TRAINING REPORT

SIEMENS

“Application Proxy Firewall for SNMP”

Submitted By
Sagar Narla
(0601012806)
B.Tech Electronics & Communication

Ambedkar Institute of Technology
(GGSIPU)

Certificate

This is to certify that Mr. Sagar Narla has completed his training successfully at Siemens Information Systems Ltd, Gurgaon from 8th June 2009 to 31st July 2009. During this time he has successfully completed the project on "Application Proxy Firewall". During this period I found him sincere and hard working. His work has been satisfactory and I wish him success in future endeavors.

Siemens Information Systems Ltd

Mr. G. E. Manoranjan
Consultant

Acknowledgment

It is a great pleasure to have an opportunity to extend my heartfelt thanks to everyone who has helped me through out the successful competition of the project. I convey my gratitude to all those who have helped me reach a stage where I have immense confidence to set to launching my career in the competitive world of Information Technology

I express my sincere gratitude towards Mr. G.E. Manoranjan for providing me the opportunity to undertake this project training at Siemens Information Systems Ltd. I am grateful to the entire CTDC E F Department, who were always there to guide me when needed.

I acknowledge the role of my institute, my respected lecturers, who have played a significant role in shaping my career. I express my profound gratitude to all teachers for gently guiding and paving my way toward a bright career.

Sagar Narla

Company Profile

Siemens Information Systems Ltd. (SISL), SISL is a wholly owned subsidiary of Siemens Ltd.- the regional company of Siemens AG in India. SISL is the IT arm of Siemens Ltd. SISL provides IT services and solutions globally. Since October 2007 SISL is an integral part of Siemens IT Solutions and Services, a Siemens Group company.

What we do
Siemens Information Systems Ltd. is a systems integrator and a total solution provider.

Siemens preferred partner for off shoring
We have been identified by the Siemens Software Initiative as a preferred offshore partner for Siemens group companies.

Mission vision strategy
We share the mission, vision and strategy of Siemens IT Solutions and Services.

Business philosophy and business model
Our business philosophy is to set benchmarks by being "best in class" in our fields and create value for our customers.

People, Quality, Innovation
SISL can be summarized in three words: people; quality and innovation.

Proven performance
SISL was established in 1992. Since then we have continued to grow steadily and profitably.

SIEMENS

CONTENTS

ABSTRACT

An Application Proxy Firewall was implemented on a Server that was at the inter-connection of two networks, one Internal and secure network and the other External and insecure network.

To protect the critical Internal network from threats posed by the external network the Firewall was implemented such that it would firstly check the integrity of the packet in terms of proper encoding that is its adherence to the SNMP standard consequently if it passed this check then various firewall rules were applied on the parsed data of the SNMP packet. These rules checked the validity of the various aspects of the SNMP. If the packet was found to adhere to all of these rules then the packet was sent to the intended destination after a regeneration from the parsed data itself. However if the packet failed to adhere to any of the rules at any stage the packet was immediately dropped and not propagated further. Additionally the Firewall was implemented in such a manner that minimal overhead was incurred in terms of speed, traffic volumes and efficiency in comparison to the existing network.

Such a mechanism ensured that all packets finally emanating from the Application Proxy Firewall were safe and free from any known malicious elements.

INTRODUCTION

With ever growing advancements in technology newer and improved features are constantly incorporated into current technologies to adapt them to the ever changing environment.

The Power Plant Automation Network was needed to be connected to Corporate and Global Networks to facilitate remote monitoring and regulation of critical application on the Power Plant Automation System. This seamless integration however introduced vulnerabilities from the external network to the critical Power Plant Network. This necessitated the need for a Firewall that integrated into the existing network and protocol stack and provided the much needed security. The firewall would sniff every packet to check for any malicious elements as well as the relevancy of the packet to the network failing which the packet is removed from the network. With such a system in place every packet is thus ensured to be safe and secure which in turn secures the entire network without compromising on functionality achieved through the inter-connection of Global Networks.

PROJECT OVERVEIW

SNMP

Simple Network Management Protocol (SNMP) is used in network management systems to monitor network-attached devices for conditions that warrant administrative attention. It is used for collecting information from, and configuring, network devices, such as servers, printers, hubs, switches, and routers on an Internet Protocol (IP) network. SNMP can collect information such as a server's CPU level, Server chassis Temperature etc. It consists of a set of standards for network management, including an application layer protocol, a database schema, and a set of data objects.

SNMP exposes management data in the form of variables on the managed systems, which describe the system configuration. These variables can then be queried (and sometimes set) by managing applications.

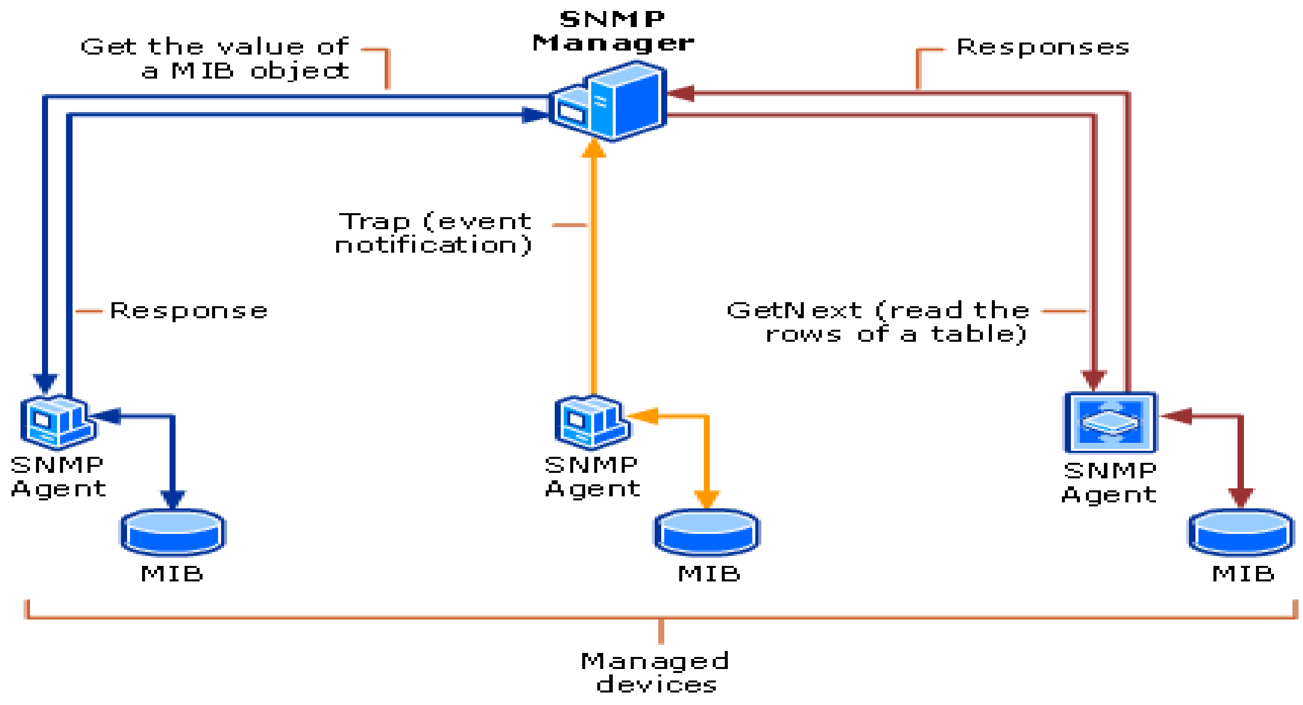
The SNMP Architecture

Implicit in the SNMP architectural model is a collection of network management stations and network elements. Network management stations execute management applications which monitor and control network elements. Network elements are devices such as hosts, gateways, terminal servers, and the like, which have management agents responsible for performing the network management functions requested by the network management stations. The Simple Network Management Protocol (SNMP) is used to communicate management information between the network management stations and the agents in the network elements.

Basic components

An SNMP-managed network consists of three key components:

- Managed device
- Agent
- Network management system (NMS)


SNMP Manager
Get the value of a MIB object
Responses
Trap (event notification)
Response
GetNext (read the rows of a table)
SNMP Agent
SNMP Agent
SNMP Agent
MIB
MIB
MIB
Managed devices

<u>A managed device</u> is a network node that contains an SNMP agent and that resides on a managed network. Managed devices collect and store management information and make this information available to NMSs using SNMP. Managed devices, sometimes called network elements, can be any type of device including, but not limited to, routers, access servers, switches, bridges, hubs, IP telephones, computer hosts, and printers.

<u>An agent</u> is a network-management software module that resides in a managed device. An agent has local knowledge of management information (such as "free memory", "system name", "number of running processes", "default route") and translates that information into a form compatible with SNMP.

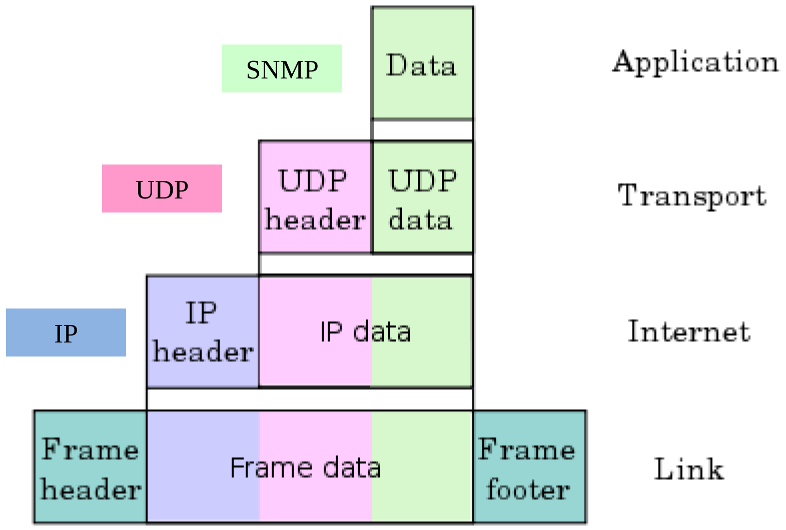
<u>A network management system (NMS)</u> executes applications that monitor and control managed devices. NMSs provide the bulk of the processing and memory resources required for network management. One or more NMSs may exist on any managed network.

In typical SNMP use, one or more administrative computers have the task of monitoring or managing a group of hosts or devices on a computer network. Each managed system executes, at all times, a software component called an agent (see below) which reports information via SNMP to the managing systems.

Essentially, SNMP agents expose management data on the managed systems as variables which are read, modified by SNMP Managers.

UDP

SNMP uses UDP at the Transport Layer.
UDP stands for User Datagram Protocol and is the opposite of TCP, Transmission Control Protocol which is a very reliable and high overhead protocol.

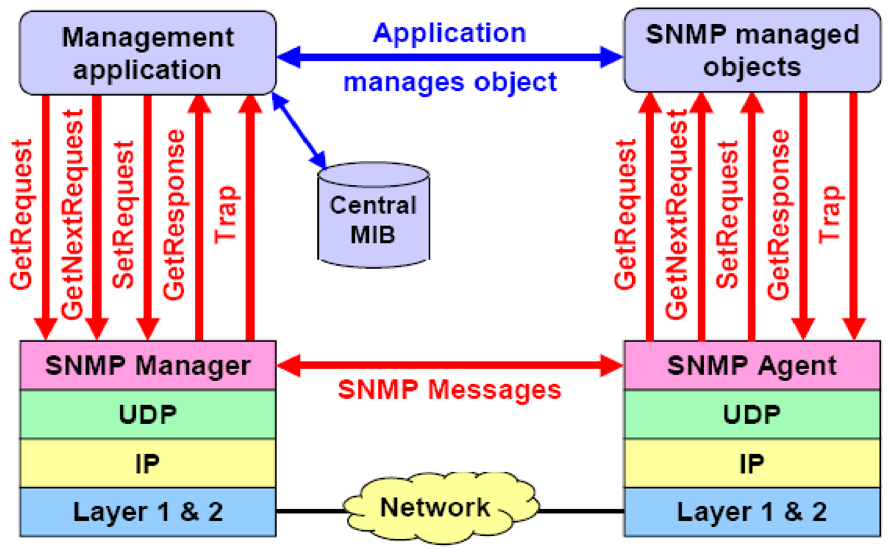


User Datagram Protocol is very low overhead, fast and unreliable. UDP is easier to implement and use than a more complex protocol such as TCP. It does however provide plenty of functionality to allow a central manager station to communicate with a remote agent that resides on any managed device that it can communicate with. The unreliability comes in the form of checks and balances whereas if TCP sends something, it waits for an acknowledgment and if it doesn't hear back, it will resend. Since logging of devices usually happens within a time period that is cyclic in nature, then it is possible to catch in the next cycle. The tradeoff between TCP or UDP is that the low overhead protocol UDP, is simple to use and doesn't need excessive bandwidth like TCP based applications.

SNMP Primitives

SNMP has five control primitives that represent data flow from the requester which is usually the Manager. These would be GET, GET-NEXT, SET, GET-RESPOSE, TRAP.

