The following paper was originally published in the
Proceedings of the Eleventh Systems Administration Conference (LISA '97)
San Diego, California, October 1997

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Implementing a Generalized Tool for Network Monitoring

Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall – Network Flight Recorder, Inc.

ABSTRACT

Determining how you were attacked is essential to developing a response or countermeasure. Usually, a system or network manager presented with a successful intrusion has very little information with which to work: a possibly corrupted system log, a firewall log, and perhaps some tcpdump output.

When hackers come up with a new technique for cracking a network, it often takes the security community a while to determine the method being used. In aviation, an aircraft's "black box"[1] is used to analyze the details of a crash. We believe a similar capability is needed for networks. Being able to quickly learn how an attack works will shorten the effective useful lifetime of the attack. Additionally, the recovered attack records may be helpful in tracking or prosecuting the attacker. Since we've developed a general purpose statistics-gathering system, we believe it will be useful for more than just security. For example, a network manager may desire an historical record of the usage growth of certain applications, or details about the breakdown of types of traffic at different times of day. Such records will provide useful information for network managers in diagnosing performance problems or planning growth.

This paper describes an architecture and toolkit for building network traffic analysis and statistical event records: The Network Flight Recorder. The NFR uses a promiscuous packet interface to pass visible traffic into an internally meta-programmed decision engine which routes information about packets and their contents into statistical or logging backends. In addition to packet analysis and collection, the NFR's internal architecture permits network managers to sample interesting portions of network traffic for logging or statistical analysis. The NFR programming language is simple, but powerful enough that you can perform reasonable analysis on traffic before choosing to record it. For example, you might analyze SMTP transactions but only choose to record those relating to a user who is sending spam or abusive E-mail. The analysis language includes a capability for generating alert messages which the rest of the system queues, multiplexes, and delivers. A simplified hyper-query interface allows extensive browsing of the NFR's stored datasets and statistics from any Java-enabled browser. The NFR is currently being deployed at a number of ISPs and commercial sites, and is available for download in source code form from www.nfr.net.[2]

Background and Motivation

In 1990, one of the authors managed a rather chaotic network, including an embryonic firewall, using NNStat as a security tool. NNStat [1] was designed as a statistical analysis system for the NSFnet backbone, not as a security tool, but possessed several attractive properties:

1. It permits accurate and highly condensed summaries of an event on the network.
2. It permits flexible specification of types of events to record.
3. It permits flexible storage of information about the events that are observed.

While NNStat's authors were concerned about, for example, how much RIP traffic was crossing the network, a security conscious network manager could use NNStat to record all RIP traffic emanating from any systems that were not on an "approved list" of routers. Suddenly, NNStat was useful as a crude tool for mapping who and what, as well as for setting an alert to fire when something happened that the network manager believed should not. NNStat, wrapped with a bunch of quick and dirty shell scripts and cron jobs, served well as a poor man's intrusion detection system. Other network managers have implemented similar systems using tcpdump, or more sophisticated special-purpose network watchers like ARPwatch [2], TCPwatch [3], Netman [4], clog [5], Netwatch [6], and Argus [7]. Other intrusion detection burglar alarms have focused on features of the host operating system, such as tcp_wrappers [8], klaxon [9], and tocsin [10]. Many of the monitoring systems implemen-

[1] They are actually Safety Orange.

[2] Use of the NFR software is free for noncommercial and research purposes. A commercial release of the software is being developed.

ted in the past contain features found in NFR. We believe that the new ground the NFR breaks is by making the filtering and analysis process internally programmed, rather than static-coded into the monitoring application.

NFR is intellectually evolved from NNStat, but includes a more generalized and powerful filtering language, as well as the ability to trigger alerts and log complete packet information. A triggering specification lets data be selected from reassembled TCP sessions, providing a powerful capability for usage measurement as well as audit. The authors intend to use NFR as a platform for exploring auditing and logging, while simultaneously providing a freely available, high quality data source for researchers working on intrusion detection.

