

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

PROJECT REPORT

MAR – JUNE 2002

R.V.C.E PROJECT CODE: RV02CS8P11

SUBJECT CODE: CS8P2

Submitted in partial fulfillment of the requirements for the 8th semester of the B.E. Computer Science and Engineering course as prescribed by the Visveswariah Technological University, Belgaum.

SUBMITTED BY

SRIVAS N. CHENNU 1RV8CS086

SUMANTH G. 1RV98CS089

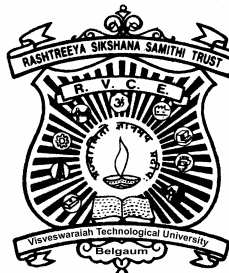
VINAY KRISHNA Y.S. 1RV98CS101

VISHWAS N. 1RV98CS104

UNDER THE GUIDANCE OF

MRS. N.P. KAVYA
CSE DEPARTMENT
R.V.C.E.
BANGALORE

MR. SHAILESH R.C.
SCIENTIST 'C'
C.A.I.R.
BANGALORE



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

R.V. COLLEGE OF ENGINEERING

BANGALORE

I. ABSTRACT

The project implements a full-featured network security solution in the form of a completely self-sufficient firewall gateway system that can be loaded directly off an independent bootable CD-ROM device. The firewall and proxying services operate over a modularized Linux OS architecture. In addition, an intelligent network anomaly and intrusion detection system has been developed and integrated with the firewalled architecture. This application infers the occurrence of abnormal activity occurring on the network by processing the firewall audit records, given a set of rule signatures, and generates quantitative alerts regarding the anomalous behavior. It thus provides the network administrator with an advanced functionality for monitoring and tracking large-volume networks.

II. SYNOPSIS

The aim of this project is to design and implement a completely self-sufficient firewall and gateway system that can be loaded directly off a bootable floppy or Compact Disc.

The system to be designed by us will use the Linux operating system architecture such that the Linux kernel, root file system and the firewall software will fit on a CD. This removable device should be able to function in the absence of a fixed disk device. Furthermore, additional features can then be added to the firewall subsystem in the form of pluggable kernel modules, so as to provide increased functionality on demand.

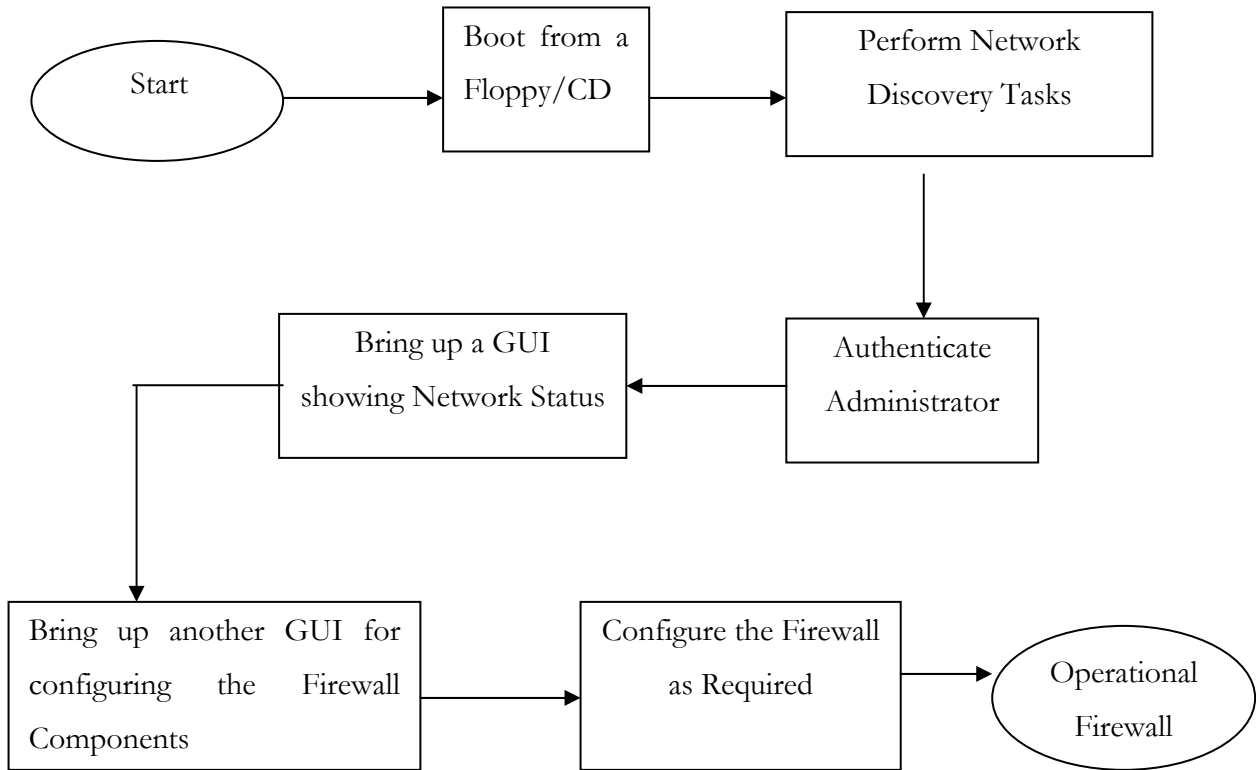
The bootable firewall system will also authenticate the network administrator before setting up the gateway. The system will then perform network hardware detection and bring up the network interfaces. The firewall and gateway software will be loaded on the configured hardware.