The manager uses the get primitive to get a single piece of information from an agent. You would use get-next if you had more than one item. When the data the manager needs to get from the agent consists of more than one item, this primitive is used to sequentially retrieve data; for example, a table of values.
You can use set when you want to set a particular value. The manager can use this primitive to request that the agent running on the remote device set a particular variable to a certain value.

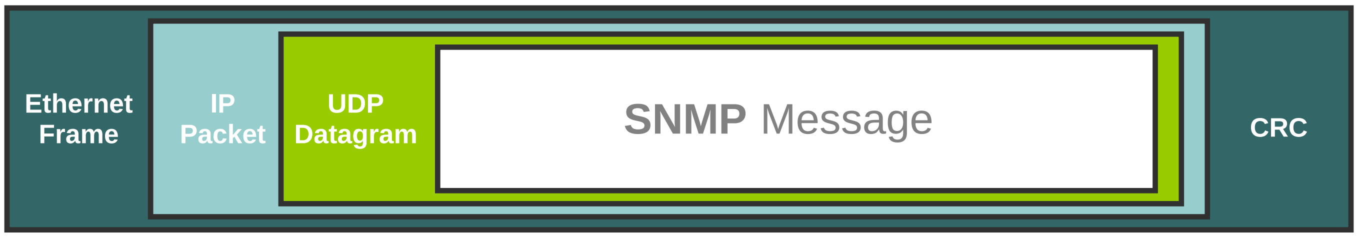


There are two control primitives the responder (manager) uses to reply and that is get-response and trap. One is used in response to the requester's direct query (get-response) and the other is an asynchronous response to obtain the requester's attention (trap). The manager doesn't always initiate – sometimes the agent can as well. Although SNMP exchanges are usually initiated by the manager software, this primitive can also be used when the agent needs to inform the manager of some important event. This is commonly known and heard of as a 'trap' sent by the agent to the NMS.

A message consists of a version identifier, an SNMP community name, and a protocol data unit (PDU).
A protocol entity receives messages at UDP port 161 on the host with which it is associated for all messages except for those which report traps (i.e., all messages except those which contain the Trap-PDU). Messages which report traps should be received on UDP port 162 for further processing.

SNMP Version 1 (SNMPv1) Message Format

SNMP an application layer protocol is implemented by the UDP at the Transport Layer. The UDP is in turn implemented by the IP at the Network Layer which is encapsulated in frames at the Physical Layer.
Thus each SNMP message finally results as shown below at the Physical Layer.



SNMP Message at the Physical Layer

General SNMP Message Format

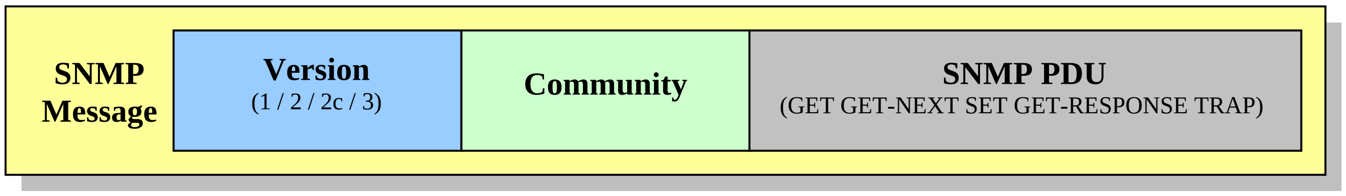
The SNMP message format specifies which fields to include in the message and in what order. Ultimately, the message is made of several layers of nested fields. At the outer-most layer, the SNMP message is a single field. The entire message is further made up of a sequence of smaller fields.

Each message carries one PDU, and the PDU is the most important part of the message. The PDU is the actual piece of information that is being communicated between SNMP entities. It is carried *within* the SNMP message along with a number of header fields, which are used to carry identification and security information.

Thus, conceptually, the SNMP message format can be considered to have two overall sections:

- **Message Header:** Contains fields used to control how the message is processed, including fields for implementing SNMP security.
- **Message Body (PDU):** Contains the main portion of the message. In this case, the message body is the protocol data unit (PDU) being transmitted.

The overall SNMP message is sometimes called a *wrapper* for the PDU, since it encapsulates the PDU and precedes it with additional fields.



General PDU Format

The fields in each PDU depend on the PDU type, but can again be divided into the following general substructure:

- PDU Control Fields: A set of fields that describe the PDU and communicate information from one SNMP entity to another.
- PDU Variable Bindings**:** A set of descriptions of the MIB objects in the PDU. Each object is described as a "binding" of a name to a value.

Each PDU will follow this general structure, which is shown in Figure, differing only in the number of control fields, the number of variable bindings, and how they are used. In theory, each PDU could have a different message format using a distinct set of control fields, but in practice, most PDUs in a particular SNMP version use the same control fields (though there are exceptions.)

SNMPv1 Common PDU Format

The common PDUs in SNMPv1 : *GetRequest-PDU, GetNextRequest-PDU*, *SetRequest-PDU* and *GetResponse-PDU*

GET-REQUEST PDU

The GetRequest-PDU is generated by a protocol entity only at the request of its SNMP application entity. This PDU is used to retrieve values of objects maintained by the SNMP managers. The GetRequest contains a unique Request ID for that particular message. Additionally the OID of the object values to be retrieved are specified in the variable bindings.

When the manager receives a GetRequest PDU it checks the OIDs specified. If any of the OID are invalid either or not present an appropriate error index and error status are generated and sent back on a GetResponse PDU with the same Request ID as the one it was generated for. However if all the values of the OID are valid and available they are sent on the GetResponse packet and error index and error status are appropriately set to indicate no error.

GETNEXT-REQUEST PDU

The GetNextRequest PDU is used to retrieve bulk data from the manager by the agent. The Agent specifies OID in a predefined lexicographical order.

Upon receiving the Manager checks the OID to be in the lexicographical order if not generates an GetResponse PDU with appropriate error index and status values. If the datat requested exceeds the maimum linit of the GetResponse PDU and also if any of the OID in the GetNextRequest are not present or invalid a GetResponse is sent with appropriate error index and error status values.

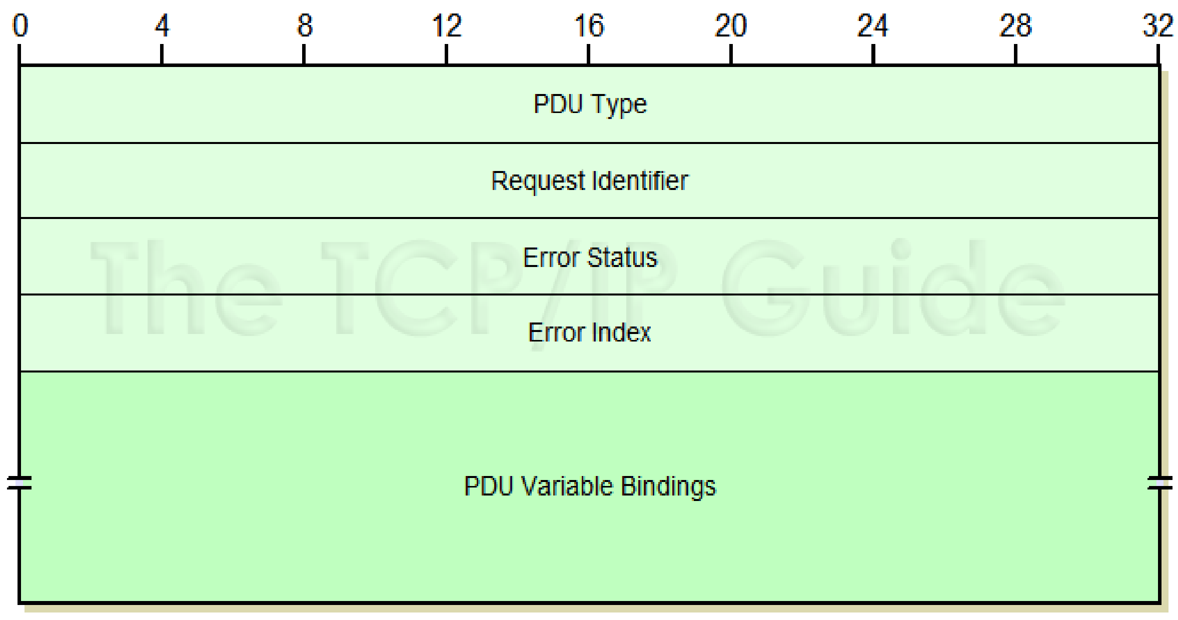
GET_RESPONSE PDU

GetResponse PDU is sent whenever a GetRequest , SetRequest or GetNextRequest PDUs are received. The GetResponse can either provide the requested OID's value or send an error message based upon the request.

SET-REQUEST PDU

SetRequest PDU allows the managers to set certain OID values to their choice. The manager accordingly sets the value of the OID mentioned if present and valid otherwise an GetRespose is sent with appropriate error index and error status values. However the agent needs to have the necessary permissions to modify the OID value the unavailability of which will also result in an error. The manager before updating value to the value requested will check for the compatitbilty of data types mentioned by the SetRequest and the actual data type of the OID if a mismatch is found an error occurs. Also certain OID do not allow their value to be altered and in which case a SetRequest for such an OID will cause an error.

However if the SetRequest is successful a GetResponse is sent with the new updated value.

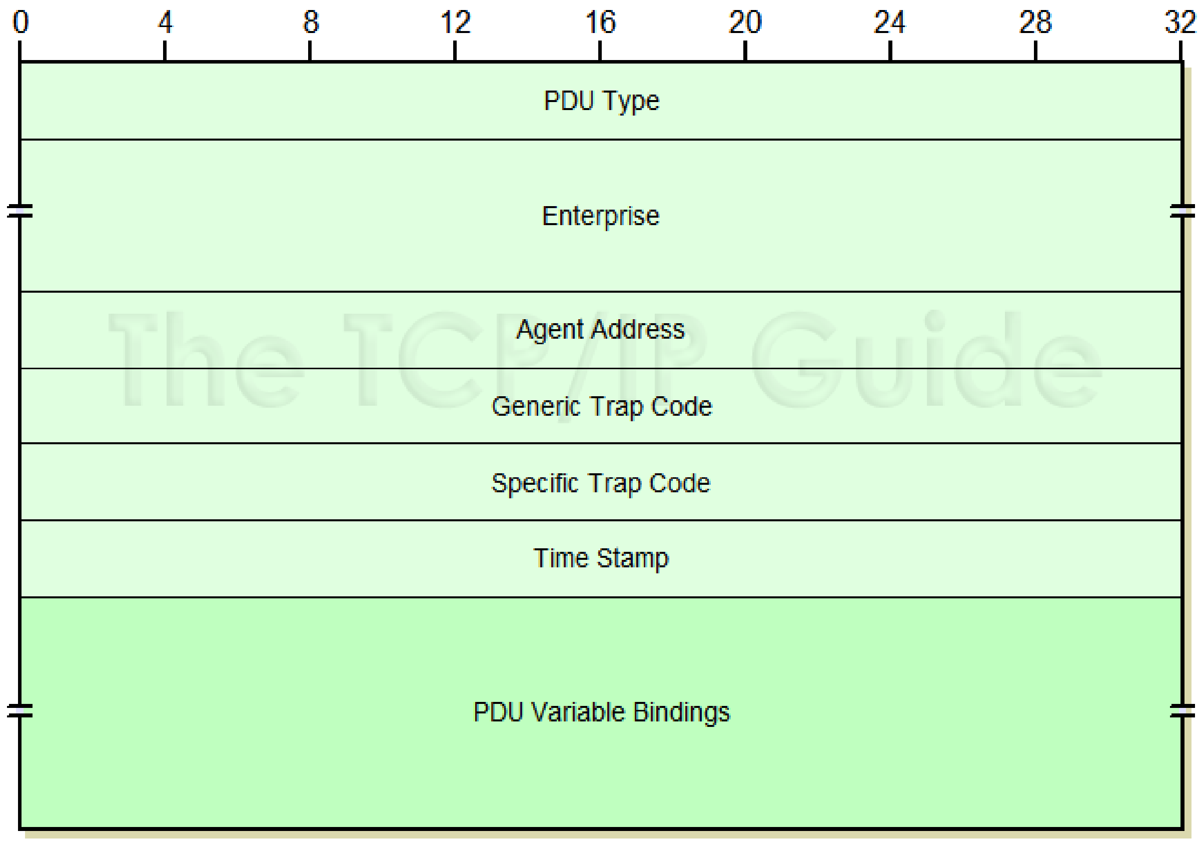


SNMPv1 *Trap-PDU* Format

The Trap-PDU is generated by a protocol entity only at the request of the SNMP application entity. The means by which an SNMP application entity selects the destination addresses of the SNMP application entities is implementation-specific.

Upon receipt of the Trap-PDU, the receiving protocol entity presents its contents to its SNMP application entity.

The significance of the variable-bindings component of the Trap-PDU is implementation-specific.