Overview of the NFR Architecture

The architecture of NFR was designed as a set of components, each tailored to a specific activity. Data is gathered by one or more packet suckers, forwarded to the decision engine for filtering and reassembly, and possibly recorded to a backend for storage or statistical processing. The query interface is kept completely separate from the input data flow to minimize the performance impact of a user's querying the system while it is collecting data.

Packet Suckers

The packet suckers we initially implemented have been based on the libpcap [11] packet capture interface. Libpcap provides a generalized packet capture facility atop a number of operating system-specific network capture interfaces. This freed us from having to deal with a lot of portability issues. We did discover, however, that some of the available packet capture facilities cannot reliably buffer high volumes of bursty traffic. Berkeley packet filter-based packet suckers running on a Pentium-200 were unable to handle even moderate network loads. This was a result of a latency interaction between BPF and our software: we do more processing than a program like tcpdump, and, though our average processing seems to be within the performance envelope of the machine, we can't always process the packet ''immediately,'' as BPF expects. To fix the problem, we increased internal buffer sizes from their default of 32K to 256K, a number more appropriate for the amount of RAM available in modern computers. Since the NFR daemon potentially monitors multiple interfaces, we performed minor modifications to the way blocking and time-outs are performed in BPF. The original BPF time-out is an inter-packet time-out based in the arrival of a packet. If you don't see a packet, you never time out. We modified it to begin the timer with the read() or select() timeout, so we can detect periods of no traffic.

Typical applications using BPF, such as tcpdump, wait in a tight loop while they read packets from the interface. Since we are trying to do additional processing, and potentially additional I/O, we have a closer-to-real-time requirement, which doesn't sit well with a wait-loop model. For high-performance networks, we are examining designing a new packet capture interface based on a memory-mapped device driver. Most of the mechanisms for packet capture require two system calls (or an interrupt and a system call) per packet; we'd like to reduce it to a single semaphore check.

When a packet is collected by a packet sucker, it is passed to the decision engine using a generalized API intended to allow packet suckers to be separate processes from the engine. The libpcap-based packet sucker is compiled into the engine, but we wanted to admit the possibility of multiprocessing packet suckers that might perform their own buffering. Packets read from libpcap include time information, which is preserved with the packet as it is passed through NFR, providing a notion of time to the entire engine.

Decision Engine

Packets are passed into the decision engine, where they are checked against a list of filters for evaluation. Filters are written in N-code, which is read into the engine, compiled, and preserved as byte-code instructions for fast execution. TCP traffic is applied against a reassembly table which preserves the state of each current TCP session. The state reassembly mechanism permits matching patterns or other events within the lifetime of a TCP stream, and keeps statistics pertaining to variance in the delivery of packets. These statistics are used to determine when the engine will stop watching a given connection – for example, connections are not considered ''closed'' until a time-out has exceeded two standard deviations of the average packet arrival rate after a FIN packet. Certain types of broken packets can be detected, and users can access byte counts of retransmitted packets – duplication of traffic – which might indicate network problems.

The filtering language binds a filter to an event or the reception of a packet. Once a packet has been applied against the filters, it is discarded. The filtering language provides programmatic access to fields within the packet, usually for the purpose of recording information from or about the fields. The main primitives for getting data out of a filter are the *alert* and *record* mechanisms. Alerts pass a free-form message to the alert management system, much like a call to syslog(). The record mechanism passes a constructed data structure to a backend recorder for further processing. Using a structured record allows the engine and the backend to pre-agree upon what type of information the recorder should expect to read, permitting for faster interpretation of the data as it arrives at the backend. Some of the backends provided with NFR are specialized to handle multiple forms of data, but if someone wanted to develop a simple backend that processed only IP addresses, they would be able to link in a simple library routine that provides the

backend a stream of pre-processed addresses. This approach saved us considerable time in development, and permitted early experiments with backends written in TCL [12]. We feel this is an important capability, since it opens the possibility of having an NFR performing gross-level filtering of traffic while passing specific records into a backend SQL database for more advanced processing.

