wide range of sites to collect global-level intelligence about attacks and track large-scale trends. They would typically not handle day-to-day attacks at a site. Other groups (DISA and AFCERT) would only be called when an attack was discovered, and they would literally hop on a plane and fly to the site that was attacked. These people knew what systems had been attacked, and they just wanted to closely monitor those few systems, tap into live connections, and terminate the connection if the attacker got too close to sensitive information. In the meantime, the group would open an investigation and try to learn who was behind the attacks. Yet another group of people would use the tool as perform forensics on a newly discovered attack.

Unfortunately, a single tool did not, nor could not, perform all these different tasks well. Thus we were motivated to develop the NMF. The ultimate goal was to custom build a specific application that was tailored to each group's needs. While in the end we did not build many specific applications for different people, the NMF library still retains this capability, and the complete framework is installed when a user installs the Network Radar application suite.

Figure 1 shows a subset of NMF objects that might be actively monitoring a network. At the bottom is the PcapTap network tap. Taps are designed to read packets from various sources. This PcapTap takes advantage of the libpcap library. We also have network taps for other sources, including one to read packet files created by the Solaris *snoop* program. Above the tap are four protocol layer objects; Ethernet Layer, IP Layer, TCP Layer, and UDP Layer. NMF also supports an ICMP Layer object that is not shown. Each protocol layer is responsible for parsing its part of the packet header and determining to which object the packet should be passed (e.g., the IP Layer can pass the data to either the UDP or TCP Layer). Each protocol layer also supports any number of proxy classes. In this example, only the IP Layer has a proxy, the IP Layer Attack Proxy. This particular proxy is tuned to detect attacks specific to the IP protocol layer.

All of the objects just described, the tap, protocol object, and protocol proxy objects, are created once at the beginning of the program. They are persistent throughout the run of the program. The NMF also has a whole class of dynamic objects that are created on the fly to analyze individual sessions. These are subclasses of NmfStream, and they do much of the novel analysis for the Network Radar applications. In Figure 1 there are three TCP/IP sessions being tracked by the TCP Layer. The right most session has a stack of five NmfStream objects analyzing its data stream. At the bottom is the TcpStream which tracks when the connection starts and stops, how much data and how many packets have been exchanged, and collects other general statistics about a TCP/IP connection. The TcpStream object also removes any duplicate data before passing the data to a higher level NmfStream object. After the data passes through the TcpStream object it is handed to the TelnetStream object. This object removes the telnet negotiation data from the stream and hands the modified data stream to the next NmfStream object, LoginStream (see Section 3.3.3 for more information about the TelnetStream object). The LoginStream identifies user names (and optionally passwords) used by the person trying to login. The session data stream is then passed to the StringStream to identify strings in the data, and finally the ThumbprintStream is handed the data to determine if this user is actually logged into multiple connections. In all, NMF supports 36 different stream classes to perform various types of analysis on network sessions.

If a developer thinks up a new novel analysis to perform on a connection, he only has to subclass the NmfStream object, write the code specific to perform his analysis, and then link the new object to an existing application like nrat (see Section 6). All the rest of the functionality for a network sensor already exists, so this greatly accelerates the time to get a new analysis technique in the field.

One example of this quick turn around in developing and deploying a new technology is our experience with thumbprinting entire network sessions. We had not originally proposed doing this research. In fact we had only thought about it while deploying Network Radar in the field. However, we thought the technique could be useful for detecting the same attack in multiple connections, and therefore identifying attacks we have never seen before. We rapidly built a prototype and deployed it as part of nrat program, and it successfully detected and tracked the Love Bug worm as it went through Rome Labs (see Section 4).

What is not shown in Figure 1 are a large number of supporting classes to create a complete application. For example, NMF has classes to accept and manage incoming

connections from clients such as security management servers. NMF has a large number of report classes to notify other applications about specific events, and these report objects can be serialize into a buffer, archived on disk, or transmitted across a network. It also has classes to provide a general-purpose filter (or throttle) to prevent clients from being overwhelmed with attack reports. In total, NMF has 278 different classes.

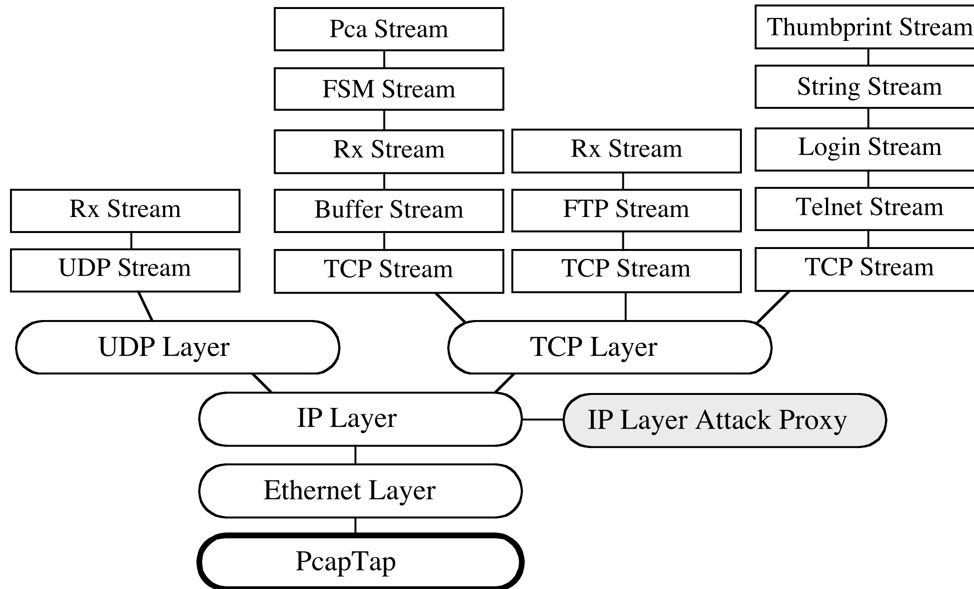


Figure 1: Network Monitoring Framework

3 Pattern Matchers

3.1 Introduction

Probably the single most successfully deployed intrusion detection technology has been the content-oriented signature detectors in network-based sensors[1]. That is, the sensors look for a string in packet data. While other techniques such as specification-based detection[2] and sequence-based detection[3] have shown success in laboratory trials, they have not been successfully adopted into operational environments. Network-based signature systems from the Network Security Monitor (NSM) in 1991, commercial systems such as ISS's RealSecure and Cisco's IDS series, and open source efforts such as Snort have dominated the intrusion detection marketplace over the years.

However, despite the market success the technology has enjoyed, the approach is not without its critics. The most frequent complaint is that the approach generates a large number of false-positives. Another frequent criticism is that the approach cannot detect new attacks or subtle variations in existing attacks.

---

[1] In addition to strings in packets' payload, the term signature is often applied to examining the protocol header fields. We are not addressing this type of signature here.

[2] C. Ko, M. Ruschitzka, and K. Levitt, "Execution Monitoring of Security-critical Programs in Distributed Systems: A Specification-based Approach," *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 134-144.

[3] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for Unix processes," *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy.*

Unfortunately, while the issues of false positives in particular and weaknesses in network-based signature detection in general have been discussed a great deal, rarely do these discussions focus on the fundamental issues. Throughout Section 3 we shed some light on these issues, and we examine how Network Radar addresses some of them.

3.2 Signature Pyramid

Figure 2 shows a pyramid of issues that must be considered when analyzing the performance of signature-based systems. The three areas, starting at the bottom, consist of (1) the actual data analyzed by the signature system, (2) the engine (or algorithm) used to look for patterns in the data, and (3) the actual pattern searched for in the data. On the left of the pyramid the arrow indicates that at the top the actual signature is actually the least important component to a system's ability to detect an attack. On the right side of the pyramid the arrow indicates that the data source is the hardest component to change and the signature is the easiest to change. We explored each of components of this pyramid in detail below.

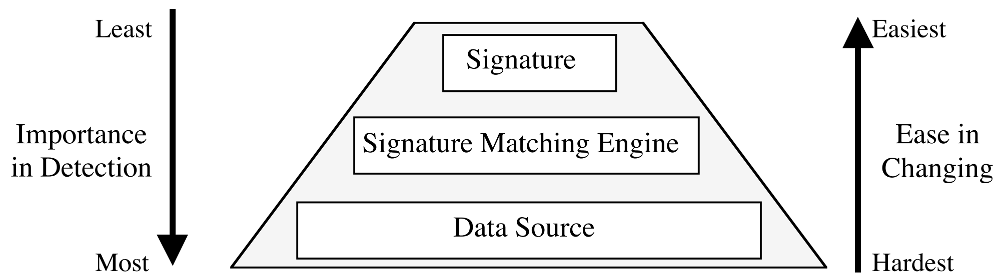


Figure 2: Signature Pyramid

3.3 Data Source

The source of the data is the most important component in detection, but it is also the hardest to change. In the next few section we look at some of the problems with respect to the data source faced by network-based sensors such as Network Radar. In Section 3.3.1 we briefly look at intractable problems with the data source that cannot be effectively solved. The problem with increasing amounts of encrypted network traffic is of particular concern. In Section 3.3.2 we examine a problem that most network sensors cannot address but which the Network Monitoring Framework used by Network Radar handles easily: dealing with context. In Section 3.3.3 we examine the problem of inline protocol data, specifically with respect to telnet connection. Embedded telnet protocol data can prevent most network-sensors from detecting some patterns, but Network Radar can modify the analyzed data stream to eliminate this problem.

3.3.1 Intractable Problems

If the attack is not visible in the data stream, no amount of work on a signature engine or signatures will help in detecting the attack. For example, if a user is logged directly into the computer from a terminal and attacks the system, there is simply no way for a network-based sensor to detect the attack because no packets are generated. Also, if a site deploys a single network sensor at the network perimeter (a common practice), but an attack is completely internal and does not cross the perimeter (e.g., the attacker is an insider, which is a common problem), then it is impossible for sensor to detect the attack. In short, unless the attack generates data that the sensor can see, no sensor of any type can detect the attack.