The network administrator will also be provided a GUI configuration and monitoring tool wherein he will be able to configure the firewall components and specify the firewall rules that will determine the access control policy of the network and organization.

Once the system is up and functioning as an operational firewall, the system administrator will be provided with options for loading additional modules and advanced network tools and utilities which enhance the capabilities of the system, and provide the administrator with increased functionality.

In addition to fulfilling these requirements, the system should be robust, efficient and generic in its design. It should be able to function on a variety of hardware environments and architectures. Extensive testing of the system will be conducted on different platforms to ensure that all these requirements will be satisfied.

The overall conceptualization of the system as visualized at the inception stage is illustrated in the diagram below.



Once a system with satisfying the conceptualization above has been designed and implemented, we will also develop an additional tool that provides the administrator with advanced network activity monitoring facilities. This tool will work in conjunction with the firewall system, and will incorporate intelligent inferencing capabilities enabling it to detect potential anomalies and intrusions into the network from external sources. Such a tool is envisioned to be ideal for deployment in large private networks with large volumes of network traffic, all of which cannot be directly monitored by the network administrator. In such a scenario, the software system to be designed by us will be ideally situated between the core firewall system and the network administrator, and provide him with alerts regarding suspicious network activity and suggest suitable courses of action.

III. ACKNOWLEDGEMENTS

We would like to thank our project guide, Mr. Shailesh Chansarkar for his guidance and support during the course of our project work at C.A.I.R. Bangalore. We also thank Mr. Kishorie, Human Resources Manager, C.A.I.R. and Dr. Athithan, Senior Scientist, C.A.I.R. for providing us the opportunity to execute the project at C.A.I.R. in area of our choice.

Additionally, we would like to acknowledge the sound advice and independence provided to us by our internal project guide at R.V.C.E., Mrs. N.P. Kavya.

SRIVAS N. CHENNU

SUMANTH G.

VINAY KRISHNA Y.S.

VISHWAS N.

IV. CONTENTS

I. ABSTRACT.....	1
II. SYNOPSIS	2
III. ACKNOWLEDGEMENTS.....	4
CHAPTER 1 - INTRODUCTION	7
1.1 NETWORK SECURITY	7
1.2 NETWORK INTRUSIONS.....	8
1.3 INTERNET FIREWALLS.....	12
1.3.1 Packet Filtering.....	13
1.3.2 Application Gateways.....	13
1.3.3 Circuit Gateways	14
1.3.4 Bastion Hosts.....	14
1.4 FIREWALL CONFIGURATIONS.....	15
1.5 DISADVANTAGES OF FIREWALLS	16
1.6 NETWORK INTRUSION DETECTION	17
1.6.1 Audit Records	17
1.6.2 Statistical Anomaly Detection.....	18
1.6.3 Rule-Based Intrusion Detection	18
1.6.4 Distributed Intrusion Detection.....	19
1.7 THE BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION.....	20
CHAPTER 2 - REQUIREMENTS SPECIFICATION.....	22
2.1 FEATURE SPECIFICATION	23
2.2 SOFTWARE REQUIREMENTS SPECIFICATION.....	27
2.2.1 Bootable Firewall CD System Requirements.....	27
2.2.2 Intelligent Anomaly Detection System Requirements	34
CHAPTER 3 - DATA FLOW DIAGRAMS	37
3.1 FIREWALL CD BOOTUP SEQUENCE.....	37
3.2 INTELLIGENT ANOMALY DETECTION SYSTEM DATA FLOW DIAGRAM.....	39
CHAPTER 4 - DESIGN.....	40
4.1 DESIGN PHASE I - THE BOOTABLE FIREWALL CD	40
4.2 THE BOOTABLE FIREWALL CD USER INTERFACE	51
4.3 DESIGN PHASE II - THE INTELLIGENT ANOMALY DETECTION SYSTEM	52
4.3.1 The iptables Firewall	52
4.3.2 The MySQL Database	53
4.3.3 The LADS Rule Data Model.....	54
4.3.4 The LADS Alert Format.....	62