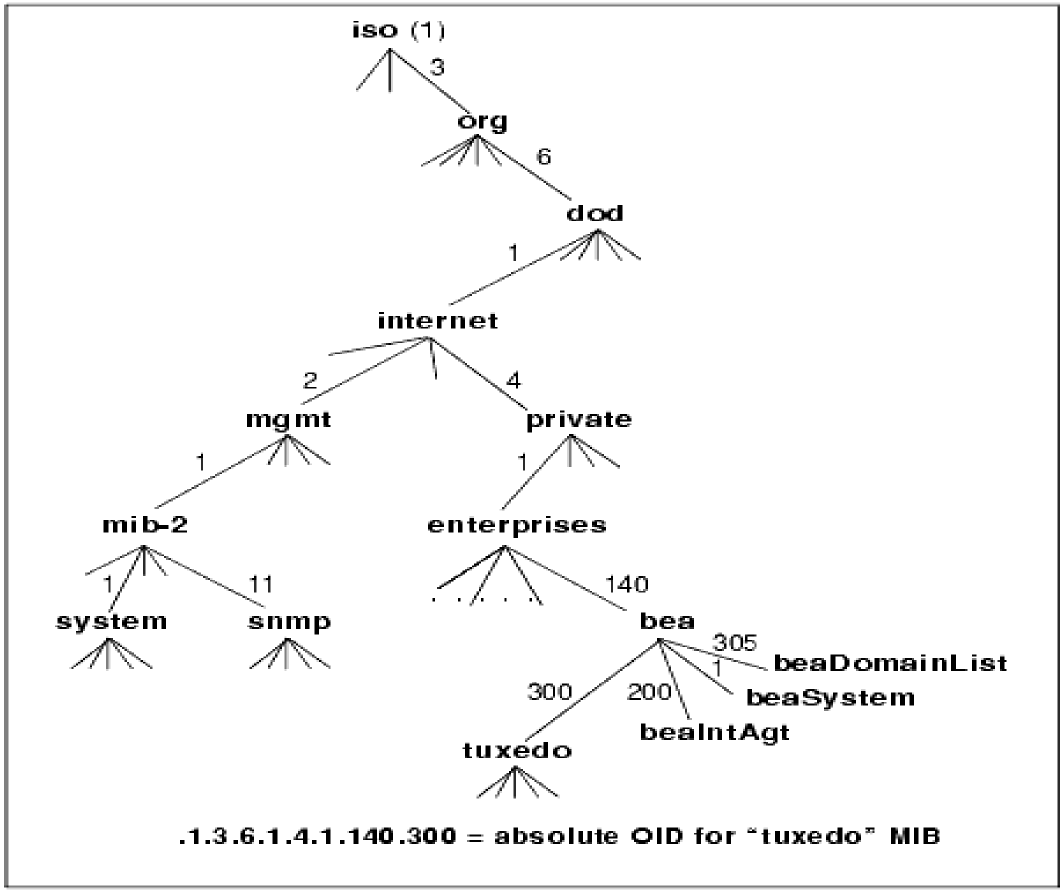


The Management Information Base (MIB)

We just learned what primitives were… the agent and the manager, exchanging data. The data they exchange also has a name. The types of data the agent and manager exchange are defined by a database called the management information base (MIB).The MIB is a virtual information store. Remember, it is a small database of information and it resides on the agent. Information collected by the agent is stored in the MIB. The MIB is precisely defined; the current Internet standard MIB contains more than a thousand objects. Each object in the MIB represents some specific entity on the managed device.

A management information base (MIB) stems from the OSI/ISO Network management model and is a type of database used to manage the devices in a communications network. It comprises a collection of objects in a (virtual) database used to manage entities (such as routers and switches) in a network.

Objects in the MIB are defined using a subset of Abstract Syntax Notation One (ASN.1) called "Structure of Management Information Version 2 (SMIv2)" RFC 2578.The software that performs the parsing is a MIB compiler.



The database is hierarchical (tree-structured) and entries are addressed through object identifiers. Internet documentation RFCs discuss MIBs, notably RFC 1155, "Structure and Identification of Management Information for TCP/IP based internets", and its two companions, RFC 1213, "Management Information Base for Network Management of TCP/IP-based internets", and RFC 1157, "A Simple Network Management Protocol".

SNMP, a communication protocol between management stations, such as consoles, and managed objects (MIB objects), such as routers, gateways, and switches, makes use of MIBs. Components controlled by the management console need a so-called SNMP agent — a software module that can communicate with the SNMP manager.

SNMP uses a specified set of commands and queries. A MIB should contain information on these commands and on the target objects (controllable entities or potential sources of status information) with a view to tuning the network transport to the current needs.

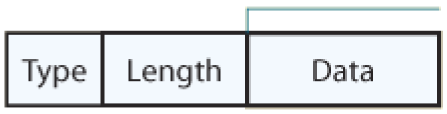
ASN.1

Constructing a message requires some knowledge of the data types specified by ASN.1, which fall into two categories: primitive and complex. ASN.1 primitive data types include Integer, Octet (byte, character) String, Null, Boolean and Object Identifier. The Object Identifier type is central to the SNMP message, because a field of the Object Identifier type holds the OID used to address a parameter in the SNMP agent. To expand the programmer's ability to organize data, ASN.1 allows primitive data types to be grouped together into complex data types.

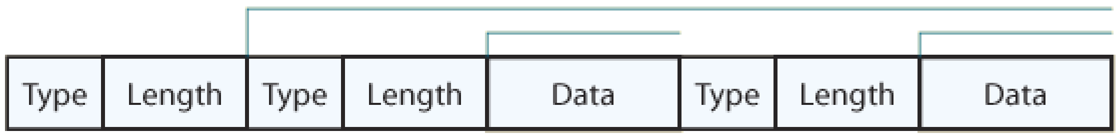
ASN.1 offers several complex data types necessary for building SNMP messages. One complex data type is the Sequence. A Sequence is simply a list of data fields. Each field in a Sequence can have a different data type. ASN.1 also defines the SNMP PDU (Protocol Data unit) data types, which are complex data types specific to SNMP. The PDU field contains the body of an SNMP message. Two PDU data types available are GetRequest and SetRequest, which hold all the necessary data to get and set parameters, respectively. Ultimately the SNMP message is a structure built entirely from fields of ASN.1 data types. However, specifying the correct data type is not enough. If the SNMP message is a Sequence of fields with varying data types, how can a recipient know where one field ends and another begins, or the data type of each field? Avoid these problems by conforming to the Basic Encoding Rules.

Basic Encoding Rules

Follow the Basic Encoding Rules when laying out the bytes of an SNMP message. The most fundamental rule states that each field is encoded in three parts: Type, Length, and Data. Type specifies the data type of the field using a single byte identifier. For a brief table of some data types and their identifiers, see Table 1. Length specifies the length in bytes of the following Data section, and Data is the actual value communicated (the number, string, OID, etc). One way to visualize encoding a field is shown



Some data types, like Sequences and PDUs, are built from several smaller fields. Therefore, a complex data type is encoded as nested fields, as shown



FIREWALL

A firewall is a part of a computer system or network that is designed to block unauthorized access while permitting authorized communications. It is a device or set of devices configured to permit, deny, encrypt, decrypt, or proxy all (in and out) computer traffic between different security domains based upon a set of rules and other criteria.

Firewalls can be implemented in either hardware or software, or a combination of both. Firewalls are frequently used to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets. All messages entering or leaving the intranet pass through the firewall, which examines each message and blocks those that do not meet the specified security criteria.

A firewall is a dedicated appliance, or software running on a computer, which inspects network traffic passing through it, and denies or permits passage based on a set of rules.

It is a software or hardware that is normally placed between a protected network and a not protected network and acts like a gate to protect assets to ensure that nothing private goes out and nothing malicious comes in.

A firewall's basic task is to regulate some of the flow of traffic between computer networks of different trust levels

There are several types of firewall techniques:

Packet filter: Looks at each packet entering or leaving the network and accepts or rejects it based on user-defined rules. Packet filtering is fairly effective and transparent to users, but it is difficult to configure. In addition, it is susceptible to IP spoofing.

Application gateway: Applies security mechanisms to specific applications, such as FTP and Telnet servers. This is very effective, but can impose a performance degradation.

Circuit-level gateway: Applies security mechanisms when a TCP or UDP connection is established. Once the connection has been made, packets can flow between the hosts without further checking.

Proxy server: Intercepts all messages entering and leaving the network. The proxy server effectively hides the true network addresses.

Application layer firewall

Another type of firewall is the application-proxy firewall. In a proxying firewall, every packet is stopped at the firewall. The packet is then examined and compared to the rules configured into the firewall. If the packet passes the examinations, it is re-created and sent out. Because each packet is destroyed and re-created, there is a potential that an application-proxy firewall can prevent unknown attacks based upon weaknesses in the TCP/IP protocol suite that would not be prevented by a packet filtering firewall. The drawback is that a separate application-proxy must be written for each application type being proxied. You need an HTTP proxy for web traffic, an FTP proxy for file transfers, a Gopher proxy for Gopher traffic, etc... Application-proxy firewalls operate on Layer 7 of the OSI model, the Application Layer

In computer networking, an application layer firewall is a firewall operating at the application layer of a protocol stack. Generally it is a host using various forms of proxy servers to proxy traffic instead of routing it. As it works on the application layer, it may inspect the contents of the traffic, blocking what the firewall administrator views as inappropriate content, such as certain websites, viruses, attempts to exploit known logical flaws in client software, and so forth.

An application layer firewall does not route traffic on the network layer. All traffic stops at the firewall which may initiate its own connections if the traffic satisfies the rules.

These generally are hosts running proxy servers, which permit no traffic directly between networks, and which perform elaborate logging and auditing of traffic passing through them. Since the proxy applications are software components running on the firewall, it is a good place to do lots of logging and access control. Application layer firewalls can be used as network address translators, since traffic goes in one ``side'' and out the other, after having passed through an application that effectively masks the origin of the initiating connection. Having an application in the way in some cases may impact performance and may make the firewall less transparent.

On inspecting all packets for improper content, firewalls can restrict or prevent outright the spread of networked computer worms and trojans. In practice, however, this becomes so complex and so difficult to attempt (given the variety of applications and the diversity of content each may allow in its packet traffic) that comprehensive firewall design does not generally attempt this approach. Hence invariably separate application firewalls are coded for different protocols.

Proxies

A proxy device (running either on dedicated hardware or as software on a general-purpose machine) may act as a firewall by responding to input packets (connection requests, for example) in the manner of an application, whilst blocking other packets.

Proxies make tampering with an internal system from the external network more difficult and misuse of one internal system would not necessarily cause a security breach exploitable from outside the firewall (as long as the application proxy remains intact and properly configured). Conversely, intruders may hijack a publicly-reachable system and use it as a proxy for their own purposes; the proxy then masquerades as that system to other internal machines. While use of internal address spaces enhances security, crackers may still employ methods such as IP spoofing to attempt to pass packets to a target network.

Application Proxy

An application proxy firewall takes apart each packet that comes in, examines it to see if it meets the criteria set, rewrites it, and sends it on its way.The proxy terminates the connection from the outside source and starts a new connection from the proxy to the destination.This offers great protection to the servers, because there is no direct interaction between the source and the destination. In addition, the proxy is greatly hardened against attacks and has a very small attack surface. It is very difficult for a hacker to take control of an application proxy firewall. These firewalls are very specific and a proxy must be written for each supported application.The advantage to this is that you will have the exact needs of your particular application addressed; however, you are at the mercy of the vendor should there be an update to your application that the firewall doesn't support. Delays may occur in upgrading your application until the firewall vendor catches up. Application proxies are usually “invisible” on the network. Often, they have no IP address themselves, or, if they do, they sometimes masquerade as the destination server.Thus, application proxies may not do address translation.

PAT

Port Address Translation (PAT) is a feature of a network device that translates TCP or UDP communications made between hosts on a private network and hosts on a public network. It allows a single public IP address to be used by many hosts on a private network, which is usually a Local Area Network or LAN.

A PAT device transparently modifies IP packets as they pass through it. The modifications make all the packets which it sends to the public network from the multiple hosts on the private network appear to originate from a single host, (the PAT device) on the public network.

In PAT there is generally only one publicly exposed IP address and multiple private hosts connecting through the exposed address. Incoming packets from the public network are routed to their destinations on the private network by reference to a table held within the PAT device which keeps track of public and private port pairs.

In PAT, both the sender's private IP and port number are modified; the PAT device chooses the port numbers which will be seen by hosts on the public network. In this way, PAT operates at layer 3 (network) and 4 (transport) of the OSI model,

Advantages of PAT

In addition to the advantages provided by NAT: The primary benefit of IP-masquerading NAT is that it has been a practical solution to the impending exhaustion of IPv4 address space.
PAT allows multiple internal hosts to share a single external IPs address.

Disadvantages of PAT

Scalability - Many hosts on the private network make many connections to the public network. Since there are only a limited number of ports available, the PAT device may eventually have insufficient space in the translation table. While there are thousands of ports available, and they are recycled quickly, some network communications consume multiple ports nearly simultaneously in a single logical transaction (an HTTP request for a web page with many embedded objects; some VoIP applications). Sufficiently-large LANs that frequently sustain this type of traffic could periodically consume all available ports.