In addition to the problem of no data, there is the problem of encrypted data. In the early years of network-based intrusion detection, there simply was not much encrypted data. However, at many sites today encrypted logins (e.g., via ssh) are more common than plain-text logins (e.g.,

telnet and rlogin), and encrypted web traffic is growing rapidly. For example, at one Air Force installation we monitored, all web servers on the inside of the firewall could only be accessed through encrypted connections. A network-base sensor that searches for patterns in the data stream will fail when looking for them in encrypted data streams. In other words, at this Air Force site where access to internal web servers must be encrypted, a network sensor could not detect attacks against these web servers.

3.3.2 Analyzing without Context

While the first two issues mentioned, lack of data and encrypted data, are effectively intractable problems, there are other problems with network-based sensors that can be addressed. One egregious error by a network sensor was a signature used by ISS's RealSecure in the mid 1990s to detect attacks that exploited the DEBUG vulnerability in sendmail[4]. Turning this signature on would flag as an attack any sendmail connection with the word "debug" anywhere in the connection.

There were at least two major flaws with this signature. First, the DEBUG sendmail vulnerability was a flaw first exploited by the Morris Worm in 1988. Probably by 1989 there were almost no machines that still contained this vulnerability. By the late 1990s there were almost certainly no attacks against this 1988 vulnerability, so the chances that the signature would detect a real attack were effectively zero. All alerts of this attack would therefore be false positives, and since the word "debug" is common in email, there would be a lot of alerts. The lesson here is that over time, as an attack is no longer active on a network, signatures for the attack should be retired.

The second flaw, and the one more directly related with our Network Radar work, is that the RealSecure signature could not understand the context in which the string "debug" was used in a sendmail connection. A sendmail connection has two phases: a command phase and a data phase. The data phase is where the actual content of the email message is transmitted, including header information such as the Subject line. The command phase is where the sendmail client and sendmail server communicate prior to and after sending the contents of the actual email message.

Figure 3, the transcript of actual data sent from a sendmail client to a server, illustrates these modes. The areas in the grey boxes are the commands sent by the client. The text between the grey boxes is the content that is delivered to the users' mailboxes. An attack against the 1988 sendmail vulnerability would have sent the string "debug" during the command phase of the connection (the grey boxes).

[4] This problem was told to us by Boeing engineers working with RealSecure.

```
EHLO [128.120.56.19]
MAIL FROM:<todd@NetSQ.com>
RCPT TO:<todd@netsq.com>
RCPT TO:<antoinette@netsq.com>
RCPT TO:<heberlei@netsq.com>
DATA
User-Agent: Microsoft-Entourage/10.1.0.2006
Date: Mon, 05 Aug 2002 10:20:43 -0700
Subject: Testing sendmail
From: Todd Heberlein <todd@NetSQ.com>
To: Todd Heberlein <todd@NetSQ.com>
CC: Antoinette Heberlein <antoinette@netsq.com>,
     <heberlei@netsq.com>
Message-ID: <B97401FB.42AE%todd@NetSQ.com>
Mime-version: 1.0
Content-type: text/plain; charset="US-ASCII"
Content-transfer-encoding: 7bit

This is a simple test of the sendmail protocol.  These words
are part of the email's body.

Todd

.
QUIT
```

Command Phase (EHLO through DATA)

Command Phase (. and QUIT)

Figure 3: Sendmail Commands and Content

RealSecure simply treated both types of data, command and content, equally, so it could not distinguish between “debug” as a command and “debug” as simply part of an email message. Network Radar can make this distinction through inline parsers.

An inline parser is a subclass of the NmfStream class (see Section 2). It is a stateful object, and one is allocated for each connection to be analyzed. The object parses the data stream and applies the appropriate analysis to the appropriate parts of the connection. While we did not create a sendmail parser as part of the standard Network Radar distribution (we were not to worried about the DEBUG attack), we did create one for web traffic (http) and one to capture login information (user names and passwords entered at “login:” and “password:” prompts). The http parser can distinguish between the primary command (GET, POST, and HEAD), the header information, and any content sent along by the client (e.g., the form data sent with a POST command). In our case, we applied thumbprints analysis to the URL supplied in a GET or POST command.

To summarize this issue, while the necessary data to detect an attack may be available to a network analyzer, in some cases the network analyzer may need to understand additional context information about the data stream such as whether a sendmail connection is in command or content modes. In short, the analyzer should parse the data and apply different types of analysis to different portions of the connection. Most network analyzers such as RealSecure cannot (or at least could not) do this, while Network Radar, through the use of inline parsers, can.

3.3.3 Inline Protocol Data

Another data source problem faced by network sensors is that the data is not always clean. That is, there is often additional data beyond the content you want to analyze embedded in the data stream, and this additional data prevents a network analyzer from being able to detect the content string. The most well known problem in this area is with telnet connections; however, other protocols also have the problem.

Figure 4 illustrates the problem. Panel A shows the data from a telnet connection during an initial login attempt. The periods indicate non-printable data bytes. These non-printable data bytes, combined with the quotation marks, are part of the telnet negotiations that go on between the client and server and are not displayed to the user. The NeXT operating system generates more negotiation data than most, but all versions of telnet on all operating systems that we have observed generate some telnet negotiation data. If you have seen transcript printouts of telnet logins by sensors such as ASIM, you will probably have seen funny characters, especially at the beginning of the connection. This funny data may have been telnet negotiation data.

If you are trying to look for a particular user logging in (e.g., heberlei), you might search for the pattern "login: heberlei". Unfortunately, in this case, that search would fail because the telnet negotiation data in the middle of the pattern would prevent the match from being detected. Likewise, our NmfStream object that extracts user names from interactive login programs such as telnet would accidentally add all the negotiation data to the user's login name.

Once again, however, because of the Network Monitoring Framework's architecture, we were able to construct an NmfStream object that strips the telnet negotiations data from the stream before passing the rest of the data to other analyzers such as pattern matchers and login name detectors. Network Radar can properly detect the string "login: heberlei", and it can generate transcript files without all the distracting telnet negotiation data in the middle of it (see Figure 4, panel B). Other sensors cannot do this.

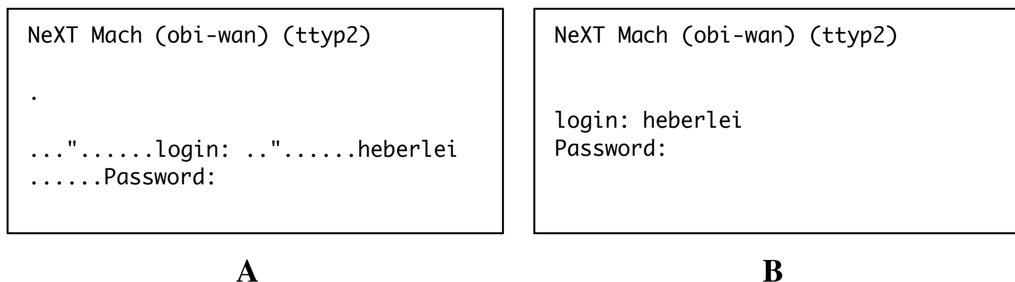


Figure 4: Telnet Negotiation

3.4 Signature Matching Engine

After the availability and quality of the data, the next most important component in a pattern matching system is the pattern-matching engine. There are at least three important questions to ask about any pattern-matching engine:

What is the engine's efficiency?

What are the engine's requirements?

How expressive are the patterns that run on the engine?

An engine's efficiency is typically expressed in terms of how much work must be performed for each byte of input it must process. For example, in some pattern matching algorithms such as naïve string matchers (e.g., strcmp()) and non-deterministic regular expression

matchers, a single byte of network traffic might have to be analyzed several times. The engine's requirements include whether the entire pattern in the data stream must be in memory at the same time for a match to occur and how much storage is required to encode the pattern. How expressive the engine is describes how complex a pattern the system can handle. In general, this has usually come down to either simple strings (used by most intrusion detection systems) or regular expression.

While engine's expressiveness usually comes down to simple strings vs. regular expressions, we do want to touch on one additional point outside this current thread. Regular expressions (considered the most powerful matcher in this line of analysis) only support context-free pattern matching. Because our NmfStream objects are stateful (e.g., we can provide the context), we can record additional state and create an even more powerful pattern matcher. For example, we have an FTP stream analyzer that can apply regular expression matches to only directory names accessed by a user (as opposed to any type of data) because we parse the FTP protocol. So in effect, while in this discussion we talk about regular expressions being the most powerful analysis technique (and more powerful than most used today), Network Radar has already surpassed this.

In the following sections we look at four pattern matching engines. The first one, Boyer-Moore, is used by the Snort intrusion detection system. Network Radar does not support this algorithm, but it would be trivial for us to add it. At this point Boyer-Moore does not add any advantage that we do not already have with the Knuth-Morris-Pratt algorithm we implemented. The other three algorithms are all available in Network Radar.

3.4.1 Boyer-Moore (BM)

The Boyer-Moore (BM) algorithm is used by the Snort intrusion detection system and is favored by many in the computer science for both its efficiency and relative simplicity. While in theory BM may have to process the same input bytes many times, a process known as backtracking, in practice it rarely does. BM can only represent simple strings such as "hello". If the pattern to be searched for is in the data stream, the entire portion of the data stream containing the pattern must be in memory at the same time. In practice, from a network monitoring perspective, BM cannot match a pattern than spans more than one packet. For most attacks this isn't a problem, but in the early days of ASIM, matching patterns in keystrokes was very important, so BM would not work. Also, if an attacker wanted to, they could easily divide their attack across many packets potentially rendering the current Snort useless.

3.4.2 Knuth-Morris-Pratt (KMP)

The Knuth-Morris-Pratt (KMP) algorithm was used in the UC Davis Network Security Monitor (NSM) and its descendents, including the Air Force's ASIM sensor, Lawrence Livermore National Laboratory's NIDS, and the Defense Information Systems Agency's JIDS. KMP is guaranteed to process each byte in the data stream only once, and is therefore always faster than the naïve string matching algorithms. KMP also has a very efficient footprint for the memory required for each pattern. Like BM, KMP can only match simple strings such as "hello". Perhaps the most important feature of KMP is that it can trivially match patterns that cross packet boundaries. This allows it to capture keystroke patterns in services such as telnet and rlogin, and KMP cannot be as easily fooled by attackers as BM can be if an attacker divides his attack across many packets..

Network Radar supports a KMP matching engine via the KmpStream object class.