Backends

Originally, we had anticipated developing a large number of specialized backends, each of which might maintain a different type of statistic. As the design evolved, we chose instead to produce a small number of multi-purpose backends, which accept and process a wider latitude of data types. Histogram[3] and list are the two primary backends in use at this time. Histogram maintains a columnar table of data, either totaling discrete values in the columns or incrementing them.

Unlike the traditional histogram in statistics, the NFR histogram recorder saves data in an arbitrary number dimensions, rather than in a single dimension. For example, a histogram that stores a list of IP addresses and strings might be used to represent the number of times clients on a network retrieved a particular URL. A different instance of histogram with the same record structure might store the number of times user-ids had logged into various systems within the network. Histograms can store multiple columns of data simultaneously, permitting a lot of information to be stored in a very compact space. The histogram recorder provides a very flexible means of generating alerts based on the appearance of previously-unseen values, or values exceeding a specified range. For example, a histogram can easily produce a "new/duplicate IP address detector" by throwing hardware ethernet addresses and IP addresses into a histogram, then generating an alert whenever a new entry appears.

The list backend maintains chronological records, as opposed to histogram's additive records. The list backend provides more typical logging functions by not collapsing its columns into totals. In the histogram example of client IP addresses and URLs, if the same data record were sent to list, it would maintain a log of individual accesses to the URLs by client, over time. List's data storage tends to be less space efficient than histogram, since it maintains data about each record it is sent, rather than a total. The additional detail list maintains is important for some applications, since it lets you know not only how many events happened, but exactly when and in what order.

Query Backends

One of the problems we faced early on was figuring out how to get data out of the NFR without interrupting the flow of data *in*. If the engine were blocked during a query, it might lose a large number of packets, since it's possible that a query against a large dataset might take several seconds to process. The eventual design decision was to have the

[3]Apparently we are using the term "histogram" somewhat incorrectly and may confuse statisticians. We should have called it "spreadsheet."

```
Time TCP Hash Client      Server           Command
Mon Sep  8 15:02:08 1997
     18 208.218.124.77  208.218.124.42  GET / HTTP/1.0
     18 208.218.124.77  208.218.124.42  If-Modified-Since: Wednesday, 06-Nov-96
                                            12:32:03 GMT; length=530
     18 208.218.124.77  208.218.124.42  Connection: Keep-Alive
     18 208.218.124.77  208.218.124.42  User-Agent: Mozilla/3.0Gold (X11; I;
                                            BSD/OS 3.0 i386)
     18 208.218.124.77  208.218.124.42  Host: cornfed
     18 208.218.124.77  208.218.124.42  Accept: image/gif, image/x-xbitmap,
                                             image/jpeg, image/pjpeg, */*
     18 208.218.124.77  208.218.124.42
     18 208.218.124.77  208.218.124.42  GET /apache_pb.gif HTTP/1.0
     18 208.218.124.77  208.218.124.42  If-Modified-Since: Wednesday, 03-Jul-96
                                             06:18:15 GMT; length=2326
     18 208.218.124.77  208.218.124.42  Referer: http://cornfed/
     18 208.218.124.77  208.218.124.42  Connection: Keep-Alive
     18 208.218.124.77  208.218.124.42  User-Agent: Mozilla/3.0Gold (X11; I;
                                             BSD/OS 3.0 i386)
     18 208.218.124.77  208.218.124.42  Host: cornfed
     18 208.218.124.77  208.218.124.42  Accept: image/gif, image/x-xbitmap,
                                             image/jpeg, image/pjpeg
     18 208.218.124.77  208.218.124.42
```

Figure 1: Historical data from a list recorder.

backends maintain their datasets entirely on disk, and to have a secondary backend – the *query backend* – which contains the logic for mining datasets produced by its matching backend. A backend may perform buffering of its own data, but since the query backend "knows" how the backend operates it can coordinate with the backend as necessary. Since one of the goals of NFR is to provide a reliable history of events, we feel that too much buffering is risky – we'd rather buy faster disks.

