Rubicon: An Extensible Integrated Traffic Shaping Firewall and Intrusion Detection System

Final Report, June 2002

Ian Peters, MEng ISE
Imperial College

Marker:	Emil Lupu
Second Marker:	Naranker Dulay

Abstract

One of the traditional bastions of security in a computer network is the firewall. Whilst only a single element of a secure network topology, they are a vital feature which allows some amount of network traffic filtering at security perimeters in an organisation. Another useful element has been the Intrusion Detection System, both network-based (NIDS) which looks at network traffic and protocol contents, and Host-based (HIDS) which reside on hosts and monitor the behaviour of the services and users on the Host. The IDS is used as an information resource – to inform system administrators when possible problems are occurring on their systems, and as such normally requires administrator intervention in order to deal with the problems that are occurring. However, as networks have become more complex, and users have had increasingly more complex requirements, the system administrators are often overcome by a mountain of network incidents requiring investigation, which they are not able to deal with in a sufficiently timely manner.

The primary concern of this project was to attempt to ease this load by integrating a NIDS into a firewall, allowing routing to be dynamically affected by network behaviour. In order to make such a system easy to use, and scale in the enterprise, the configuration for the system is through the use of an administrator-configured policy. The project produced a software application entitled Rubicon, written in C for Unix platforms. In order to maximise the potential of the product, it is aimed to analyse and modify traffic on layer 2 and above of the OSI 7-layer model, i.e. Ethernet and above.

Acknowledgments

Firstly I would like to thank my first and second markers, Emil Lupu and Naranker Dulay, for allowing me the space to pursue my own ideas whilst keeping them grounded whenever they became a little airy. Many thanks to Lance Spitzner and other members of several security mailing lists for the positive feedback offered and pointers to several useful resources.

I would very much like to thank certain anonymous benefactors for their sponsorship in the form of the loan of several computers. Without these resources I would have had a great deal more difficulty in developing this software.

Finally, thanks are due to my fellow students for joining with me in the imbibing of vital liquid refreshment whenever I felt I was banging my head against a brick wall.

Contents

Table of Contents

Table of Figures

1 Introduction

1.1 Aims

The aim of this project is to produce a software package that will integrate Intrusion Detection and firewall technologies in an easily extensible and controllable format. This software package, named 'Rubicon', solves current problems in network administration (as described below) by reducing the need for human intervention when dealing with security events. This reduction speeds reaction time to security events, and reduces scope for human error in dealing with events, both of which increase network security for a well-configured system.

The system is aimed to work on the common internet protocols (namely Ethernet, IP, TCP, UDP, ICMP). It is configured by an XML policy, which can also include rules in 'snort' format. It is also aimed to provide most 'normal' firewall functionality in addition to pattern matching intrusion detection, and be capable of making routing and logging decisions based on all packet headers and intrusion detection results. Finally, in order that further development may be easily made without modification of existing code, the design is modular and extensible.

1.2 Motivation

Computer networks normally take the form of multiple interconnected host computers, which communicate with other computers via a small number of defined gateways. These gateways, as well as simplifying routing, provide natural chokepoints for network traffic at which border controls and filtering, often called firewalling, may be performed. Firewalls in general decide whether to allow or deny network connections and traffic to pass through based on a set of rules related to the contents of the incoming network packets. These rules are normally relatively static, and only depend on a subset of the packet header fields. Whilst firewalls can reduce the vulnerability of computer networks to security issues, they are not foolproof. Errors in rules, rules not being updated in a sufficiently timely manner, and the shortcomings in not being able to make routing decisions based on packet data all reduce the reliability of firewalls. Additionally, by definition only traffic passing through a firewall may be filtered, and so both internal traffic and network traffic passing through unsecured gateways are impossible to filter. It is therefore desirable that the network be monitored for unauthorised access.

In order to gain this information on possible attempts to intrude on computer networks, and break into computer systems, Intrusion Detection systems are used. These take three forms: 1) Host-based (HIDS) - Software that sits on each host and watches for suspect activity, such as privileged files being accesses, 2) Network-based (NIDS) - Systems that monitor network traffic for suspect activity, 3) Hybrid - Systems that have both HIDS and NIDS elements, normally with a central manager to correlate data from multiple HIDS and NIDS 'sensors'. Intrusion Detection Systems (IDSs) are passive information resources, which collect data and then present it in some form to human administrators. These administrators must then investigate flagged suspect network events and then respond accordingly. This investigation may be complex and time-consuming as all IDSs suffer from misreporting activity, termed 'false-positives' and 'false-negatives'. A false positive is the reporting of activity as suspect when in reality it is not. A false negative occurs when an IDS does not flag traffic that is in reality suspect. Due to the large amount of network traffic that passes through a busy network, even a very low 'false-positive' rate can lead to very large numbers of events falsely flagged, each of which in theory needs to be investigated. This therefore leads to a long delay between notification of dangerous activity and the required response to it, which greatly reduces any benefit derived from the IDS. Additionally, due to shortcomings in many computer protocols, and different implementations of these protocols, it can be possible to disguise suspect traffic, thus causing false-negatives.

It is this length of this response loop in dealing with potentially suspect traffic that this project is aimed to deal with. By integrating intrusion detection and firewall technologies it is possible to automate reaction to perceived threats. This reaction may take many forms, from the simple notification currently offered by IDSs, to the closing of network connections, blocking of certain systems, and inline-modification of packets to neutralise threats. When configurable in a sufficiently flexible way, the reaction may be tailored to the severity and false-positive rate of a given threat, in order that normal users are not affected. A modular design allows good extensibility leading to the provision of new types of analysis and response. Indeed, it will be possible for the system to perform non-security related networking behaviour, such as Quality of Service support and Load Balancing. This unified system therefore allows for very good homogenisation of computer networks by only requiring support for a single software package, and a single policy language, as compared to the highly heterogeneous networks that are currently the norm. Better homogenisation increases ease of support and configuration, as well as improved security in some areas such as up-to-date patching.

The automation of response may already be performed in basic form through the use of Unix scripting. However, this approach is not scalable and depends entirely on the technical skills of the system administrators. The benefits that may be accrued through the proper implementation of an integrated system include better scalability, lower support, and the important reduction in delay between threat detection and response, leading to greatly improved security. Some proprietary software performs a limited amount of automated response, and there is some work in the public domain for this. Current techniques however involve 'wrapping' an IDS with some element of response capability. There is currently no truly integrated system in the public domain with the capabilities desired. For further information on the current state-of-the-art the reader is referred to section 3(below).

Whilst a naïve implementation of this would not be especially difficult to form, by hard-coding protocols, functionality etc into a single process, this would not offer the level of customisability and extensibility desired. It is a great deal more challenging to design a system that can allow new protocols to be specified and then fully supported at runtime, whilst still retaining sufficiently fast throughput.

A more in-depth look at Network security, firewalls, IDSs and the current state-of-the-art in these areas is given below in section 3.1.

1.3 Deliverables

The primary deliverable for this project is the 'Rubicon' software package. The software package allows for local configuration in the form of XML policies, and 'snort' format rules. Routing decisions may be made based on any non-optional field in the Ethernet, IP, TCP, UDP and ICMP protocols, as well as pattern matching intrusion detection. Responses include local logging in file or syslog format, remote logging using the IETF Intrusion Detection Message Format, the standard firewalling rules of allowing, dropping and rejecting packets, and the inline modification of traffic. Sources for network traffic include network, files, and the Linux kernel. Additionally, a report, user and developer’s manuals, and a short presentation will be produced.

1.4 Reading guide

Following this section, a brief summary of background information on the subject matter is given in section 2. This background provides a brief introduction to the purposes and current technologies in Intrusion Detection, Firewalls and Integration of Firewalls and IDSs, together with an example situation in which these technologies would be used. The background then covers different methods of receiving network packet data, and then surveys the different analyses that may be performed on received traffic. Examination is made of the different forms of output that may be implemented. Finally the background covers the issue of configuration and programmability through plugins and policies.

Following this, an overview of the entire system design is given in section 3, with coverage given to system-wide decisions, overall architecture, and a functional representation for this architecture, and an example highlighting the functionality desired. The component parts of the system are identified and the interfaces between them defined.

In sections 4, 5, and 6, more in-depth implementation details are given. The design of the central loop is covered in section 4, with a look into the implemented plugins given in section 5. Finally section 6 deals with the policy, it's querying, transmission and local and remote storage.

Testing and evaluation are covered in section 7, followed by conclusions and recommendations for further work. After this appears a full bibliography and guides for both users and developers for the system.

The code developed for this thesis, and full documentation and user manuals are available online at:
http://www.ianpeters.net/project/

2 Background

Prior to any system design, the current state of the art is investigated for the different constituent parts of the system. Firstly, an introduction to Intrusion Detection systems is given in which the fundamental ideas behind Intrusion Detection are discussed followed by a look at current products and techniques. Firewall technology is then similarly dealt with. This is then followed by a review of current methods for integrating Firewalls and IDSs.

Following this is an analysis of different methods for a program to receive network traffic. A summary of several different ways to log and respond to threats is then given. An examination of methods of providing extensibility in the form of plugins is then briefly given, followed by a review of some of the different languages for specifying policies, as well as methods for handling these languages (for example: combining them).

2.1 Intrusion Detection Systems

It is one of the paradigms of computer and network security that there is not such thing as 100% secure. The development of Intrusion Detection Systems (IDSs) resulted from this, and a desire to detect when security breaches have occurred. Specifically, the purpose of IDSs is to flag behaviour that may correspond to a security breach or ongoing attack. They have also been used to check for many breaches of computer policy, such as the accessing of restricted or adult material, the playing of games, and so on. Once flag, it is normally up to system administrators to investigate whether any such breach, or attack has occurred. Therefore, IDSs are purely an information service, not a pro-active computer security defence.

There are three forms of IDS: Network (NIDS), Host (HIDS) and Hybrid systems. Network IDSs monitor network traffic through some form of ‘tap’ (figure 1, below), i.e. an interception point. These then perform analysis on network traffic as it passes by, and log any suspect activity. Host based systems reside on the hosts of the computer network, i.e. the servers and desktop computers of the users. Rather than analysing network traffic, these watch the behaviour of the hosts on which they reside for evidence of breaches and attacks. This may be performed purely from the log files produced by the host operating system, or the IDS software may hook into the operating system itself to watch system calls and such. The final form of IDS is hybrid IDSs. These comprise both Host and Network based, performing correlation between them.

In the Enterprise, and high-speed networks, hybrid IDSs are the norm. These normally comprise or 'sensors', 'engines' and 'consoles'. The sensors are HIDS and/or NIDS placed strategically throughout the network. These perform some form of pre-processing and forward the results to the engines. These correlate the outputs of the different sensors, allowing the display of the results on the consoles.

One of the aims of the Rubicon system is to implement NIDS functionality, however HIDS functionality is currently not planned. Therefore, for specific discussion of HIDS techniques the reader is referred to [IDS (2000)]

NIDS analysis can be performed using many different techniques. The traditional NIDS uses pattern matching to look for previously identified patterns in network traffic, for example, a known exploit or similar. This technique is still used by most IDS systems, such as the open-source IDS snort [snort 2002 [online]; Roesch 1999]. Pattern matching, whilst a very simple approach, suffers from many shortcomings. Firstly, it is potentially a very expensive process in terms of processing time, due to the large number of byte-byte comparations required. Several algorithms can be used to reduce this, such as the Boyer-Moore fast string-searching algorithm [Boyer (1977)], as used in snort. Coit et al [Coit et al (2001)] propose an optimisation for this, and demonstrate a 130% to 330% improvement in throughput when incorporated into snort. This optimisation does however modify the algorithm in such a way that some snort rules patterns no longer match correctly. A significant shortcoming in pattern matching is that by definition only known patterns may be searched for. Thus unknown attacks may not be detected as the pattern has not yet been written. Certain 'generic' patterns - such as NOP slides (long lists of no-operation machine code) - allow some such attacks to be detected, but by no means all.

An alternative to pattern matching is statistical and anomaly analysis. These may take many forms, but in general work by comparing the behaviour of the network, and the packets thereof, against some form of heuristic. This analysis is therefore looking for anomalous behaviour. [Caberera et al(2000)] examine the application of Statistical Traffic Modelling for detecting novel attacks. They look at lower level network activity, and find that denial-of-service and probing attacks do leave statistical traces. Additionally they consider Application level protocols, specifically telnet, and use the Kolmogorov-Smirnov test to show that attacks look statistically different to normal telnet traffic. [Nong et al (2001)] look into a purely statistical process, using what they term a Chi-square test, and a learning misuse detection technique called clustering. They demonstrate very good results for the Chi-square test, although less good for clustering. [Shan-Zheng et al. (2001)] discuss a knowledge based IDS technique that builds upon traditional techniques by attempting to track the state of network connections and observing abnormal behaviour. They propose that this offers good anomaly detection, and better efficiency and accuracy than many other techniques. [Ho-Yen-Chang et al (2001)] discuss a combination of both traditional knowledge based and newer statistical techniques, implemented as a timed Finite State machine. They discuss this with reference specifically to the OSPF routing protocol. They achieve 100% detection with no false positives in their evaluation, however only run a total of 3 attacks, and so this is only limited evidence of accuracy.

Anomaly analysis has the benefit of being able to detect unknown attacks. However, they have some shortcomings. The primary shortcoming is the heuristic used for measuring anomalous behaviour. For non-knowledge based systems, this heuristic normally must be 'learnt' be training with non-anomalous data. This increases complexity and development time. Once trained, if any changes are made to the allowed traffic, i.e. the definition of an anomaly is changed, the system must be retrained or else the now allowed traffic may be flagged as anomalous(false positive), and newly disallowed traffic will not be flagged (false-negative). Knowledge based systems suffer less from this effect as anomalies are hard-coded as knowledge into the system by the developers and/or administrators. However, these can suffer from false positives solely due to 'broken' implementations of protocols by some vendors, and false negatives from attacks that are not anomalous in terms of protocols, but rather are relying on 'bugs' in implementations on certain systems. Knowledge of these may be incorporated, although this therefore reduces the ability of the system to identify unknown attacks.

For more information on Intrusion detection, the reader is referred to [IDS (2001)]

2.2 Firewalls

Firewalls are used to regulate the traffic passing to and from a computer network. In order to do this, they are normally placed on the borders between networks (figure 1), and so are often termed border control devices. The aim is that by restricting traffic passing to and from networks to only that traffic which has been specifically allowed, the security of the network may be improved. For example, one common method of breaking into Microsoft Windows machines is to connect to its file sharing service on port 139. By blocking all traffic to and from port 139 at the firewall, the network can be made more secure by disallowing any connections to this service from outside the trusted network.

In general, firewalls operate by examining the contents of specific protocol headers, for example: IP source and destination address, tcp/udp port. Rules are then parsed to determine what behaviour to perform dependent on the contents of these headers. As a minimum, this behaviour is to either ALLOW it, forwarding it through its destination, or to DROP it. Other alternatives include logging, and rejecting (sending an ICMP unreachable message).

Firewalls can be either stateful or stateless. Stateless firewalls examine each incoming packet on its own merits. Stateful firewalls store some form of state information, for example the status of a TCP connection, and can perform analysis using this stored state in addition to the packet header. Stateful firewalls are more secure as traffic that is valid in some states is not in others, and so rules may be more strict than with a stateless firewall, were the rules must allow traffic through irrespective. For example, TCP is a connection-oriented protocol, which starts with a 3-message handshake. If, for example, a stateless firewall is used to disallow connections incoming to a private network, then it can block any incoming packets which have their syn (create new connection) flag and no others, set. However, this rule must allow all other traffic through it has no way of determining whether the traffic is part of a connection started from the inside of the network. If, however, a stateful firewall is used, then it can be set to allow any outbound packets and only inbound packets related to a connection started from inside the private network. This would therefore block traffic that a stateless firewall would have to let through, allowing for better security.

Firewalls are a relatively mature technology. Recent work has mainly focused on increasing the number and type of protocols inspected, as well as improving efficiency in order to handle the fast networks in use today. There are several shortcomings to firewalls, some of which are being and have been addressed. The first of these is that they in general only inspect the lower-layer protocols, namely IP, TCP, UDP and ICMP. They cannot, for example, filter out higher layer protocols. This has been addressed through the use of 'proxies' and 'application firewalls', which effectively filter at the application layer. The second shortcoming is that firewalls cannot inspect encrypted traffic, unless they have access to the key(s) in use.

Unix and Linux have long had firewalls installed by default, with the most notable being the netfilter/iptables [NETFILTER 2002 [online]] firewall software. This is a stateful firewall, fully integrated with the operating system kernel.

2.3 Example Network Showing Uses for NIDS and Firewalls

An example network showing some of the key features of a secure network, and the placement of Network Intrusion detection systems is shown in figure 1.