3.4.3 Regular Expression (RX)

Regular Expression (RE) matchers were initially used by the WheelGroup's NetRanger sensor. The critical factor in an RE matcher is that it can search for much richer patterns than either BM or KMP. For example, it can match "GET.*cgi-bin.*etc/passwd", a single pattern that can capture a large number web attacks against CGI scripts. Regular expression pattern matcher can be implemented as non-deterministic finite state machines (NFSM) or deterministic finite state machines (DFSM). An NFSM implementation (e.g., used by WheelGroup's NetRanger via the regex series of UNIX library calls) has an efficient footprint in memory but can require many passes across each byte of input as it perform backtracking. An NFSM implementation also requires that the entire pattern in the data stream be in memory at the same time. In other words, it cannot detect patterns that cross packet boundaries. A DFSM implementation, on the other hand, may require a large memory footprint for each pattern, but it only processes each byte once (so it is efficient), and it can detect patterns that cross packet boundaries.

Network Radar supports two regular expression pattern matchers: one with an NFSM implementation and one with a DFSM implementation (discussed in the next section). The RxStream object class is a non-deterministic finite state machine implementation of a regular expression pattern matcher. It uses GNU's librx library which must be installed on the system (see Section 5.1.2).

3.4.4 Finite State Machine (FSM)

Network Radar also supports a deterministic finite state machine regular expression pattern matcher via the FsmStream object class. As mentioned in the previous section, this implementation often requires larger amounts of memory than the BM, FSM, or even the non-deterministic RE matchers, but it only processes each input byte once (so it can be much more efficient than a NFSM implementation) and it can match patterns that span multiple packets. However, the biggest advantage of our FsmStream matcher comes from something we call signature compression. This is covered in the next section.

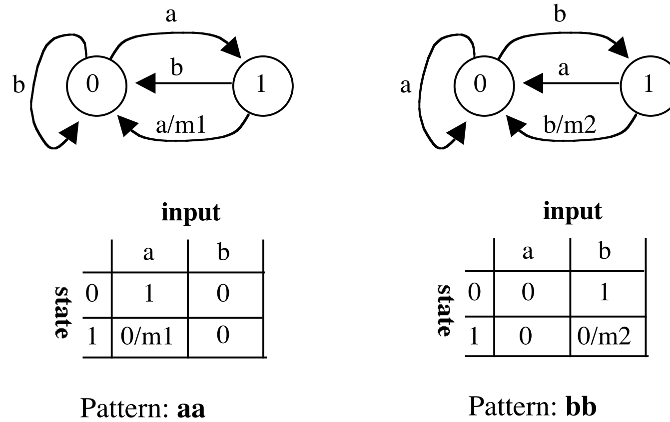
3.4.5 Signature Compression

Besides Network Radar's FsmStream advantages in efficiency (no backtracking), detecting patterns that cross packet boundaries, and expressiveness (matching regular expressing instead of just simple strings), the FsmStream can match multiple patterns simultaneously. In other signature-based systems the first signature is checked against the data in a packet, then the next signature is checked against the same packet data, and so on until all relevant signatures are checked against the data. Checking for 30 signatures takes 30 times longer than checking for just one signature.

The FsmStream class uses an approach we call signature compression that merges two or more signatures into a single super signature (sometimes called a "meta signature"). The FsmStream class can process this super signature is the same amount of time it takes to check a simple signature. In other words, checking for 30 signatures takes the same amount of time as checking for just one signature!

As networks become more congested, or as network speeds increase, network-based sensors are prone to drop packets and therefore miss potential attacks. As a sensor spends more time analyzing a single packet (e.g., to look for many signatures in the packet's data), it increases the chances that packets and attacks will be missed. Signature compression reduces the analysis time required for each packet, thus reducing the chance of missing an attack.

Figure 5 through Figure 8 illustrate the process of merging two signatures into a single signature. Figure 5 shows a state machine diagram and table for two patterns: “aa” and ‘bb”. We are using trivial signatures here because the steps can be very difficult to follow with anything more complex. The state machine at the top of the left column matches “aa”. We begin in state 0. If we get an input of ‘a’ we move to state 1, but if we get an input of ‘b’ we remain in state 0. If we are in state 1 and get an input of ‘a’, we return to state 0 and report the match m1 (“m1” will stand for matching the pattern “aa”). On the other hand, if we are in state 1 and get an input of ‘b’, we silently move back to state 0. The table below the state machine shows the same information in tabular form, and the state machine and table in the right column show the matching information for the signature “bb”.



state	a	b
0	1	0
1	0/m1	0

Pattern: **aa**

state	a	b
0	0	1
1	0	0/m2

Pattern: **bb**

Figure 5: Two Simple Patterns

The first step in compressing two signatures is to take the cross product of the two signatures. Figure 6 illustrates the process. Whereas the tables for patterns “aa” and “bb” each had two states (0 and 1), the new table has four “meta states”: (0,0), (0,1), (1,0), and (1,1). In general, when merging a signature that has *n* states with a signature that has *m* states, the resulting compressed signature will initially have *n*x*m* states.

In the “meta states”, the first number represents a state from the first signature, and the second number represents the state from the second signature. For example, the “meta state” (0,1) is associates with the first signature’s state 0 and the second signature’s state 1. If we look at the table for the first signature (“aa”), when we are in state 0 and get an input of ‘a’ we move to state 1. Likewise, if we look at the table for the second signature (“bb”), when we are in state 1 and get an input of ‘a’ we move to state 0. Thus, in the new table, if we are in state (0,1) and get an input of ‘a’ we move to state (1,0).

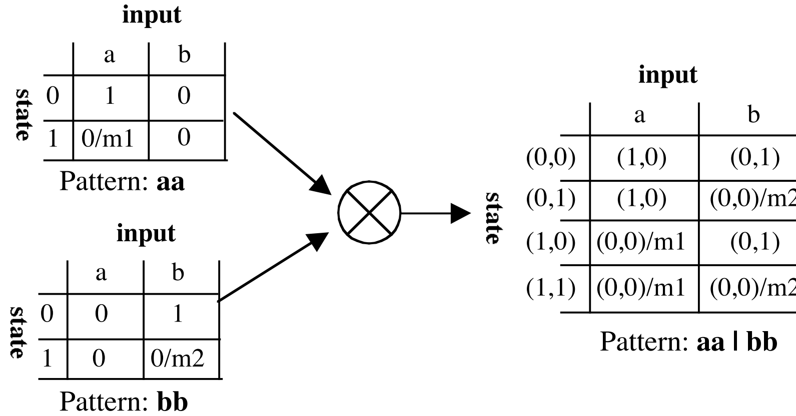


Figure 6: Signature Cross Product

Figure 7 shows the table and state machine representation of our new signature. It detects the patterns "aa" and "bb" simultaneously. If we look closely at the state machine representation on the right, we notice that there are no inputs to the "meta state" (1,1). Since we always start in state (0,0), there is no possible input stream that can lead us to state (1,1). We can also see this in the table; although, it is not as obvious to see. Inside the table there are no "next state" values of (1,1). We can therefore eliminate "meta state" (1,1) in a process called pruning.

We prune the graph by starting at state (0,0) and performing a breadth first search though the state machine graph. At the end of the search, any state not reached by the search can be eliminated.

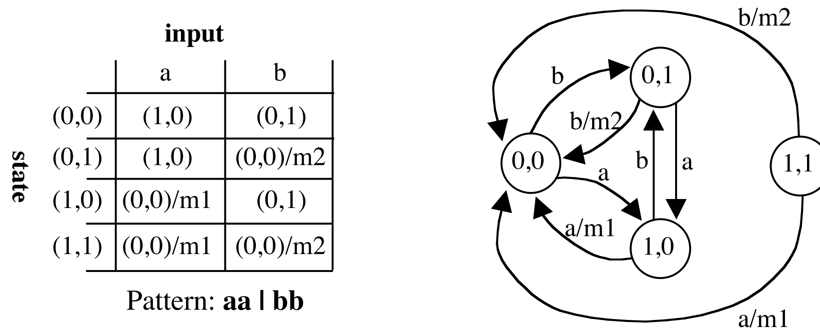
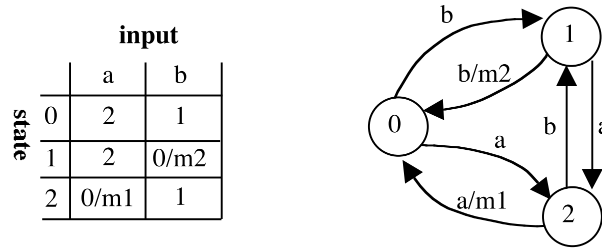


Figure 7: Table and Diagram for "aa | bb"

Figure 8 shows the results of the pruned signature. We have also renamed "meta states" (0,0), (0,1), and (1,0) to states 0, 1, and 2. The resulting graph efficiently searches for patterns "aa" and "bb" simultaneously. For example, if the input stream is "bb", we begin at state 0, move to state 1, and then move back to state 0 while reporting match m2. If the input stream is "baa", we begin at state 0, move to state 1, move to state 2, and them move to state 0 while reporting match m1.



state \ input	a	b
0	2	1
1	2	0/m2
2	0/m1	1

Figure 8: Eliminate Unreachable States

Network Radar includes a program called merge_fsm that takes two FsmStream signature representations and merges them into a single super signature. The resulting super signature can be merged with other FsmStream signatures or super signatures. As more and more complex super signatures are merged, however, the time to compute the next super signature grows considerable. While theoretically there is no limit to the number of signatures that can be merged into a single super signature, in practice we have found that the time required to merge more than 30 signatures into a single super signature can test one's patients.

3.5 Signatures

Finally, we have reached the last component in our Signature Pyramid introduced in Section 3.2, the actual signatures used to detect a particular activity on the network. More often than not, when a signature-based system generates a large number of false positives, the problem is not the signature system (e.g., the data stream and the signature engine) but rather the signatures themselves. The signatures simply are not refined enough to properly detect the activity of interest and skip the unrelated activity. Fortunately, because neither the data stream nor the signature engine needs to be modified, this problem is usually the easiest to resolve. Below is a brief historical example to illustrate the issue.