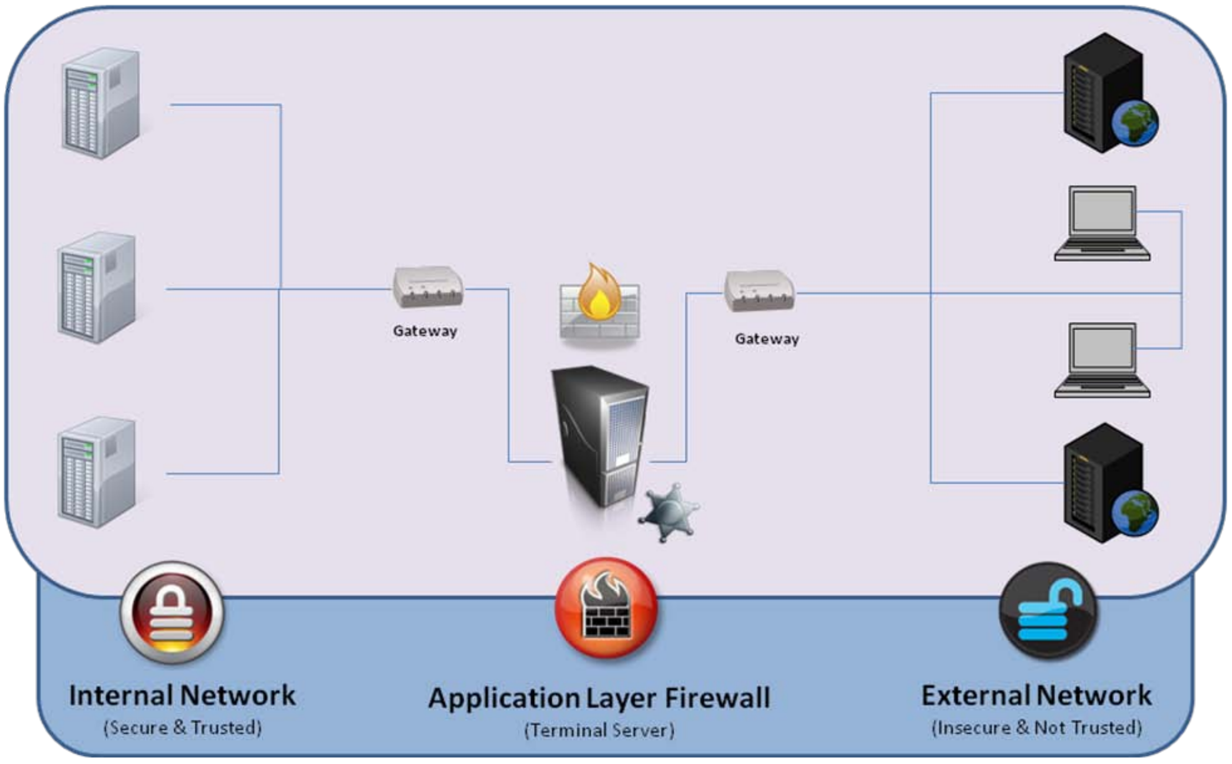
Firewall complexity - Because the inside addresses are all disguised behind one publicly-accessible address, it is impossible for outside machines to initiate a connection to a particular inside machine without special configuration on the firewall to forward connections to a particular port. This has a considerable impact upon applications such as VOIP, videoconferencing, and other peer-to-peer applications.

IMPLEMENTATION

Network Overview

The Application Layer Firewall was implemented on a Terminal Server that resided at an inter-connection of an Internal Network (Secure and Trusted) with an External Network (Insecure and Not Trusted). Gateways at the interconnections implemented PAT (Port Address Translation) to improve security by masking IP Addresses of the Internal and the External Networks from each other. The Internal Network consisted of servers that had Power Plan Automation software, that was monitoring the Power Plant operation. The External Network consisted of terminals that were on the Siemens Networks and on Client Networks with possible internet connections. The External Network provided a means to monitor and control the onsite Power Plant Automation Software remotely through VPN (Very Private Network) or via the internet. This however introduced vulnerabilities to the Internal Network thus necessitating the need for a Firewall.

The network employed SNMPv1 to perform the various tasks of managing and monitoring the system. SNMPv1 was chosen due to its simplicity and efficacy coupled with the fact that it could be easily implemented on hardware. But SNMPv1 lacks security features thus mandating a need for a Firewall.

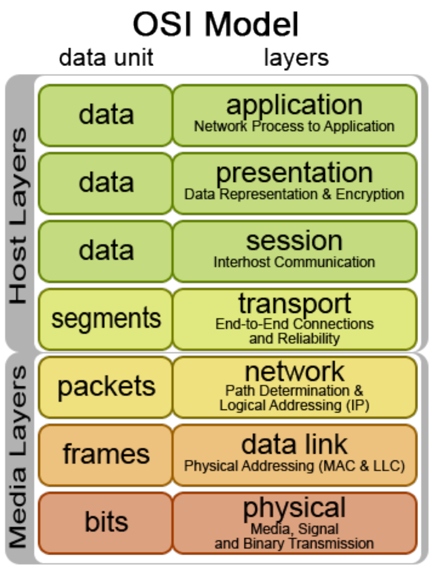


Platform

The Terminal Server was a Linux Box with a Debian based Operating System on it. This required that the Application Layer Firewall be able to run on Linux Machines. Thus the platform for development and deployment of the Application Layer Firewall was chosen to be Linux. Subsequently the entire project was adapted to a Linux environment and all development was carried out on a Linux Box.

IP Layer vs Application Layer Firewall

A Firewall for the network configuration and protocol (SNMP) could be implemented at various levels of the network model. However each has its own pros and cons.


OSI Model
data unit
layers
Host Layers
Media Layers
data
application
Network Process to Application
data
presentation
Data Representation & Encryption
data
session
Interhost Communication
segments
transport
End-to-End Connections and Reliability
packets
network
Path Determination & Logical Addressing (IP)
frames
data link
Physical Addressing (MAC & LLC)
bits
physical
Media, Signal and Binary Transmission

To retrieve Application layer (SNMP) packets at the each frame received from the data link layer has to be cleared from the various headers and tails included at various levels. Subsequently a single SNMP packet may be spread over many frames. As frames are of fixed length unlike Application layer packets the tracking of contents of each SNMP packet over many frames becomes cumbersome and susceptible to Buffer Overflow attacks, as the limited buffers may be made to overflowed by unnecessarily extending the packet over many frames. However traditional IP Layer firewalls (IP Filters) do not concern themselves with Application Layer content and treat them as a raw stream of bytes hence making them impervious to such Buffer Overflow Attacks.

Additionally data such as CRC and Checksums would have to be performed manually by the firewall. These would hamper the performance and efficiency of the firewall. At the IP layer the application would be put in a position where it would receive all the packets from all the other IP network protocols (TCP, HTML, NTP) as well. This would obscure the application with unnecessary and unwarranted packets with which it would have to deal with. This would adversely affect the speed and performance of the firewall.

As IP Layer is very close to the physical components it is handled by the kernel in the kernel space of the physical memory. To function in the IP layer the application would have to be programmed into the kernel space. This would hamper the efficiency of the Operating System's kernel itself. Also the application would have a serious constraint on memory and resources. Furthermore many standard library functions available with C would have to be re-implemented in kernel space in order to use them.

Net-SNMP

Net-SNMP is a suite of software for using and deploying the SNMP protocol (v1, v2c and v3 and the AgentX subagent protocol). It supports IPv4, IPv6, IPX, AAL5, Unix domain sockets and other transports. It contains a generic client library, a suite of command line applications, a highly extensible SNMP agent, perl modules and python modules. Net-SNMP is a complete and exhaustive package for SNMP. Net-SNMP being open-source is widely used by many networks world-wide. With a huge developer community Net-SNMP is efficient and safe. The command line utilities provided by the Net-SNMP enable users to implement SNMP.

Additionally, Net-SNMP package provides APIs to developers to utilize the modules provided in the package which are used by the command line utilities provided by the package as well. Various modules can be incorporated into separate applications by developers with the aid of API documentation provided on the internet. These modules only require the Net-SNMP shared libraries be installed on the local system and provide an efficient and easy way to implement various aspects of SNMP in custom applications.

Of the many modules available from the Net-SNMP package the SNMP parser was utilized. The Net-SNMP parser was found to be efficient, robust and secure. With the SNMP parser many aspects of programming the Application Proxy Firewall were simplified.

Design Criterion

1. Listening at Multiple ports: The Application Layer Firewall runs on the Terminal Server that is at the interconnection of the Internal Network and the External Network. To protect the Internal Network a Gateway Firewall was used to filter each packet and also to mask the internal IP addresses to provide additional security. To aid in masquerading the Internal Network, Port Address Translation was employed by the Gateway Firewall.

   The PAT implementation lead to IP address being mapped to corresponding port numbers based on a configuration file specified with the Gateway Firewall. Thus the Terminal server was required to receive the intended packets at multiple ports simultaneously. Hence to listen at multiple ports the Application Layer Firewall had to bind to all the ports and listen to all of them simultaneously. This complicacy was simplified by the use of ‘select()’ system call.

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
        fd set *exceptfds, struct timeval *timeout);
```

   The select() method can be a powerful tool when multiplexing network sockets. Specifically, the method will indicate when a procedure will be safe to execute on a socket port without any delays. For instance, a programmer can use these calls to know when there is data to be read on a socket. By delegating responsibility to select(), the program doesn't have to constantly check whether there is data to be read. Instead, select() can be placed in the background by the operating system and woken up when the event is satisfied or a specified timeout has elapsed. This process significantly increases execution efficiency of a program. The 'select' system call provides a mechanism to listen at multiple ports simultaneously. When a socket receives a packet, the system call returns and subsequently the socket ready with the packet to be read is identified from the pool of sockets being 'listened' by the 'select'. Once recognized the packet is read from the socket. In case multiple sockets received packets simultaneously the same process is applied to all the sockets.

```
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

2. Parsing SNMP Messages: As the Application Layer Firewall operates at the Application Layer it is responsible to read SNMP packets and interpret them independently. This is net necessary because Application Layer Firewall needs to apply certain filtering rules on the SNMP packets to identify spurious packets. This mandates the Application Layer Firewall to have a means of interpreting ie. Parsing the raw SNMP message received from the Transport Layer to a meaningful format and finally building a data structure to hold the SNMP Message. Parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. A parser breaks data into smaller elements, according to a set of rules that describe its structure. Most data can be decomposed to some degree. The SNMP Message is encoded in a predefined notation the Abstract Syntax Notation 1.

   However the ASN1 is a set of complex and elaborate notation rules. Additionally the parsers often require a separate lexical analyzer to create tokens from the sequence of input characters., the output of which is then syntactically analyzed followed by semantic analysis. Also the parser needs to be comprehensive enough to decode the SNMP message applying most of the ASN1 rules.

   Fortunately, as SNMP is a widely used Protocol across many networks and even the internet, many open source, robust, efficient and reliable SNMP parsers are available. Thus to optimize the Application Layer Firewall one such parser available in the Net-SNMP package was used.

```
//Encoding
int  snmp_build (u_char **pkt, size_t *pkt_len, size_t
*offset, netsnmp_session *pss, netsnmp_pdu *pdu)

//Decoding
int  snmp_parse (netsnmp_pdu *pdu, u_char *data, size_t
*length, u_char **after_header, netsnmp_session *sess)
```

However the API provided by the package was customized to cater to the needs of the Application Layer Firewall.

3. Speed: The Application Layer Protocol plays a pivotal role in the entire network. As it is at the interconnection of the Internal Network and the External Network all the SNMP traffic was routed through the Terminal Server running the Application Layer Firewall. The Application Layer Firewall was thus needed to be very fast so as to be able to handle the vast volumes of traffic at the interconnection. Thus the Terminal Server constantly received messages and had to subsequently forward them to their intended destination after applying the various Filtering Criterion on each and every packet. As many of these operations such as Parsing were time consuming it was imperative of the Terminal Server and Application Layer Firewall to be fast in processing and then forwarding the packet. If each packet were not to be processed quickly the internal buffers were to overflow and hence finally lead to dropping of packets.

   This situation required the Application Layer Firewall to be very fast in processing each packet and at the same time must have an internal buffer to store packets that have arrived and are waiting to be processed. Thus the need for an internal message queue and multithreading arose.