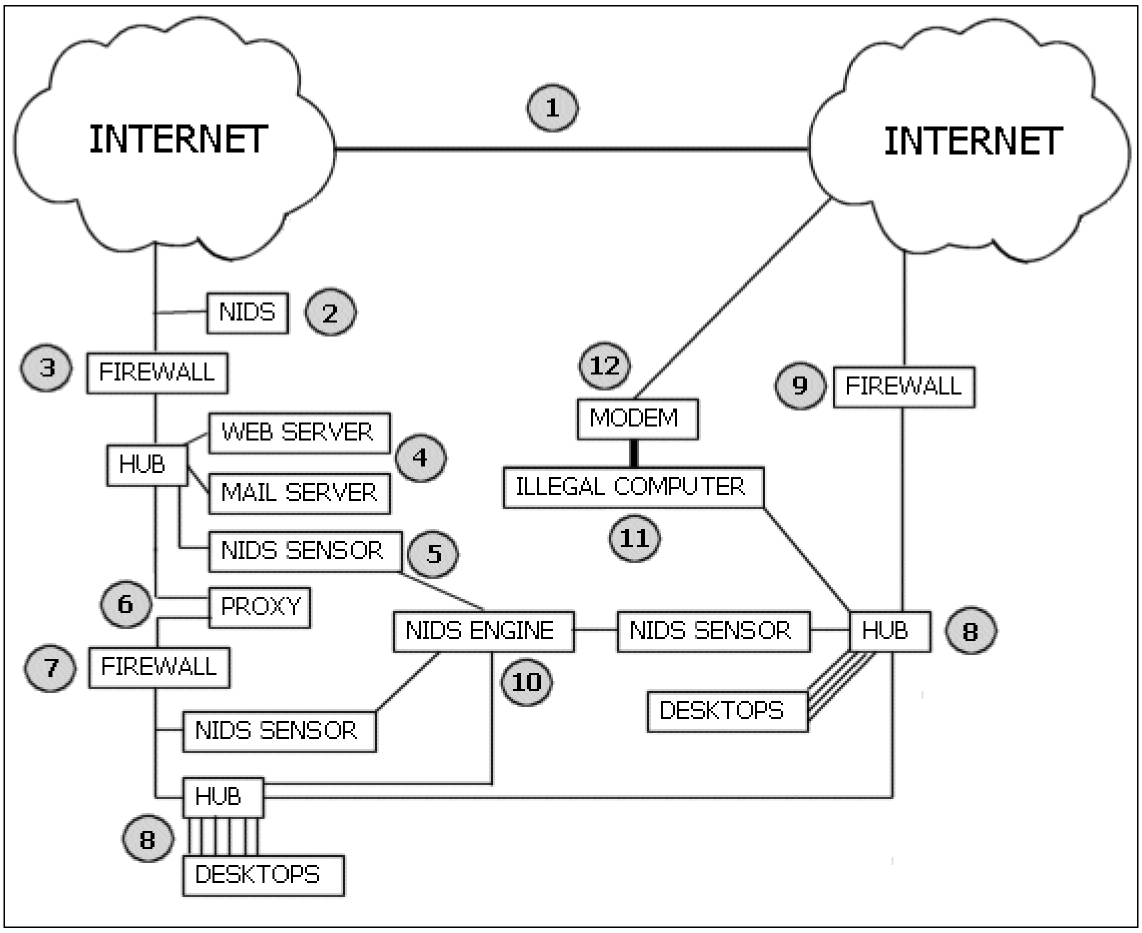

Figure 1 - An example secured network.

In this figure, we can see several key points, which are numbered. The network is connected to the internet (1) in three locations. The first of these is the primary connection. Placed at this boundary is a firewall (3) which allows traffic to and from the public web and mail servers (4), and the proxy server used by the network (6). The internal network may only normally communicate with the internet via this proxy. In addition there is a second, internal, firewall (7). This architecture is normally called a screened-subnet with dual-homed host. The purpose is to keep the public servers secure instead a 'DMZ' (demilitarised zone), whilst recognising that they may be broken into. If this were to occur, the intruder would then still have to compromise the proxy server and internal firewall before being able to gain access to the internal network. This concept of defence in depth is a key tenet of secure networks. Outside the firewall is positioned a NIDS (2), which taps the traffic passing to and from the external firewall. This can be used for information gathering reasons, to see how hostile the internet is, and check that the firewalls are operational. This is a standalone system so that in the event of a compromise the security of the entire system wouldn't be damaged.

The network is primarily composed of two hubs, each connected to multiple desktop computers (8). The second connection to the internet is via a backup firewall (9). This is normally off, but may be activated in the event of a breakdown occurring to (3), (6) or (7). Due to the lack of defensive depth this is a much less secure set-up. Monitoring the entire network is a NIDS Engine (10) that is accessible by the network for administration reasons. It would be better to have a totally separate network in parallel to the users network for administrative reasons, but this is very rarely implemented. The NIDS engine retrieves data from numerous sensors (5) around the network. In addition to detecting security events, these can also be used to diagnose breakdowns in the network. The use of a central NIDS engine to correlate information can be demonstrated if, for example, a person inside the secure network makes a hacking attempt. The NIDS should ideally detect this, and after comparing the results from its sensors should be able to identify that the attack was made internally, rather than from the internet.

In addition to detecting hacking attempts which have managed to progress through the firewalls, and attempts made internally, violations of company policy are also detectable. For example, policy may specify that it is not allowed to connect laptops to the network. If this does however occur (11), the NIDS will detect traffic emanating from a system not on the list of allowed systems, and warn the administrators accordingly. However, it must be remembered that NIDS can only examine network traffic that passes them. For example, if a modem is connected to a computer on the network, which is then used to connect to the internet, a very dangerous backdoor into the network is created. Moreover, this connection will not be detectable, as no traffic will necessarily pass on the network to indicate the existence of the modem. This is a situation that installing HIDS on all the systems on the secure network can detect.

2.4 Integration of Firewalls and IDSs

As mentioned, the purpose of IDSs is to inform administrators about suspect traffic. The network administrators then must investigate the incident, and if perform any remedial actions necessary. For even rules with very low false positive rates, this may still lead to very large numbers of incidents requiring investigation, often swamping available manpower resources, causing substantial delays in dealing with actual security events.

Even when the delay is low in human terms, the requirement for human interaction greatly delays processing in machine terms. IDSs are reactive in nature, and so by the time a person is informed of an event, it has already happened, and the event will have caused any damage it was likely to cause.

In order to reduce the delay between an event being noticed by the IDS, and being properly handled, it is desirous to automate the system to some extent. Rules with low false positive rates, when combined with sensible reactions, can greatly enhance the security of a network by intercepting and dealing with changing threats.

There are many different ways of performing this analysis and reaction. Firstly, it is possible for the IDS to forge network packets to interfere with any suspect connections. For example: by sending a forged TCP reset packet to either end of an ongoing connection, the IDS may force the closure of suspect connections. Snort natively has this facility, as do several commercial IDSs. This has the benefit of being already implemented, but can only react once events have happened.

Alternately, a program can listen to the IDS log files, and perform actions based on these. For example, a script could listen for messages specifying IP addresses that have port-scanned the network, and then configure the relevant firewall to block the specified address. Programs such as swatch can be used for this purpose. This is more extensible than relying on built-in IDS features, but again is only reactive in nature. This is however easily written and extended.

Another alternative is to slightly modify the IDS code to perform the desired response, and expand the configuration language of the IDS to allow specification of the responses. The hogwash program [HOGWASH [online]] works this way by wrapping and modifying snort. This has the benefits of building on already existing code, and being slightly more powerful and efficient than listening to logs. Whilst it is possible to make this wrapped system act proactively, by placing the system inline with the gateway device so that all traffic must pass through it. Whilst a strictly functional system may be relatively easily written, this will not prove as efficient when running as a well written bespoke system, due to the system fundamentally remaining an IDS at its core.

The opposite to adding firewalling functionality to an IDS is to instead add IDS functionality to a firewall. This could be performed, for example, through the addition of an IDS 'table' to the netfilter firewall software. This would again reduce the amount of new code to be written, and improve efficiency for non-IDS rules due to the already existing firewalling functionality. However, this would add a reliance on the netfilter software being installed on any system to be used as an IDS/firewall - which would cause problems as earlier versions of Linux, still widely used, came with earlier firewall software already installed, software that would conflict with iptables. Additionally, many firewalls are run, for performance reasons, inside the operating system kernel. Intrusion detection is a highly processor intensive process, and so running intrusion detection algorithms inside the kernel - as would be required if IDS functionality is added to an existing firewall implementation – would have a serious negative effect on the performance and stability of the host system.

Finally, the IDS and firewall software could be integrated into a bespoke system. This would be the most costly alternative in terms of development time due to the requirement of writing the entire software package from ground up. It would however provide the benefits of being designed specifically for the purpose and so would be the most efficient of designs.

Several commercial firewalls have recently been integrated with IDS. Checkpoint Smartdefense [CHECKPOINT], now in beta testing, and BlackICE PC Protector [BLACKICE] only implement the basic functionality offered by snort, namely sending RESET packets to close connections, in addition to normal logging. These are both systems aimed at the home and small business user. Realsecure network defense [REALSECURE], and Symantec Intruder Alert [SYMANTEC] are both aimed at the enterprise market and so have features (as well as price tags) that reflect this. Neither natively support the modification of firewall rules nor do they perform full filtering themselves as Rubicon aims to. The do however, both have the facility for running any script of executable on the admin host, and so gain the functionality Rubicon aims for by taking the wrapping the IDS with firewalling code technique.

2.5 Network Traffic Reception

In order to provide any form of processing of network traffic, as is required by a firewall or NIDS, the processing software must acquire the network traffic. Therefore, some form of access to the network interface is required. Additionally, it is desirable for testing and offline processing reasons that it be possible to import data from files.

Several formats for storing network traffic data exist, with the most common and most cross application of these being the tcpdump/pcap format, as used by tcpdump [TCPDUMP 2002 [online]], ethereal [ETHEREAL 2002 [online]], snort [SNORT 2002[online]] and many others.

For collection from a network interface, three possible methods have been identified. The first of these is through the use of a 'raw' socket (Winsock or Berkley). A raw socket retrieves and sends raw data to/from a network interface, without any protocol support or modification by the host operating system. Benefits for this include its cross-platform support. However, the system must itself be running a BSD derived stack with the correct socket types (SOCK_RAW/SOCK_PACKET). There are many bugs associated with this, such as is mentioned in the Raw IP Networking FAQ [RAW IP FAQ 2002 [online]]. Additionally, this can be a quite complex method due to the archaic nature of certain of the networking system calls.

The second method is to acquire traffic from firewall software installed by default on some operating systems, namely netfilter/iptables [NETFILTER 2002 [online]]. This is a kernel packet filter (i.e. firewalling software) implemented in later versions of the Linux kernel. Netfilter provides the capability to create rules to pass certain packets to userspace applications for processing, which provides a useful and easy way to obtain these packets. However, this functionality is limited to only those newer kernels which support netfilter, and the kernel must have IP support and netfilter either compiled in, or as a loadable module.

Finally, an attempt to make a cross-platform network capture API has been made in the form of libpcap [LIBPCAP 2002 [online]]. Libpcap is currently able to run on most commonly used operating systems and platforms, including many flavours of Unix, and Microsoft Windows support. This truly is the most cross-platform of the methods therefore. Additional benefits include it’s support for opening tcpdump file format network dumps, having a simple yet powerful API, being widely used and as it directly connects to the network interface device itself, does not require an IP stack on the host computer. It does however require that libpcap be installed on the host machine, as well as elevated privileges if promiscuous network access is desired.

2.6 Logging and Response

There are three classes of output for a combined IDS/firewall system. The first of these are the logging outputs normally provided by Intrusion Detection Systems. The second are the outputs to be found in firewalls. Finally there are outputs that are relatively specific for combined IDS/firewall systems or similar.

It is expected of an Intrusion Detection system that it be able to log suspect packets. Normally this includes syslog, text files, and binary packet logs. Additionally, SMB popup and SNMP trap messages may be sent to administrators in the case of an emergency. In addition to this, of interest is work being performed by the IETF on an Intrusion Detection Message Exchange Format [Curry et al. 2002] and Intrusion Detection Exchange Protocol [White et al. 2002]. An implementation of these protocols is available in the form of libidxp [LIBIDXP 2002 (online)]. This exchange protocol is envisaged as becoming the standard method for communication between IDS sensors and engines, and so any new IDS should support them.

Firewalls by definition must be able to either forward packets, or drop them. Additionally, most provide the functionality to reject packets, by dropping them and replying to the packet source with an ICMP unreachable packet, and masquerading. Masquerading is a form of network address translation (NAT), whereby the source IP of outgoing packets are changed to hide their originator. When the reply is received, the masquerading firewall translates the destination address of the incoming packet to the original source address of the outgoing one. The primary usage of this is to enable multiple hosts on a private subnet to access the internet via a single world-accessible IP address. NAT can take many forms, each of which essentially just translate the source and/or destination IP addresses and/or TCP/UDP ports on specific filtered packets.

The addition of IDS analysis to firewall rules allows certain other outputs to be used. For example, certain network traffic could be selectively packet/protocol scrubbed, as described in [Malan et al(2000,2001)], were it is suggested that by enforcing rigid compliance with protocol guidelines and the removal of ambiguity by altering passing packets many security gains may be made. For example, one method of bypassing IDS detection is by taking advantage of the different ways that differing operating system protocol stacks handling overlapping IP fragments. IP packets fragment their data into several smaller packets for forwarding. If this occurs, a bit is set in the header, and an offset is used to determine where a fragment fits in the overall transmitted data. However, the behaviour of a protocol stack in reassembling fragmented data is ambiguous in the case of overlapping fragments. Some stacks use the contents of the first packet received as the value for the overlapped region, whereas others use the last. If an attacker believes that an IDS uses a differing technique to the system they are attacking, they can hide the exploit from the IDS by using an overlapped region. By scrubbing packets as they pass, overlapping fragments can be modified so that they no longer overlap, removing any ambiguity and hence making the network more secure. Another output is the connection resetting mentioned above. For example, whenever a dangerous command is intercepted in a TCP connection, for example an 'rm -rf /*' on a telnet connection, the system could drop the packet and close the connection by sending forged TCP packets with their RESET field set to both the client and server, pretending to be from the other end of the connection. Finally, load balancing [Bryhni et al 2000; Gupta et al 1999] (possibly using NAT), and Quality of Service [Matta 1998] support may be added.

It is easy to log to syslog and text files, however it is somewhat harder to output to the network in a cross-platform manner. Raw sockets (SOCK_RAW/SOCK_PACKET) may be used to output as well as input data, and so the benefits and problems are as discussed under Network Traffic Reception, above. An alternative is an API entitled libnet [Schiffman 2000 [online]], which allows good cross-platform access to network devices, but similarly to libpcap, requires installing on the host machine.

2.7 Programmability and Configuration

One of the aims of the system is that it be highly extensible and controllable. In order to be easily extensible, modular design must be used, with the facility to load and unload modules at runtime. Several techniques were investigated for this, all of which ultimately devolved to the same technique - the use of libraries, and specifically shared libraries (also known as DLLs on Microsoft Windows machines). These are accessible via the POSIX system calls dlopen, dlclose and dlsym, and so should be capable of working on any POSIX compatible platform.

In order to configure the system it was desired that a high-level policy language be used, in order to take advantage of abstraction where possible. A summary of each of the key policy papers is presented below.

Gary N. Stone, Bert Lundy, Geoffrey G. Xie. “Network Policy Languages: A Survey and a New Approach”
Current network policy languages are surveyed, with some criticism of each. A summary of current methods for detecting policy conflicts is then given. Finally, a new policy language, Path-Based Policy Language, is presented which is aimed at solving and/or alleviating problems mentioned in the previous survey. A useful paper primarily for the breadth of policy languages considered, and the criticism of each.

B. Moore, E. Ellesson, J. Strassner, A. Westerinen “Policy Core Information Model – Version 1 Specification”
The Internet Engineering Task Force’s (IETF) objected oriented policy language is here presented. It has been designed as an extension to the IETF Common Information Model (CIM). As a language it appears to have many shortcomings, such as no support for events, and it has been reported that there are significant problems involved in creating software to support it.

B. Moore, L.Rafalow, Y.Ramberg, Y. Snir, R. Chadha, M. Brunner, J. Strassner, A. Westerinen “Policy Core Information Model Extensions”
This is a draft document extending the Policy Core Information Model (PCIM) above. Both additions and modifications to PCIM are described. It is unknown whether these changes will provide better support for usage of PCIM.

Yasusi Kanada "A representation of network node QoS control policies using rule-based building blocks"
This brief paper considers modelling network nodes and domains through 'building blocks'. A language (SNAP) is then presented to utilise this methodology in a stream-oriented context. However, no details of implementation or language specification are given.

Randeep Bhatia, Jorge Lobo, Madhur Kohli. "Policy evaluation for network management"
This paper considers policy evaluation of the PDL policy language. It produces a policy evaluation algorithm and discusses theoretical and empirical behaviour. It concludes that whilst the problem of evaluation is ultimately intractable, through a few realistic assumptions the algorithm submitted is relatively efficient at enforcing policies in complex networks.

Burns-J, Cheng-A, Gurung-P, Rajagopalan-S, Rao-P, Rosenbluth-D, Surendran-AV, Martin-DM-Jr. "Automatic management of network security policy"
A new very high-level policy language, Security Policy Language (SPL), is presented in this paper, representing a high level approach to network policy design. The approach attempts to separate what the authors term 'invariants' (policy specification) from the task of finding the changes to be made to a network in order to maintain the policy (policy management). They attempt to describe network policy through the definition of reachability and availability of resources and services, and specific definition of administrative edges. Some implementation details are also considered.

Damianou N, Dulay N, Lupu E, Sloman M "The Ponder Policy Specification Language"
This paper presents the Ponder policy language. The language is object oriented, strongly typed, and has software produced for it in Java. It is role-based, supporting delegation, obligation and authorization policy rules, making it suitable for describing Host-based policies. Support for network policies is less good, although the extensibility offered by the language assuages this.

Kanada-Y "Policy division and fusion: examples and a method-or, multiple classifiers considered harmful"
Kanada considers the problems associated with translating two or more high-level policies into one low-level policy (fusion) and one high-level policy into two or more low-level (division). A method entitled Virtual Flow Labels (VFL) is presented to attempt to assuage some of these problems. Ultimately however the paper concludes certain of the problems may prove intractable.

3 Design Overview

This section defines the high level requirements for the system, and the design decisions made due these requirements. Following this, the overall architecture of the system is given and justified, followed by the overall flow of the system. Firstly, an example situation for the system is discussed in depth, to show a possible use for the system, and to help identify the functionality desired.