The second network-based signature ever created, the string "daemon", was added to the Network Security Monitor (NSM) around 1990. The purpose of the string was to detect someone opening, copying, transferring, or otherwise manipulating a password file, because at that time weak passwords were the primary method for penetrating systems and usually the first target of an attacker. The string "daemon" was in every UNIX-based password file. The pattern was not designed to detect a specific attack but to detect an indicator of possible misuse (accessing a password file). The first day the string was used we detected an attack using the signature.

Unfortunately, the string "daemon" occurs in many sources not tied to the password file. For example, when a mail message bounces, the error message accompanying the bounced message would day it came from the mail "daemon". The result is that the string generated many false positives.

Fortunately being both the operator of the sensor and the creator of the signatures, I was able to easily compare three pieces of information: (1) the signature pattern, (2) the activity I was trying to detect (the content of a password file), and (3) samples of data that were creating false positives. From this I made a simple extension to the pattern by appending a colon, ':' to the end of the signature. "daemon**:**" had a much lower false positive reporting level than "daemon," but it still easily detected someone opening or transferring a password file.

The lesson is that signature systems deployed with stringent quality assurance testing of the signatures can still be very valuable. Combine this with other advances made in Network Radar such as regular expression pattern matching across packets, signature compression, and protocol parsing to add context, and signatures can be a very powerful detection force.

3.6 Pattern Matching Summary

To summarize, while quality assurance of the signatures used inside a signature-based system is outside the scope of the Network Radar work, it is important to understand this is an important issue when analyzing the efficacy of a signature-based system. Too often we have heard "*Signature systems produce too many false positives. We need a new technology*." When facing a problem in a signature-based system, the first things to examine are the signatures themselves.

However, once the quality of a signature has reached its limit, you may then want to look at the other components of the Signature Pyramid. Does the signature engine need to be more powerful? For example, moving from a simple string based system such as the early version of ASIM and today's Snort to a regular expression based system such as Network Radar (as implemented by the Network Monitoring Framework's RxStream and FsmStream classes). Or does the data stream need to be processed differently? A parser required to provide context to a pattern matcher might help. For example, Network Radar provides a simple http parser to analyze only the URL component in an http request. Or perhaps the data stream needs to be modified to achieve the correct results. For example, Network Monitoring Framework's TelnetStream class removes telnet negotiation protocol from the data stream so proper string matching can be done in telnet connections.

The issue of how well a signature-based intrusion detection system works is complex. Network Radar tackles several of these issues through novel NmfStream subclass objects.

4 Research Example: Cluster-based Detection

4.1 Introduction

Thumbprints were first proposed in 1992 in a paper titled "Internetwork Security Monitor: An Intrusion-Detection System for Large-Scale Networks"[5]. The primary purpose for thumbprints as proposed in that original paper as well as a follow up paper describing an initial prototype[6] was to track users hopping across the network via interactive login services such as telnet and rlogin. Because of trust relationships between hosts (.rhosts and hosts.equiv), default passwords, weak passwords, and shared passwords across multiple systems, attackers could penetrate deeply into the Internet with these interactive login services.

Unfortunately, by the late 1990s when we started deploying a thumbprint-based sensor (Network Radar) in an operational environment, most sites which were concerned with security had largely disabled the use of the interactive plaintext telnet and rlogin services. When interactive login was necessary, these sites used the Secure Shell protocol, SSH. Because our thumbprint approach analyzed the content of the interactive login sessions (e.g., keystrokes and the data displayed on the user's screen), and because SSH encrypts this content, thumbprint analysis was no longer as effective as it would have been in the early 1990s.

In 1999 and 2000 we investigated other uses of thumbprints, and we applied thumbprints to entire network sessions to look for the same content crossing multiple network connections.

[5] L.T. Heberlein, B. Mukherjee, K.N. Levitt., "Internetwork Security Monitor: An Intrusion-Detection System for Large-Scale Networks," Proc. 15th National Computer Security Conference, pp. 262-271, Oct. 1992.

[6] S. Staniford-Chen, and L.T. Heberlein, "Holding Intruders Accountable on the Internet". Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, CA, 8-10 May 1995, pp. 39-49.

The purpose of this approach is to detect new attacks launched against multiple servers or worms spreading across a network. The thumbprint can even be used as a signature, and when deployed in interdiction devices (e.g., firewalls) thumbprints could potentially be used to stop a newly detected attack.

In section 4.2 we present what a thumbprint is. In section 4.3 we compare thumbprints with the more traditional signature-based approaches. We also discuss what thumbprints can do that classical signatures cannot: detect new attacks by looking at repeated activity. In section 4.4 we present a simple operational model using these concepts to stop a fast moving worm. Finally, in section 4.5 we wrap-up our tour of thumbprints with an actual example of using this approach to track the Love Bug worm sweeping through the Air Force's Rome Labs.

4.2 Thumbprint Concept

A thumbprint is simply a small numerical representation of some content. It is related to the more familiar hash functions and checksums. Figure 9 shows the basic approach. The original content, "The quick brown fox jumped over the lazy dog." is sent through a hash function that generates a number. The original content consisted of 360 bits (8 bits per character times 45 characters) and the result in a single 32-bit number (a typical unsigned integer on most computers).

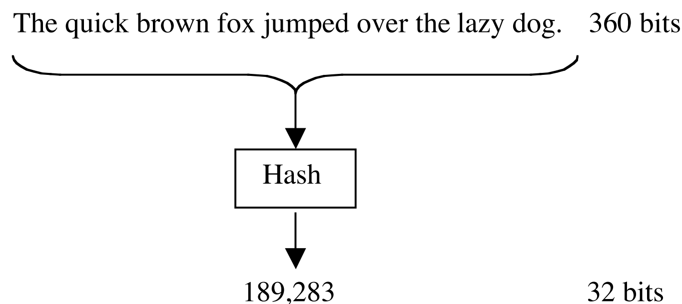


Figure 9: Numerical Representation

This number can serve as type of compact representation of the original content. For example, suppose a document in processed by this technique. A hash number is computed for each sentence in the document, and then we store all the numbers representing all the sentences in a hash table. Later, if a user provides us with a sample sentence and asks us if it is in the document, we can use the following algorithm to very quickly determine the answer. First, compute the hash value of the sample sentence. Second, look in the hash table to see if that number exists in the table. If it is not in the table, then the sample sentence is not in the document and we are done. Third, if we do find a matching hash value in the table, then we examine the sentence (or sentences) in the original document that created the hash value and determine if it matches our sample sentence.

This is roughly the way we use thumbprints to identify duplicate content crossing the network. We compute a thumbprint of each network connection observed (essentially the sample sentence provided by the user in the above example), and then we look in our hash table to determine if we have potentially seen the same content in another connection.

Unfortunately, traditional hash functions do not work well for our problem area. Most hash functions are designed to produce a completely different hash number even if the content only varies by a single byte. For example, in Figure 10 we modified the original sentence by making the word "dog" plural, "dogs." This single change produces a completely different hash number. In fact, in traditional hashing functions, looking at just the resulting numbers would not

indicate that sentences **A** and **B** were very similar, and sentence **C** was completely different than both of these.

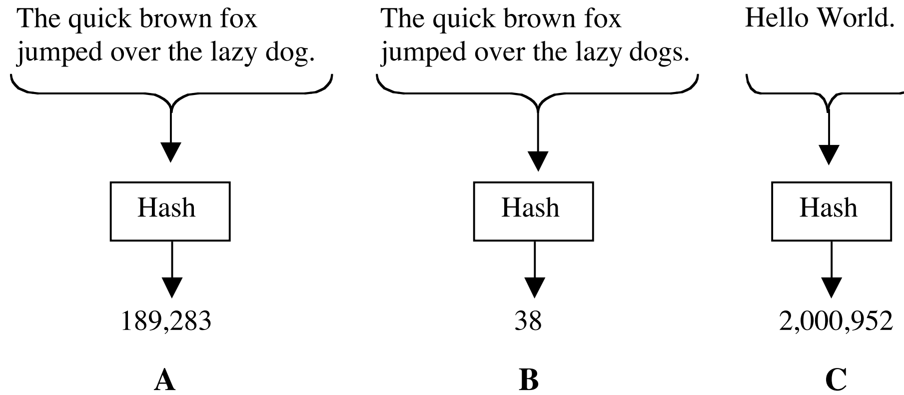


Figure 10: Traditional Checksums

The reason that this traditional approach does not work for us is that the same attack, as observed from our network sensor, might appear slightly differently from one instance to another. For example, in the case of email worms, the content of the email connection would include the sender and recipient's email addresses, and these would vary each time the worm attempts to infect a new machine. Therefore, we need a hashing function that preserves the similarity. Our hash numbers for similar content should be similar. Blain Burnham described this approach as a "robust checksum."

Figure 11 demonstrates how our thumbprint hashing function might treat the three sample sentences from the earlier example. Now the second sentence, which varies from the original sentence by the addition of the single letter 's', has a hash value that is only slightly different from that of the original sentence. Now when the user presents a sample sentence to our hypothetical document analysis system, we can quickly determine whether the exact sentence, *or one very close to the sample sentence*, exists in the original document.

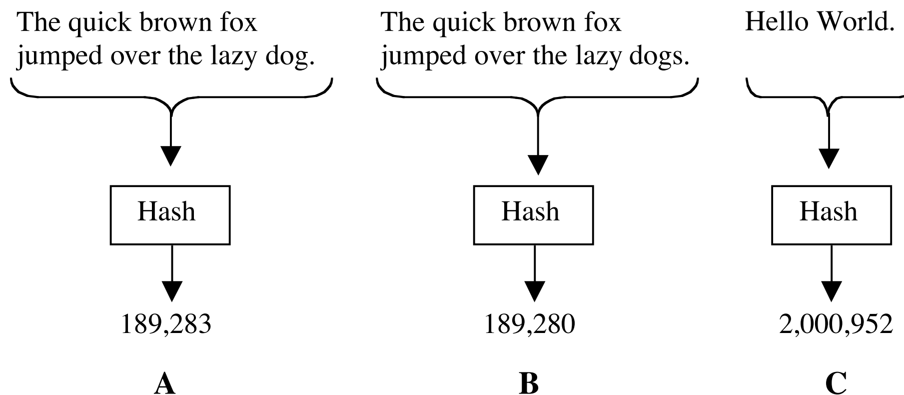


Figure 11: Thumbprint Checksums

These are the basic concepts of a thumbprint: condense a potentially large amount of content to a single number, and similar content should generate similar numbers. There can also be variations. For example, we might condense the content to two numbers instead of one (think of a two dimensional vector). Once again, for similar content, each pair of numbers should be similar. By using two numbers, however, we reduce the probability of a false positive, where two dissimilar content samples would have both pairs of numbers similar. One approach that satisfies

all of these needs, and the one we use to in our implementation, is a popular technique in multivariate statistical analysis called principal component analysis (PCA). Other algorithms could potentially be used.