4.4 IADS INTERNAL ARCHITECTURE.....	65
4.5 THE IADS USER INTERFACE	68
CHAPTER 5 - IMPLEMENTATION	69
5.1 HARDWARE DEPENDENCIES	69
5.2 SOFTWARE DEPENDENCIES	70
5.3 IADS SOURCE CODE STRUCTURE.....	71
5.3.1 <i>The Rule Engine</i>	71
5.3.2 <i>The Rule File Parser</i>	73
5.3.3 <i>The Main Module</i>	73
5.3.4 <i>The Log Facility</i>	74
5.3.5 <i>The Log Processor</i>	74
5.3.6 <i>The Alert Handler</i>	75
5.3.7 <i>The Packet Handler</i>	76
5.3.8 <i>The Packet Processor</i>	76
CHAPTER 6 - TESTING	77
6.1 TESTING PHASE I - THE BOOTABLE FIREWALL CD TEST PLAN.....	77
6.2 TESTING PHASE II - THE INTELLIGENT ANOMALY DETECTION SYSTEM TEST PLAN.....	80
CHAPTER 7 - CONCLUSION	82
7.1 FUTURE IMPROVEMENTS.....	82
CHAPTER 8 - USER MANUAL	83
8.1 CONFIGURING THE SYSTEM BIOS	83
8.2 BOOTING FROM THE CD-ROM.....	83
8.3 THE SYSCONF CONFIGURATION PROGRAM	83
8.4 CONFIGURATION OF SYSTEM SERVICES.....	84
8.4.1 <i>The dnsmconf DNS service configuration program</i>	84
8.4.2 <i>The iptconf Iptables configuration program</i>	84
8.4.3 <i>The apacheconf web server configuration program</i>	85
8.4.4 <i>The smconf Sendmail configuration program</i>	85
8.4.5 <i>The iadsconf IADS configuration program</i>	86
8.4.6 <i>Network configuration</i>	86
8.4.7 <i>Configuration of miscellaneous services</i>	86
BIBLIOGRAPHY.....	88
APPENDICES.....	89
APPENDIX A - THE CRYPT() ENCRYPTION LIBRARY.....	89
APPENDIX B - THE MYSQL C API LIBRARY	90
APPENDIX C - THE NEWT WINDOWING SYSTEM	93
APPENDIX D - THE IADS XML RULE DTD	105
APPENDIX E - THE IDMEF DATA MODEL.....	110
APPENDIX F - SOURCE CODE	111

CHAPTER 1 - INTRODUCTION

1.1 NETWORK SECURITY

With the widespread use of distributed and networked computer systems, protecting data being stored on such systems and being transmitted between them has become necessary. Network security refers to this important aspect of computer security that deals with securing network communication and storage. With the spread of the Internet, the need for dependable network security solutions has become increasingly evident.

Network security applications thus need to perform both preventive as well as remedial functions in today's networks. Some of these key functions of network security services are listed below.

- **Confidentiality** – This function ensures that information transactions are accessible only to authorized parties.
- **Authentication** – Ensures that the origin of a message or electronic document is correctly identified, with the assurance that the identity is not false.
- **Integrity** – Ensures that only the authorized parties are able to modify system assets and transmitted information.
- **Non-repudiation** – This function requires that neither the sender nor receiver of a message can deny its transmission.
- **Access Control** – Ensures that access to secure information is controlled in a graded manner.
- **Availability** – This function requires that the computer system assets be available to authorized parties as and when requested.

1.2 NETWORK INTRUSIONS

Network intrusions can be defined as unauthorized entries and attacks into a network which compromise the security of the information stored in it. Due to the Internet, individuals anywhere in the world can initiate such attacks toward private and corporate networks. Though the motivation behind such attacks varies, the result is the compromise of the security of the private network.

Most of the attacks common on networks today can be classified into the one of the following types.

- **Interruption** – An asset of the system is destroyed or become unavailable or unusable. This is an attack on **availability**.
- **Interception** – An unauthorized party gains access to an asset. This is an attack on **confidentiality**.
- **Modification** – An unauthorized party not only gains access to a system asset, but also tampers with it. This is an attack on **integrity**.
- **Fabrication** – An unauthorized party inserts counterfeit information into the system. This is an attack on **authenticity**.

Some real life examples of common attacks from the above classification are detailed below.

Probes and Scans

Probes are characterized by unusual activity occurring during attempts to gain access to a system or to discover information about it. Large numbers of such probes performed by an automated program are called Scans. An example of such an attack is TCP portscan.