3.1 Example Situation

Let us examine and example situation in which Rubicon may be used. The Rubicon software is installed on a host with four network cards, as show in figure 2. One of these interfaces is assigned a fixed address .10 (i.e. xx.xx.xx.10 where xx.xx.xx represents the subnet). Another is assigned a non-private, i.e. world addressable IP address. The other interfaces are left in a 'down' state, with no IP address assigned and no operating system network support.

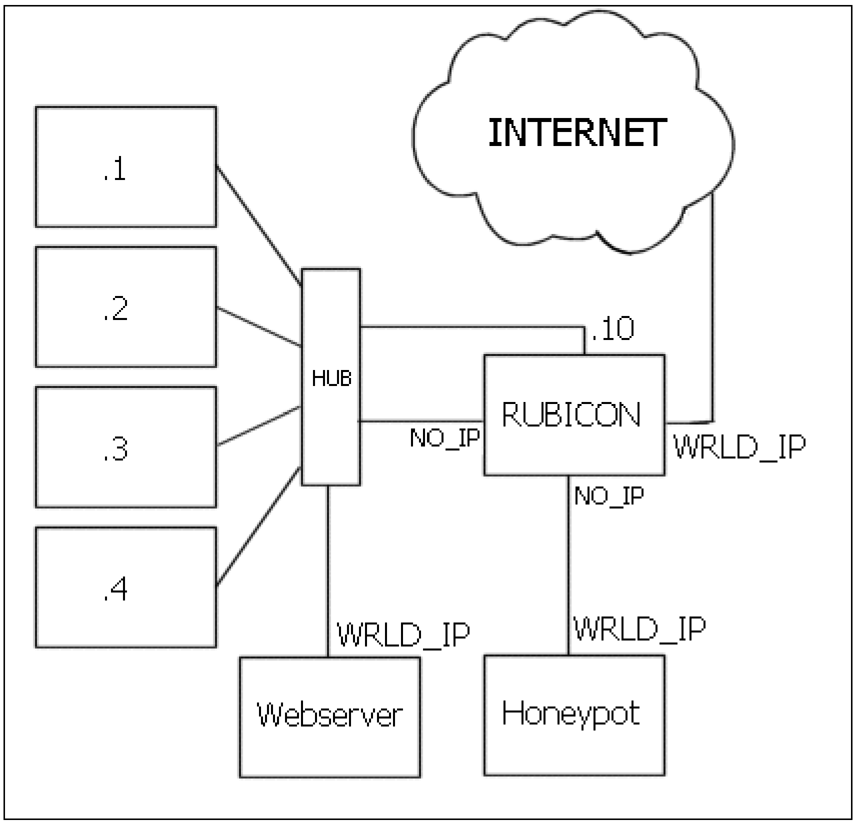

Figure 2 - Example situation for Rubicon.

The interface with a world routable IP is connected to the Internet. The .10 interface, and one of the other interfaces are connected to the hub. The final interface is connected directly to a honeypot host. This host is an exact copy of the web server, including having the same world routable IP address as the external interface of the Rubicon host. Attached to the hub is a LAN of desktop computers each with a private IP address from .1 to .4, and a web server. All machines apart from the Rubicon host are configured such that they do not have a default gateway IP address, but instead have their network interface as the default gateway device. This has the effect that whenever any of the hosts wish to communicate with any other host, they output the packet on their interface, and hence onto the LAN, which forwards the packet to all the connected interfaces.

The following policy could then be applied to the Rubicon host. Inbound refers to packets coming from the internet, and outbound refers to packets destined for the internet. An explanation of these rules follow the list: -

1) Deny any packet not explicitly allowed.
2) Allow any inbound traffic related to an existing connection that was started from inside the (private) network.
3) Allow any outbound connections from hosts .1 -.4 from a source port greater than 1024.
4) All traffic from these hosts should be masqueraded.
5) If the web server attempts to make an outbound connection, send a message via syslog to warn the admin, and drop the packet.
6) The only host allowed to connect to .10 is .1
7) Run the IDS pattern-matching algorithms on any traffic inbound to the web server with a destination port of 80 (http). Allow the traffic if no warning is issued by the IDS. If the IDS flags the traffic as suspect, transparently redirect it to the honeypot.
8) If traffic inbound to the web server is destined for the https port and is encoded with SSL, then decode it and run rule (6) on the resulting http traffic.
9) Run the IDS pattern-matching algorithms on any traffic outbound from the web server. If the traffic is flagged as suspect, close the connection.
10) Transparently redirect any traffic inbound to the web server on any port other than (http) and (https) to the honeypot.
11) Run the IDS pattern-matching algorithms on any traffic outbound from the honeypot. If the IDS flags the traffic as potentially containing an exploit, set every byte of the data to 0. Allow the modified traffic. Log the data prior to modification.
12) Once a packet from an IP address on the internet has been flagged as suspect, all traffic from that system is transparently redirected to the honeypot for the next 6 hours, unless it is present on the list of trusted hosts (e.g. upstream routers etc).

The purpose of the rules is in essence to allow the hosts .1 to .4 to browse the internet by being masqueraded, but only from non-privileged ports. The internet is able to access the web server, however if anything suspect is noticed, then this data, as well as all data from that host for the next 6 hours is transparently redirected to the honeypot. If this was a false positive, then the user is not affected, as the honeypot is an exact copy of the main web server. If the suspect data was in fact an attack, then the attacker will be breaking into the honeypot rather than the web server, and so a security breach is avoided.

Rule (1) is the default rule. All firewalls should deny traffic by default, as they then fail-safe. Rules 2-4 allow the internal hosts to access the internet. Rule 5 detects whether the web server has been successfully attacked, as there is no such thing as a 100% secure system. Rule 6 limits the list of hosts that may connect to the Rubicon host in order to configure it to only the .1 host. This increases the security of the system by not providing an easy route for attackers to break into the Rubicon host. Ideally it would be impossible to connect to the Rubicon host; instead all configuration should be performed locally. Rules 9 and 10 protect the web server from the majority of attacks from the internet. Rule 11 protects the administrators from potential legal liability issues that may result if the honeypot was used as a source computer for breaking into another system. It does this by breaking any exploit included in the packet by setting all the data to Nulls. Finally, rule 12 is used to hide from any attacker the fact that they have been redirected to a honeypot. For example, if the attacker defaced the web pages on the honeypot, they would expect to see the defaced web page when they browsed the website immediately after the attack. However, we do not wish to permanently redirect the IP address as it may be a dial-up connection and so the IP address may refer to many different people even over only a relatively short period of time.

This example highlights one possible use of the Rubicon system, that of incorporating a honeypot into a production or ‘live’ network. Additionally, we may from this example garner some of the functionality that Rubicon could provide: -

- -Outputs
    - Logging/sending syslog messages
    - Forwarding, Dropping and Mangling packets

    - Masquerading and transparently redirecting traffic
- Analysis
    - Connection state tracking (i.e. being able to identify whether a packet is part of a connection or not).
    - Intrusion Detection pattern-matching
    - Source/destination address/port filtering
    - Flagging of suspect IP addresses for periods of time
    - Detection of the direction a packet is travelling in.

3.2 System Design Requirements

Listed below are the detailed requirements of the system. These are listed to facilitate targeted testing and evaluation, and to enable decisions to be made on the system design.
The below specific requirements must be met: -

1. The system must be extensible without modification of core source code.
2. The system must provide a basic subset of normal firewall rule targets, specifically FORWARD, DROP and REJECT.
3. Routing decisions must be able to be made from any field in the Ethernet, Ipv4, TCP, UDP or ICMPv6 protocols.
4. Routing decisions must be able to be made based on the results of a traditional intrusion detection system.
5. The system must operate in a stable way on a specific hardware/OS platform.
6. The system must be capable of being fully transparent at all network layers above the lowest, hardware specific, protocols (i.e. from Ethernet/ATM upwards).
7. The system must be controlled by an easily understood human-readable language.
8. The system must be able to operate on raw network devices, i.e. no Operating System network support/network stack.
9. The system itself must be secure to common network attacks.
10. The system must only allow packets through when permission is given by the configuration.
11. The system must have an operating performance suitable for medium bandwidth networks, i.e. 5MB+ throughput for a basic configuration.

The below requirements are additional targets that it would be good to meet: -

12. The system should operate in a stable way on many hardware/OS platforms.
13. The system should be controlled by a high-level policy language.
14. The system should have an operating performance suitable for medium bandwidth networks when using a complex configuration.
15. The system should be able to dynamically alter network packet contents between reception and retransmission.
16. The system should be able to make routing decisions for more complex problems, for example load balancing or Intserv/Diffserv quality of service routing.

3.3 System Design Decisions

As mentioned, prior to any system implementation being possible, certain overall design decisions must be made. The first of these is the hardware platform(s) and operating systems that will be used. Many possible operating systems may be selected, but ultimately the decision is between a Microsoft Windows based and a Unix based, system. It was decided that as efficient low-level access to resources is required, with emphasis placed on networking, a Unix based system would be most useful. The Linux operating system is well known and supported, free, and Unix-based, and so it was decided to use a recent version of Linux. Due to previous experience of the Mandrake Linux distribution, it was determined that the software platform of choice would be Linux Mandrake 7.2 with a Linux 2.4.9 kernel. Due to ease of acquirement, and as it is the standard and most common hardware platform in use, an x86-based platform will be used.

One of the key design decisions made was which programming language(s) to develop with. Choices looked at included scripting languages (perl, bash), C, C++, and Java.

In order to access the raw network devices and gain access to layer 2 protocols (as specified in requirements 6,8), low-level access to the operating system is required. Additionally, handling network traffic is a highly performance critical job, and it is desired that efforts be made to perform in a relatively efficient manner (requirements 11, 14).

Scripting languages are interpreted languages, which slows their performance down greatly. Bash cannot by itself realistically access low-level functionality, although it can run processes written in other languages to do this. Perl has some functionality for low-level access via certain of its 'modules', and also can be compiled in some ways. However, even when compiled the performance is still not fast enough for the critical nature of this application. Ultimately, both scripting languages are aimed as being 'glue' languages for tying differing processes together, and as such are not really capable of the requirements imposed, primarily due to performance issues.

C is a relatively low-level language that is normally compiled into binary form before running, although C interpreters do exist. Of all the languages examined it has the fastest performance. As a relatively mature language it has compilers available on most platforms, and generally very good support. Due to it's low-level nature it's level of abstraction is not very high, indeed there is close to a 1:1 mapping between many features of the C language, and machine code. This minimal level of abstraction does lead to increased errors, more complex and slower programming, and modularity is less well enforced than is the case with more high-level languages.

C++ is essentially a superset of C. C++ is a higher-level language than C, although the two languages have many elements in common. Most importantly, C++ is object-oriented; allowing a higher level of abstraction, with the benefits in decreased development time, better code reuse and modularity, and better suitability to modern programming techniques that this conveys. Although newer than C there is still good support available, and relatively mature compilers are available for many platforms. C++ often compiles to a larger and slower binary than C, although this difference varies wildly dependent on compiler. Both C and C++ have good low-level access to hardware.

Java is a truly object-oriented language. Although compiled, it is normally compiled into pseudo-machine-code that is then interpreted. Due to this interpretation Java has good cross-platform compatibility. Ordinarily, the java bytecode is interpreted inside a ‘sandbox’ that limits the ability of code to access the operating system to only user-specified areas. This increases security, but makes direct access of computer hardware much more difficult. It is possible to integrate C with Java for this low-level access. Due to java bytecode being interpreted, the performance of Java is not as good as code compiled into native binary form.

This problem does potentially suite an object-oriented language and design approach due to the desired modular design and the self-contained nature of protocol headers. The requirements for good performance and low-level system access meant that Java was not suitable however. After a survey of relevant libraries and freely available code, it was noted that the majority of available libraries were solely written in C. Additionally, the author has substantially greater experience with C. These factors, combined with the slightly better performance and software support offered by C led to the selection of C for the programming language.

3.4 Architecture

The architecture describes the structure of the system at the highest level of abstraction. When designing the architecture, several factors must be considered, firstly how should the system be divided into modular parts, and what these modular parts should be, secondly, how these components will co-exist with each other, and thirdly, how they will communicate.

3.4.1 Components

From the above example situation, two groupings are identified namely Outputs and Analysis. To this must be added Inputs, as the system in some way needs to receive network traffic. The rules themselves also form a part of the system, and so the collection of all the rules is the Policy. Finally, a Central loop is required to pull all the parts together. If we further consider the Input, Analysis and Output parts, we can see that they all operate in some fashion with the network traffic (inputting, analysing and outputting it). It has already been specified that the system be extensible, and the key area of extensibility is through adding new ways to handle traffic. It can be seen therefore that all three of these parts together form the superset Plugins. It therefore follows that by providing facility for the dynamic incorporation of these plugins into the Central system the requirement for extensibility will be satisfied. This is therefore the highest level system abstraction, as shown in figure 3.

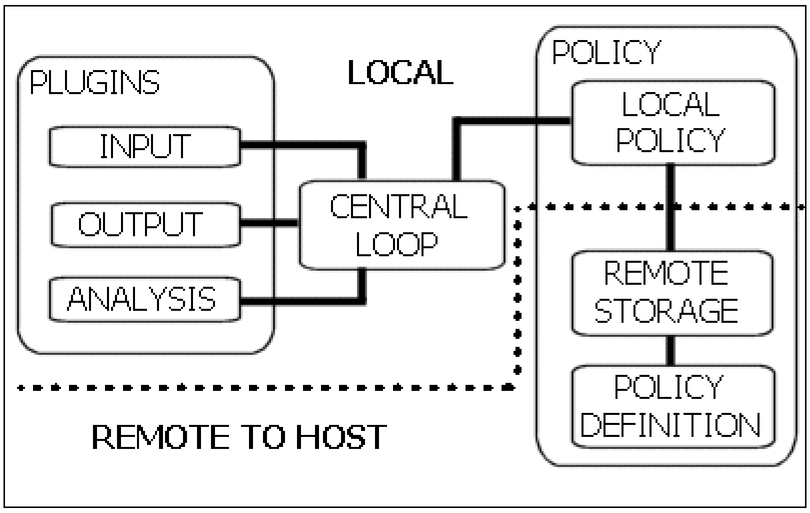

Figure 3 - Rubicon system components

3.4.2 Component Communication

There are several ways these parts could be connected in order to form the complete system. At the one extreme, each can be a separate process, communicating with each other via some form of Inter-process communication. This has the advantage of being parallelised, and so taking advantage of multi-processor systems. Another advantage of this parallelism is that if a particular process, for example the Intrusion Detection Analysis, is taking a very long time, this will not impinge on other processes, minimising the delays of packets. However, a great deal of intercommunication, semaphoring, mutual exclusion, etc will be required, increasing the complexity and thus likelihood of errors and potentially deadlocks. Additionally, there are no truly cross-platform methods for interprocess communication, and so the use of multiple processes would cause problems with respect to this.

At the other extreme is a single monolithic process, which handles packets in a first-in-first-out method. This has the advantage of simplicity of both programming and debugging. Additionally, the performance penalty resulting from interprocess communication is removed. However, if any single packet analysis takes a large amount of time then all traffic will be delayed, potentially leading to packet loss in a busy network.

Another alternative is to use threads (i.e. lightweight processes). These suffer less of a performance overhead than processes, whilst still retaining many of the benefits. Threads do however still increase the programming and debugging complexity, and suffer from cross-platform compatibility issues.

It was decided that the primary concern be to simplify the system complexity. Therefore, the monolithic design was decided upon, with a single thread for the plugins and central parts. The Policy module will be required to download and parse new policies, which is a processor intensive but not very time critical operation. Therefore, it was decided to have a second thread for the Policy module to run in.

3.4.3 Policy subsystem

Policies are defined at a central location, and enforced on the Rubicon host. The policy definition is in a high-level language at a relatively high level of abstraction on the central console. The purpose of this language is to facilitate network configuration by the administrators. The focus of the policy representation on the Rubicon host is to maximise performance. As a general rule, in order to maximise performance, a verbose language or representation should be used. However, one of the desired qualities of a high-level policy language is that it not be overly verbose. Indeed, the more abstract the policy the less verbiage is desirable. Due to the differing requirements of the policy language/representation at either end of the subsystem there must therefore be a translation between the two languages.

A decision must be made on whether the central console should directly push new policies to the Rubicon host, or should some form of centralized or distributed storage be used for caching. Additionally, at what stage translation between languages should occur – before remote storage, between remote storage and transmission, or after local reception on the Rubicon host.

Use of remote storage improves scalability, as storage may be distributed to be nearer policy enforcement points, and the load shared between multiple storage hosts. Centralised storage, when properly managed, offers good synchronisation between all systems that use stored policies as all policy updates occur on one centralised system, however, distributed storage tends to scale better due to the improved proximity of storage to client. Using central console(s) only, simplifies the system and provides immediate updates as required. However, the lack of caching means that if, for example, the firewall host crashes and reboots, it may not be possible for it to immediately acquire an up-to-date policy.

Local translation ensures that the policy gets translated into the correct version of the correct language. Additionally, the policy after translation will be larger than before translation and so policy transmission will require less bandwidth with local translation. However, any form of parsing, translation or compilation is notoriously resource and time intensive process, and so remote compilation removes this load from the Rubicon host, freeing resources for it's primary task of traffic analysis and forwarding. Additionally, if translation occurs before caching in central or distributed storage, then the policy only needs to be translated once and it is guaranteed that all firewalls using it have exactly the same policy *after* translation.