4.3 Comparisons to Traditional Detection Techniques

The thumbprint approach just described is simply a mechanism that can be used for quickly comparing content. If an exact match is needed, a traditional checksum can be used. If a more robust match is required, our PCA approach can be used. In this section we compare this approach with traditional content-based detection approaches used in intrusion detection systems. We also discuss how this approach can be used in way that traditional signature-based detection cannot be: looking for original attacks on the network.

Most traditional network-based intrusion detection systems use some type of key-word detection algorithm. The early UC Davis Network Security Monitor (NSM), Air Force's ASIM monitor, LLNL's NID, and DISA's JIDS intrusion detection systems all used the Knuth-Morris-Pratt pattern matcher. Snort uses a Boyer-Moore pattern matcher. Early versions of WheelGroup's NetRanger used the default UNIX regular expression matcher, which in turn used a basic non-deterministic, backtracking state machine.

All these approaches essentially do the same thing: they all look for strings in a network data stream. The strings are hand-generated by analysts. And the systems look for each string one at a time. If you double the number of strings you are searching for, you double the analysis time that must be spent on each packet. (The one exception to this is Network Radar's FsmStream pattern matcher described in Section 3.4.5.)

Our thumbprint-based approach, on the other hand, calculates a vector (perhaps just a single number (e.g., a single hash value) representing a one-dimensional vector or maybe several numbers for a multi-dimensional vector). To compare a vector of a new connection against known attacks, we simply compute the distance between our new vector and vectors of known attacks. However, because the vectors are numerical in nature, we can hash the vectors into buckets, so the new vector can be compared against all known attack vectors (perhaps thousands) simultaneously by simply looking into the appropriate hash bucket. Thus, when trying to solve the traditional problem of looking for known patterns, this approach scales to larger signature sets much better than the traditional signature algorithms.

The thumbprint approach can also do something traditional signature-based systems cannot: detect new attacks automatically.

In a traditional operational setting, somehow a new attack is detected. Since most signature-based systems primarily detect what they already know about, new attacks are often "detected" through some other mechanism than the intrusion detection system. Perhaps an attack tool is posted to BugTraq. Perhaps a site captures the new attack with a honey pot. However the new attack is detected, the appropriate evidence is shipped to the analysts (e.g., AFIWC). An analyst carefully examines the attack and then hand-generates a string or pattern that will detect the new attack.

Our approach, on the other hand, can detect the new attack if it is repeated several times, such as when an attack is part of a worm or used against multiple servers (e.g., the attack is sweeping a site). Thus the new attack can be detected in theory on just the second time it passes our sensor. Furthermore, since the vector *is* the signature, the signature is *automatically* created.

In the traditional operational setting, because the attack must be detected "by other means", and the signature must be generated by hand by a human analyst, a good turn around

time from first attack to a signature ready for deployment is a day; at best a several hour turn around time can be expected. This is far too slow to affect a fast spreading worm.

Our approach, on the other hand, may have a signature ready to go within a second of the first observed attack. Thus our approach can be used to interdict fast moving attacks.

4.4 Example Operational Model

Figure 12 shows an example scenario of how this thumbprint-based approach, combined with cluster detection just described, can be used to quickly stop a new worm from spreading. The Air Force network in the upper left corner is attacked multiple times, perhaps by an email worm. Our sensor detects the attack by identifying the cluster of content vectors (1). The vector *is* the signature, and it is reported to a central distribution center (2). The distribution center pushes the signature to interdiction devices such as email gateways and firewalls (3). An email gateway at a university detects the email worm, and places the offending email into a queue to be processed or deleted by a system administrator at a later time (4). Because the detection is automatic, the signature generation is automatic, and the distribution the signature is automatic, the interdiction devices around the globe can interrupt the spread of the worm in minutes from the attack's initial detection at an Air Force network.

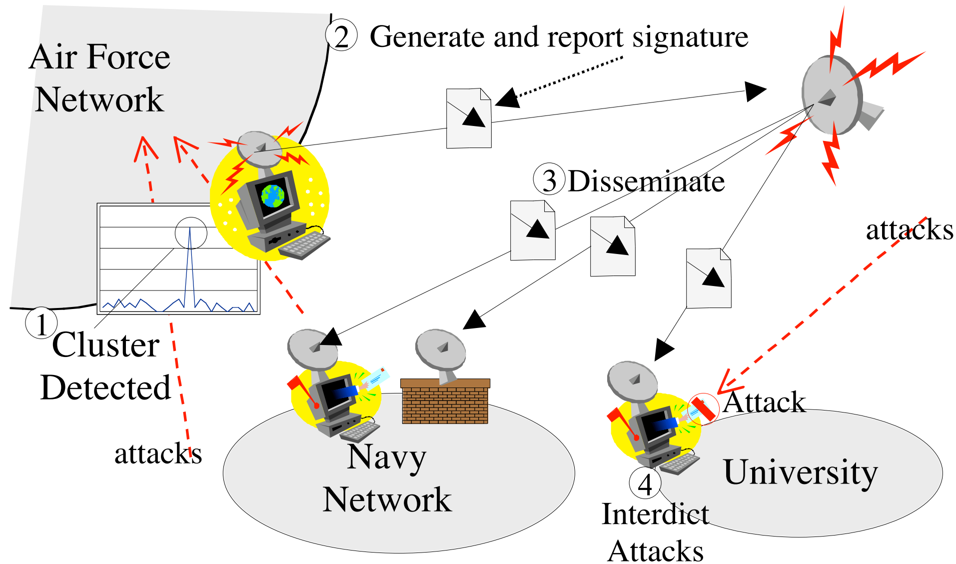


Figure 12: Operational Model

4.5 Example of Detection in Action

Figure 12 shows a hypothetical scenario. However, Network Radar does thumbprint network sessions (as well as interactive logins), and we have operational experience demonstrating that the approach can work. Network Radar was running at the Air Force's Rome Labs on May 4th, 2000 when the ILOVEYOU virus (a.k.a., the Love Bug worm) hit the Internet. We harvested the data collected by the monitor and generated an animated movie that shows the cluster spikes as the worm moved through Rome Labs. Figure 13 shows several selected frames from the movie.

Since ILOVEYOU was an email-based worm, for several reasons the best place to apply our vector-based cluster detection technology is in an email gateway (as opposed to a network sensor). First, a single sendmail connection can and often does transmit multiple email messages over the same connection. However, from our network perspective we treat the entire connection as a single transaction. Second, as email worms travel from user to user, header information is

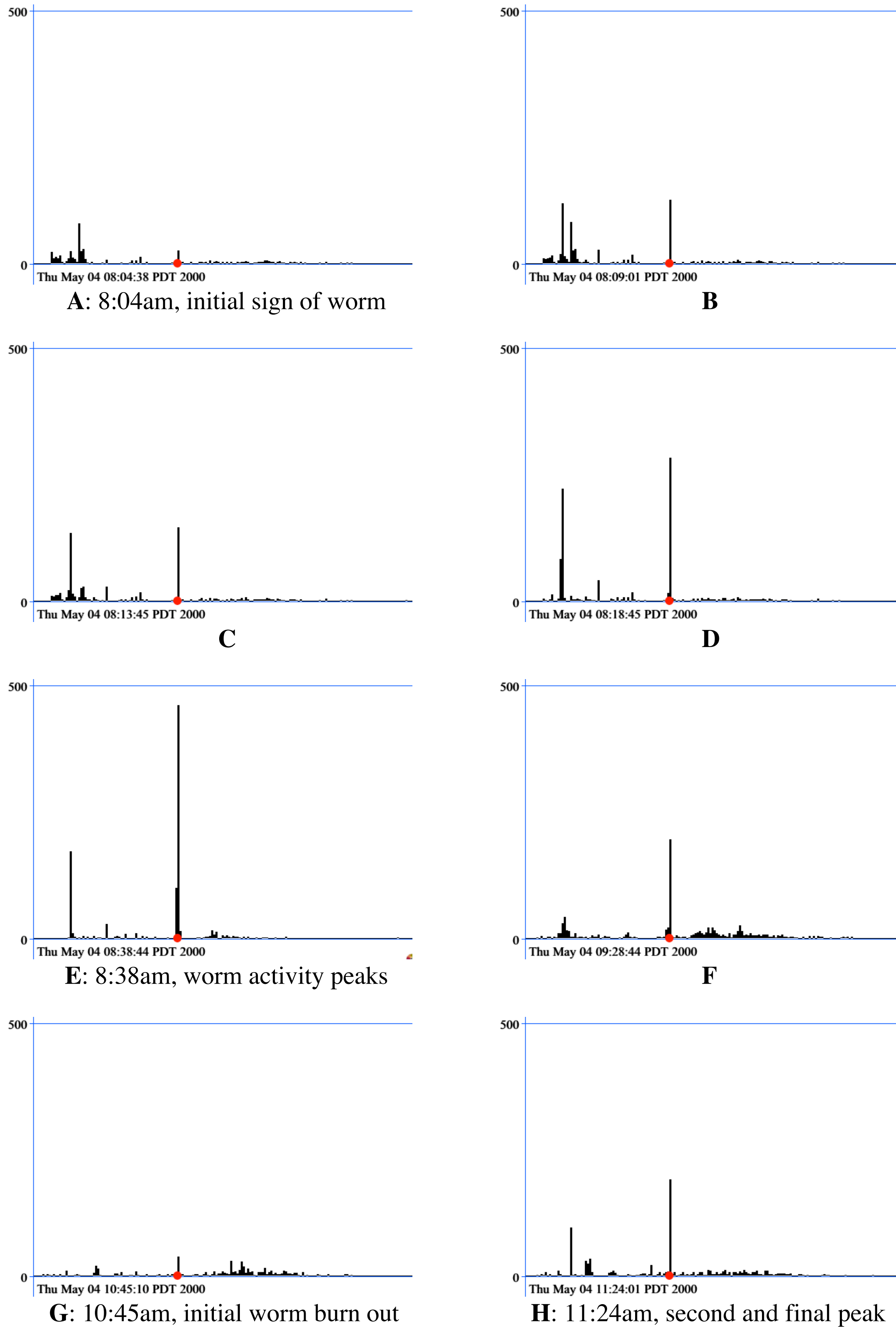
constantly changing, so from a network perspective that cannot distinguish header, body, and attachments in an email message, the representative vector will vary slightly as the headers vary. We could potentially address this with Network Radar since it can easily build a sendmail parser, but we are unlikely to get firewall vendors that will need to interdict the attack later to do the same thing. Third, email systems encode attachments using several different algorithms, so from the network perspective the same worm will look different depending on the encoding scheme used by the email client. An email gateway can decode the attachment and calculate the vector on the original data.