   A thread of execution results from a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in most cases, multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

   The Application Layer Firewall is thus a Multithreaded application. The process spawns threads for

   Listening at multiple ports
   Processing each packet

   One thread is dedicated to receive packets from the networks. This ensures that the packet buffers of the Terminal Server do not overflow and at the same time it decouples receiving packets from processing them as it splits these two processes in two different threads. By creating a different thread for processing packets, the time consuming functions of parsing and filter checks were independent of arrival of packets at different ports. This led to packet processing and listening at ports to be continuous processes. These threads share the process' resources but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a single process to enable parallel execution on a multiprocessor system. This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines because the threads of the program naturally lend themselves to truly concurrent execution.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
const pthread_attr_t *restrict attr,
void *(*start_routine)(void*), void *restrict arg);
```

On a single processor, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, the threads or tasks will generally run at the same time, with each processor or core running a particular thread or task. This means a processor is able to execute only one thread at a time, but a processor with multiple cores is able to run multiple threads. Fortunately most servers have processors with multiple cores with at least two if not more cores. This ensures the efficiency of the multithreaded scheme. Further, to utilize the all the cores of a server the Application Layer Firewall can be made to spawn multiple threads for packet processing for speeding up the packet processing packets.

Speed of the packet processing thread is of concern as the thread overburdened with a diverse set of tasks from parsing to applying firewall rules. This necessitates the need for every function of the packet processing be quick and efficient. For parsing an external component from the Net-SNMP package was used. This in itself was compact efficient and quick ensuring that the parsing was done as fast as possible. Other functions of the processing thread were applying firewall rules. These rules constantly required the thread to access data structured that housed the predefined permissions for each IP Address obtained from the configuration file. Thus to optimize and improve the speed of this look up that has to be performed for each packet multiple times these values stored in a hash table. The socket file descriptor was used as the hash key. As hashing obviates the need for searches the entire process of lookup is optimized. Furthermore keeping in mind that socket descriptors are allotted sequentially to a process a hash function with good avalanche property was to be employed at the same time the hash functions had to be as fast as possible to reduce overhead involved in computing the hash key. Thus keeping in mind these requirements the Robert Jetkins Hash Functions was selected and implemented. The Hash function operated on 32 bit data types and the socket file descriptors were also of the same size.

```
uint32_t hash( uint32_t a)
{  a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;}
```

4. <u>Ensuring a Thread Safety:</u> With multiple threads being employed for improving efficiency many issues concerning multi threaded applications come into the picture. Thread safety is a concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.

   Thread safety is a key challenge in multi-threaded programming. In a multi-threaded program, several threads execute simultaneously in a shared address space. Every thread has access to virtually all the memory of every other thread. Thus the flow of control and the sequence of accesses to data often have little relation to what would be reasonably expected by looking at the text of the program, violating the principle of least astonishment. Thread safety is a property aimed at minimizing surprising behavior by re-establishing some of the correspondences between the actual flow of control and the text of the program.

   It is not easy to determine if a piece of code is thread-safe or not. However, there are several factors that endanger thread safety of an applications some of them maybe accessing global variables or the heap, allocating/reallocating/freeing resources that have global limits (files, sub-processes, etc) and indirect accesses through handles or pointers
   However a subroutine is reentrant, and thus thread-safe, if the only variables it uses are from the stack, execution depends only on the arguments passed in, and the only subroutines it calls have the same properties.

   There are various methods to ensure safety albeit inclusion of one or more of risks.
   Re-entrancy: Writing code in such a way that it can be partially executed by one task, reentered by another task, and then resumed from the original task. This requires the saving of state information in variables local to each task, usually on its stack, instead of in static or global variables.
   Mutual exclusion: Access to shared data is serialized using mechanisms that ensure only one thread reads or writes the shared data at any time. Great care is required if a piece of code accesses multiple shared pieces of data—problems include race conditions, deadlocks, livelocks, starvation, and various other ills enumerated in many operating systems textbooks.
   Thread-local storage: Variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be reentrant.
   Atomic operations: Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter what other threads access it. Atomic operations form the basis of many thread locking mechanisms.

   Measure such as the use of only local storages was not possible because a mechanism had to exist for raw packets received from the listening thread to be given to the processing thread.

This was due to the need for a global message queue that acted as a buffer. Hence some data had to be shared among the threads. This fact necessitates the need for mutual exclusion.

Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code where a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have critical section in it without any mechanism or algorithm, which implements mutual exclusion.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

A mutex lock provided by the multithreading libraries in Linux was used to implement mutual exclusion among the threads. As the locking mechanism does not involve spin locks the processor is kept free as the waiting thread is put to sleep and awakened when the lock requested is released. Spin locks on the other hand constantly check the status of the lock in a infinite loop thus using up of processor resources.

5. <u>Incorporating Port Address Translation:</u> Another important feature of the network was the use of Port Address Translation for masking the IP Addresses of the terminals in the Internal Network to protect them from any direct attack. But this adds complications to the mechanism of packet forwarding at the Terminal Server.

   Two-way communication process for the SNMP at the Terminal Server had to be re-implemented to incorporate the PAT for common message types such as the GetRequest, GetNextRequest, SetRequest and the GetResponse. As Traps are unidirectional and not requiring two-way communication no modification was required to ensure proper operation for SNMP Traps. As the external terminals never directly address packets to their intended destination terminals for requests or responses for the requests, the packets must be modified to enable them to reach the actual intended destination and also is the intended order. Thus if a Request is send by a terminal on the Internal Network to a terminal in the External Network the request must be delivered to only that terminal and subsequently the response generated in consequence to the request must be delivered to the terminal where the request had originated from.

   PAT implemented on the Gateway Firewall modified the intended destination on each of the packets to the IP Address of the Terminal Server but to a port that indicated the intended destination. Thus for the Terminal Server the arrival of a packet at a specific port indicates its intended destination on the internal network. Thus after the packet has gone through the filtering process it is forwarded to the intended address. However as External Network is unable to see the internal IP Addresses in the Internal Network similarly the Internal Network terminals are unable to see external IP Addresses. Thus to keep track of requests and their responses, the Application Layer Firewall changes the Request ID on all incoming requests to any new request id which it keeps track in its internal state tables. This conversion helps to initiate a new request response communication between the Terminal Server and one of the Network, thus eliminating the direct communication among the Internal and External Networks. The new request ids generated are related to the old ones, against which they were generated to help determine the destination of the response packets. This change in request id meant that new ids generated needed to be unique. A combination of the old and new request id helps the Application Layer Firewall trace a path for each packet between the Internal Network and the External Network.

6. Configurability: A configuration is an arrangement of functional units according to their nature, number, and chief characteristics. Often, configuration pertains to the choice of hardware, software, firmware, and documentation and customization on their part. The configuration affects system function and performance. Configurability helps to achieve and incorporate flexibility in the system and also to provide the vital parameters and information for the various filtering checks that are done on each packet by the Application Layer Firewall. Application Layer Firewall had to be so designed so as to be configurable and to accommodate minor changes in the network. These changes could include changes IP Addresses of the Terminal Server, the change in PAT mapping.

   The configuration file was also used to provide vital security information to the Application Layer Firewall. These include types of messages to be restricted. Certain messages such as Traps or SetRequests could be used for malicious purposes. Thus eliminate any such threat these messages could be made to be dropped by Application Layer Firewall. However this being an extreme resort of sorts isn’t generally used.

   One of the most essential and vital information that is provided to the Application Layer Firewall by the configuration file is the individual permissions to each member in the network. With this each member’s packets are forwarded based on heir adherence to the permissions provided in the file. If any of the packets are found to violate any of them are simply dropped. This helps in providing special rights and restrictions at various terminals to ensure that secure terminals are entrusted with more rights and permissions than others. Such a differential scheme is however highly dynamic and thus mandating a flexibility which was provided by the use of configuration file.

   The configuration file also provided with certain useful information such as the number of cores in the Terminal Server processor assisting in determining the optimum number of threads to be spawned in order maximize efficiency and at the same time keeping overheads at a minimum. As stated above the processors ability to run a process concurrently is restricted by the number of cores it has such vital information on hardware can enable the Application Layer Firewall to optimize the hardware usage as much as possible.

Finally the configuration files also provided the Port Address Translation mappings of IP addresses to port numbers as being done by the Gateway Firewalls. This was one of the main reasons for the use of a configuration file. PAT mappings had to be specified in advance to the Application Layer Firewall in order for its proper operation.

7. Logs: A server log is a log file (or several files) automatically created and maintained by a server of activity performed by it. Logging enables programmers to diagnose faults and reasons in an event of a firewall crash. Logs provide vital information about the activities being performed by the firewall before its crash. The Application Layer Firewall maintains a log for all its activities that may help to reconstruct the sequence of events of events in an event of a crash. This may provide vital information to diagnose the reason for a crash. Inorder to keep the logs up to date the files buffers are periodically flushed at vulnerable intervals to improve the efficacy of the log content.

Filtering Rules

Data Integrity Check

Data integrity is a term used in computer science and telecommunications that can mean ensuring data is "whole" or complete, the condition in which data is identically maintained during any operation (such as transfer, storage or retrieval), the preservation of data for their intended use, or, relative to specified operations, the a priori expectation of data quality. Put simply, data integrity is the assurance that data is consistent and correct.

As the SNMP messages are encoded in a well defined and accurate notation (ASN1) any message or part of a message that does not abide by the notation is considered to be invalid and malformed. The Application Layer Firewall dropped any malformed packet. A packet is malformed if it fails the data integrity check ie the raw bytes are not encoded properly in the ASN1 notation. This check was enforced inherently during parsing. The parsing failed if any packet not following the ASN1 notation was passed to it. Thus if an error was returned by the parser it signified that packet did not follow ASN! Notation completely. No checks on the SNMP Message content are forced during parsing. All SNMP message content abiding the notation rules is considered to be valid and accepted.

Unsolicited responses

SNMP GetResponse plays an important role in SNMP. It is generated as a consequence of many other type of messages and carriers important information such as values of requested variable or error value. However a GetResponse can never be generated with a Request initiating it. Hence any unsolicited Resonses are considered to be threats. Thus the firewall maintains a state of all the pending requests in the recent past. Each Response packet that passes through the firewall is checked against this list of pending Requests. If a Request ID is found to match that of the incoming Response then the packet is forwarded and subsequently the Request ID is removed from the pending list.

Duplicate Requests

The SNMP dictates that each Request must have a unique Request ID. In an event of the same Request ID being used twice the SNMP would consider that the message is a duplicate message sent twice. Hence the message with the duplicate Request ID is dropped by the Application Layer Firewall. However in the event of the Request being serviced by a corresponding response a duplicate occurrence of the same Request ID would be considered as a new message and Request altogether. Thus by dropping duplicate Requests the traffic on the Networks is regulated to some extent.

Message type restricted on the network

Certain messages in the SNMP have an immense potential for misuse such as the SetRequest and the Trap. If the network finds any these to be of any use then the message types can be altogether be barred on the network. The Application Layer Firewall will drop any of the messages of these types. However such a provision is an extreme resort as it restricts the functionality of the SNMP itself.

Specific permissions to individual IPs

Each IP Address on either network can be restricted to sending or receiving certain message types only. This is done by granting individual permissions to each IP Address on the network. Thus each message whenever forwarded is checked for its source IP Address and destination IP addresses to see if they are permitted to send or receive such messages. This helps in protecting vulnerable and vital terminals from being targeted and at the same time provides the functionalities of the other terminals onto the network. This scheme is pivotal to the functioning of the Application Layer Firewall. Such a scheme is however dynamic and is provided to the Application Layer Firewall by the configuration file. In the case no restrictions are specified for an IP Address that IP Address is allowed to send and receive all the message types allowed on the network. With such measures in place the efficacy of the Application Layer Firewall is greatly increased.

Rate Check

In the network certain spurious terminals might try to overload the network by sending by constantly sending messages at a high rate. As the Application Layer Firewall would have to buffer all of these messages a constant proliferation of SNMP messages on the network would mandate a very large buffer on the Terminal Server and hence on the Application Layer Firewall. But as known from the previous estimates about the network that the traffic may not exceed under any condition above a certain limit the proliferation of SNMP traffic can be restrained by dropping the SNMP packets originating from the spurious IP Addresses for a specified amount of time. This prevents the network from being overloaded by potentially malicious programs and also reduces the chances of a Buffer Overflow on the Terminal Server. Also with such rate checks in place the chances of Application Layer Firewall crashing in high traffic situations also reduces. As traffic originating from each IP is constantly monitored spurious elements elevating the traffic can also be identified by the network administrators.

SIEMENS

SOURCE CODE

File : app-poxy.c

```
/*
 * APP-PROXY : An Application Layer Proxy Firewall for SNMP v1
Protocol
 */

#include<sys/socket.h>
#include<netinet/in.h>
#include<stdlib.h>
#include<sys/un.h>
#include<sys/time.h>
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include <net-snmp/net-snmp-config.h>
#include <net-snmp/net-snmp-includes.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>


// GLOBAL DEFINITIONS

#define MAXLINE 4096
#define MAXPORTSUPPORT 256
#define BARR_LENGTH 6
#define SNMPTRAP 162
#define SNMPNONTRAP 161

// STRUCTURES

struct msg_buffer
{
   u_char pkt[MAXLINE];
   int length;
   int sock;
   struct msg_buffer *next;
   struct msg_buffer *prev;
};
```

```
struct reqid_list
{
   long reqid;
   long myreqid;
   struct reqid_list *next;
};

struct hash_ip
{
   struct in_addr ip;
   //int16_t port;
   int sock;
   struct hash_ip *next;
};

typedef struct hash_ip hash_map;
typedef struct reqid_list id_q;
typedef struct msg_buffer rd_q;