It was therefore decided, for scalability and caching reasons, to use remote storage of the policy (if possible). Policy compilation would occur before remote storage, in order to maximise efficiency of the firewall host. However, rather than a 1-stage translation, a 2-stage translation is performed. The primary translation stage occurs before remote storage for the reasons already mentioned. This translates the high level language to an intermediate language. This intermediate language is then downloaded onto the Rubicon host where the second translation stage occurs, and the intermediate language is translated into a low-level representation. In order to minimise the performance requirements of this latter stage, there is a roughly 1:1 mapping of intermediate to low-level language elements. The purpose of the intermediate language is to remove compatibility issues, for example byte-ordering, allowing a single intermediate level language to be used for all Rubicon hosts, irrespective of the architecture of their platforms.

3.4.4 Interfaces

As can be seen in figure 3 above, the component parts which are situated on the Rubicon host interface with each other at four points; the three plugin-central loop interfaces, and the central loop-policy (internal policy) interface. Additionally, the local policy sub process interfaces with the other policy components on its external interface. Each component module acts like a black box, exporting the relevant functionality through these interfaces. By defining these, modularity is improved, with beneficial effects on the ease of development of extensions. Each of these interfaces is briefly dealt with below.

Plugin – Central Loop Interfaces

Plugins are loaded through the use of the C dlopen, dlsym and dlclose functions. The dlopen function loads a shared object into the heap of a program. The dlsym function can then be used to search the loaded shared object/library for a specific public function, a pointer to which is returned.

There are three classes of plugin: input, analysis and output, as described above. There are certain operations, namely initial registration, that all classes need to implement, and there are certain operations that are plugin type dependent, namely performing input, analysis, and output. Importantly, any plugin may implement more than one class, for example the defrag plugin, used to defragment fragmented packets, is both an analysis plugin – for sensing whether a packet is fragmented, and an input plugin – i.e. it is a source for new packets once they have been defragged.

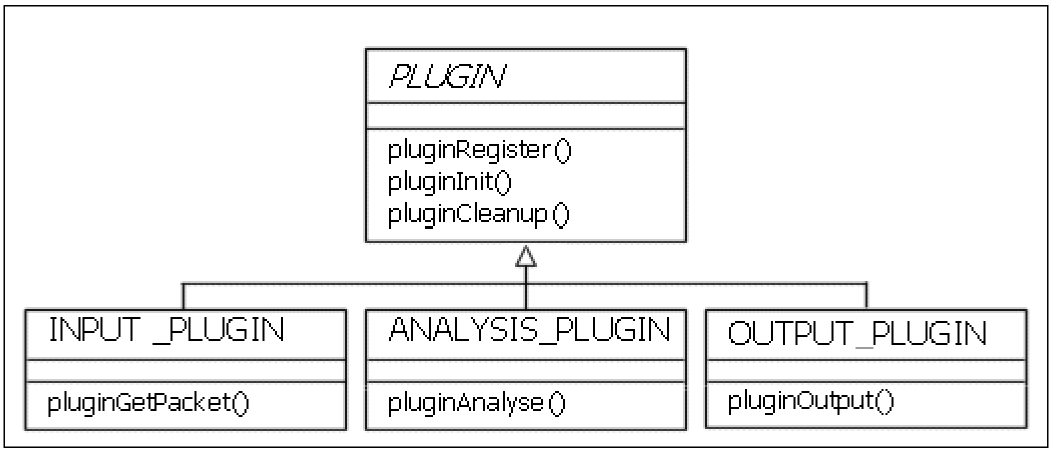

Figure 4 - UML diagram illustrating inheritance of specific plugin types from an abstract common plugin type.

In order to support this the following registration process occurs: -

1) Each .plg (plugin) file in the plugin directory is dlopen'ed in turn.
2) Each file has dlsym called to find the pluginRegistration function.
3) The registration function passes by reference a pointer to three structures, one each for input, output and analysis plugins.
4) The plugin nulls the structures that are not relevant and populates those that are.
5) The function returns and the main program internally registers the plugins.
6) Finally the main program obtains references to the other interface functions in the plugin.

The functions common to all plugins are: -

- pluginRegister – Called to register the plugin.
- pluginInit – Called to initialize the plugin
- pluginCleanup – Called to ensure the plugin cleans up after itself. This is called prior to exit, or whenever the system is sent a HUP signal.

The first three parameters of the pluginRegister function are pointers to structures that are populated by the plugin to indicate its capabilities. The final argument is a pointer to a function that performs protocol decoding. This is used by the plugin to decode received packets if particular protocols are required.

The other functions are as specified below. A UML diagram showing how the plugins inherit from a common abstract class is given in figure 4.

Input Plugin

The Input plugin is used for the acquirement of packets, and so it is no surprise that the function implementing this functionality is titled pluginGetPacket. During registration the INPLUG structure is populated with the plugin name, data types it can provide, and the source of the data e.g. file, network, defrag.

The pluginGetPacket is called specifying the data type and/or the source desired for reception. The function returns pointing to a structure wrapping the packet.

Analysis Plugin

There are two types of analysis functionality, and a plugin can implement either one, or both. The first type offers protocol-decoding functionality. These have the public access function pluginAnalyse called, specifying which protocol to attempt to decode. The plugin should then identify whether the packet can immediately decode that protocol i.e. there are no intermediate protocols undecoded. For example, if an attempt is made to decode for a TCP header and the IP header has yet to be decoded, then the call will fail.

The second type of functionality is performing analysis. This analysis includes the type offered by the Intrusion Detection plugins and also the tests for optional headers in protocol decoding plugins.

Differentiation between these types is provided by the analysis type argument. The data type and other arguments parameters are used for further specification of the exact behaviour desired of the plugin.

Output Plugin

Output plugins are used for both network output, and logging output. These are offered through the pluginOutput function. This function allows the provision of the data types to be output/logged, and other arguments.

Policy Internal Interface

The policy subsystem interfaces with the main program at two points. The first of these is the acquisition stage, where the policy must be swapped from the policy thread to the main thread. The second is the querying of the policy when a packet is received.

When the policy thread receives a policy, it first compiles it into a local representation. A mutual exclusion flag (mutex) is placed on a shared variable whilst it is set to point to the new policy, and a semaphore is incremented to indicate to the main thread that a new policy is available. When the main thread tests for this semaphore and finds it set, a mutex is placed on the shared variable whilst this pointer is copied into the thread. The old policy is passed back to the policy thread via this same process, whereupon it is garbage collected. The semaphore is then reset.

When a packet is received, the HandlePacket function is called. This descends into the policy and performs all the tests and outputs indicated by it. No other handling is required. This is further described in section 8.4 below.

Policy External Interface

The external interface is treated with more detail in section 8.2 and 8.3 below. It comprises two sections, a listening TCP port which receives messages to inform when a new policy is available, and LDAP client code to connect to a LDAP server in order to download new policies. This downloaded policy is in XML format, as specified in section 8.2.

3.4.5 Summary

The overall architecture is split into 5 component parts on the Rubicon host, and an additional 2 parts on the remote side. The host system comprises the Central process, Input, Output and Analysis plugins, and a local Policy component. This latter policy component communicates with the remote storage and central console components. The host software comprises two threads, one running the local policy component, the other running the remaining 4 components.

3.5 Program flow

A diagram showing the program flow can be seen in figure 5. As can be seen, the central component essentially loops, getting a packet, then handling it based on the policy. The local policy component loops in parallel to this, downloading, parsing and storing new policies as they arrive. Other possible approaches include: 1) Integration of local policy and central components to run in series, 2) Farming new packets for processing to several packet-handling threads running in parallel in a round-robin fashion, 3) Interlacing the reception and handling of multiple packets at once.

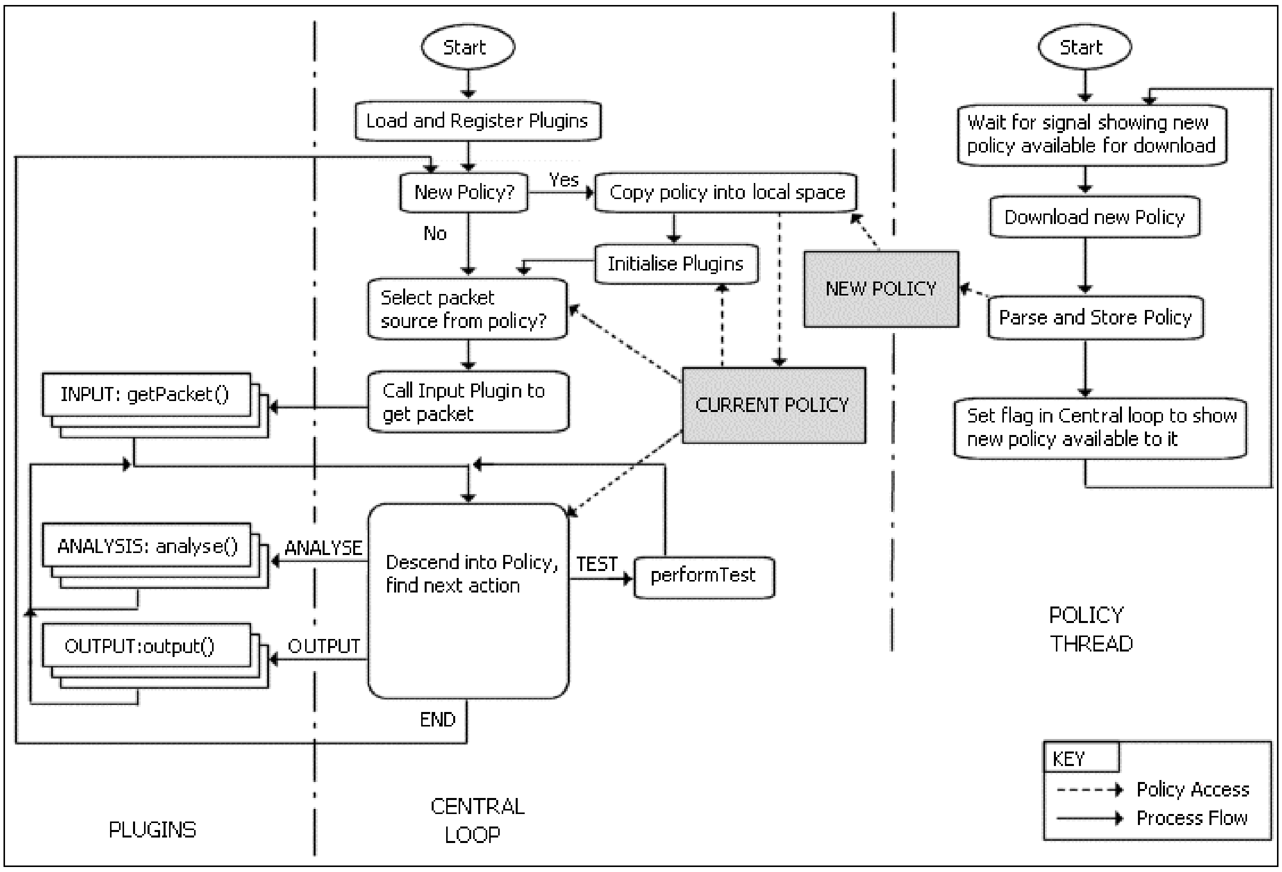
Start
Load and Register Plugins
New Policy?
Yes
No
Copy policy into local space
Initialise Plugins
Select packet source from policy?
Call Input Plugin to get packet
INPUT: getPacket()
CURRENT POLICY
NEW POLICY
ANALYSIS: analyse()
ANALYSE
Descend into Policy, find next action
TEST
performTest
OUTPUT:output()
OUTPUT
END
PLUGINS
CENTRAL LOOP
Start
Wait for signal showing new policy available for download
Download new Policy
Parse and Store Policy
Set flag in Central loop to show new policy available to it
POLICY THREAD
KEY
Policy Access
Process Flow

Figure 5 - Rubicon Program Flow

Whilst alternative approaches (2) and (3) allow multiple packets to be handled in parallel, thus minimising the delays on other packets due to slow processing for a given packet, they would also greatly complicate the system, and cause issues with regards handling related packets in order. Alternative approach (1) provides the other extreme. This is the simplest possible design, however will suffer from the need to regularly pause from handling packets in order to poll for, and potentially download, parse and store, new policies. Therefore, it was decided to design the system as described briefly above, and in more detail below.

First let us consider the central component part. The process starts by loading all the plugins in the specified directory source, and registering their capabilities. The process then waits whilst the initial policy arrives, is parsed and stored. When the new policy arrives, the loaded plugins are initialised as defined by the policy. The packet acquirement and handling loop is then entered.

The process first selects which of the Input plugins to get the next Data packet from, dependent on the policy. The desired Input plugin is then accessed to obtain the next packet. The policy is then descended into, performing tests, analysis and output as required. Protocol headers are tested inside the central component. Analysis, such as IDS pattern matching, is performed via calls to the relevant Analysis plugin.

Outputs, whether logging or network modification, are performed via calls to the relevant Output plugins. This continues until the policy specifies that no more actions should be performed, at which point the process returns back to the start, polls for a new policy, and obtains a new packet.

Running in parallel is the local policy subsystem, the purpose of which is the loading of new policies, and cleaning up of old ones, as required. This component listens on a user-defined port for commands from the central server. Specifically, the process waits until commanded that a new policy is available for downloading. When this command occurs, the most recent policy is downloaded. When downloaded, the policy is parsed and stored in its low-level local form. This new policy is then made available to the central component, and a flag shared between threads is set to inform the central component that a new policy is available. When the central component flags the policy component to say that the new policy has been loaded successfully, the old policy is cleaned up. The thread then loops back to it's start, waiting for a new policy to be made available.

4 Central Component

As can be seen from the Architecture given in section 3.4, the primary purpose of the central component is to glue together the other components. The central process does not of itself provide any functionality, but instead handles calls between other component parts. The method for performing these calls, and the overall structure of the central component are discussed below. Also discussed are the design of certain global structures and the method used for handling and identifying protocols.

4.1 Main Program Loop

As stated, the central component handles calls between the other component parts, namely plugins and the policy. When first started up, the core initialises internal data, and starts the separate policy reception and translation thread. It then waits for a policy to be received. Once received, it loads, registers, and initialises plugins as specified by the policy. A loop is the entered into of querying the policy to identify what source to acquire network traffic from (i.e. the input plugin), obtaining the next packet, and then calling the policy querying code to handle the packet. It also periodically polls the policy reception thread to check whether a new policy is available. If one is, the central component instructs all the plugins to clean up, then loads the new policy, reinitialises the plugins, and restarts the loop.

Two sets of callback functions are provided for use by the packet-handling/policy-querying and plugin components. The first of these is the decode callback. This function has as its parameters a packet, and a list of desired protocols. It then attempts to decode the packet headers until one of the desired protocols is found, or it is determined that none are present, at which point it returns. This is used by most plugins that require specific protocols. For example, the snort plugin requires IP with TCP/UDP/ICMP, and so if the packet has not already had these decoded then it calls the decode callback function to obtain them. The advantages of this technique are that protocols only need be decoded once, and that the process of protocol identification and decoding is separated from that of protocol usage. For example, if support for certain protocols, e.g. Ethernet, IP, TCP/UDP/ICMP, was hard coded into a plugin then it would not be able to handle say, IP traffic over an ATM network. The disadvantage of this system is that it is slightly less efficient and more complex than hard-coded support.

The list of decoded and identified protocols for a given packet are stored in the PACKET structure, discussed below.

The second set of callback functions are used during protocol testing and analysis. As such they are primarily only called by the policy-querying subroutine, although they are accessibly to plugins. There are three functions in this set: Protocol-Tests (typed), Protocol-Tests (masked) and Analysis.

The typed protocol test is used to test a specific field in the header of a specific header. It does this by typecasting the given field to the relevant type (e.g. an unsigned 2 byte field is typecast to an unsigned short), and then performing a normal mathematical comparison, namely whether the value is less than (<), greater than (>) equal to (=) or not equal to (!) a specified figure. The masked test is used to test multiple fields at once, or to test fields not suited to direct typing such as an IP address. This test operates by masking both the test value and the relevant part of the header with a specified value, then performing a bitwise comparison between these masked values to determine whether they are equal (=) or not equal (!). For example, to test that the source IP address of a packet is in the subnet 192.168.0.0/24 and the destination is in subnet 15.16.17.0/24, the mask "FFFFFF00FFFFFF00" is bitwise AND’ed with the 8 bytes starting with the 12th byte in the IP header, and the resulting value then compared with the desired, also masked, value of "C0A800000f101100" (i.e. 0xC0=192, 0xA8=168, 0x0f=15, 0x10=16, 0x11=17).

The advantage of these two types of test is that multiple fields may in some cases be compared at once, and yet at the same time ranges of values can also be compared. If only masked tests were used, testing for values of a field in the range 1-1024 (e.g. source port for a TCP packet) would take 1024 separate tests, compared to the 1 required by the typed test. However, if only typed tests were used, then to check for a specific IP address would require 4 tests (one for each byte), whereas by masking, only one test incorporating all four comparisons need be called. This greatly increases performance, although at the cost of added complexity. An additional advantage is that by just specifying the offset of the relevant fields into the packet, the packet decoding can be separated from the testing. Certain protocols, especially higher level ones, do not have headers as such (POP, SMTP) or have optional headers (e.g. TCP, IPv6), and so cannot be tested in this manner. The method used for dealing with this is that a call can be made into the relevant analysis plugin (i.e. the plugin that decoded the protocol), to test these fields.

The analysis callback just identifies the desired analysis plugin, and calls its pluginAnalyse function with the desired arguments.

4.2 The PACKET Structure

In order to keep all information pertinent to a specific packet together, the PACKET structure was developed. This structure holds the received packet, results of some analysis tests, and both decoded and identified protocols. Protocols are stored within the structure so that any protocol only needs to be decoded once.