For all these reasons, email-based worms should be detected (and interdicted) via email gateways as opposed to a network monitor that we used. However, despite this, our network monitor version, a first generation prototype at that, was clearly able to identify the spike from the attack.

In each graph in Figure 13, the horizontal axis shows the value a network connection was mapped to (i.e., the hash number described earlier); that is, we were only using a one-dimensional vector to represent each connection. A small circle is placed on this axis to indicate the score our system assigned to a copy of the worm that was sent to us (via our DARPA security mailing list, no less). The vertical axis represents the rates at which connections were observed matching a given vector score. The maximum value on our graph is 500 connections per hour for a given vector value.

Figure 13's panel A shows the initial indications of the virus. We see a small cluster, or spike, at the circle matching the score we received, and we also see a second cluster towards the left. Because the two clusters tracked each other so closely, we believe the second cluster was the same worm where the email attachment was encoded with a different algorithm. Unfortunately (or fortunately depending on your perspective) we only received a single copy of the worm, so we could not verify the second encoding hypothesis. The spread of the worm continues to grow through panels B, C, and D, until it reaches it maximum peak in panel E at 8:38am. Following the peak the virus rate declines in F and reaches almost the level of background noise in panel G at 10:45am. Finally the virus shows another increase beginning around 11:00am and a second smaller peak in panel H at 11:24am. This second period of growth corresponds to the eight o'clock hour on the west coast – just when people are showing up to work and opening their email there.

This real-world example demonstrates two important points. First the worm spread very quickly, reaching a peak about 35 minutes after initial penetration and almost burning itself out in less than three hours. The classic intrusion detection and interdiction systems cannot compete with this time scale. Second, the basic approach of vector-based cluster detection does work, even in this less than optimal situation.



A: 8:04am, initial sign of worm
B
C
D
E: 8:38am, worm activity peaks
F
G: 10:45am, initial worm burn out
H: 11:24am, second and final peak

Figure 13: ILOVEYOU Worm Strikes Rome Labs

5 Installing Network Radar

In this section we describe how to install the Network Radar package. The installation includes the Network Radar application suite as well as the Network Monitoring Framework (NMF). NMF allows a developer to build their own custom applications using the C++ header files and library. As a demonstration of this, we developed an additional set of libraries and applications for Rome Labs to allow their developers to tap into the Network Radar Audit Trail (NRAT) reports from C programs instead of using C++ programs. Finally, before any of these packages can be installed, two external libraries must be installed on the system. These are libpcap, the packet capture library originally from Lawrence Berkeley Laboratory, and librx, GNU's regular expression library.

All of these packages have been provided to Rome Labs and the Defense Advanced Research Agency (DARPA) Technology Integration Center (TIC) on CDs. For ease of installation, they should all be placed in the same directory. The packages are as follows:

PACKAGE	Description
libpcap.tar.Z	The packet capture library used by most network monitoring applications.
rx-1_5.tar.gz	GNU's regular expression library.
NMF.11Nov00.tar.gz	The Network Monitoring Framework (NMF) and Network Radar application suite.
C_NR_BRIDGE.tar.gz	C-based Network Radar bridge libraries. This is a set of libraries to provide a C-based interface (as opposed to C++ interface) to access reports from Network Radar's nrat sensor. All reports will be fully parsed and returned to the calling function as a C data structure.
NR_BRIDGE_APPS.tar.gz	C-based Network Radar bridge applications. These sample applications can be used as is or modified to meet Rome Labs's needs.

The installation below uses the command line and assumes that the user has installed all tar packages in the same directory. The installation also assumes the user is already in that same directory. The installation also assumes that the user carries out the steps from Section 5.1 through Section 5.4 exactly as shown. The Network Radar suite will be installed in the user's home directory under the name NetRadar.

5.1 Installing Third-Party Libraries

Sections 5.1.1 and 5.1.2 show how to build the two third-party library packages used by the Network Monitoring Framework and Network Radar application suite. These must be installed before installing Network Radar.

5.1.1 Building Packet Capture Library

```
% zcat libpcap.tar.Z | tar xf -
% cd libpcap-0.4
% ./configure
% make
```

5.1.2 Building Regular Expression Library

```
% cd ..
% gzip -cd rx-1_5.tar.gz | tar xf -
% cd rx-1.5
% ./configure
% make
```

5.2 Installing NMF and Network Radar

This section shows how to build and install the Network Monitoring Framework (NMF) and the Network Radar application suite. By default, Network Radar will be installed in the user's home directory under the name NetRadar. To override this location, the user can provide the additional command-line option "`--prefix=`*`dir`*" to the configure script below. The value of *dir* is the location where the user wants the package to be installed (e.g., `--prefix=/usr/local/NetRadar`).

```
% cd ..
% gzip -cd NMF.11Nov00.tar.gz | tar xf -
% cd NMF
% ./configure --with-cxx=g++ --with-pcap=../libpcap-0.4 \
  --with-rx=../rx-1.5
% make
% make install
```

5.3 Building C-based Libraries and Applications

The previous steps installed Network Radar; however, some users wanted more specific examples of how their C-based applications can tap into the information from the Network Radar nrat sensor. The following sections show how to build a C-based library and two C applications (nr_client and aide_test). The two applications were developed specifically for users and developers and Rome Labs, but they are general purpose enough that others may find them useful too.

5.3.1 Building Network Radar Bridge Libraries

```
% cd ..
% gzip -cd C_NR_BRIDGE.tar.gz | tar xf -
% cd C_NR_BRIDGE
% ./configure
% make
% make install
```

5.3.2 Building Sample C Applications to Connect to Nrat Sensor

```
% cd ..
% gzip -cd NR_BRIDGE_APPS.tar.gz | tar xf -
% cd NR_BRIDGE_APPS
% ./configure
% make
```

5.4 Copy Read-only Template Files

Finally, several Network Radar template files need to be copied from their read-only directories into the Network Radar Support directory. Because these files are typically modified by users in order to meet their local needs, we wanted to make sure there were always a copies of the original files around (in case they messed up a file and needed to start from scratch again). The following command assumes that Network Radar was installed in a directory called NetRadar in the user's home directory (the default behavior). If the user changes the installation location (see Section 5.2), the user needs to use that directory location instead.

```
% cd ~/NetRadar/Support/Templates
% cp * ../.
```

6 Running NRAT

The Network Radar Audit Trail (nrat) application is the primary Network Radar sensor and usually the only application most users run. Nrat began life as a simple application to generate an audit trail of network activity, but eventually it grew into a highly configurable multi-purpose sensor that most people used as an intrusion detection sensor. In fact, there are over 70 configuration switches a user can set.

Before a user runs nrat, he must set the environment variable RADAR_DIST to point to the Network Radar installation and he must add Network Radar's binary directory to his path. The user can set these values through his appropriate shell configuration files (e.g., .cshrc) so it is performed automatically, or the user can set it from the shell prompt at any time. To set these variables for the C shell, the user can use the following commands:

```
% setenv RADAR_DIST $HOME/NetRadar
% set path=($RADAR_DIST/bin $path)
```

Once these variables are set, the user can start the nrat sensor simply by typing the command "nrat". However, because nrat opens a network tap in promiscuous mode, most operating systems require that the application run with root privileges. This can be done in one of three ways. First, the user can set the nrat application to be an SUID root application. Second, the user can simply `su` to root and run the application. This may require the user to reset the RADAR_DIST and path variables to pick up the Network Radar information. Third, the user can use the `sudo` command to temporarily run the application as root. This last approach is probably the best one, but it may require support from the system administrator to properly install and configure sudo. Thus, to run nrat, use the following command:

```
% sudo nrat
```

Nrat will begin generating audit records in the current working directory. To place the audit trails in another directory, make sure you change the current working directory to the desired location before running the program. By default, new audit files will be created every hour on the hour.

As mentioned, nrat supports many options. These are listed in Section 6.1. Each option has a default value, but you can changes the default value by adding a line in the nrat6.defaults file that can be found in the Network Radar Support directory. The value of a variable is changed by adding a line to nrat6.defaults with the variable name, followed by white space, followed by the value for the variable. Several variables have already been set in the nrat6.defaults file to meet with Rome Labs need.

The nrat6.defaults file was designed to make semi-permanent changes to the way nrat is run. That is, you use nrat6.defaults to set nrat6 to behave as you usually want it to run. If,

however, you want to make a change to nrat's behavior for just one run of the application, you can also use a command-line option. The command-line option will override all other settings. All command line options consist of a flag representing a variable, followed by a space, followed by the value for the variable.

For example, nrat includes the monitor name in each report that it sends over the network. This allows a central security management station to know which sensor sent each report. However, if you want to only test a change to some component in the system, you can set the monitor to some temporary name (e.g., "test_run"), and then you can later remove these records from your security management station's long-term database by simply looking for records from "test_run". To change the monitor name sent with each report, use the following command:

```
% sudo nrat -mn test_run
```

All configurations switches that can be changed are described in the next section.

6.1 NRAT Configurations

This section describes the switches that can be configured by the user. All of these variables have a default value. This default value can be overridden on a semi-permanent basis by adding the variable name and value to a line in the file nrat6.defaults file found in the Network Radar Support directory. Several changes have been added to this file already to suite Rome Labs' needs. Furthermore, a value can be set at run time through a command-line option that will only affect the current run of the application. The command line option consists of a flag followed by a value (see example in previous section). All the information described below can also be found online in the file nrat6.definitions, also in the Network Radar Support directory.