// FUNCTIONS PROTOTYPES

void *rd_service();
int push_rd_q(int);
void end(int);
netsnmp_pdu* parse(u_char*,size_t );
int assess(netsnmp_pdu*,int);
int barred(netsnmp_pdu*);
int config_init(void);
int binder(int,char*,int16_t);
int externmapper(int,struct in_addr);
int internmapper(int,struct in_addr);
int forward(char*,int,struct in_addr,int);
long genmyreqid();

// GLOBAL VARIABLES

//struct sockaddr_in servaddr[2];
int *sockfd,sock_no;
int barr[BARR_LENGTH];
FILE *log;
char log_str[50];
//char *community;
pthread_t rd_thread;
//sem_t rd_mutex;
pthread_mutex_t rd_mutex;
rd_q *rd_front,*rd_rear;
id_q *id_head;
```

```
hash_map *internmap,*externmap;



// C CODE

void initialize()
{
    int i,er;
    //Binding and Address allocation

    printf("Initializing");

    //OPEN LOGFILE
    log=fopen("/var/log/applog.log","w");
    if(log == NULL)
    {
         printf("Log file cannot be opened");
         end(-1);
    }
    fputs("Logging Enabled \n",log);

    //ENABLING SIGNAL HANDLER
    signal(SIGINT,end);

    //INITIALIZE VARIABLES

    rd_rear = NULL;
    rd_front = NULL;
    id_head = NULL;
    sock_no = 0;

    //READING CONFIG FILE
    if(config_init())
    {
         fputs("Configuration File Error",log);
         end(-1);
    }

    // SPAWN NEW READ THREAD

    pthread_create(&rd_thread,NULL,rd_service,NULL);

    // INITIALIZE PTHREAD MUTEX

    pthread_mutex_init(&rd_mutex,NULL);
    //rd_mutex = PTHREAD_MUTEX_INITIALIZER;
    fflush(log);

}



void *rd_service()
{
    rd_q *delnode;
    u_char msg[MAXLINE];
    int ctr,len;
    //int16_t port;
    int sock;
    netsnmp_pdu *status;

    printf("Read Service thread running");

    while(1)
    {
        //sem_wait(&rd_mutex); //CRITICAL SECTION BEGINS
        pthread_mutex_lock(&rd_mutex);

        if(rd_front == NULL && rd_rear == NULL)
        {
            pthread_mutex_unlock(&rd_mutex);
            //sem_post(&rd_mutex); //CRITICAL SECTION ENDS
            continue;
        }
        else
        {
            memset(msg,0,MAXLINE);

            delnode=rd_front;
            for(ctr=0;ctr<delnode->length;ctr++)
                msg[ctr]=delnode->pkt[ctr];
            len = delnode->length;
            //port = delnode->port;
            sock = delnode->sock;
            if(rd_front == rd_rear)
            {
                rd_front = NULL;
                rd_rear = NULL;

            }
            else
            {
                    rd_front = delnode->next;
                    //free(delnode);
               }
               free(delnode);
               delnode = NULL;
               pthread_mutex_unlock(&rd_mutex);
               //sem_post(&rd_mutex); //CRITICAL SECTION ENDS
          }

          status=parse(msg,len);
          if(status)
          {
               fputs("\nIntegrity Check Passed : ",log);
               if(!barred(status))
                    if(assess(status,sock))
                         fputs(" Droping packet\n",log);
               free(status);
          }
          else
          {
               fputs("\nIntegrity Check Failed : Droping
packet\n",log);
          }
          fflush(log);

     }


}


int push_rd_q(int sockfd)
{
  rd_q *q_node;
  //char msg[MAXLINE];

  int len;

  q_node=(rd_q *)malloc(sizeof(rd_q));
  if(q_node == NULL)
  {
       fputs("Malloc failed (push_rd_q)",log);
       return -1;
  }
```

```
    len=recv(sockfd,q_node->pkt,MAXLINE,0);
    q_node->length=len+1;
    q_node->sock=sockfd;

    q_node->next = NULL;
    q_node->prev = NULL;

    //ADDING NODE TO RD_Q LINKLIST

    //sem_wait(&rd_mutex); //CRITICAL SECTION BEGINS
    pthread_mutex_lock(&rd_mutex);

    if(rd_front == NULL && rd_rear ==NULL)
    {
         //rd_front = q_node;
         //rd_rear = q_node;
         rd_front = q_node;
         rd_rear = q_node;
    }
    else
    {
         //rd_rear->next=q_node;
         //rd_rear=q_node;
         q_node->prev = rd_rear;
         rd_rear->next = q_node;
         rd_rear = q_node;
    }

    pthread_mutex_unlock(&rd_mutex);
    //sem_post(&rd_mutex); //CRITICAL SECTION ENDS

    return 0;
}

int main()
{
    int i=0,rd_sock;
    struct timeval timeout;
    fd_set socks;

    initialize();

    printf("\nmain ");

    for(;;)
    {
         FD_ZERO(&socks);

         //FD_SET(sockfd[0],&socks);
         for(i=0;i<sock_no-1;i++)
              FD_SET(sockfd[i],&socks);



         timeout.tv_sec=1;
         timeout.tv_usec=0;

         rd_sock=select(sockfd[sock_no-1]+1,&socks,(fd_set
*)0,(fd_set *)0,NULL);
         //printf("Select returned");

         if(rd_sock<0)
         {
              fputs("Select Error",log);
              end(-1);
         }
         if(rd_sock==0)
              fputs(".",log);
         else
         {
              for(i=0;i<sock_no-1;i++)
              {
                   if(FD_ISSET(sockfd[i],&socks))
                   {

                        if(push_rd_q(sockfd[i]))
                             fputs("Cannot be serviced\n",log);

                   }
                   else
                        fputs(".",log);
              }
         }

         fflush(log);
    }

    return 0;
}

void end(int value)
{
    //sem_destroy(&rd_mutex);
```

```
    pthread_mutex_destroy(&rd_mutex);
    fclose(log);
    //free(sockfd);
    exit(value);
}

/////////////////////////////


//size_t newpktlen=0,offset=0;
//u_char *newpkt;
//size_t pktlen=46;

netsnmp_pdu* parse(u_char *pkt,size_t pktlen)
{

    size_t newpktlen=0,offset=0;
    u_char *newpkt;
    netsnmp_session session;
    netsnmp_pdu *pdu;
    netsnmp_variable_list *vars;
    int status=45,ctr,i;

    newpktlen=0;
    offset=0;

    pdu = (netsnmp_pdu *) malloc(sizeof(netsnmp_pdu));
         if (pdu)
    {
              //pdu->version = SNMP_VERSION_1;
              //pdu->command = SNMP_MSG_RESPONSE;
              //pdu->errstat = SNMP_DEFAULT_ERRSTAT;
              //pdu->errindex = SNMP_DEFAULT_ERRINDEX;
              pdu->securityModel = SNMP_DEFAULT_SECMODEL;
              pdu->transport_data = NULL;
              pdu->transport_data_length = 0;
              pdu->securityNameLen = 0;
              pdu->contextNameLen = 0;
              pdu->time = 0;
              pdu->reqid = snmp_get_next_reqid();
              pdu->msgid = snmp_get_next_msgid();
         }
    else
    {
         sprintf(log_str,"Malloc Failed (parse)");
         fputs(log_str,log);
         return NULL;
    }
    //pkt=msg;
    //pkt_len=msg_len;

    printf("Message before parsing\n");
    for(ctr=0;ctr<(int)pktlen;ctr++)
         printf("%x",pkt[ctr]);
    printf("\n\n");

    status=snmp_parse(&session,pdu,pkt,pktlen);
    if(status != 0)
         return (NULL);

    printf("\n\n");
    printf("Status %d",status);


    for(vars = pdu->variables; vars; vars = vars->next_variable)
          {
         print_variable(vars->name, vars->name_length, vars);

    }
    return (pdu);

    //--THIS PART DOES NOT EXECUTE--

         printf("pdu:\nversion:%ld\ncommand:%d\n",pdu->version,pdu-
  >command);


    //printf("Decoding done. \n ");
    printf("Decoding done. \n Now encoding");

    //return (pdu);

    status=snmp_build(&newpkt,&newpktlen,&offset,&session,pdu);
    printf("\nstatus = %d\n",status);
    printf("\nnewpktlen = %d\n",newpktlen);
    printf("\noffset = %d\n",offset);
    if(status==0)
    {
         //newpktlen=sizeof(newpkt);
         //printf("seems successful\n Here's the new packet:\npktlen
  :%d\n",newpktlen);
         printf("Packet dump : \n");
         for(ctr=newpktlen-offset;ctr<newpktlen;ctr++)
         {
                printf("%X",newpkt[ctr]);
            }
            printf("\nPacket dump - END \n");
            for(ctr=newpktlen-offset,i=0;i<46 &&
ctr<newpktlen;i++,ctr++)
            {
                if(newpkt[ctr]!=pkt[i])
                {
                    printf("\nMISMATCH");
                    return(NULL);
                }
            }
            printf("\nMATCH!!!!\n Parse successful\n");
            free(newpkt);
            newpkt = NULL;
            return pdu;

        }
        else
        {
            printf("Parse failed");
            //free(newpkt);
            return (NULL);
        }

}


/////////////////////////////

int forward(char *pkt,int pkt_len, struct in_addr ip,int command)
{
    struct sockaddr_in servaddr;
    int sockfd;

    memset(&servaddr,0,sizeof(servaddr));

    servaddr.sin_family = PF_INET;

    if(command == SNMP_MSG_TRAP)
        servaddr.sin_port = htons(SNMPTRAP);
    else
        servaddr.sin_port = htons(SNMPNONTRAP);

    servaddr.sin_addr = ip;

    if((sockfd=socket(PF_INET,SOCK_DGRAM,0))<0)
    {
         fputs("Socket Error\n",log);
         return(-1);
    }

    if(sendto(sockfd,pkt,pkt_len,0,(struct sockaddr
*)&servaddr,sizeof(servaddr)) == -1)
    {
         fputs("Send failed",log);
         return -1;
    }

    return 0;
}

int add_request(long reqid)
{
    id_q *newnode;

    newnode=(id_q *)malloc(sizeof(id_q));
    if(newnode == NULL)
    {
         fputs("Malloc failed (add_request)",log);
         return -1;
    }

    newnode->next=NULL;
    newnode->reqid=reqid;
    newnode->myreqid = genmyreqid();

    if(id_head == NULL )
    {
         id_head = newnode;
    }
    else
    {
         newnode->next = id_head;
         id_head = newnode;
    }

    return 0;

}

long chk_response(long reqid) //Check against myreqid
{
    id_q *nownode,*prevnode;
```

```
   nownode = id_head;
   prevnode = id_head;
   for(;nownode != NULL;nownode=nownode->next)
   {
        if(reqid == nownode->myreqid)
        {
             prevnode->next = nownode->next;
             free(nownode);
             return nownode->reqid;   //FOUND
        }
        prevnode=nownode;
   }

   return 0;           //NOT FOUND

}

int chk_request(long reqid) //Check against reqid
{
   id_q *nownode;

   nownode = id_head;
   for(;nownode != NULL;nownode=nownode->next)
   {
        if(reqid == nownode->reqid)
        {
             return 0; //FOUND
        }
   }

   return -1;               //NOT FOUND

}

int resend_packet(netsnmp_pdu *pdu,int sock)
{
   int status,ctr,i;
   size_t newpktlen=0,offset=0;
   u_char *newpkt,msg[MAXLINE];
   netsnmp_session session;
   struct in_addr ipaddr;

   newpktlen=0;
   offset=0;

   fputs("Forwarding packet : ",log);
```

```
    printf("\n ReqID : %ld", pdu->reqid);
    status=snmp_build(&newpkt,&newpktlen,&offset,&session,pdu);
    if(status==0)
    {
         fputs(" Encoding Successfull \n",log);
         fputs("Packet dump : \n",log);
         for(ctr=newpktlen-offset;ctr<newpktlen;ctr++)
         {
              msg[ctr+offset-newpktlen] = newpkt[ctr];
              sprintf(log_str,"%X",newpkt[ctr]);
              fputs(log_str,log);
         }
         fputs("\nPacket dump - END \n",log);
         //free(newpkt);

         //GET IPADDR
         if(retreive(sock,&ipaddr))
         {
              fputs("Socket mapping retreival failed",log);
              return -1;
         }
         if(forward(msg,offset+1,ipaddr,pdu->command))
         {
              fputs(" Message Forwarding failed",log);
              return -1;
         }


         return 0;
    }
    else
    {
         fputs("Encoding Failed",log);
         //free(newpkt);
         return -1;