A decoded protocol is essentially a tuple containing a pointer to the start of the protocol header, a pointer to the start of the data contained in the protocol, the length of the header and data, the name of the protocol, and the name of the next protocol (if given by the protocol header - e.g. IP has a protocol field which specifies what the next protocol is). The naming of protocols poses an interesting problem, as each protocol must be uniquely named, and yet be easily searched for. The technique used, and justification for it, is given below. Additionally, the results of analysis may be appended to the list of protocols in the form of a 'special' protocol. This enables the results from, for example, previous calls to the snort plugin to be used if a second call is made, or if a certain analysis relies on a snort analysis having been performed.

A naive implementation of this system would require a descent down the list of protocols decoded whenever it was desired to determine whether a specific protocol has been decoded. Whilst this would be satisfactory when only a small (3 or less) number of protocols have been decoded, this would potentially be a performance choke-point when larger number of protocols have been decoded, and/or analysis has been performed. Therefore, also stored in the packet structure is a 'map' of which protocols have been decoded. This is described below. Also to facilitate fast access, the most recent 'next protocol' name and protocol data pointer are also stored in the PACKET structure.

4.3 Protocol Identification and Naming

The requirements for protocol naming are that the names be unique, efficiently testable, and conducive to combining together in the form of a 'map' of decoded protocols. Options include using string names, or some form of numeric identifier. Strings have the benefit of being human-readable, and can be unique. However, numeric identifiers are substantially faster to compare than strings, due to their size. Both strings and simple incremented numeric identifier (i.e. each protocol has a different number, each at least 1 different from any other) suffer from not being easily applicable to a protocol map. The only ways of mapping these are some form of indexing into a table, which is memory inefficient and also does not scale to new protocols not defined in the code, or a simple list, which is not conducive to fast testing. Indeed, there would therefore be no reason for the protocol map, as the protocols themselves are stored as a list.

An alternate approach, and the one used is to have numeric identifiers that may be combined together in a reversible operation. Specifically, by allocating each protocol a particular bit-field, they may be combined and separated at will. For example, if IP is binary 00001, and TCP is 00100, we can see that 00101 refers to a TCP/IP packet. Therefore the protocol map is formed by a bitwise OR of all protocols decoded, and tested with a bitwise AND of the desired protocol identifier. This technique suffers from a serious shortcoming due to the limited number of bits in a number. For example, a 'long' is only 32 bits in length and so only 32 unique protocols may be stored. In order to solve this problem, the most significant 2 bits are used as a 'level' identifier, with the remaining 30 bits unique for that layer. These layers relate to the layers of the OSI 7-layer model, together with a 'special' layer used for naming analysis results and similar. Thus the protocol map is several 'long' variables, one for each 'layer'. An example would be an ARP packet on Ethernet. When fully decoded, the protocol map will contain Ethernet (0x00000001) in layer 2, and ARP (0x00020000) in layer 3. Outside of the map, the ARP packet would be identified as being protocol 0x020020000 (i.e. ARP bitwise OR'ed with LAYER3 (0x02000000)).

This approach is fast to test and process, but still suffers from the serious shortcoming of having support for only a limited number of protocols, specifically 30 at each layer. An additional benefit of using a numeric identifier is that the number can relate to the order that protocols may occur. For example, when testing for the existence of an Ethernet protocol, if the IP protocol is shown as having been decoded, we may be sure that some other layer 2 protocol, for example ATM or AppleTalk, is in use, and thus there is no requirement for attempting to find the protocol by decoding the packet until no more protocols are left in order to search for Ethernet. Another serious shortcoming of this scheme however is that no support for transparently 'tunnelled' protocols exists. For example, the layer 2 PPP protocol can be transmitted over an IP connection. However, as IP is of a higher layer than PPP, it is impossible for the system using the naming herein discussed, to find and decode the PPP protocol. This can be solved by removing the efficiency improving mechanism of stopping searching for a protocol once one with a higher 'identifier' is encountered. However, this would not deal with the case of tunnelling IP over IP. As these are both IP, they would have identical numeric identifiers, and so it proves impossible to access the tunnelled IP packet.

Due to the several shortcomings of this naming scheme, it is likely that the naming and mapping scheme will need significant modification. In recognition of this, and also from good coding practices, effort has been made to facilitate any changes by minimising the reliance on naming, and encapsulating access to naming inside a small number of functions.

5 Plugins

Each of the currently implemented plugins is dealt with below. For each one, a brief description of its purpose, the action it performs and/or functionality it adds, and how it works, is given.

5.1 Input Plugins

Three plugins have been constructed which implement the Input plugin interface. Two of these are for the reception of network traffic. The last is to receive reassembled/defragmented packets.

5.1.1 PCAP - Raw network device access

The PCAP plugin provides cross-platform support for reading live network traffic from network devices, and logged traffic from files. It does this by calling into the libpcap packet capture library. This allows for packets to be received on an interface without relying on any networking support from the operating system, including both promiscuous and 'normal' access. In order to run this, the libpcap library needs to be installed

5.1.2 IPQ - Iptables/Netfilter

One limitation of the PCAP technique is the requirement of having the library installed. Some alternative for when this is not the case is therefore desirable. Additionally, it may be desirable to install the host on an already existing firewall, and initially only filter and react to only a small subset of network traffic. For example, when installed in a production environment it would be desirable to phase the system in to ensure no unnecessary downtime. The IPQ input plugin provides this functionality by interfacing with the libipq library to obtain packets from netfilter/iptables. The administrator must add rules to netfilter via iptables in order to send desired packets to the userspace (i.e. libipq) target. Once this is configured, this plugin is used identically to the other input plugins.

There are two important advantages of this plugin over the PCAP one. Firstly, it enables Rubicon to be gradually phased into a production (i.e. live) system without any negative affect on operations. Secondly, rules purely relying on IP, TCP, UDP and ICMP headers can be implemented using netfilter, with Rubicon only used when more complex requirements exist. This is of benefit as netfilter is highly efficient code, implemented inside the kernel, and so is a great deal more efficient than the equivalent test being performed by Rubicon. IPQ does suffer an overhead when receiving packets due to the requirement that each transferred packet must be copied from kernel to user space. The key disadvantages are that netfilter is only available on some operating systems, and so this method is less cross-platform than PCAP, and that the host system must have an IP protocol stack to be able to install netfilter. This also limits the protocols that may be handled, namely to those listed earlier.

5.1.3 DEFRAG - Packet defragmentation.

The defrag plugin performs packet defragmentation. Certain protocols allow data to be fragmented over several packets, and then reassembled when the packets reach their destination. This is useful as different protocols have different maximum packet sizes, and so data may be fragmented for transmission over protocols with smaller MTU (Maximum Transmission Unit) sizes. However, attackers may purposefully fragment traffic for various reasons. By spanning an exploit over several packets, they may attempt to hide it from pattern matching IDSs. Additionally, as described in section 2.6 exploits may be hidden in overlapping fragments. Finally, denial of service attacks can be performed through sending confusing fragments to some systems.

It is therefore necessary to be able to both detect, and reassemble, fragmented data. Defragmentation (i.e. packet reassembly) is currently implemented for fragments where the underlying protocol is Ethernet or IP. In other words, if an IP packet is fragmented, but the link layer protocol is Ethernet, or if a TCP or similar packet is fragmented and the transport layer protocol is IP, then defragmentation can be performed. Defrag works in two stages. Firstly there is an analysis plugin. pluginAnalysis is called for a given packet to check whether it is a fragment. If it is then the packet is stored and no further handling of the packet should occur. As new fragments are received, they are appended to the received fragment chain until a complete packet has been received. At this stage the analysis function call returns indicating that a newly reassembled packet has been formed. The defrag plugin is then used as a packet source (i.e. input plugin) next time a packet is to be received, in which case it returns the oldest non handled, reassembled packet.

5.2 Analysis Plugins

There are two classes of analysis plugin - protocol decoding and testing, and other analysis, such as pattern-matching IDS.

5.2.1 Protocol Decoders

One of the design aims of Rubicon was that it be extensible, and able to handle any protocol if the correct extensions were made. Therefore, no specific protocols were hard-coded into the system; instead reliance was placed on plugins to 'decode' protocols. As described in section 4.3, each protocol is assigned a unique identifier, which does not have to be hard coded into the program, but must be assigned centrally to ensure uniqueness. During the registration of protocol decoding plugins, the plugin specifies which protocols it can decode by their unique identifiers.

The decoding process involves finding the desired protocol by iteratively decoding each protocol encountered based on the next-protocol specified by the previous protocol, and/or analysis of packet data. A pointer to the start of the protocol header, start of the protocol data, and data length are then appended to the PACKET structure as discussed above.

Many common protocols are however not solely formed of fixed-size headers. Some have footers, and some have optional headers, of which both IP and TCP are options. It is desirous that these optional fields also be 'testable' during the policy querying process. To perform this a call into the requisite protocol decoding analysis plugin is made. This takes the form of a pluginAnalyse call, with the analysis type set to Protocol testing. Currently, decoders for Ethernet, IP, TCP, UDP and ICMP have been produced; the latter three in the "IP-TCP-UDP-ICMP" plugin, and the former in the "ETH" plugin.

A possible extension of the functionality of protocol decoding plugins is explained under 'further work' in section 8.4.

5.2.2 SNORT - Pattern matching intrusion detection

The primary aim of Rubicon is to integrate IDS and firewall technologies, and so this IDS analysis plugin provides core functionality. The purpose of this plugin is to take a packet and perform pattern-matching analysis for a specified subset of patterns. If a match occurs, information on the match is appended to the PACKET structure, and a string describing the match type is returned (i.e. NONE, EXPLOIT, DOS (denial of service), etc). This returned match type can be used to compare against desired results in the policy, and the information stored in the structure can be used in any further analysis or output.

Rather than implementing a signature/pattern matching analysis from scratch it was decided to use the code developed for the snort IDS, with some modification. This sped up the development process and also allowed the signatures developed for the snort IDS to be easily used without modification inside Rubicon.

Signatures are loaded into the SNORT plugin during initialisation, and grouped together by the headers they depend on, for example all the signatures that operate on http traffic and so have protocol tests such that the SNORT analysis is called on any traffic to the web server on port 80. This group is allocated an identifier, which is then used when calling the plugin to perform the analysis.

5.2.3 DEFRAG - Packet defragmentation.

The analysis aspect of defrag is covered in 5.1.3 above.

5.2.4 MASQ - IP Address Masquerading

Address masquerading is described briefly in section 2.6. This is a very commonly used feature of many firewalls as it allows multiple hosts with private IP addresses to share a single world routable IP address, as well as providing some additional security. This has been implemented based on the masquerading support of netfilter, and previous versions of Linux firewalls.

5.2.5 COUNTER - Simple variable support

Some firewalls provide support for user-defined counters. This can prove very useful when, for example, only a certain number of outbound connections, or inbound UDP packets etc, are allowed within a specific time period. This is useful for both Quality of Service and security reasons. For example, limiting the numbers of certain types of packet removes one Denial of Service technique.

The ‘counter’ plugin provides this functionality. During initialisation, any number of unique counter variables may be defined. For each variable a default value can be given, a timeout period after which the variable is reset to this default may be specified, as well as a timeout period after which a specific operation occurs. For example, a variable can be set with a default value of 0, which is incremented every time a packet matching certain criteria is received, but decrements every 10 seconds, and resets to 0 after 1 minute with no activity. All variables are stored internally as unsigned long and so have values in the range 0 to 2^32-1.

Provided for analysis are certain arithmetic tests and operators, namely the ==, <, <=, >, >=, and != tests and the +, -, and = operations. This provides sufficient flexibility for keeping relatively large amounts of state information.

5.3 Output Plugins

There are two classes of output plugin, logging and network transmission plugins. The logging plugins provide the functionality normally offered by NIDS, namely logging to file and syslog, as well as the new form of IDS specific logging offered by IDMEF/IDXP. The network transmission plugins offer the functionality provided by firewalls to forward, reject and drop data, as well as packet modification/mangling.

5.3.1 NET-CORE - Network - Forwarding, rejecting, dropping

This is all required core functionality. Forwarding of network packets requires the placing of a packet received on one interface onto another interface for transmission onwards towards its destination. On most firewalls, were the operating system handles the reception and transmission, the operating system handles the lower (i.e. Ethernet and below) protocols. Thus the output packet has its Ethernet source and destination addresses modified. It is often desirous to place a firewall, and hence Rubicon, transparently. This is done by not modifying the lower level packet headers, and by forwarding rather than responding to lower level mapping protocols (such as Arp). Rejection is performed by sending an ICMP unreachable packet to the source of the rejected packet. Dropping is simply performed by not performing any action.

5.3.2 NET-MANGLE - Network - Packet mangling

[Malan et al(2000,2001)], as discussed in the background section 2.6, propose that rigid enforcement of network protocols can improve security. Whilst the system could do this by dropping or rejecting traffic, which does not sufficiently closely follow the respective protocols, this would lead to widespread problems due to the large number of incorrect and incomplete implementations of protocols that exist. Instead, it would prove sufficiently advantageous, where possible, to transparently enforce compliance through the modification of traffic as it passes through. For example, fragmented packets may be modified so that they do not overlap. Another possible reason for mangling/modifying packets is Quality of Service reasons. Specifically the Type Of Service (TOS) field in IP packets may be modified dependent on the type and quantity of data carried. This field is used by many network routing devices to provide differentiated services.

A security related use of packet modification is when honey pots are being implemented. A honeypot is a computer system that is placed in such a way that hackers attempt to break into it. The behaviour of the hacker when breaking into this heavily logged system can then be observed both during and after any attack to gain insight into methodologies, purposes and technologies used by hackers. One problem with allowing hackers to break into a system is that these hackers may then use your host to break into another system, with potential liability concerns therefore being raised. Whilst all outbound traffic could be blocked, this may warn hackers that something bizarre is happening to the host, possibly 'spooking' them. Instead, it would be useful to 'break' any outbound attacks so that they are not successful, but to do so in such a way as to not warn the hackers. One way of performing this is to modify all the data in any outgoing packet that is flagged as being an exploit by the IDS analysis plugin. Specifically, every byte of the data may be set to null, or the packet may be trimmed to a smaller length, thus defeating most buffer overflow style attacks.

The NET-MANGLE plugin provides support for all these techniques, dependent on protocol mangling type. As currently implemented, it can modify the data for IP, TCP, UDP and ICMP traffic, and recompute checksums if desired. It can resize IP and UDP packets, however cannot resize TCP packets due to their usage of sequence numbers to guarantee ordering and acknowledged reception, and cannot resize ICMP packets due to their fixed length (dependent on message).

5.3.3 LOG - File/stream logging

One of the core requirements of any security related software is full logging support, which this plugin provides to Rubicon. This plugin allows text and binary logging to be performed to a number of different files, which are specified when the output function is called. This allows, for example, interesting events to be logged in binary format for later investigation by administrators, and text logging to different (possibly remote) files dependent on the severity etc of the threat.

The plugin has been implemented to be able to output any non-optional field of the Ethernet, IP, TCP, UDP and ICMP protocols (if present in the packet). Discussion of possible extensions to this is given in section 8.4.

The output text is specified in a similar style to printf, allowing great flexibility of output format. Other data, such as headers for unknown protocols, or the packet data, can also be displayed. The format for this is slightly cryptic, and requires knowledge of protocol numbers and specifics of the protocol. The format specifier is: %proto-offset-Len-type: -

- The proto is the numeric identifier of the protocol, for example 0x01000004 for Ethernet.
- The offset is the positive integer number of bytes/octets into the header to start the output from. Note that the packet will always be in network order.
- The len is the number of octets to display, 0 if a specific data type is specified in type or f if the entire header.
- The type is how to display the data. If len is 0, this may be either l or s representing a long or short respectively. If type is one of these values, then a hexadecimal, host-ordered, value is displayed. Otherwise, type can be either h or t, representing hexadecimal or text respectively.

Examples for this latter specifier include: -

“%0x01000004-0-6-h” - To display the destination MAC of an Ethernet packet in hex
“%0x02000001-1-0-s” - To display the host ordered data length of an IP packet in hex.
“%0x10000001-0-f-t” - To display the payload of a fully decoded packet as a string.

This specifier therefore supports protocols that are not built into this plugin, improving extensibility. However, it does not support optional headers for these, or indeed currently any, protocol.

5.3.4 SYSLOG - Unix system logging

In addition to logging to text files, the Unix operating systems support a system logger daemon, syslog. Syslog is accessed by specifying a facility (i.e. the source of the message) and a priority. A string is then sent to the system logger, which determines what to do with it, such as store it, email it out, etc, dependent on the facility and priority and a separate configuration file. The SYSLOG plugin allows output to the syslog daemon. Any facility and priority may be specified. The format of the message is the same as that for LOG, above.

5.3.5 IDXP - Libidxp - Intrusion detection transfer protocol

The IDXP protocol and IDMEF message format were briefly discussed in section 2.6. These represent an attempt by the IETF (Internet Engineering Task Force) to specify a common, non-vendor specific, protocol for the interchange of Intrusion Detection messages and logs. An implementation of this exists in the form of libidxp, which this plugin accesses. This plugin currently works by outputting the result of a snort analysis, and specifically the data attached to the PACKET structure. The remote target is specified during initialisation. The IDXP protocol relies on TCP/IP and as such may only be used on systems with operating system support for TCP/IP, and on devices with a specified IP address. It is envisaged that this plugin will therefore operate on the same network interface as the policy reception thread.

6 Policy Process

6.1 High level language

A high-level policy language was desired in order to configure rules for Rubicon for several reasons. First, high level languages are greatly abstracted over lower level ones allowing for more concise policies to be written, more easily, and with less of an chance of errors than is the case when writing in lower level languages. Secondly, many higher-level languages have been developed with tools to aid policy creation, simulation and analysis. If one could be found which offers this then significant development time improvements would be made.