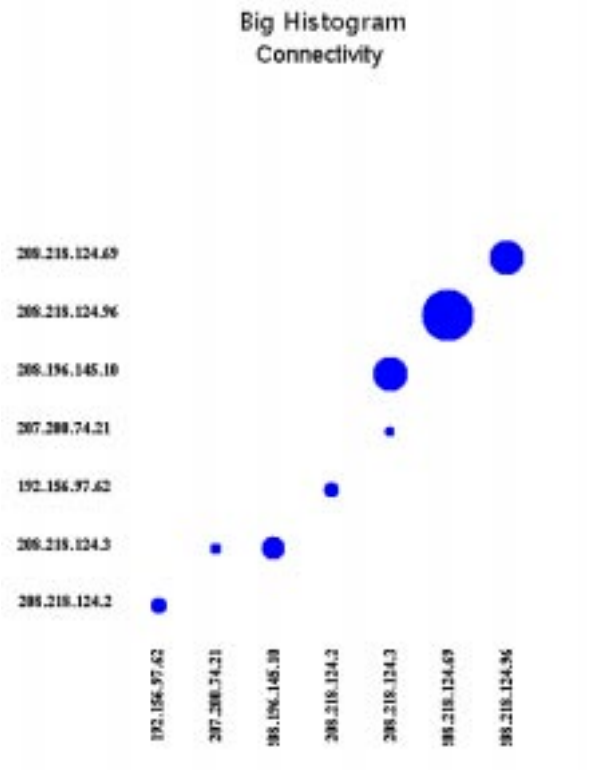

Figure 2: Connectivity graph with 10% cutoff in effect.

Each of the query backends provides its own CGI interface, parsing a URL into a query. Histogram and list both support options to compact out fields and eventually will support sorting of fields. Another feature we added later is threshold cutoffs – we discovered that some queries produced much, much more information than we could process sensibly. For example, an origin/source map between any two systems that have sent packets produces a scatter-plot that contains a large number of single-packet events (mostly DNS traffic). While the information is worth having, it is much easier to glance at when there is a cutoff in place. (See Figure 1)

The first version of our query backends represented information uniformly, which we found made it a bit more difficult to visualize. Since the data going into the backends is abstract, we needed to provide a mechanism for the end user to apply "eye candy" to the dataset to make it more comprehensible. User-specified mappings are matched against tokens in the output fields of histogram, changing color or replacing the strings. (See Figure 2)

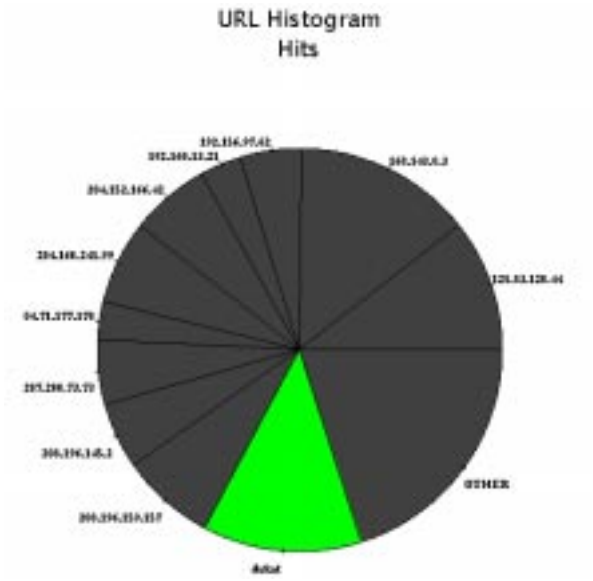

Figure 3: Pie chart with user-specified color and name mapping.