    }

  }

  int assess(netsnmp_pdu *pdu,int sock)
  {
    long chngid;

    if(pdu->command == 160)  //GET
    {
         if(chk_request(pdu->reqid))
```

```
          {
               if (add_request(pdu->reqid))
                    return (-1);
               else
                    return (resend_packet(pdu,sock));
          }
          else
          {
               sprintf(log_str," Duplicate Request : Request Id :
%ld",pdu->reqid);
               fputs(log_str,log);
               return (-1);
          }
     }
     else if(pdu->command == 162) //GETRESPONSE
     {
          chngid = chk_response(pdu->reqid);
          if(chngid)
          {
               pdu->reqid = chngid;
               return resend_packet(pdu,sock);
          }
          else
          {
               sprintf(log_str," Unsolicited Respose : Request Id :
%ld",pdu->reqid);
               fputs(log_str,log);
               return (-1);
          }
     }
     return 0;
}

int barred(netsnmp_pdu *pdu)
{
     int ctr;
     for(ctr=0;ctr<10;ctr++)
     {
          if(pdu->command == barr[ctr])
          {
               sprintf(log_str," Message Type %d has been Barred :
Dropping Packet \n",pdu->command);
               fputs(log_str,log);
               return -1;
          }
     }
     //fputs(pdu->community,log);
```

```
    return 0;
}

int config_init(void)
{
    char str1[20],str2[20],config_str[50];
    int num,ctr=0,read;
    int16_t port;
    FILE *config;
    int *tmp;

    if(mapper_init())
    {
         fputs("Mapper Failed",log);
         return -1;
    }

    config = fopen("/etc/appconfig.conf","r");
    if(config == NULL)
    {
         fputs("Config file not found\n",log);
         fclose(config);
         return -1;
    }
    else
    {
         fputs("Config file found\n",log);
         sock_no = 1;
         sockfd = (int*)malloc(sizeof(int)*sock_no);
         sockfd[0]=0;
         while(!feof(config))
         {
              fgets(config_str,sizeof(config_str),config);
              //str=strstr(config_str,"restrict");

              read = sscanf(config_str,"%s %d",str1,&num);
              if(strcmp(str1,"restrict") == 0 && read == 2 && ctr <
BARR_LENGTH)
              {
                   barr[ctr++]=num;
                   sprintf(log_str,"Restricting Message type :
%d\n",num);
                   fputs(log_str,log);
              }

              read = sscanf(config_str,"%s %s",str1,str2);
```

```
            if(strcmp(str1,"internip") == 0 && read == 2 &&
binder(0,str2,0))
            {
                 fputs(" Internal IP Address invalid",log);
                 fclose(config);
                 return -1;
            }
            if(strcmp(str1,"externip") == 0 && read == 2 &&
binder(1,str2,0))
            {
                 fputs(" External IP Address invalid",log);
                 fclose(config);
                 return -1;
            }

            read = sscanf(config_str,"%s %d %s",str1,&port,str2);
            if(strcmp(str1,"internmap") == 0 && read == 3 &&
binder(2,str2,port))
            {
                 sprintf(log_str," Internal Bind : port %d to %s
failed\n",port,str2);
                 fputs(log_str,log);
                 fclose(config);
                 return -1;
            }
            if(strcmp(str1,"externmap") == 0 && read == 3 &&
binder(3,str2,port))
            {
                 sprintf(log_str," External Bind : port %d to %s
failed\n",port,str2);
                 fputs(log_str,log);
                 fclose(config);
                 return -1;
            }

       }
       fclose(config);
       return 0;
  }

}
int sock_add(int sock,int16_t port)
{
  //sockfd = (int*)realloc(sockfd,++sock_no*(sizeof(int)))
  if(realloc(sockfd,(++sock_no)*sizeof(int)) == NULL)
  {
       fputs("Realloc failed (config_init)",log);
		return -1;
	}
	else
	{
		sockfd[sock_no-1] = sockfd[sock_no-2];
		sockfd[sock_no-2] = sock;

		if(port>sockfd[sock_no-1])
			sockfd[sock_no-1] = port;
	}

	return 0;
}

int sock_bind(struct sockaddr_in servaddr, int16_t port)
{
	//struct sockaddr_in servaddr;
	int sockfd;
	//struct in_addr ipaddr;

	servaddr.sin_port = htons(port);
	if((sockfd=socket(PF_INET,SOCK_DGRAM,0))<0)
	{
		fputs("Socket Error\n",log);
		return(-1);
	}

	if(bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)) !=
0)
	{
		fputs("Bind Error\n",log);
		return(-1);
	}

	if(sock_add(sockfd,port) != 0)
	{
		fputs("Socket Update error",log);
		return (-1);
	}

	return sockfd;

}

int binder(int func,char *ipstr,int16_t port)
{
	static struct sockaddr_in internaddr,externaddr;
```

```
    struct in_addr ipaddr;
    int sockfd;
    //memset(&internaddr,0,sizeof(internaddr));
    //memset(&externaddr,0,sizeof(externaddr));
    internaddr.sin_family = PF_INET;
    externaddr.sin_family = PF_INET;

    switch(func)
    {
         case 0:
              if(inet_aton(ipstr,&ipaddr) == 0)
                   return -1;
              else
                   internaddr.sin_addr = ipaddr;
         break;

         case 1:
              if(inet_aton(ipstr,&ipaddr) == 0)
                   return -1;
              else
                   externaddr.sin_addr = ipaddr;
         break;

         case 2:
              sockfd = sock_bind(internaddr,port);
              if(sockfd == -1)
                   return -1;
              else
                   if(inet_aton(ipstr,&ipaddr) == 0)
                   {    sprintf(log_str,"Intern IP Address map : %s
  invalid",ipstr);
                        fputs(log_str,log);
                        return -1;
                   }
                   else
                        if(internmapper(sockfd,ipaddr))
                             return -1;
         break;

         case 3:
              sockfd = sock_bind(externaddr,port);
              if(sockfd== -1)
                   return -1;
              else
                   if(inet_aton(ipstr,&ipaddr) == 0)
                   {    sprintf(log_str,"Extern IP Address map : %s
  invalid",ipstr);
```

```
                          fputs(log_str,log);
                          return -1;
                     }
                     else
                          if(externmapper(sockfd,ipaddr))
                               return -1;
           break;

           default:
                return -1;
           break;
      }
      return 0;
  }

  uint32_t hash(uint32_t key) //ROBERT JETKINS HASH FUNCTION
  {
      key =  (key+0x7ed55d16) + (key<<12);
      key =  (key^0xc761c23c) + (key>>19);
      key =  (key+0x165667b1) + (key<<5);
      key =  (key+0xd3a2646c) + (key<<9);
      key =  (key+0xfd7046c5) + (key<<3);
      key =  (key+0xb55a4f09) + (key>>16);

      key = key % MAXPORTSUPPORT;
      return key;
  }

  int mapper_init()
  {
      externmap = (hash_map*)malloc(sizeof(hash_map)*MAXPORTSUPPORT);
      if(externmap == NULL)
      {
           fputs("Malloc failed (mapper_init)",log);
           return -1;
      }

      internmap = (hash_map*)malloc(sizeof(hash_map)*MAXPORTSUPPORT);
      if(internmap == NULL)
      {
           fputs("Malloc failed (mapper_init)",log);
           return -1;
      }
      return 0;
  }

  int externmapper(int sock,struct in_addr ipaddr)
```

```
{
   uint32_t key;
   static int collide_no;
   hash_map *collide,*ptr;

   key = sock;
   key = hash(key) % MAXPORTSUPPORT;

   if(externmap[key].sock == 0)
   {
        externmap[key].ip = ipaddr;
        //externmap[key].port = port;
        externmap[key].sock = sock;
        externmap[key].next = NULL;
   }
   else
   {
        collide = (hash_map*)malloc(sizeof(hash_map));
        if(collide == NULL)
        {
             fputs("Malooc failed (mapper)",log);
             return -1;
        }
        for(ptr = &externmap[key];ptr->next != NULL;ptr = ptr-
>next);

        ptr->next = collide;

        collide->ip = ipaddr;
        //collide->port = port;
        collide->sock = sock;
        collide->next = NULL;
   }


   return 0;
}

int internmapper(int sock,struct in_addr ipaddr)
{
   uint32_t key;
   static int collide_no;
   hash_map *collide,*ptr;

   key = sock;
   key = hash(key);
```

```
     if(internmap[key].sock == 0)
     {
          internmap[key].ip = ipaddr;
          //internmap[key].port = port;
          internmap[key].sock = sock;
          internmap[key].next = NULL;
     }
     else
     {
          collide = (hash_map*)malloc(sizeof(hash_map));
          if(collide == NULL)
          {
               fputs("Malooc failed (mapper)",log);
               return -1;
          }
          for(ptr = &internmap[key];ptr->next != NULL;ptr = ptr-
>next);

          ptr->next = collide;

          collide->ip = ipaddr;
          //collide->port = port;
          collide->sock = sock;
          collide->next = NULL;
     }


     return 0;
}
int retreive(int sock, struct in_addr *ip)
{
     int key;
     struct in_addr ipaddr;
     hash_map *node;

     key = hash(sock);
     node = &internmap[key];

     do{
          if(sock == node->sock)
          {
               *ip =  node->ip;
               return 0;
          }
          else
               node = node->next;
```

```
    }while(node != NULL);


    node = &externmap[key];
    do{
         if(sock == node->sock)
         {
              *ip = node->ip;
              return 0;
         }
         else
              node = node->next;

    }while(node != NULL);

    return -1;
}

long genmyreqid()
{
    static myreqid = 0;
    return (++myreqid);
}

/////////////////END OF PROGRAM//////////////////////////
```

File: /etc/appconfig.conf

```
restrict 164

thread 3

internip 127.0.1.1
externip 127.0.2.1

internmap 2000 127.0.1.10
internmap 2001 127.0.1.11
internmap 2002 127.0.1.12
internmap 2003 127.0.1.13
internmap 2004 127.0.1.14

externmap 3000 127.0.2.20
externmap 3001 127.0.2.21
externmap 3002 127.0.2.22
externmap 3003 127.0.2.23
externmap 3004 127.0.2.24
```

File: /var/log/applog.log

```
Logging Enabled
Config file found
Resticting Message Type : 164
..
Integrity Check Failed : Dropping Packet
..
Integrity Check Passed : Forwarding Packet : Encoding Successful
Packet dump :

Packet dump -END
..
Integrity Check Failed : Dropping Packet
..
Integrity Check Passed : Duplicate Request : Request ID : 1
Dropping Packet
```

File: ~/net-snmp-5.3.3/snmplib/snmp-api.c

```
static int
_snmp_build(u_char ** pkt, size_t * pkt_len, size_t * offset,
            netsnmp_session * session, netsnmp_pdu *pdu)
{
#if !defined(DISABLE_SNMPV1) || !defined(DISABLE_SNMPV2C)
    u_char          *h0e = 0;
    size_t           start_offset = *offset;
    long             version;
    int              rc = 0;
#endif /* support for community based SNMP */

    u_char          *h0, *h1;
    u_char          *cp;
    size_t           length;

    //session->s_snmp_errno = 0;
    //session->s_errno = 0;

    if (pdu->version != SNMP_VERSION_1) {
   printf("SNMPvany_other_than_1  bad idea");
   return -2;
   //return snmpv3_build(pkt, pkt_len, offset, session, pdu);
    }

    switch (pdu->command) {
    case SNMP_MSG_RESPONSE:
        netsnmp_assert(0 == (pdu->flags &
UCD_MSG_FLAG_EXPECT_RESPONSE));
        /*
         * Fallthrough
         */
    case SNMP_MSG_GET:
    case SNMP_MSG_GETNEXT:
    case SNMP_MSG_SET:
        /*
         * all versions support these PDU types
         */
        /*
         * initialize defaulted PDU fields
         */

        if (pdu->errstat == SNMP_DEFAULT_ERRSTAT)
            pdu->errstat = 0;
        if (pdu->errindex == SNMP_DEFAULT_ERRINDEX)
            pdu->errindex = 0;
        break;

    case SNMP_MSG_TRAP2:
        netsnmp_assert(0 == (pdu->flags &
UCD_MSG_FLAG_EXPECT_RESPONSE));
        /*
         * Fallthrough
         */
    case SNMP_MSG_INFORM:
#ifndef DISABLE_SNMPV1
        /*
         * not supported in SNMPv1 and SNMPsec
         */
        if (pdu->version == SNMP_VERSION_1) {
            //session->s_snmp_errno = SNMPERR_V2_IN_V1;
            return -2;
        }
#endif
        if (pdu->errstat == SNMP_DEFAULT_ERRSTAT)
            pdu->errstat = 0;
        if (pdu->errindex == SNMP_DEFAULT_ERRINDEX)
            pdu->errindex = 0;
        break;

    case SNMP_MSG_GETBULK:
        /*
         * not supported in SNMPv1 and SNMPsec
```