Development of high-level policies is a skilled process requiring a great deal of testing and work in order to successfully create a sufficiently powerful yet abstract language. This can be seen by the large amounts of work being performed on the subject. It was therefore imperative that a language well suited to network security policies be identified. Unfortunately, each of the languages considered suffered from shortcomings for the task desired of them. The IETF PCIM model apparently suffers from several problems, and does not appear to have any implementations yet. Very few of the languages discussed in 2.7 were found to be natively suitable and have sufficient pre-written software. The Ponder language had the advantage of having an well-developed software suite, but was primarily aimed at host-based policies using Role-based access controls. Although the language is extensible, it was decided that as the high level language was of a relatively low priority, and extending Ponder would require considerable time and effort, focus would instead be placed on developing a human-parsable intermediate language, reserving the higher level policy language for further study at a later date.

6.2 Intermediate language

The purpose of the intermediate level language is two-fold. Firstly, the language is to be platform-independent, allowing it to be translated into the lowest level representation on the Rubicon host irrespective of the platform Rubicon is on, thus avoiding compatibility uses from, for example, byte ordering. The second purpose is that a human readable and if necessary, human writable, configuration may be produced as it was decided to postpone development of the high level policy.

The XML data type definition decided upon is included in section 10. It is essentially a 1:1 mapping of the structure for the 'local representation' described below, allowing fast and logical translation between intermediate and low level representations. Some additional elements are defined for plugin initialisation, and to simplify the construction of larger policies.

An example pseudo policy is given in section 6.4.2 below, and an XML representation of the policy is also given there. For information on how to construct the XML policy, the reader is referred to the user manual in section 10.

6.3 Transmission and Remote Storage

As mentioned in (8.1) and (3.2.5) above, substantial research has been performed into policies. One common thread between many of the implementations is that they utilize directory services for the storage and transmission of policies. Specific examples include Ponder, and some work on the IETF PCIM, both of which can use the LDAP directory service. Due to this support, and the potential ease of use it was decided to support the retrieval from LDAP directories, in addition to local configuration file setup.

LDAP, as a directory service, is a pull technology, i.e. the client accesses the LDAP server whenever it wishes to download data. This alone is not satisfactory for Rubicon as we wish to be able to order the system to reload as and when desired, often within a relatively short timeframe. In order to do this, some mechanism for commanding the system to refresh its policy from the directory server was required.

There are three methods for performing this: one in-band and two out-of-band. The in-band method is that Rubicon is commanded to look for certain patterns of packets arriving on one of it’s sensor interfaces, and then treating these in-band packets as commands. This has inherent security problems, but does benefit from not requiring an additional ‘command’ network interface. The out-of-band methods rely on the reception of either TCP or UDP traffic on a separate administrative network interface. This has the benefit of separating control from data and thus improving both performance and security. It does however increase the requirements for the number of network devices on the computer as an additional, separate network device will be required for the administrative access.

On balance, it was decided that the out-of-band method was safest, and easiest, and so it was chosen. It was further decided that the transmission guarantees offered by TCP over UDP meant that a TCP solution would be used.

The OpenLDAP [OPENLDAP] project offers a GNU licensed library implementing the LDAP API, and so it was decided to use this rather than starting from scratch. Other possible sources for LDAP libraries include Mozilla [MOZILLA] and Novell [NOVELL].

The policy thread, when waiting for commands from the server, opens a listening socket and blocks on accept. When the server connects, the following transaction takes place: -

Rubicon Host		**Server**
Blocks on accept()		
	<-------------	Opens connection to Rubicon
Sends “HELO myid”	------------->	
	<-------------	Sends “HELO myid, srvid, command”
Sends “ACK”	------------->	
Performs action…..		
Sends “RESP myid, srvid, command = res”	------------->	
	<-------------	Sends another command or “BYE”
If “BYE”, close connection		If “BYE”, close connection

The myid variable is a unique identifier for the Rubicon host. The srvid variable is a unique identifier for the server. The command is currently only “DIE” or “GET location” which correspond to exit, or a command to get a new policy. If the latter, an attempt is made to retrieve the policy from the LDAP server. The response to the server after this may be one of “OK”, “NO_SERVER”, “NO_POLICY”, “PARSE_ERR”, “ERR”, corresponding to the new policy being loaded successfully, the LDAP server not being found, the policy not being found on the server, an error parsing the policy, and an unknown error, respectively.

6.4 Local Representation and Access

6.4.1 Description

Some consideration of the representation of policy at the firewall has been made. One possibility would be the use of the Java environment already designed for Ponder, and modified to run on the firewall host. This has the benefit of shortening certain aspects of the development cycle, such as the compilation and policy state management, but unfortunately has a number of possible flaws due to it’s implementation in Java. Firstly, as the project software is being written in C, the java and C would need to be interfaced at some point – a non-trivial exercise due to the complexity of interfacing through the sandbox, the different paradigms and binary format of the code. Secondly, the requirement for a java environment on the firewall host will increase the difficulty in bug-tracking and fixing. Finally, the substantial performance hit that will occur due to the use of Java in a highly utilized section of code will hinder the overall throughput.

Another possibility would be to use some form of low-level compiled format to store and access the data. This would almost certainly prove more efficient than Java at runtime especially if the optimizations mentioned by Lakshman and Stiliadis(1998) or Coit et al (2001) be used, but would introduce a significant delay in the system development as the format would have to be designed, encoded and tested, and the Ponder software would require modification to compile into this low-level format.

It was finally decided that a low-level implementation be used due to the complexities and performance hit that would occur if integrating C with Java.

The local representation of a policy was essentially a tree of structures specifying the tests to perform, and the outputs to be made based on the test results. A policy always begins with a Policy Head which points to the top of the tree, and a default output, as shown in figure 6. Following this is a linked list of protocols, in ascending ‘value’, where ‘value’ represents the position of a protocol in the OSI Seven Layer model, such that for example IP follows Ethernet and ATM, and is followed by TCP and UDP. When querying, the process will determine whether the packet contains a given header and if so proceeds down that protocols linked list of ‘matches’. These matches are what has been termed here as protocol tests. These tests are the simple comparisons one may make on a protocol header, such as checking whether the packet falls in a certain subnet.

Descending from each ‘match’ is a linked list of ‘outputs’. These outputs can be any output plugin, with parameters, such as logging, packet forwarding and so on. They can additionally be null, but point to an additional test to perform. These tests may be the aforementioned protocol tests, potentially checking for other protocols than the ‘protocol’ chain the test falls in, or they may be analytic tests. Analytic tests require calling into an analysis plugin, such as an Intrusion Detection engine, with a plugin-dependent set of possible results. Following each test is a linked list of possible ‘results’, each of which points to a linked list of ‘outputs’.

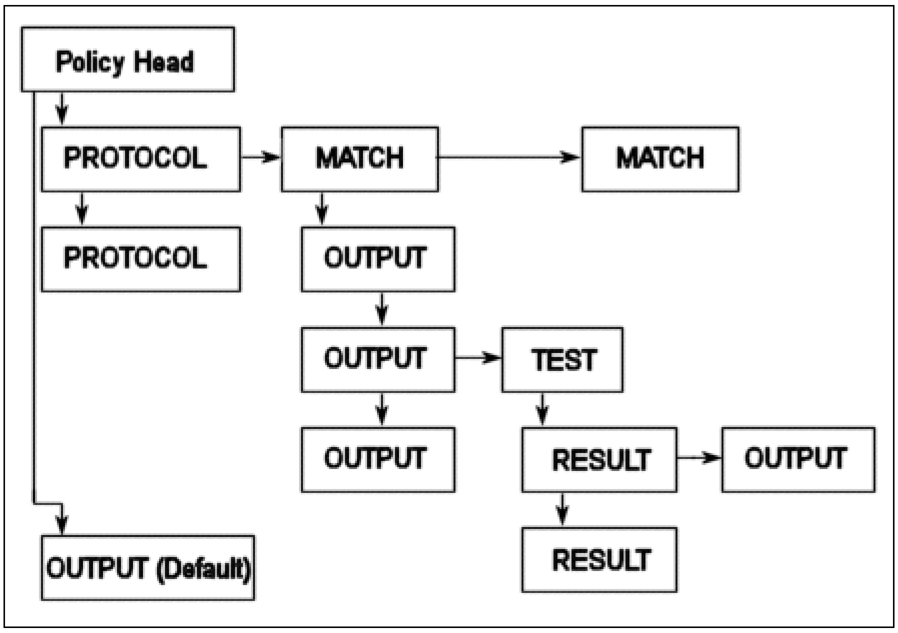

Figure 6 - Structure of local policy representations.

When querying, the querying process descends the linked list, effectively performing a depth-first search. As tests are encountered, they are performed and the results compared to the desired results, with the output linked list of the matching result (if any) then being descended. Querying ceases immediately upon an EOF being reached. Until that point, or the end of the search, *all* outputs encountered are performed and so care must be taken when designing the structure. If the end of the policy is reached before an EOF flag occurs, the default output is called.

Packet headers are decoded as necessary by calls into the relevant decoding plugins. The algorithm for this is as follows: -

- Each protocol is allocated a unique identifier such that lower level protocols have lower identifiers.
- The 'protocols' linked from the policy head are in ascending order as the list descends.
- Each packet has a list of which protocols have thus far been found.
- As the process descends the 'protocol' linked list, the identifier of the protocol is compared to that of the protocols identified in the packet.
- If there is a match, that protocols match linked list is then descended into.
- If the protocol id is higher than the highest identified protocol in the packet, and the packet appears to be able to be further decoded, then this decoding occurs until either a match is found, no more protocols can be decoded, or the highest protocol in the packet is higher than the current 'protocol'.

This organisation in local memory has the advantages of being easily understandable and logical, being able to handle complex rulesets and being able to be quickly parsed. It does however trade off efficiency in terms of processing speed with efficiency in terms of space. For larger policies the local representation does take up a substantial amount of space due to the way that branches are never combined even when identical. For example, two separate rules may have identical results, which could potentially be combined, and yet due to the tree-like nature of the structure, these branches are not combined. This is primarily for simplicity reasons, and could be modified if it was found desirable.

6.4.2 Example Pseudo-policy

In order to demonstrate this process, a small example is shown in figure 7. We have a simple network with Rubicon situated as shown. We wish the following rules to be in place on outgoing traffic: -

1) Only the Ethernet MAC address 00:11:22:33:44:55 is allowed to transmit outbound.
2) Only the IP address 1.2.3.4 is allowed to transmit outbound (this is the same machine as in (1)).
3) Any outbound traffic should have the snort plugin run on it
    a. If the result is "OK" then just forward it
    b. If the result contains "WARNING" then forward and log
    c. If the result contains "DANGER" then log it and drop it.
4) If the traffic is encrypted via SSL, decrypt it before running snort.
5) Default is to drop

The first stage in configuring the system to use these rules is to create the required XML description, as shown below.

```
<Rubicon>
<policy>
       <default>
              <output name=”NET-DROP” eof=”yes”>
       </defaut>
       <protocol proto=”ETH” field=”src” val=”001122334455” negate=”yes”>
              <output name=”NET-DROP” args=”” eof=”yes”/>
       </protocol>
       <protocol proto=”IP” field=”src” val=”1.2.3.4”>
              <protocol proto=”0x20000001”>  <!—0x2000001 is the identifier for SSL -- >
                     <!—no action required as SSL will have been decoded if present -- >
              </protocol>
              <analysis name=”snort” args=”all_rules”>
                     <analysisResult match=”OK”>
                            <output name=”NET-FORWARD” eof=”yes”>
                     </analysisResult>
                     <analysisResult match=”WARNING”>
                            <output name=”LOG” args=”LOGFILE: Warning: Packet=%ip.all:”>
                            <output name=”NET-FORWARD” eof=”yes”>
                     </analysisResult>
                     <analysisResult match=”DANGER”>
                            <output name=”LOG” args=”LOGFILE: Danger: Packet=%ip.all:”>
                            <output name=”NET-DROP” eof=”yes”>
                     </analysisResult>
              </analysis>
       </protocol>
</policy>
</rubicon>
```

These rules, when encoded into a local representation, could create a structure like that in figure 7.

Note how similar the XML configuration file is to the local representation.

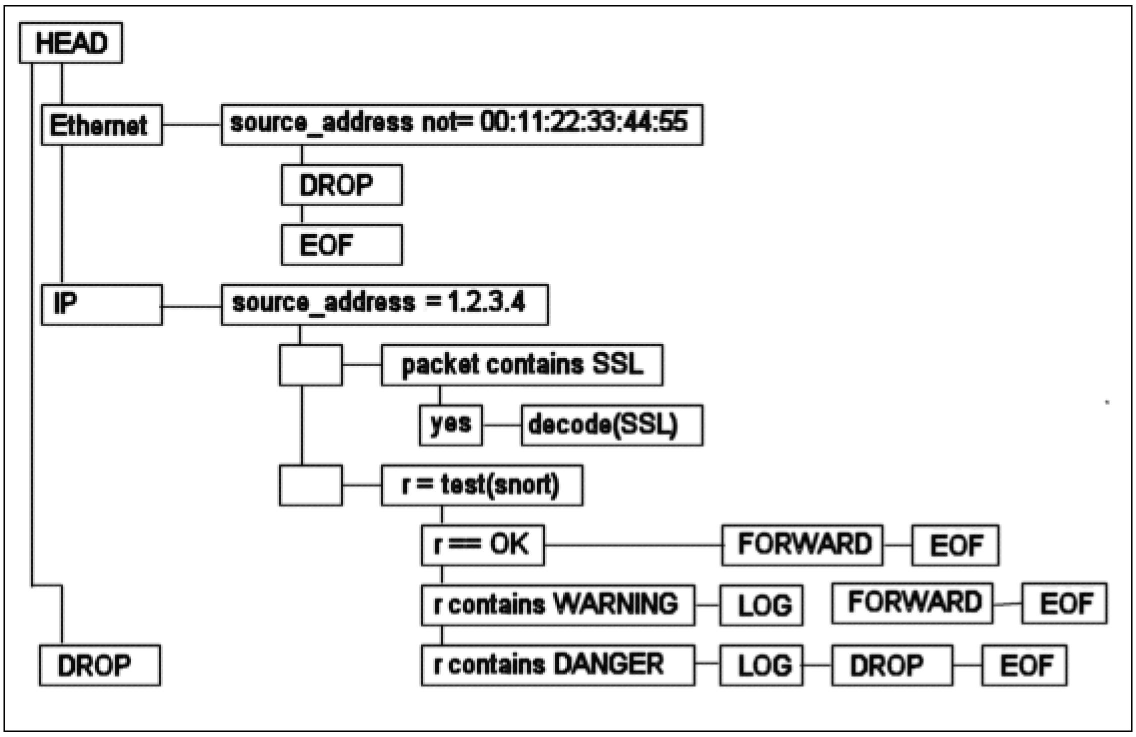

Figure 7 - Local representation for example ruleset.

Now let us follow the querying of this policy for a sample of packets. Firstly, let us deal with an ARP packet from the MAC address 00:11:22:33:44:55.

i) The packet arrives and has had no headers yet decoded.
ii) The query starts at HEAD and descends to Ethernet. A check of the packet shows no header of type Ethernet, but the packet has headers yet to decode. The lowest protocol is decoded and found to be Ethernet.
iii) Therefore, the Ethernet match linked list is descended. The source address is *not* not-equal to 00:11:22:33:44:55 (i.e. it is equal), and so no match is found.
iv) The query returns to the ‘protocols’ stage and descends downwards.
v) The IP protocol is encountered. A check of the packet shows no header of type IP, but the packet has headers yet to decode. The next protocol header is decoded and found to be Arp.
vi) The Arp protocol is lower than the IP protocol, and so we should continue decoding. However, no other protocols are shown for the packet, hence the search ends.
vii) As no EOF was encountered, the default behavior occurs and the packet is dropped.

Next let us consider an IP packet with IP source address of 1.2.3.4 and MAC address of 66:66:66:66:66:66.

i) The packet arrives and has had no headers yet decoded.
ii) The query starts at HEAD and descends to Ethernet. A check of the packet shows no header of type Ethernet, but the packet has headers yet to decode. The lowest protocol is decoded and found to be Ethernet.
iii) Therefore, the Ethernet match linked list is descended. The source address is not equal to 00:11:22:33:44:55, and so the output chain for this protocol ‘match’ is descended.
iv) The packet is dropped, and an EOF encountered. The processing of the packet immediately finishes as an EOF has been reached.

Now let us consider the somewhat more complex situation of an IP packet from 1.2.3.4 with MAC 00:11:22:33:44:55 which is a packet from a secure http session.

i) The packet arrives and has had no headers yet decoded.

ii) The query starts at HEAD and descends to Ethernet. A check of the packet shows no header of type Ethernet, but the packet has headers yet to decode. The lowest protocol is decoded and found to be Ethernet.

iii) Therefore, the Ethernet match linked list is descended. The source address is *not* not-equal to 00:11:22:33:44:55 (i.e. it is equal), and so no match is found.

iv) The query returns to the 'protocols' stage and descends downwards.

v) The IP protocol is encountered. A check of the packet shows no header of type IP, but the packet has headers yet to decode. The next protocol header is decoded and found to be IP.

vi) The IP match list is therefore descended. The source address is equal to 1.2.3.4 and so a match is found and the output list descended.

vii) The first output is empty, so now output is performed. However, there is a test for 'packet contains SSL', which is actually an Analysis test calling the SSL plugin. Due to the depth-first nature of the system, this test is performed before further descent along the output list.

viii) The SSL plugin is called, which returns 'yes' i.e. the packet does contain SSL traffic. The results list is descended and a match for this return value found.

ix) The output for this result match is to call the decode plugin to decode the SSL header (i.e. decrypt the packet). As no EOF is encountered, the process returns to the empty output mentioned in (vii).

x) The output is further descended and another empty output encountered. This requests an analysis test with the snort plugin, which is forthwith performed.

xi) Snort returns ERROR. The results lists is descended but no match found.

xii) As the end of the policy has now been encountered with no EOF, the default behavior occurs.