For each variable that can be changed, the following information is provided.

Variable. This is the name of the variable. If you are going to change the default value by adding a line in nrat6.defaults, this is the name you use to identify the variable.

Values. This is the set of values that you can set the variable to. Sometimes this might be a simple "yes" or "no" value, sometimes it may be a list of specific names, and sometimes it may be a range of numbers.

Default Value. This is the default value for the variable. This value can be overridden by adding the appropriate line to the file nrat6.defaults. Both the default value and the value in the nrat6.defaults value can be overridden by using a command-line option. Note: the nrat6.defaults file has already overridden several variables.

Command Line. This is the flag used at run-time to override the default value or the value in the nrat6.defaults file. Each flag must be followed by a value. See the previous section for an example.

Description. This is a brief description of the variable's purpose.

Variable:	UseFileTime
Values:	yes, no
Default Value:	yes
Command Line	-uft

Description:	UseFileTime directs the monitor to base its starting time on the first packet it can read. This works for both packets from a file and from a live network tap. However, if you are reading from a live network tap (1) the very first packet observed will not be processed (it is only used for settin the time), and if you have a *very* quiet network, the program may block until it observes the first packet. Neither of these problems occur when the packet source is a file. If you set UseFileTime to "no", then the current operating system time is used. When looking for sweeps or multiple hops across a network, this can cause problems when the packets comes from a data file.

Variable:	MonitorName
Values:	string
Default Value:	none, name retrieved from file MonitorName
Command Line	-mn
Description:	MonitorName replaces the use of the file $RADAR_DIST/Support/MonitorName. Every attack or detail event includes the name of the monitor which generated it. The default value is NR (for Network Radar). If you run multiple monitors feeding into a central database or network management station, it is probably a good idea to give each monitor a unique name.

Variable:	TapType
Values:	FileTap TcpdumpTap PcapTap DlpiTap SnoopTap
Default Value:	PcapTap
Command Line:	-tap
Description:	TapType defines the type of tap used by the monitor. Each tap is designed to read packets from a specific data source. All taps may not be supported on all platforms. Once started, the monitor cannot switch tap types or data sources.

Variable:	TapInput
Values:	string, depends on the TapType
Default Value:	depends on the TapType
Command Line:	-i
Description:	TapInput defines a particular packet source (e.g., a network interface or a data file) from which a Tap will read packets. Once started, the monitor cannot switch data sources.

Variable:	StopTapOnFail
Values:	yes | no
Default Value:	yes

Command Line:	-stof
Description:	StopTapOnFail directs the monitor to exit when it fails to read a packet. This is typically used when reading from a data file.

Variable:	InstallEpicDetails
Values:	yes | no
Default Value:	no
Command Line:	-ied
Description:	InstallEpicDetails specifies whether the monitor should set up the classic EPIC detail report server. Shutting down a server terminates all existing connections to it.

Variable:	EpicDetailsPort
Values:	number 1-65535
Default Value:	1234
Command Line:	-edp
Description:	EpicDetailsPort specifies the port at which the EPIC Detail server will be installed.

Variable:	EpicDetailsNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-edn
Description:	EpicDetailsNetwork specifies the Network interface at which the EPIC Detail server will be installed.

Variable:	InstallBinaryDetails
Values:	yes | no
Default Value:	no
Command Line:	-ibd
Description:	InstallBinaryDetails specifies whether the monitor should set up the classic binary detail report server. Shutting down a server terminates all existing connections to it.

Variable:	BinaryDetailsPort
Values:	number 1-65535
Default Value:	21234
Command Line:	-bdp
Description:	BinaryDetailsPort specifies the port at which the binary detail server will be installed.

Variable:	BinaryDetailsNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-bdd
Description:	BinaryDetailsNetwork specifies the network interface at which the binary detail server will be installed.

Variable:	InstallEpicAttacks
Values:	yes | no
Default Value:	no
Command Line:	-iea
Description:	InstallEpicAttacks specifies whether the monitor should set up the classic EPIC attack report server. Shutting down a server terminates all existing connections to it.

Variable:	EpicAttackPort
Values:	number, 1-65535
Default Value:	2345
Command Line:	-eap
Description:	EpicDetailsPort specifies the port at which the EPIC attack server will be installed.

Variable:	EpicAttackNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-ean
Description:	EpicAttackNetwork specifies the network interface at which the EPIC attack server will be installed.

Variable:	InstallBinaryAttacks
Values:	yes | no
Default Value:	no
Command Line:	-iba
Description:	InstallBinaryAttacks specifies whether the monitor should set up the classic binary attack report server.

Variable:	BinaryAttackPort
Values:	number, 1-65535
Default Value:	22345
Command Line:	-bap
Description:	BinaryDetailsPort specifies the port at which the binary attack server will

	be installed.

Variable:	BinaryAttackNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-ban
Description:	BinaryAttackNetwork specifies the network interface at which the binary attack server will be installed.

Variable:	InstallEpicZones
Values:	yes | no
Default Value:	no
Command Line:	-iez
Description:	InstallEpicZones specifies whether the monitor should set up the classic EPIC zone change report server.

Variable:	EpicZonePort
Values:	number, 1-65535
Default Value:	2346
Command Line:	-ezp
Description:	InstallEpicZones specifies the port at which the EPIC zone change server will be installed.

Variable:	EpicZoneNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-ezn
Description:	EpicZoneNetwork specifies the network interface at which the EPIC zone server will be installed.

Variable:	InstallEpicUsage
Values:	yes | no
Default Value:	no
Command Line:	-ieu
Description:	InstallEpicUsage specifies whether the monitor should set up the classic EPIC object usage server. This server reports the number of dynamic objects (e.g., TCP Streams) in allocated, in use, and the total number requested.

Variable:	EpicUsagePort
Values:	number, 1-65535
Default Value:	3456

Command Line:	-eup
Description:	EpicUsagePort specifies the port at which the EPIC object usage server will be installed.

Variable:	EpicUsageNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-eun
Description:	EpicUsageNetwork specifies the network interface at which the EPIC usage server will be installed.

Variable:	InstallEpicStrings
Values:	yes | no
Default Value:	no
Command Line:	-eas
Description:	InstallEpicStrings specifies whether the monitor should set up the classic EPIC pattern match report server.

Variable:	EpicStringsPort
Values:	number, 1-65535
Default Value:	7890
Command Line:	-esp
Description:	EpicStringsPort specifies the port where the EPIC pattern match report server will be installed.

Variable:	EpicStringsNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-esn
Description:	EpicStringsNetwork specifies the network interface at which the EPIC strings server will be installed.

Variable:	InstallDistributedObjects
Values:	yes | no
Default Value:	no
Command Line:	-ido
Description:	InstallDistributedObjects specifies whether the monitor should set up the distributed object server. Currently this is primarily used to tap individual network sessions. Shutting down a server terminates all existing connections to it.

Variable:	DistributedObjectsPort
Values:	number, 1-65535
Default Value:	4567
Command Line:	-dop
Description:	DistributedObjectsPort specifies the port where the distributed object server will be installed.

Variable:	DistributedObjectsNetwork
Values:	any | IP address
Default Value:	any
Command Line:	-don
Description:	DistributedObjectsNetwork specifies the Network interface at which the distributed objects server will be installed.

Variable:	DoEthernetLayer
Values:	yes | no
Default Value:	yes
Command Line:	-del
Description:	DoEthernetLayer directs the monitor to process Ethernet packets.

Variable:	DoIpLayer
Values:	yes | no
Default Value:	yes
Command Line:	-dil
Description:	DoIpLayer directs the monitor to process IP packets. An appropriate link layer must also be turned on. (Currently the only valid link layer is EthernetLayer).

Variable:	DoIpAttackProxy
Values:	yes | no
Default Value:	no
Command Line:	-diap
Description:	DoIpAttackProxy directs the monitor to used the IpAttackProxy to identify IP protocol layer attacks.

Variable:	DoIpFilter
Values:	yes | no
Default Value:	no
Command Line:	-dif
Description:	DoIpFilter directs the monitor to apply the filter at the IP layer. The specific filter is specified by the variable IpFilterType.

Variable:	IpFilterType
Values:	IpZoneFilter
Default Value:	IpZoneFilter
Command Line:	-ift
Description:	IpFilterType identifies the type of filter to apply to the IP layer. Currently, only one type is supported, IpFilterType. It loads the filter configuration identified in NratZoneFilter.conf.

Variable:	DoIpReassembly
Values:	yes | no
Default Value:	no
Command Line:	-dir
Description:	DoIpReassembly directs the monitor to reassemble fragmented IP packets. This also detects a number of IP overlap attacks.

Variable:	DoTcpLayer
Values:	yes | no
Default Value:	yes
Command Line:	-dtl
Description:	DoTcpLayer directs the monitor to process TCP packets. The IP layer must also be turned on.

Variable:	DoTcpSessions
Values:	yes | no
Default Value:	yes
Command Line:	-dts
Description:	DoTcpSessions directs the monitor to create TCP session objects to track the behavior of individual TCP sessions.

Variable:	MaxTcpSessions
Values:	number
Default Value:	none
Command Line:	-mts
Description:	MaxTcpSessions directs the monitor to used the supplied number as the maximum number of TCP sessions tracked. If the number is reached no new sessions will be tracked until an existing session closes.

Variable:	DoTcpAttackProxy
Values:	yes | no
Default Value:	no
Command Line:	-dtap
Description:	DoTcpAttackProxy directs the monitor to used the TcpAttackProxy to identify TCP protocol layer attacks.

Variable:	ReportTcpSessionsToDetails
Values:	yes | no
Default Value:	yes
Command Line:	-rtd
Description:	ReportTcpSessionsToDetails directs the monitor to send all TCP sessions (both requests and fully established connections) to the details server.

Variable:	DoTaggedTcpPorts
Values:	yes | no
Default Value:	no
Command Line:	-dttp
Description:	DoTaggedTcpPorts directs the monitor to report attempted access to specific ports as an attack. For example, a connection attempt to port 31337 may be looking for BackOrifice. You can also use this feature to establish honey pots or identify scans looking at only common server ports that generally aren't accessed (e.g., chargen at port 19). The specified ports are defined in the file TcpTaggedPorts.conf in Network Radar's Support directory.