Figure 4: Query interface showing field collapsing.

We intend to continue to improve the query interface; that is an area of ongoing research. Currently, the query interface allows the user to specify field values and fields to represent, as well as how to represent them. Since each backend and query backend is a separate software system, there is some duplication of effort; we considered the problem of developing a generalized query language but concluded that the level of effort was too high. General query capability is best provided through an external database with SQL. Using an SQL database will greatly increase the cost, administrative cost, and setup cost of the system

in return for a marginally greater flexibility in query capability. We can imagine some cases where advanced query capability will justify the cost and effort, but we didn't want to try to integrate or develop a database engine as part of the project.

GUI

Each backend has a number of graphical user interface elements. The GUI elements for each backend control its individual configuration attributes, as well as its query interface. The query interface generates correctly configured CGI queries aimed at its matching backend. To simplify the user's view of the system, we've created a construct, which exists purely in the GUI, of "packages." Packages are a grouping of backends which may operate independently or may cooperate by sharing common code or variables. The latter case is particularly useful if you have a filter that does substantial work to analyze a protocol, and you want to record more than one type of information about it. For example, you may wish to analyze E-mail traffic and record sender/recipient to a list log and a matching count to a histogram, without having to duplicate the filter that performs the analysis.

While the Java language holds great promise, we found that Java is not yet ready for use in developing serious applications. Primarily, the difficulty is that the client browser implementations perform badly or unpredictably. While the core language itself appears solid, the browser supplied functions, which include all of the GUI objects and network communications, often behave erratically. Eventually, we developed a stable base of classes that seem to work on popular platforms, but we did it by testing the target systems to determine what parts of the published interface were actually reliable. The process was not as fun or pleasant as the Java hype would lead one to expect.

For a program like NFR, it was vital that we be able to have a secured remote management capability. Using Java and Web technology allowed us to layer the NFR interface underneath whatever security the browser and web server can provide. This freed us from worrying about export control regulations that apply to software incorporating cryptography. It also freed us from portability problems, in a conceptual sense, though the browser-version-specific bugs we coded around were another form of portability problem. "Write once, run anywhere" does not promise that your code will actually *work* once it is running. In spite of our difficulties with Java, it was still easier to implement our U/I than it would have been had we been using Microsoft Windows and X Toolkit to do our own secure remote user interface. Java has a much simpler GUI model than either X or Windows, but, when it works, it is suitable for real applications. We hope that future versions of Java, such as Java 1.1, will overcome some of the difficulties we faced.

Alert Queue Manager

NFR has a generalized mechanism for processing alerts, which is used to manage alerts that are produced from N-code, alerts from backends, and internal error-alerts generated by the engine. In order to make NFR a useful network management tool, we developed alertd, which prioritizes and routes alerts. Alerts can include free-form or formatted strings, which are matched against a number of delivery facilities. The facilities represent delivery mechanisms such as printer, E-mail, and FAX. Alert events can further be marked as alerts requiring acknowledgment, which maintains them in a queue that a network manager can clear once the situation causing the alert has been resolved. Records of who acknowledged an alert, as well as comments, are preserved with the alert.

Space Manager

Since NFR is intended to provide historical records, we needed to provide a capability for rotating logs and archiving them or deleting them. Spaceman, the space manager, runs periodically to check the amount of disk space available for data storage, and to "expire" old datasets. Each instance of a backend can be specifically tuned for storage lifetime and archiving. Backends that are preserving especially important data might have an expiration time that is much higher, and an archiving process that copies the data to CD-R or tape instead of merely deleting it.

N-code Filtering

The N programming language is a derivation of an interpreted language designed years ago for use in a computer game.[4] The interpreter operates on a byte-code instruction set that implements a simple stack machine. One advantage of this approach is that NFR filters occupy very little memory, yet are quite fast to evaluate. N is a complete programming language including flow control, procedures, variables with scoping rules, and list data types. Unlike many programming languages, however, N has primary data types such as "IP address." Since NFRs may be used on large networks, we chose to implement counter data types as 64-bit integers, to reduce the chance of overflow.