Finally, let us consider an interesting situation that shows why the policy must be accurate and complete. In this example, the Rubicon interface receives an AppleTalk packet, containing traffic with a source IP address of 1.2.3.4.

i) The packet arrives and has had no headers yet decoded.

ii) The query starts at HEAD and descends to Ethernet. A check of the packet shows no header of type Ethernet, but the packet has headers yet to decode. The lowest protocol is decoded and found to be AppleTalk. The AppleTalk identifier is lower than Ethernet (although this example could equally apply were the identifier higher), and so the packet is further decoded.

iii) The newly decoded protocol is found to be IP, which has an identifier larger than Ethernet. Therefore, the Ethernet protocol has not been encountered, and the process continues to descend the 'protocols' list.

iv) The IP protocol is encountered in the 'protocols' list, and a check finds that the IP protocol has been found in the packet.

v) Therefore, the IP protocol 'match' list is descended. After further testing the packet is then forwarded.

In this last example, we can see that a packet was forwarded even though it may not have been desirous to do so. This results from a potential ambiguity in the policy, were the possible existence of non-Ethernet packets has not been considered. Therefore, policies must be complete – checking for the existence of a specific protocol if that protocol must exist. This example does however demonstrate one positive aspect of Rubicon; the transparent way it handled the different link-layer protocols.

6.4.3 API

The programming interface to the low-level implementation of the policies is relatively simple. Construction is performed by effectively building the tree from the top down, using a number of construction functions. In build order, these functions are: -

- newPolicy() – Creates a new POLICY_HEAD.
- addProtoToHead() – Adds a new 'protocol' into the protocol linked list hanging from the policy head. Used for the very first protocol to be entered.
- addProtoToChain() – Adds a new 'protocol' into the linked list. Calls addProtoToHead() if this is the first protocol in the list.

- addMatchToProto() – Add a match to the match chain for the given protocol.
- addAnalysisTestToMatch() – Appends an analysis-style test to the outputs for a protocol match.
- addProtoTestToMatch() – Appends a protocol-style test to the outputs for a protocol match.
- addOutputToMatch() – Appends an output to the output linked list for a protocol match.
- addResultToTest() – Appends a result to the result linked list for a given test of either type.
- addOutputToResult() – Appends an output to the output linked list for a test result.
- addAnalysisTestToResult() – Appends an analysis-style test to the output linked list for a test result.
- addProtoTestToResult() – Appends a protocol-style test to the output linked list for a test result.

When querying, the HandlePacket() function is called by the main program loop. This descends into the policy calling in order: -

- descendProtoMatch() – descend down the matches for a given protocol. This calls descendOutput() as required.
- descendOutput() – descends down an output linked list, performing outputs as required. Calls performTest() whenever a test is encountered.
- performTest() – Performs the specified test, descends down the results linked list to find a match, and calls descendOutput() on the result output linked list if a match is found.

7 Evaluation

7.1 Requirements Testing

One reason for specifying the design requirements in section 3.2 is to formalise the evaluation of the system. Specifically, each requirement is considered, and a decision is given specifying to what extent the requirement has been satisfied, together with supporting evidence.

Requirement 1

"The system must be extensible without modification of core source code."

Evidence to support the claim of fully this meeting requirement can be seen by examination of the system design. Specifically, the use of dynamically loaded plugins separates the core source code from the plugin source code. Additionally, through the extensibility inherent in the method used to specify and handle protocols, it can be seen that source code will not have to be modified even when handling new protocols.

Requirement 2

"The system must provide a basic subset of normal firewall rule targets, specifically FORWARD, DROP and REJECT."

The evidence for this is in the form of the plugin specified in section (5.3.1) above. This plugin provides the desired functionality.

Requirement 3

"Routing decisions must be able to be made from any field in the Ethernet, Ipv4, TCP, UDP or ICMPv6 protocols."

The plugins specified in section (7.2.1), when combined with the policy querying as described in (8.4) show that this requirement is satisfied. Specifically, any field in the header of any of these protocols is trivially testable after the protocol has been found in the packet, as is performed via the pluginAnalyse function call. Additionally, the same function call can be performed with differing arguments in order to inspect optional protocol fields. All the options known to the author after inspection of the relevant documents, notably the IP, TCP, UDP, ICMP Requests for Comment (RFC) [RFC791; RFC792; RFC793; RFC768], are testable.

Requirement 4

"Routing decisions must be able to be made based on the results of a traditional intrusion detection system."

Through the construction of the snort-based pattern-matching intrusion detection plugin, the system has been enabled for routing based on an IDS. Therefore, this requirement has been satisfied.

Requirement 5

"The system must operate in a stable way on a specific hardware/OS platform."

The system has been developed on an x86 architecture computer, running the Mandrake distribution of the Linux operating system. During the stress and other quantitative tests a small memory leak was noticed, which does negatively affect stability after the reception of several million packets. Apart from this the system behaviours in a very stable manner.

Requirement 6

"The system must be capable of being fully transparent at all network layers above the lowest, hardware specific, protocols (i.e. from Ethernet/ATM upwards)."

As can be seen in section (7.3.1) where the implementation of the plugin to forward network traffic, support is included for specification of all protocol settings from Ethernet upwards. By setting the output packet headers to be identical to the input packet headers full transparency can be enforced. As this behaviour can be specified in the rules for the system, this requirement has therefore been satisfied.

Requirement 7

"The system must be controlled by an easily understood human-readable language."

The use of XML as the intermediate policy language, as described in section (8.2), satisfies this requirement. However, whilst this requirement is technically satisfied, the incomplete nature of development on the high level language does not ensure full satisfaction.

Requirement 8

"The system must be able to operate on raw network devices, i.e. no Operating System network support/network stack."

This requirement is satisfied through the use of the libpcap based plugin as described in section 7.1.1. As mentioned, libpcap accesses the raw network device, thus requiring no network stack to be installed.

Requirement 9

"The system itself must be secure to common network attacks."

The system itself opens a listening port for reception of commands. There is currently no encryption, or validation of identity and so the system is not currently secure. This was due to time issues, and is an area requiring further development. Apart from this weakness, the system is relatively secure. Buffer length validation has been an especial concern throughout the development process and so the system should be secure from most buffer-overflow style attacks. However, further testing is required to confirm this. Additionally, several external libraries are called, weaknesses in any of which could have knock-on effects on the system, and so testing and source-code validation of these is also required.

Requirement 10

"The system must only allow packets through when permission is given by the configuration."

As can be seen from the design of the system, packets can only pass through the Rubicon system if the system actively calls the forwarding output plugin. Testing and source code review have confirmed that as far as is known, it is not possible to incorrectly call the wrong plugin, and so this requirement is satisfied for the system. Consideration must however be made of the host operating system. This must also be configured not to forward network traffic

Requirement 11

"The system must have an operating performance suitable for medium bandwidth networks, i.e. 5MB+ throughput for a basic configuration."

The system was run with a relatively simple ruleset on a 10MB network that was then saturated by several hosts up to a maximum bandwidth of approximately 8MB. The system ran satisfactorily at this point when no screen logging was taking place, with no packet loss. However, when a small amount of output was written to the screen for each packet, packets began to be dropped by the PCAP interface. When writing 5 characters to screen per packet at 8MBps bandwidth, it was found that the PCAP plugin, courtesy of the libpcap library, was having to drop approximately 3% of packets.

Requirement 12

"The system should operate in a stable way on many hardware/OS platforms."

As mentioned in section (4.5) the GNU autoconf, automake and libtool tools were used during development. When properly used, these tools guarantee support for a large number of Unix based platforms, as well as some others. The system has successfully been compiled and run on Linux Mandrake with a 2.4.9 kernel, and a 2.2.17 kernel, Redhat 6.1 with a 2.4.9 kernel, and OpenBSD 2.9 (although some problems occurred). Whilst this is not conclusive evidence, as a small sample it does suggest that the system should compile on a number of Linux and Unix platforms, once the required libraries and tools have been installed.

Requirement 13

"The system should be controlled by a high-level policy language."

Unfortunately time did not allow for a high-level policy language to be selected, modified and tested (as required). This is due in part to the relative recency of work on high level policy languages for network security configuration. Whilst the IETF PCIM language and/or Ponder offer possible benefits, it was decided to concentrate on the lower level implementations, thus not allowing enough time for work to be completed on the higher level. The use of XML as the intermediate language does however make follow-up work to perform this a great deal easier. Indeed, this was one of the reasons XML was used, as Ponder has software support for XML output (although not of the format used here), and some work has been performed on mapping between XML and PCIM. This work may however require a revamp of the specific XML Data Type Definition currently used by Rubicon.

Requirement 14

"The system should have an operating performance suitable for medium bandwidth networks when using a complex configuration."

This requirement could not be tested due to the limitations of the equipment available. Specifically, only a 10MB network was available and so it proved impossible to drive the network at above approximately 8MBps. The system was capable of operating at this bandwidth running the majority of the standard snort rules without any packet loss. it must be noted however that this was essentially random data, and the efficiency of operation greatly depends on the topography of the network data.

Requirement 15

"The system should be able to dynamically alter network packet contents between reception and retransmission."

The implementation of the network packet mangling plugin described in section (7.3.2) satisfies this requirement. Specifically the capability of modifying any part of the payload, or any headers provides full satisfaction. It is to be noted however that the recalculation of Checksums causes a significant performance hit if a large number of packets are being mangled. Additionally, due to sequence numbers and similar being present in many protocols it is often not possible to add or remove data from the packet due to the probable loss of synchronisation that would occur.

7.2 Quantitative Analysis

In order to evaluate the software, some quantitative tests were performed to compare the Rubicon software with quantitative requirements.

The first of these was to compare Rubicon with the snort IDS. This was performed by running and timing each program with an identical – everything on – ruleset to parse the packet captures from the SANS 2000 ID'net event. Snort parsed the 45MB of logs in approximately 2 minutes. Rubicon took slightly longer, as reported by GNU time, taking around 2:45 minutes. When a comparison was made between the logs of the two processes, it was determined that they were the same, thus helping show that the implementation of the IDS analysis is satisfactory.

One item that requires fixing with further development is the existence of a small memory leak. This did not prove very important when dealing with only 45MB of data, but in the following test Rubicon became unstable due to its growth in process size.

The second set of quantitative tests to be run were stress tests. Rubicon was set up with first a minimal subset of rules, and later a more complex set. The 4 computers attached to the network were then commanded to produce as much network traffic, with relatively random characteristics, as possible to attempt to saturate the network. This it did, with a relatively stable 8MB being reached as saturation point.

The system ran satisfactorily at this point with both sets of rules when no screen logging was taking place, with no packet loss occurring. However, when a small amount of output was written to the screen for each packet, packets began to be dropped by the PCAP interface. When writing 5 characters to screen per packet at 8MBps bandwidth, it was found that the PCAP plugin, courtesy of the libpcap library, had to drop approximately 3% of packets. Additionally, when the network was at saturation a small memory leak became very apparent, specifically a leak of around 1kB per packet, which very rapidly causes problems when around 10,000 packets are being handled per second.

Further quantitative data would be desirable but unfortunately in order to make statistically relevant live tests would take a great deal of time, and any non-live test will naturally produce skewed results.

8 Conclusions

8.1 Summary

In conclusion, the project has been a qualified success. The system developed fulfils most of the design requirements, and is an improvement on the current state-of-the-art in some areas. It does however suffer from several shortcomings, each of which could be solved through more development work. There are several areas that further work may be performed, both modifying the existing system to make it work more efficiently, and expanding on the current code base.

8.2 Achievements

Successfully fulfilled key requirements
As can be seen from the evaluation (section 7.1) the majority of the key requirements have been fulfilled. Specifically this means that the Rubicon system successfully integrates a substantial subset of the functionality of IDS and firewall technologies. Filtering may be performed through both traditional header contents and IDS style signature analysis. In addition to the outputs and responses normally found in these separate technologies, support for the new IDXP/IDMEF protocol is included, as is an implementation of a novel way to transparently defeat many attackers, through packet mangling.

Unique combination of functionality
The functionality so far implemented in this system represents a step forward in the current state-of-the-art. No open-source or commercial software contains the combination of functionality and programmability offered by Rubicon. Specifically, the ease of extensibility to new protocols, and the new responses offered by packet mangling have not previously been seen. Additional functionality such as that offered by the counters plugin also improves the scope for complex dynamic configurations.

Well suited for Research and Development environments
Rubicon has been aimed at Research and Development markets. It sacrifices efficiency in terms of throughput in favour of ease of extension, and power of programmability. The resulting system, whilst not suited for high bandwidth production environments is very suited for developing with and on. For example, if developing a new IDS algorithm, rather than writing an entire module from scratch the developer can instead write an analysis plugin for Rubicon, then writing the required XML policy rules to access it. This also eases the comparing of such a new module with existing implementations, for example the snort pattern-matching method currently included. One key area of use for Rubicon is in the field of research into the behaviour of 'hackers' in the wild, through the use of honeypots and honeynets. The transparent redirection and exploit nullification through packet mangling facilities enable more advanced honeynets than are currently easily possible.

8.3 Limitations

As with any project, there are always shortcomings and limitations, and Rubicon is no exception. There are several limitations with the system. Almost all can however been removed with additional development work.

Lack of High Level Policy Support
No support for specification of rules through a high-level policy has been developed. Currently, rules must be written in XML which is relatively easy for a technically skilled person, but is prone to error and is by no means ideal.

Policy Process Security
There is currently no identity validation, or encryption involved in any part of the policy acquirement process. As such the system is currently very insecure. This can be easily rectified however with some further development work.

Insufficient Testing
Insufficient stress testing has been performed, especially under very high load. As has been noted in the quantitative analysis during evaluation, the system has been run on a saturated 10MBps network (i.e.

around 7MBps average bandwidth), but more work is required to identify bugs and performance choke points.

Memory leak
It has been noted that under heavy loads the memory size of the Rubicon process grows significantly and rapidly. This is due to an as-yet unidentified memory leak, which must be fixed to ensure long-term process stability.

Barriers to uptake of Rubicon
A non-technical shortcoming of the project as a whole is that, as is the case with most technologies, there are significant barriers in replace with regards uptake of the product, due to current market incumbents. This is a limitation that education and technical superiority will overcome, with time. Exploitation of the product in the form of research and white papers, and presentations at key conferences and security events will help in this.

Efficiency
As has been mentioned, efficiency in terms of throughput has been sacrificed in some areas in favour of speed of development and ease of extensibility. This represents a limitation in terms of bandwidth, but one that may be slightly overcome by further development.

Protocol Identifiers
As discussed in section 4.3, there are several serious shortcomings with the technique currently used for uniquely naming protocols. Notable amongst these are limitations on the number of protocols implementable (30 at each 'layer'), and the lack of support for 'tunneled' protocols.

Protocol Plugins
Whilst the protocol decoding plugins offer significant levels of extensibility, certain functionality is currently missing and could be added by an extension to the interface. Specifically, there is a reliance on hard-coded protocols and duplication of effort in some output plugins, and the XML to local representation policy translator, when mapping named protocol fields (e.g. IP.source_address) to their relevant offset and lengths. This is addressable by adding a public function to protocol decoding plugins to provide this mapping, thus removing the need for this to be done inside the output plugins, thus improving the modular nature of the system.

8.4 Further work

There is scope for large amounts of further work, which in general falls under two categories. Firstly, there is remedial work to address limitations discussed above, and secondly there is extension work to provide support for additional protocols, analysis, and so on.

Protocol Identification *(REMEDIAL)*
Work may be performed to investigate alternate ways of uniquely identifying protocols. This would most likely result in some form of strings-based solution describing the protocol in terms of it's name, and the protocols it may have both above and below. Additionally, this investigation would need look at the PACKET protocol map, and alternative efficient methods for performing this

Protocol Decoding *(REMEDIAL)*
The protocol-decoding interface may be extended to address the concerns noted under the limitations above. This extension would probably involve the addition of a new 'mapping' function which takes a field name, and value as arguments and returns with the correct offset of the field into the packet, and so on, which may then be used for accessing the packet for both testing and output purposes. There are additional methods of implementing this. The aim is that all transactions requiring special knowledge of any given protocol be performed inside that protocols' protocol decoding plugin, thus improving modularity.

High level language and configuration *(REMEDIAL)*
Development of a mechanism for specifying Rubicon rules in a high level policy language, and being able to translate this into the required XML intermediate policy. This can be thought of as adding functionality but is being termed remedial as this was one of the unfulfilled aims of the original system.

New plugins *(EXTENSION)*

One obvious area for further work is in the extension of the system by implementing new plugins to add functionality. This includes support for new protocols, new methods of outputting, inputting and analysis.

More testing *(REMEDIAL)*
Also desirable is to further test the system. Substantial testing is required before the software is suitable for use on a business critical platform.

Policy Process Security *(REMEDIAL)*
There is currently no identity validation, or encryption involved in any part of the policy acquirement process. As such the system is currently very insecure. This can be rectified however with some further development work.

Support for reading writing in different languages *(EXTENSION)*
Finally, work on the construction of the Rubicon policy file from policies for other systems, e.g. Checkpoint FW-1, would prove useful. This would aid uptake as it would easy configuration of the system in heterogenous network environments.