Variable:	ReportTcpAccessDenied
Values:	yes | no
Default Value:	no
Command Line:	-rtad
Description:	ReportTcpAccessDenied reports deined access to TCP (and eventually UDP) ports. A denied TCP session is defined by the server returning a RST when it receives a SYN request. A denied UDP session is defined by the server returning an ICMP port unreachable message. This feature can be useful for detecting a simple scans of a host (e.g., just a few ports, but not enough to set off a sweep analysis). However, this can cause enormous numbers of attack reports to be generated during a heavy network scan.

Variable:	DoUdpLayer
Values:	yes | no
Default Value:	yes
Command Line:	-dul
Description:	DoUdpLayer directs the monitor to process UDP packets. The IP layer must also be turned on.

Variable:	DoUdpSessions
Values:	yes | no
Default Value:	yes
Command Line:	-dus

Description:	DoUdpSessions directs the monitor to create UDP session objects to track the behavior of individual UDP sessions.

Variable:	MaxUdpSessions
Values:	number
Default Value:	none
Command Line:	-mus
Description:	MaxUdpSessions directs the monitor to used the supplied number as the maximum number of UDP sessions tracked. If the number is reached no new sessions will be tracked until an existing session closes.

Variable:	DoUdpAttackProxy
Values:	yes | no
Default Value:	no
Command Line:	-duap
Description:	DoUdpAttackProxy directs the monitor to used the UdpAttackProxy to identify UDP protocol layer attacks.

Variable:	ReportUdpSessionsToDetails
Values:	yes | no
Default Value:	yes
Command Line:	-rud
Description:	ReportUdpSessionsToDetails directs the monitor to send all UDP sessions to the details server.

Variable:	DoTaggedUdpPorts
Values:	yes | no
Default Value:	no
Command Line:	-dtup
Description:	DoTaggedUdpPorts directs the monitor to report attempted access to specific ports as an attack. For example, a connection attempt to port 31337 may be looking for BackOrifice. You can also use this feature to establish honey pots or identify scans looking at only common server ports that generally aren't accessed (e.g., chargen at port 19). The specified ports are defined in the file UdpTaggedPorts.conf in Network Radar's Support directory.

Variable:	DoIcmpLayer
Values:	yes | no
Default Value:	yes
Command Line:	-dicmpl
Description:	DoIcmpLayer directs the monitor to process ICMP packets. The IP layer must also be turned on.

Variable:	DoTraceroutes
Values:	yes | no
Default Value:	yes
Command Line:	-dtr
Description:	DoTraceroutes directs the monitor to create traceroute session objects to track the behavior of individual traceroute sessions. A traceroute session is identified by TTL time exceeded ICMP error messages.

Variable:	DoPingSessions
Values:	yes | no
Default Value:	yes
Command Line:	-dps
Description:	DoPingSessions directs the monitor to create ping session objects to track the behavior of individual ping sessions. A ping session is identified by ICMP echo requests and replies.

Variable:	MaxTraceroutes
Values:	number
Default Value:	none
Command Line:	-mtr
Description:	MaxTraceroutes directs the monitor to used the supplied number as the maximum number of traceroute sessions tracked. If the number is reached no new sessions will be tracked until an existing session closed.

Variable:	MaxPingSessions
Values:	number
Default Value:	none
Command Line:	-mps
Description:	MaxPingSessions directs the monitor to used the supplied number as the maximum number of ping sessions tracked. If the number is reached no new sessions will be tracked until an existing session closes.

Variable:	DoIcmpAttackProxy
Values:	yes | no
Default Value:	no
Command Line:	-dicmpap
Description:	DoIcmpAttackProxy directs the monitor to used the IcmpAttackProxy to identify ICMP protocol layer attacks.

Variable:	ReportPingSessionsToDetails
Values:	yes | no

Default Value:	yes
Command Line:	-rpd
Description:	ReportPingSessionsToDetails directs the monitor to send all Ping sessions (ICMP echo requests/replies) to the details server.

Variable:	ReportTraceroutesToDetails
Values:	yes | no
Default Value:	yes
Command Line:	-rtrd
Description:	ReportTraceroutesToDetails directs the monitor to send all traceroute sessions (a series of ICMP Time exceeded messages) to the details server.

Variable:	ReportPingsAsAttacks
Values:	yes | no
Default Value:	no
Command Line:	-rpaa
Description:	ReportPingsAsAttacks reports a ping session (ICMP echo request/reply) as an attack.

Variable:	ReportTraceroutesAsAttacks
Values:	yes | no
Default Value:	no
Command Line:	-rtaa
Description:	ReportTraceroutesAsAttacks reports a traceroute session (a series of ICMP Time exceeded messages) as an attack.

Variable:	DoTcpHosts
Values:	yes | no
Default Value:	yes
Command Line:	-dth
Description:	DoTcpHosts directs the monitor to track behavior if IP hosts.

Variable:	MaxTcpHosts
Values:	number
Default Value:	none
Command Line:	-mth
Description:	MaxTcpHosts directs the monitor to used the supplied number as the maximum number of hosts tracked. If the number is reached no new hosts will be tracked until an existing host session closes.

Variable:	DoSweepAnalysis
Values:	yes | no

Default Value:	no
Command Line:	-dsa
Description:	DoSweepAnalysis directs the monitor to perform additional sweep analysis when a client appears to be starting a large number of sessions or has received a number of failed session requests.

Variable:	ReportStreamInfoToDetails
Values:	yes | no
Default Value:	yes
Command Line:	-rsid
Description:	ReportStreamInfoToDetails directs the monitor to send all stream reports to the details server.

Variable:	SavePasswordInfo
Values:	yes | no
Default Value:	no
Command Line:	-spi
Description:	SavePasswordInfo directs the monitor to save password information when processing login information by the stream LoginStream.

Variable:	SaveLogs
Values:	yes | no
Default Value:	yes
Command Line:	-sl
Description:	SaveLogs specifies whether the monitor should save audit records for the various stream modules.

Variable:	RoleLogs
Values:	number defining the minutes for each log file
Default Value:	60 (1 hour)
Command Line:	-rl
Description:	RoleLogs specifies the number of minutes between log file changes.

Variable:	DoTaggedRpcs
Values:	yes | no
Default Value:	no
Command Line:	-dtrpc
Description:	DoTaggedRpcs directs the monitor to report remote procedure calls (RPCs) to specific program functions as attacks. For example, the RPC server portmapper supports the function called dumpit(). This function reports all registered RPC programs and is often used by attackers to profile your system. You can call this program with the program rpcinfo: `% rpcinfo -p han`

	The program and function numbers are specified in the file RpcStreamSensitive.conf in Network Radar's Support directory.

Variable:	UsePatternExpertSystem
Values:	yes | no
Default Value:	no
Command Line:	-upes
Description:	UsePatternExpertSystem specifies whether pattern matches will be mapped to attacks by the simple Pattern Expert System. These rules are defined in the file PatternES_rules found in Network Radar's Support directory. The pattern names used by PatternES_rules are defined in the RX.conf, KMP.conf, or the FSM.conf files also found in support.

Variable:	DoVectorCollect
Values:	yes | no
Default Value:	no
Command Line:	-dvc
Description:	DoVectorCollect directs the monitor to archive two 128 dimensional vectors representing the counts of byte values observed being sent by the client and server in individual sessions. The monitor still needs to be directed to place the VectorCollectStream on the appropriate streams; this is done in the NratTcpStreams.conf and NratUdpStreams.conf files.

Variable:	DoWebCollect
Values:	yes | no
Default Value:	no
Command Line:	-dwc
Description:	DoWebCollect, like DoVectorCollect, collects vectors representing session activity. However, whereas DoVectorCollect uses VectorCollectStream to collect a single vector for each session, DoWebCollect uses WebCollectStream which can collect several vectors for Web activity. We needed to create a separate stream class for web traffic for two reasons. First, we are only interested in analyzing a portion of a web transaction (e.g., the data requested and not the data types a client can accept). Second, newer web browsers and web servers will allow multiple connections over a single connection, so each connection may need to generate several vector sets (one for each web request).

Variable:	DoSavePackets
Values:	yes | no
Default Value:	no
Command Line:	-dsp
Description:	DoSavePackets directs nrat6 to save packets certain packets. For the packets to be saved, it has to cross a boundary between a protected zone and an unprotected zone, and it either its source or destination ports must

	belong to a specific set of numbers. The file defining the "protected zone" is $RADAR_DIST/Support/session_addr_files It contains a set of addresses and netmask pairs. Together they define the protected zone. Sessions crossing the zone will be candidates for having their packets saved. The file defining the sensitive ports is: $RADAR_DIST/Support/session_port_file If a session crosses the zone defined above, we then check to see if either of its ports is in the session_port file. If so, we save the packets.

Variable:	ThumbprintPossibleCount
Values:	number 1-65535
Default Value:	1
Command Line:	-tpc
Description:	ThumbprintPossibleCount directs the monitor to generate a multi-hop alert (when someone connection from machine A to B to C), when the two sessions appear similar for ThumbprintPossibleCount time periods. By default, a time period is two minutes, and an alert is triggered when the first match is detected. At some sites, in particular those with large login banners, two connections occasionally trigger a false alert because two independent users logged in, received the voluminous login banner, and did nothing else. In these cases, you might want to increase ThumbprintPossibleCount to 2 or 3. That is, you must balance the timeliness of a report and the certainty the chance of false alerts.

Variable:	ThumbprintProbableCount
Values:	number 1-65535
Default Value:	3
Command Line:	-tprc
Description:	Like ThumbprintPossibleCount, ThumbprintProbableCount directs the monitor to generate a multi-hop alert when two sessions appear similar for ThumbprintProbableCount time periods. The primary difference is that (assuming ThumbprintProbableCount is larger than ThumbprintPossibleCount), the certainty of the alert will be higher. Later alerts will by default be generated every 10 time periods (roughly 20 minutes) the connections appear to be the same.

Variable:	DoUdpDhcpProxy
Values:	yes | no
Default Value:	no
Command Line:	-dudp
Description:	DoUdpDhcpProxy directs the monitor to use the UdpDhcpProxy to

	process DHCP activity. Right now, the DHCP proxy is just a stub that prints out a very small amount of information.