```
         */
#ifndef DISABLE_SNMPV1
        if (pdu->version == SNMP_VERSION_1) {
            //session->s_snmp_errno = SNMPERR_V2_IN_V1;
            return -3;
        }
#endif
        if (pdu->max_repetitions < 0) {
            //session->s_snmp_errno = SNMPERR_BAD_REPETITIONS;
            return -4;
        }
        if (pdu->non_repeaters < 0) {
            //session->s_snmp_errno = SNMPERR_BAD_REPEATERS;
            return -5;
        }
        break;

    case SNMP_MSG_TRAP:
        /*
         * *only* supported in SNMPv1 and SNMPsec
         */
#ifndef DISABLE_SNMPV1
        if (pdu->version != SNMP_VERSION_1) {
            //session->s_snmp_errno = SNMPERR_V1_IN_V2;
            return -6;
        }
#endif
        /*
         * initialize defaulted Trap PDU fields
         */
        pdu->reqid = 1;         /* give a bogus non-error reqid for
traps */
        if (pdu->enterprise_length ==
SNMP_DEFAULT_ENTERPRISE_LENGTH) {
            pdu->enterprise = (oid *)
malloc(sizeof(DEFAULT_ENTERPRISE));
            if (pdu->enterprise == NULL) {
                //session->s_snmp_errno = SNMPERR_MALLOC;
       printf("malloc error");
                return -1;
            }
            memmove(pdu->enterprise, DEFAULT_ENTERPRISE,
                    sizeof(DEFAULT_ENTERPRISE));
            pdu->enterprise_length =
                sizeof(DEFAULT_ENTERPRISE) / sizeof(oid);
        }
        if (pdu->time == SNMP_DEFAULT_TIME)
```

```
            pdu->time = DEFAULT_TIME;
        /*
         * don't expect a response
         */
        pdu->flags &= (~UCD_MSG_FLAG_EXPECT_RESPONSE);
        break;

    case SNMP_MSG_REPORT:      /* SNMPv3 only */
    default:
        //session->s_snmp_errno = SNMPERR_UNKNOWN_PDU;
        return -7;
    }

    /*
     * save length
     */
    length = *pkt_len;

    /*
     * setup administrative fields based on version
     */
    /*
     * build the message wrapper and all the administrative fields
     * upto the PDU sequence
     * (note that actual length of message will be inserted later)
     */
    h0 = *pkt;
    switch (pdu->version) {
#ifndef DISABLE_SNMPV1
    case SNMP_VERSION_1:
#endif
#ifndef DISABLE_SNMPV2C
    case SNMP_VERSION_2c:
#endif
#if !defined(DISABLE_SNMPV1) || !defined(DISABLE_SNMPV2C)
#ifdef NO_ZEROLENGTH_COMMUNITY
        if (pdu->community_len == 0) {
            if (session->community_len == 0) {
                //session->s_snmp_errno = SNMPERR_BAD_COMMUNITY;
                return -8;
            }
            pdu->community = (u_char *) malloc(session-
>community_len);
            if (pdu->community == NULL) {
                //session->s_snmp_errno = SNMPERR_MALLOC;
                return -9;
            }
            memmove(pdu->community,
                    session->community, session->community_len);
            pdu->community_len = session->community_len;
        }
#else                           /* !NO_ZEROLENGTH_COMMUNITY */
        if (pdu->community_len == 0 && pdu->command !=
SNMP_MSG_RESPONSE) {
            /*
             * copy session community exactly to pdu community
             */
            if (0 == session->community_len) {
                SNMP_FREE(pdu->community);
                pdu->community = NULL;
            } else if (pdu->community_len == session-
>community_len) {
                memmove(pdu->community,
                        session->community, session-
>community_len);
            } else {
                SNMP_FREE(pdu->community);
                pdu->community = (u_char *) malloc(session-
>community_len);
                if (pdu->community == NULL) {
                    //session->s_snmp_errno = SNMPERR_MALLOC;
             printf("malloc error2");
                    return -1;
                }
                memmove(pdu->community,
                        session->community, session-
>community_len);
            }
            pdu->community_len = session->community_len;
        }
#endif                          /* !NO_ZEROLENGTH_COMMUNITY */

        DEBUGMSGTL(("snmp_send", "Building SNMPv%d message...\n",
                    (1 + pdu->version)));
#ifdef USE_REVERSE_ASNENCODING
        if (1) {
            DEBUGPRINTPDUTYPE("send", pdu->command);
            rc = snmp_pdu_realloc_rbuild(pkt, pkt_len, offset,
pdu);
            if (rc == 0) {
                return -10;
            }

            DEBUGDUMPHEADER("send", "Community String");
            rc = asn_realloc_rbuild_string(pkt, pkt_len, offset, 1,
                                           (u_char) (ASN_UNIVERSAL
|
                                                     ASN_PRIMITIVE
|

ASN_OCTET_STR),
                                           pdu->community,
                                           pdu->community_len);
            DEBUGINDENTLESS();
            if (rc == 0) {
                return -11;
            }


            /*
             * Store the version field.
             */
            DEBUGDUMPHEADER("send", "SNMP Version Number");

            version = pdu->version;
            rc = asn_realloc_rbuild_int(pkt, pkt_len, offset, 1,
                                        (u_char) (ASN_UNIVERSAL |
                                                  ASN_PRIMITIVE |
                                                  ASN_INTEGER),
                                        (long *) &version,
                                        sizeof(version));
            DEBUGINDENTLESS();
            if (rc == 0) {
                return -12;
            }

            /*
             * Build the final sequence.
             */
#ifndef DISABLE_SNMPV1
            if (pdu->version == SNMP_VERSION_1) {
                DEBUGDUMPSECTION("send", "SNMPv1 Message");
            } else {
#endif
                DEBUGDUMPSECTION("send", "SNMPv2c Message");
#ifndef DISABLE_SNMPV1
            }
#endif
            rc = asn_realloc_rbuild_sequence(pkt, pkt_len, offset,
1,
                                            (u_char) (ASN_SEQUENCE
|

ASN_CONSTRUCTOR),
                                            *offset -
start_offset);
            DEBUGINDENTLESS();

            if (rc == 0) {
                return -13;
            }
            return 0;
        } else {

#endif                          /* USE_REVERSE_ASNENCODING */

  //Ditching code here

            /*
             * Save current location and build SEQUENCE tag and
length
             * placeholder for SNMP message sequence
             * (actual length will be inserted later)
             */
            cp = asn_build_sequence(*pkt, pkt_len,
                                   (u_char) (ASN_SEQUENCE |
                                            ASN_CONSTRUCTOR), 0);
            if (cp == NULL) {
                return -14;
            }
            h0e = cp;

#ifndef DISABLE_SNMPV1
            if (pdu->version == SNMP_VERSION_1) {
                DEBUGDUMPSECTION("send", "SNMPv1 Message");
            } else {
#endif
                DEBUGDUMPSECTION("send", "SNMPv2c Message");
#ifndef DISABLE_SNMPV1
            }
#endif

            /*
             * store the version field
             */
            DEBUGDUMPHEADER("send", "SNMP Version Number");
            version = pdu->version;
            cp = asn_build_int(cp, pkt_len,
                               (u_char) (ASN_UNIVERSAL |
ASN_PRIMITIVE |
                                         ASN_INTEGER), (long *)
&version,
                               sizeof(version));
            DEBUGINDENTLESS();
            if (cp == NULL)
                return -15;

            /*
             * store the community string
             */
            DEBUGDUMPHEADER("send", "Community String");
            cp = asn_build_string(cp, pkt_len,
                                  (u_char) (ASN_UNIVERSAL |
ASN_PRIMITIVE |
                                            ASN_OCTET_STR), pdu-
>community,
                                  pdu->community_len);
            DEBUGINDENTLESS();
            if (cp == NULL)
                return -16;
            break;

#ifdef USE_REVERSE_ASNENCODING
        }
#endif                         /* USE_REVERSE_ASNENCODING */
        break;
#endif /* support for community based SNMP */
    case SNMP_VERSION_2p:
    case SNMP_VERSION_sec:
    case SNMP_VERSION_2u:
    case SNMP_VERSION_2star:
    default:
        session->s_snmp_errno = SNMPERR_BAD_VERSION;
        return -17;
    }

    h1 = cp;
    DEBUGPRINTPDUTYPE("send", pdu->command);
    cp = snmp_pdu_build(pdu, cp, pkt_len);
    DEBUGINDENTADD(-4);          /* return from entire v1/v2c
message */
    if (cp == NULL)
        return -18;
    /*
     * insert the actual length of the message sequence
     */
    switch (pdu->version) {
#ifndef DISABLE_SNMPV1
    case SNMP_VERSION_1:
#endif
#ifndef DISABLE_SNMPV2C
    case SNMP_VERSION_2c:
#endif
#if !defined(DISABLE_SNMPV1) || !defined(DISABLE_SNMPV2C)
        asn_build_sequence(*pkt, &length,
                           (u_char) (ASN_SEQUENCE |
ASN_CONSTRUCTOR),
                           cp - h0e);
        break;
#endif /* support for community based SNMP */

    case SNMP_VERSION_2p:
    case SNMP_VERSION_sec:
    case SNMP_VERSION_2u:
    case SNMP_VERSION_2star:
    default:
        session->s_snmp_errno = SNMPERR_BAD_VERSION;
        return -19;
    }
    *pkt_len = cp - *pkt;
    return 0;
}

int
snmp_build(u_char ** pkt, size_t * pkt_len, size_t * offset,
           netsnmp_session * pss, netsnmp_pdu *pdu)
{
    int             rc;
    rc = _snmp_build(pkt, pkt_len, offset, pss, pdu);
    if (rc) {
        if (!pss->s_snmp_errno) {
            snmp_log(LOG_ERR, "snmp_build: unknown failure");
            pss->s_snmp_errno = SNMPERR_BAD_ASN1_BUILD;
        }
        SET_SNMP_ERROR(pss->s_snmp_errno);
        //rc = -1;
    }
    return rc;
}
```

```
static int
_snmp_parse(netsnmp_session * session,
            netsnmp_pdu *pdu, u_char * data, size_t length)
{

    u_char          community[COMMUNITY_MAX_LEN];
    size_t          community_length = COMMUNITY_MAX_LEN;

    int             result = -1;

    /*
     * Ensure all incoming PDUs have a unique means of
identification
     * (This is not restricted to AgentX handling,
     * though that is where the need becomes visible)
     */
    pdu->transid = 1;

    pdu->version = snmp_parse_version(data, length);

  printf("pdu-version\n%ld",pdu->version);

    switch (pdu->version) {
    case SNMP_VERSION_1:
      data = snmp_comstr_parse(data, &length,
                               community, &community_length,
                               &pdu->version);
        if (data == NULL)
            return -1;


        /*
         * maybe get the community string.
         */
        SNMP_FREE(pdu->community);
        pdu->community_len = 0;
        pdu->community = (u_char *) 0;
        if (community_length) {
            pdu->community_len = community_length;
            pdu->community = (u_char *) malloc(community_length);
            if (pdu->community == NULL) {
                //session->s_snmp_errno = SNMPERR_MALLOC;
                return -2;
            }
            memmove(pdu->community, community, community_length);
        }
```

```
   printf("Community String is : %s , length : %d",pdu-
>community,pdu->community_len);

        //DEBUGDUMPSECTION("recv", "PDU");
        result = snmp_pdu_parse(pdu, data, &length);
        if (result < 0) {
            /*
             * This indicates a parse error.
             */
            //snmp_increment_statistic(STAT_SNMPINASNPARSEERRS);
       return -3;
   }
        DEBUGINDENTADD(-6);
        break;


    case SNMPERR_BAD_VERSION:
        printf("error parsing snmp message version");
        snmp_increment_statistic(STAT_SNMPINASNPARSEERRS);

   return -4;
        break;
    case SNMP_VERSION_sec:
    case SNMP_VERSION_2u:
    case SNMP_VERSION_2star:
    case SNMP_VERSION_2p:
    default:
        printf("unsupported snmp message version");
        //snmp_increment_statistic(STAT_SNMPINBADVERSIONS);

        /*
         * need better way to determine OS independent
         * INT32_MAX value, for now hardcode
         */
        if (pdu->version < 0 || pdu->version > 2147483647) {
            //snmp_increment_statistic(STAT_SNMPINASNPARSEERRS);
        }
        //session->s_snmp_errno = SNMPERR_BAD_VERSION;
   return -5;
   break;

    }

    return 0;
}
```

```
//static int snmp_parse(void *sessp,netsnmp_session * pss,
netsnmp_pdu *pdu, u_char * data, size_t length)

int
snmp_parse(netsnmp_session * pss,
           netsnmp_pdu *pdu, u_char * data, size_t length)
{
    int             rc;

    rc = _snmp_parse(pss, pdu, data, length);
    return rc;
}
```

CONCLUSION

The Application Proxy Firewall designed and implemented met all the requirements and successfully fulfilled the efficiency and performance parameters.

The firewall facilitated the much needed integration of the Power Plant Automation Network with Global and Corporate Networks spanning across countries without comprising on security and integrity.

REPORT CONCLUSION

It was a wonderful experience working eith Siemens Information Systems Ltd. Siemens Information Systems Ltd is a big company and is involved in many projects. I got a chance to be a part of their various projects.

These projects gave me exposure to industrial apllications.

I worked under <DEPARTMENT> for <WORK>. I had set my goals in the very beginning as to understand the needs and requiremenst of Industry today, and in the near future too, I also wanted to analyze my standing, and also wished to figure oput the direction I had to move in.

With utmost pleasure, I would like to say that I have successfully accomplished my goals, as I have learnt a lot in this Organisation, gained experience of Network Programming . I am now aware of my strengths to build upon and weakenses to work on. I have gained clear vision of the use of Network Programming from the point of view of industrial applications.

I know where I have put in efforts to evolve myself into a through professional.