Packet values are accessible in N-code using a syntax of *thing.thing*, in which a higher level attribute references a lower level element. For example: *ip.src*, *ip.dst*, *syslog.message*, and *tcp.hdr* are all valid N-code packet fields. The packet ripping routines perform lazy evaluation with cached responses, so the runtime cost of packet ripping is kept low. At present, the NFR engine's packet rippers can handle IP, TCP, and UDP packets and ethernet frames. We are considering adding routines to decode other protocols or even application protocols, such as *syslog*. Mixing

[4] UberMUD, a meta-programmed multi-user dungeon game by Marcus J. Ranum

application specific protocols into packet value access enables some unusually powerful productions such as:

```
if (ip.src == $myfirewall &&
    index(syslog.message,"BADSU")
        >= 0 ) { ...
```

In cases where the built-in packet fields are insufficient, other fields can be located by N-code functions. It is possible, though slightly slower, to write N-code subroutines to return byte offsets within packets. This can extend as far as analyzing high-level protocols. For example, we have implemented an N-code filter that extracts protocol fields such as sender and recipient from an SMTP dialog. More N-code can then act on that information:

```
if ( $sender == $known_spammers )
    { ...
if ( $n_recipients > 10000 )
    { ... / possible spam
```

Events

Triggers within N-code occur upon receipt or detection of an event that the code is attached to. Events can be triggered with limitations on source, destination, ports, client or server side (if known), or patterns within the TCP stream. The syntax looks like:

```
filter mailtrack tcp ( client,
    dport: 25 ) {
```

The filter above is a simple TCP stream trigger that will monitor the client side of SMTP connections. The ‘‘client’’ and ‘‘server’’ notion is based on the reassembly engine’s recollection of which system initiated the connection that is being observed.

Keywords that can be placed within an event are:

- client – from the caller
- server – from the called
- start: ‘‘string’’ – begin matching
- stop: ‘‘string’’ – end matching
- opensession – on start of connection
- closesession – on end of connection
- port – IP port number (source or dest)
- sport – source port
- dport – destination port
- host – source or destination address
- net – source or destination network
- dst – destination address
- src – source address

A typical use is to configure an event to call N code for as small a subset of received data as is practical, then implement any further filtering in N code. To detect spam, for example, you might select TCP traffic for port 25/SMTP. The N code would then:

- Observe the transaction and determine the sender and recipient
- If the sender is the spammer, record the sender, recipient, and originating host to a histogram recorder
- If the sender is the spammer, record the message header and the first 10 lines of the message text to a log recorder

```
filter server tcp ( client,
          port: 80,
          start: "GET ",
          stop: " " ) {
     record  ip.src, ip.dst,
         tcp.sport, tcp.dport,
         tcp.bytes to urlRecorder;
}
```

Figure 4: N-code implementing a simple HTTP URL detector.

The example HTTP URL detector sets a trigger filter on TCP port 80, for patterns that look like HTTP GET commands, and logs information about the activity to a recorder. In the example, tcp.bytes represents the matched TCP data between the start and stop triggers. More complex filters can attach symbolic values as local variables for each TCP stream – permitting a filter to do complex activity like counting the To: recipients in an SMTP stream, or recording them in an array for later logging. The N-code for more advanced filters, with comments, can run to several hundred lines.

Performance

Performance is an open question with an NFR. We haven’t had enough access (yet) to truly large networks that can properly stress the system. For some reason, sites with large, interesting, backbones are not enthusiastic about having someone else put a meta-programmable traffic analysis engine on their network. As of this writing we have not had a chance to test the NFR software on more than 10Mb/s ethernet. We believe that performance should be adequate up to at least 60-70Mb/s, based on experiments other sites have performed with TCPdump on 100-base-T hubs, using comparable hardware.