9 References

Bhatia R., Lobo J., Kohli M. "Policy evaluation for network management" In *Proceedings IEEE INFOCOM 2000*, 3, 1107-1116

BLACKICE PC Protector [Accessed June 2002]
http://www.iss.net/products_services/hsoffice_protection/blkice_protect_pc.php

Bryhni H., Klovning E., Kure O. "A comparison of load balancing techniques for scalable Web servers" *IEEE Network*, 14(4), July/August 2000, 58-64

Burns J., Cheng A., Gurung P., Rajagopalan S., Rao P., Rosenbluth D., Surendran A.V., Martin D.M(Jr). "Automatic management of network security policy" In *Proceedings IEEE DISCEX'01*, 2001, 2, 12-26

Caberera J.B.D., Ravichandran B., Mehra R.K. "Statistical traffic modeling for network intrusion detection" In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication System*, 466-73

Chang H.Y., Wu S.F., Jou Y.F. "Real-time protocol analysis for detecting link-state routing protocol attacks" *ACM Transactions on Information and Systems Security*, 4(1), Feb 2001, 1-36

Changkun W. "Policy-based Network Management" In *Proceedings IEEE WCC 2000 - ICCT 2000*, 1, 101-105

CHECKPOINT http://www.checkpoint.com/products/protect/smartdefense.html [Accessed June 2002]

Cobb S. "Establishing Firewall Policy" *IEEE Southcon*, 1996, 198-205

Coit C.J., Staniford S., McAlerney J. "Towards faster string matching for intrusion detection or exceeding the speed of Snort" In *Proceedings IEEE DISCEX'01*, 2001, 1, 367-373

Curry D., Debar H. "Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition" http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-06.txt [Accessed May 2002]

Damianou N.C. 2002 "A Policy Framework for Management of Distributed Systems" Thesis(PhD), Imperial College

Damianou N., Dulay N., Lupu E., Sloman M. "The Ponder Policy Specification Language" Available at http://www.doc.ic.ac.uk/~mss/Papers/Ponder-Policy01V5.pdf [Accessed February 2002]

ETHEREAL[online] http://www.snort.org/ [Accessed February 2002]

Falk R., Trommer T. "Managing Network Security – A Pragmatic Approach" In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems*, 398-402

Ganesarajah D., 2001 "Web Service Workflow" Thesis(MEng), Imperial College

Gangadharan M., Kai-Hwang. "Intranet security with micro-firewalls and mobile agents for proactive intrusion response" In *Proceedings 2001 IEEE International Conference on Computer Networks and Mobile Computing*, 325-332

Gooch D.J., Hubbard S.D., Moore M.W., Hill J. "Firewalls – Evolve or Die" *BT Technology Journal*, 19(3), 89-98

Gupta S., Narasimha-Reddy A.L. "A client oriented, IP level redirection mechanism" In *Proceedings IEEE INFOCOM '99*, 3, 1461-1469

HOGWASH [online] http://hogwash.sourceforge.net/ [Accesses May 2002

Hunt R. "Internet/Intranet firewall security – policy, architecture, and transaction services" *Computer Communications*, 21, 1107-1123

IDS 2000 [online] www.securityfocus.com/infocus/1520 [Accessed May 2002]

Kanada Y. “A representation of network node QoS control policies using rule-based building blocks” In *IEEE IWQoS 2000*, 161-163

Kanada Y. “Policy division and fusion: examples and a method-or, multiple classifiers considered harmful” In *IEEE/IFIP International Symposium on Integrated Network Management Proceedings*, 545-560

Kewley D., Fink R., Lowry J., Dean M. “Dynamic approaches to thwart adversary intelligence gathering” In *Proceedings DISCEX’01*, 2001, 1, 176-185

Lakshman T.V., Stiliadis D. “High-Speed Policy-based Packet Forwarding Using Efficient Multi-Dimensional Range Matching” In *Proceedings IEEE SIGCOMM* 1998, 203-214

LIBIDXP [online] http://ftp.codefactory.se/pub/idxp/doc/index.html [Accessed May 2002]

LIBPCAP[online] *http://freshmeat.net/projects/libpcap/* [Accessed Feb 02]

Malan G.R.., Watson D., Jahanian F., Howell P. “Transport and application protocol scrubbing” In *Proceedings IEEE Infocom 2000*, 1381-1390

Malan G.R., Watson D., Smart M., Jahanian F. “Protocol scrubbing: network security through transparent flow modification” In Proceedings IEEE DISCEX’01, 2001, 2, 108-18

Matta I., Bestavros A. “A load profiling approach to routing guaranteed bandwidth flows” In *Proceedings. IEEE INFOCOM '98*, 3, 1012-1021

Moore B., Ellesson E., Strassner J., Westerinen A. “Policy Core Information Model – Version 1 Specification” *Internet RFC 3060*, February 2001

Moore B., Rafalow L., Ramberg Y., Snir Y., Chadha R., Brunner M., Strassner J., Westerinen A. “Policy Core Information Model Extensions” *Draft Extension of RFC 3060*, November 2001

MOZILLA [online] http://www.mozilla.org/directory/ [Accessed May 02]

Mukherjee B., Heberlein L.T., Levitt K.N. “Network intrusion detection” *IEEE Network*, 8(3), May-June 1994, 26-41

NETFILTER [online] http://netfilter.samba.org/ [Accessed February 02]

Nong Y., Emran S.M., Xiangyang L., Qiang C. “Statistical process control for computer intrusion detection” In *Proceedings DISCEX'01*, 2001, 1, 3-14

NOVELL [online] http://www.novell.com/products/edirectory/details.html [Accessed May 02]

Ogura T, Fukuda K, Iseda K, Okuda M. “A policy server for an access network (PSAN)” In *Proceedings IEEE ICC 2001*, 8, 2404-2409

OPENLDAP [online] http://www.openldap.org/ [Accessed May 02]

PLUGIN1 [online] http://www.isi.uu.nl/ix/programdoc/my_first_plugin.htm [Accessed February 02]

PLUGIN2 [online] http://www-106.ibm.com/developerworks/linux/library/l-dll.html?dwzone=linux [Accessed February 02]

PLUGIN3 [online] http://www.winamp.com/nsdn/winamp2x/dev/plugins/ [Accessed February 02]

RAW IP FAQ 2002 [online] http://www.whitefang.com/rin/rawfaq.html [Accessed February 02]

REALSECURE [Accessed June 2002]
http://www.iss.net/products_services/enterprise_protection/rsnetwork/index.php

RFC768 “User Datagram Protocol” http://www.faqs.org/rfcs/rfc768.html [Accessed May 02]

RFC791 “Internet Protocol” http://www.faqs.org/rfcs/rfc791.html [Accessed May 02]

RFC792 “Internet Control Message Protocol” http://www.faqs.org/rfcs/rfc792.html [Accessed May 02]

RFC793 “Transmission Control Protocol” http://www.faqs.org/rfcs/rfc793.html [Accessed May 02]

Roesch M. “Snort – Lightweight Iontrusion Detection for Networks” In *Proceedings USENIX LISA ’99*, 1999. Available at: - http://www.snort.org/docs/lisapaper.txt [Accessed February 02]

Schiffman M.D. “Libnet 1010, Part 1: The Primer” [Accessed February 02] http://www1.securityfocus.com/focus/linux/articles/libnet101.html

Sekar R., Guang Y., Verma S., Shanbhag “A high-performance network intrusion detection system” In *Proceedings 6th ACM Conference on Computer and Communications Securit*y, 1999, 8-17

Shan-Zheng, Chen-Peng, Xu-Ying, Xu-Ke “A network state based intrusion detection model” In Proceedings IEEE 2001 International Conference on Computer Networks and Mobile Computing, 481-486

Snapp S.R., Brentano J., Dias G.V., Goan T.L., Grance T., Heberlein L.T., Ho C.L., Levitt K.N., Mukherjee B., Mansur D.L., Pon K.L., Smaha S.E. “A system for distributed intrusion detection” In *Proceedings IEEE COMPCON Spring '91*, 1991, 170-176

SNORT:[online] http://www.snort.org/ [Accessed February 02]

Stone G.N., Lundy B., Geoffrey G. Xie. “Network Policy Languages: A Survey and a New Approach” *IEEE Network*, Jan/Feb 2001, 10-21

SYMANTEC Intruder Alert [Accessed June 2002] http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=48&PID=12166302&EID=0

Thomsen D., O’Brien R., Payne C. “Napoleon: Network Application Policy Environment” In *Proceedings IEEE RBAC ’99*, 1999, 145-152

Treadway A. 2001 “A Mobile Agent Architecture to Provide Virtual Homan Environment Services to Nomadic Users” Thesis(Meng), Imperial College

White J., Feinstein B., Matthews G. “The Intrusion Detection Exchange Protocol (IDXP)” http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-04.txt [Accessed May 2002]

10 User Guide

The system is only configured through two ways. Firstly, the location of plugins, and a policy file may be specified from the command line. Secondly, through the XML policy, whether specified on the command line or via the remote command/LDAP route.

10.1 Command line

Two arguments are required to the rubicon executable:-
 -p <path> = The location of the plugin (.plg) files
 -i <policy> = The initial policy file.

Both arguments require **full** paths, e.g.
/usr/local/bin/rubicon –i /usr/local/etc/initial_policy.txt –p /usr/local/lib/rubicon/

10.2 XML Policy file

The XML policy file implements the Data Type definition given in section 10.3 below. In essence the XML policy is just a textual version of the local representation of a policy. All rubicon rules must be contained within: -

```
<?xml version=”1.0”?>
<Rubicon>
<policy>

<!------ rules are inserted here ----->

</policy>
</Rubicon>
```

At the root level, there are 6 possible elements that may be used:-

- default – Contains the default actions to be performed.
- snort – Either points to a file containing snort rules to parse and add to the policy, or the snort rules inlined into the policy. The parameters are *source*, which can be either ‘file’ or ‘include’, and then if the *source* is ‘file’ there <u>must</u> be a *filename* parameter giving the full location of the file.
- protocol – A protocol type test. Contains the actions to be performed in the event of a match occurring. The *proto* parameter must be included, which has the value specifying the protocol to match. this may be the protocol numeric identifier, or the special strings “ETH”, “IP”, “UDP”, “TCP” and “ICMP”. If one of the special strings is used, the additional parameters *field* (specifies by name which field in the header to test), *val* (which specifies the value to test for), and *cmpType*, which specifies the type of comparison. Alternately, on of the parameter sets of (*val*, *offset*, *type*, *cmpType*) or (*offset*, *len*, *num*, *mask*, *negate*) may be specified. The first of these is used for typed protocol tests, and the latter for masked.
- init – Specifies the initialization string for a plugin. The *name* parameter specifies the plugin, and the *arg* parameter gives the initialization string to use. This is an EMPTY element.
- define – Used to define a group of actions and tests which can be inserted elsewhere in the policy. Takes an *id* parameter which must be unique, an contains the elements to insert elsewhere.
- instance – This takes an *id* parameter which links to a previous ‘define’. Any place in the policy that an instance element occurs it is replaced with the contents of the relevant define. It must be noted that care must be taken to ensure only valid replacements occur, as no checking is performed until load-time on whether legal elements are being included.

Protocol elements may contain the following elements, which are called in order from top to bottom in the event of a protocol match occurring.

- output – an output plugin. This element must contain a *name* parameter which specifies the name of the plugin to call. Optionally an *eof* parameter may be given the values “yes” or “no” to specify whether this is the end of handling the given packet. The element contains the string to use as an argument.

- protocol – As described above.
- analysis - Calls an analysis plugin. Requires the *name* of the plugin as a parameter, and optionally its *args*. Contains analysisResults which specify the different strings to match against the output of the analysis.
- instance – As described above.

As mentioned, the analysis element contains analysisResult elements. The elements contain the protocol, analysis, output, and instance elements that are run if the relevant *match* parameter of the analysisResult element matches the result from the parent analysis test.

For example:

```
<?xml version=”1.0”?>
<Rubicon>
<policy>
   <default>
      <output name=”log” args=”This is the default output”>
   </default>
   <init name=”plg” arg=”The initialization string for the plg plugin”>
   <define id=”copy_me”>
      <output name=”log”>I want this log to appear in </output>
      <output name=”log”>every location where an instance </output>
      <output name=”log”>of the copy_me define occurs </output>
   </define>
   <protocol name=”0x4000001”> <!—check for the existence of a certain proto→
      <analysis name=”if it does” arg=”then run this analysis”>
         <analysisResult match=”this”>
               <output name=”something” eof=”yes”></output>
         </analysisResult>
         <analysisResult match=”that”>
               <output name=”something else” eof=”yes”></output>
         </analysisResult>
         <analysisResult match=””> <!— the default result -->
               <instance id=”copy_me_”> <!—print all those logs -->
         </analysisResult>
      </analysis>
   </protocol>
   <protocol name=”IP”>
      <instance id=”copy_me”> <!—print those outputs again -->
   </protocol>
   <snort source=”file” filename=”/etc/snort.rules”/>
   <snort source=”include”>
      Some snort rules to be parsed
   </snort>
</policy>
</Rubicon>
```

10.3 XML Configuration File Data Type Definition (DTD)

```
<!---
     Data Type Definition for RUBICON
     Version 1.0 (May 2002)
     This dtd should be modified to enfore parameters, as specified below.
-->

<!-- .elem - Entities to describe what elements different elements may contain
-->
<!ENTITY % Rubicon.elem
```

```
                           '(policy)'>
<!ENTITY % policy.elem
                           '(default|snort|protocol|init|define|instance)*'>
<!ENTITY % protocol.elem
                           '(output|protocol|analysis|instance)*'>
<!ENTITY % define.elem
                           '(output|protocol|analysis|instance)*'>
<!ENTITY % analysis.elem
                           '(analysisResult)*'>
<!ENTITY % analysisResult.elem
                           '(output|protocol|analysis|instance)*'>
<!ENTITY % default.elem
                           '(output)*'>

<!-- .attrs - Attributes for the different element types -->
<!-- The protocol element should be converted into different elements to
reflect the different valid combinations of parameters. Namely if 'field' is
specified, then 'val' is required, if 'type' is given then 'cmpType', 'offset',
and 'val' are required, and if 'mask' is specified, 'negate', 'len', 'offset'
and 'num' are required. Additionally, only one of 'field', 'type' or 'mask' may
be specified. Therefore -  the protocol element should be expanded to 3 new
elements -->
<!ENTITY % varTypes
                           '(BYTE|UBYTE|SHORT|USHORT|LONG|ULONG)'>
<!ENTITY % protocol.attrs
                           'proto      CDATA            #REQUIRED
                            field      CDATA            #IMPLIED
                            val        CDATA            #IMPLIED
                            type       %varTypes;       #IMPLIED
                            cmpType    (&gt; &lt; = !)  "="
                            offset     CDATA            #IMPLIED
                            len        CDATA            #IMPLIED
                            num        CDATA            #IMPLIED
                            mask       CDATA            #IMPLIED
                            negate     (yes|no)         "no" '>
<!-- if source==file, then filename must be given -->
<!ENTITY % snort.attrs
                           'source     (file|include)   "include"
                            filename   CDATA            #IMPLIED'>
<!ENTITY % define.attrs
                           'id         ID               #REQUIRED'>
<!ENTITY % analysis.attrs
                           'name       CDATA            #REQUIRED
                            args       CDATA            #IMPLIED'>

<!ENTITY % analysisResult.attrs
                           'match      CDATA            #IMPLIED'>
<!ENTITY % output.attrs
                           'name       CDATA            #IMPLIED
                            args       CDATA            #IMPLIED
                            eof        (yes|no)         "no"'>
<!ENTITY % instance.attrs
                           'id         IDREF            #REQUIRED'>
<!ENTITY % init.attrs
                           'name       CDATA            #REQUIRED
                            arg        CDATA            #REQUIRED'>

<!-- ELEMENT definitions--->
<!-- The primary element. Used so that definitions for different systems or
different versions of Rubicon may be included in the same XML file -->
<!ELEMENT Rubicon %Rubicon.elem;>
```

```
<!ATTLIST Rubicon
                           %Rubicon.attrs;>
<!-- The rubicon policy -->
<!ELEMENT policy %policy.elem;>
<!-- A protocol-style test -->
<!ELEMENT protocol %protocol.elem;>
<!ATTLIST protocol
                           %protocol.attrs;>
<!-- Snort-format rules to be parsed -->
<!ELEMENT snort (#CDATA)>
<!ATTLIST snort
                           %snort.attrs;>
<!-- define a set of tags for inclusion elsewhere in the policy -->
<!ELEMENT define %define.elem;>
<!ATTLIST define
                           %define.attrs;>
<!-- an analysis style test -->
<!ELEMENT analysis %analysis.elem;>
<!ATTLIST analysis
                           %analysis.attrs;>
<!-- The results of the analysis-style test -->
<!ELEMENT analysisResult %analysisResult.elem;>
<!ATTLIST analysisResult
                           %analysisResult.attrs;>
<!-- a call to an output plugin -->
<!ELEMENT output EMPTY>
<!ATTLIST output
                           %output.attrs;>
<!-- an instance of a definition (above). For example, if the same list of
outputs and tests are required in multiple positions in a policy, rather than
defining them at all the desired points they only need to be defined within a
'define' tag, and then instances of this may be included wherever desired -->
<!ELEMENT instance EMPTY>
<!ATTLIST instance
                           %instance.attrs;>
<!-- a list of the default outputs -->
<!ELEMENT default %default.elem;>
<!-- initialisation strings for plugins -->
<!ELEMENT init EMPTY>
<!ATTLIST init
                           %init.attrs;>
```

11 Developer Guide

The developed code is heavily commented, and documentation is available in html and man page format at www.ianpeters.net and so no detail will be given here. Each source file has a summary of its purpose, and each function has it’s purpose, arguments and return variables explained. These comments, combined with this document should inform the developer of all that is needed.