Probably the biggest performance problem that we worry about is the NFR’s programmability. Since it is highly customizable, it’s hard to predict how many filters the average user will install, and how much data they will be recording. For a network that runs a steady 80Mb/s load, with a filter recording all the packets to disk, the NFR would not be able to keep up, unless the I/O subsystem were capable of sustained disk writes, through the filesystem, at comparable speeds. We’ve found that, in general, filters compile down to be quite tiny and require few ‘‘instructions’’ to operate, but it’s conceivable that a user might develop an extremely N-code-intensive filter which bogged a system down unacceptably. We fully anticipate that this will happen, and plan to performance tune the system as we get more feedback from installed sites.

Lessons Learned

The first hurdle in designing the system was figuring out the right place to put each piece of functionality. TCP reassembly, for example, was originally planned for implementation in a backend, but we realized that some of the filters we wanted to install depended on reassembled streams. TCP reassembly moved back into the engine. Some of the statistical capabilities of the backends were originally slated for residence in the engine, but we realized that queries against the engine itself might cause it to spend too long processing the query and lose packets. As a result, the backends show duplication of functionality, but are standalone programs. The decision to separate querying completely from the backend into a separate program was risky but has proven to work fairly well at the cost of duplication of functionality between the query backend and the recorder backend.

The most hard-won lesson of the project, so far, is that Java isn't really ready for writing major user interfaces. It's great for nifty animated objects, but until JDK1.1 – and widely supported 1.1 virtual machines – are available, Java will be a problem. We also learned that designing a general-purpose language for traffic analysis was harder than we originally expected it to be. We began with a packet-oriented event model, which did not survive well in the face of TCP streams and matching. NFR seems to be evolving toward a model in which filters are callbacks triggered by pre-specified conditions within the engine. Now that we've tried it the other way, it makes a lot of sense!

Future Work

In future releases of NFR we intend to expand the engine's ability to rip apart packets and application traffic. As we discover more types of things we want to record, it is likely we'll extend the language. Doubtless we will have cause to seek and find ways to improve the NFR's performance; there is no such thing as "fast enough." We believe that NFR will make a useful "bottom half" of an intrusion detection system, and are considering a number of options for analysis engines that can be plugged into NFR as a backend. Another area to explore is the development of active agents which will "poke" parts of the network so that NFR can record the responses.

Conclusions

We set out to build an extremely flexible general-purpose security and network management tool, and appear to have met our goal. There is still a great deal of enhancement we hope to make to the system, but the basic functionality already exceeds what we had originally expected. In a few areas, we were surprised that things we thought would be very useful turned out to be less so – a good example is the scatter plot diagrams prior to the addition of cutoff thresholds.

Initially, we were concerned about the possibility for abusing our tool. It would, for example, make a terrific password grabber. But we feel it is too "heavyweight" for such uses – most hackers who are grabbing passwords are already using very simple single-purpose software that does the job quite nicely. NFR presents the possibility for abuse by "big brother" or well-funded snoops monitoring networks on a grand scale. Keeping NFR out of the public's hand will not prevent such activities, either, since specialized tools for particular traffic analysis are not difficult to develop. We believe that, in general, NFR will be a net benefit to the community, by providing a tool that will stimulate research in network analysis and intrusion detection.

We've been very, very pleased with the power of the system; it appears to be able to do all of the kinds of things for which we designed it. Simple statistics and traffic gathering is very straightforward, and very sophisticated programs are possible as well. We haven't really had time to explore all the possibilities of the system, and we see the depth of its capabilities as a great source for future research and discovery.

Availability

The complete NFR source code, including documentation, Java class source, decision engine, space manager, etc., is available for download from www.nfr.net. for non-commercial research use. The code is designed to operate on a wide variety of UNIX platforms and is being ported to Windows NT for commercial release.

Acknowledgements

Marcus Ranum wrote the first version of the engine interpreter and early syntax for the N language, as well as developing the overall early design of NFR. Mike Stolarchuk implemented the TCP reassembly code in the engine, and refined the engine and N language to its current state, patiently taking feedback from the rest of us. Mark Sienkiewicz implemented the backend design, including histogram, list, and the layout of many of the Java user interface elements. Kent Landfield implemented spaceman, and a number of the user interface elements. Andrew Lambeth designed and implemented alertd, and the list viewer applets. Eric Wall implemented further user interface elements, stealth mode kernel hacks, and did system configuration.

Network Flight Recorder would like to thank our investors for their kind support of our efforts.

References

[1] Robert Braden and Annette DeSchon, *NSFnet Statistics Collection System – NNStat*, USC Information Sciences Institute, December, 1992.

[2] *ARPwatch*, Lawrence Berkeley National Labs Network Research Group, http://ftp.ee.lbl.gov .

[3] *TCPwatch*, Lawrence Berkeley National Labs Network Research Group, http://ftp.ee.lbl.gov .

[4] *Homebrew Network Monitoring: a Prelude to Network Management*, Mike Schultze, George

Benko, and Craig Farrell. Curtin University of Technology, 1993

[5] Clog, Brian Mitchell.

[6] *Netwatch and Netwatch*, Texas A&M University, January 1994.

[7] Carter Bullard, Chas DiFatta, *Argus 1.5 announcement*, Software Engineering Institute, Carnegie Mellon University, ftp://lancaster.andrew.cmu.edu/pub/argus-1.5 .

[8] *Tcp_Wrappers and Logdaemon*, Wietse Venema.

[9] *Klaxon*, Doug Hughes.

[10] *Tocsin*, Doug Hughes.

[11] *libpcap*, Lawrence Berkeley National Labs Network Research Group, http://ftp.ee.lbl.gov .

[12] Ousterhout, J. K., "TCL: An Embeddable Command Language," *Proceedings of the 1990 Winter USENIX Conference*, pp 133-146, 1990.

[13] Anderson and Patterson, "Extensible, Scalable Monitoring for Clusters of Computers," *Proceedings of the 1997 USENIX LISA Conference*, 1997.

Appendix 1: A Simple Filter

```
# This filter serves two purposes:  to record client requests
# made to your web servers, and to serve as example in the LISA paper.
#       Mark Sienkiewicz / NFR
#       Copyright 1997, Network Flight Recorder, Inc.

# schema would be automatically generated by a "wizard"
watchservers_schema = [ 1, 1, 1, 6, 6, 2 ];

# list of my web servers
my_web_servers = [ 208.218.124.77 , 208.218.124.42 ] ;

# gather data the client sends to a web server.  If I didn't know that
# all my web servers are on port 80 I would make this more elaborate.
filter watch tcp ( client, dport: 80 )
{
        if (ip.dest != my_web_servers)
                return;
        declare $blob inside tcp.connSym;
        $blob = strcat ( $blob, tcp.bytes );
        while (1 == 1)
                {
                $x = index( $blob, "\n" );
                if ($x < 0)             # break loop if no complete line yet
                        break;
                $t=substr($blob,$x-1,1);        # look for cr at end of line
                if ($t == '\r')
                        $t=substr($blob,0,$x-1);        # tear off line
                else
                        $t=substr($blob,0,$x);

                # save the time, the connection hash, the client,
                # the server, and the command to a list
                record system.time, tcp.connHash, ip.src, ip.dest, $t to
                        watchservers_list;

                # keep the remainder of the blob for the next pass
                $blob = substr($blob, $x + 1);
                }
        # keep us from getting flooded if there is no newline in the data
        if (strlen($blob) > 4096)
                $blob = "";
}
watchservers_list = recorder ("bin/list packages/web/watchservers.cfg",
        "watchservers_schema" );
```