Account Compromise

An account compromise is the unauthorized use of a computer account by an intruder. This account may be a user account with limited privileges or in the worst case, a privileged super-user account. Such attacks can lead to loss of secrecy and integrity of data and other resources.

Packet Sniffing

Packet sniffers are programs that capture packets passing through the local network interface, as they travel over the network. By using such a program, an attacker can violate the integrity and confidentiality of the data in the packet.

Denial of Service

The goal of DoS attacks is not to gain unauthorized access to a network service, but to prevent legitimate users from using it. Denial of service attacks use multiple systems to attack one or more victim systems with the intent of denying service to legitimate users of the victim systems. The degree of automation in attack tools enables a single attacker to install their tools and control tens of thousands of compromised systems for use in attacks. Denial-of-service attacks are effective because the Internet is comprised of limited and consumable resources, and Internet security is highly interdependent.

Worms, Viruses and Trojans

A worm is self-propagating malicious code. Unlike a virus, which requires a user to do something to continue the propagation, a worm can propagate by itself. The highly automated nature of the worms coupled with the relatively widespread nature of the vulnerabilities they exploit allows a large number of systems to be compromised within a matter of hours. Some worms include built-in denial-of-service attack payloads or website defacement payloads; and others have dynamic configuration capabilities. But the biggest impact of these worms is that their propagation effectively creates a denial of service in many parts of the Internet because of the huge amounts of scan traffic generated, and they cause much collateral damage.

Internet Infrastructure attacks

Attack Type 1 - Attacks on the Internet Domain Name System (DNS)

Threats to DNS system, deployed widely over the Internet, include the following:

Cache poisoning - If DNS is made to cache bogus information, the attacker can redirect traffic intended for a legitimate site to a site under the attacker's control.

Compromised data - Attackers could compromise vulnerable DNS servers, giving them the ability to modify the data served to users.

Denial of service - A large denial-of-service attack on some of the name servers for a TLD could cause widespread Internet slowdowns or effective outages.

Domain hijacking - By leveraging insecure mechanisms used by customers to update their domain registration information, attackers can co-opt the domain registration processes to take control of legitimate domains.

Attack Type 2 – Attacks against or using routers

Routers are specialized computers that direct traffic on the Internet. Router threats fall into the following categories:

Routers as attack platforms - Intruders use poorly secured routers as platforms for generating attack traffic at other sites, or for scanning or reconnaissance.

Denial of service - Although routers are designed to pass large amounts of traffic through them, they often are not capable of handling the same amount of traffic directed *at* them. Intruders take advantage of this characteristic, attacking the routers that lead into a network rather than attacking the systems on the network directly.

Exploitation of trust relationship between routers - For routers to do their job, they have to know where to send the traffic they receive. They do this by sharing routing information between them, which requires the routers to trust the information they receive from their peers.

As a result, it would be relatively easy for an attacker to modify, delete, or inject routes into the global Internet routing tables to redirect traffic destined for one network to another, effectively causing a denial of service to both, one because no traffic is being routed to them, and the other because they're getting more traffic than they should. Although the technology has been widely available for some time, many networks like Internet service providers and large corporations do not protect themselves with the strong encryption and authentication features available on the routers.

The Potential impacts of such network infrastructure attacks are many.

Denial of service - Because of the asymmetric nature of the threat, denial of service is likely to remain a high-impact, low-effort modus operandi for attackers. Most organizations' Internet connections have between 1 and 155 megabits per second (Mbps) of bandwidth available. Attacks have been reported in the hundreds of Mbps and up, more than enough to saturate nearly any system on the Internet.

Compromise of sensitive information - Some viruses attach themselves to existing files on the systems they infect and then send the infected files to others. This can result in confidential information being distributed without the author's permission.

Misinformation - Intruders might be able to modify news sites, produce bogus press releases, and conduct other activities, all of which could have economic impact.

Time and resources diverted from other tasks - Perhaps the largest impact of security events is the time and resource requirements to deal with them.

1.3 INTERNET FIREWALLS

A firewall is a system or group of systems that enforces an access control policy at the interface between two networks. The firewall system performs the function of controlling network accesses between the networks that it is deployed, based on a set of security policies specified by the firewall administrator. Firewalls serve multiple purposes.

1. It restricts people to entering at a carefully controlled point.
2. It prevents attackers from getting close to your other defenses.
3. It restricts people to leaving at a carefully controlled point.

Firewalls are widely used as the primary network security application to protect private networks from attackers and malicious code on the Internet. Firewall systems can enforce security policies at different levels in the network software hierarchy.

Network Level – Packet filtering systems enforce packet based security checks to decide whether a specific packet should be permitted or not.

Application Level – Application gateways enforce higher level filtering specific to one or more Internet applications like E-mail, FTP etc.

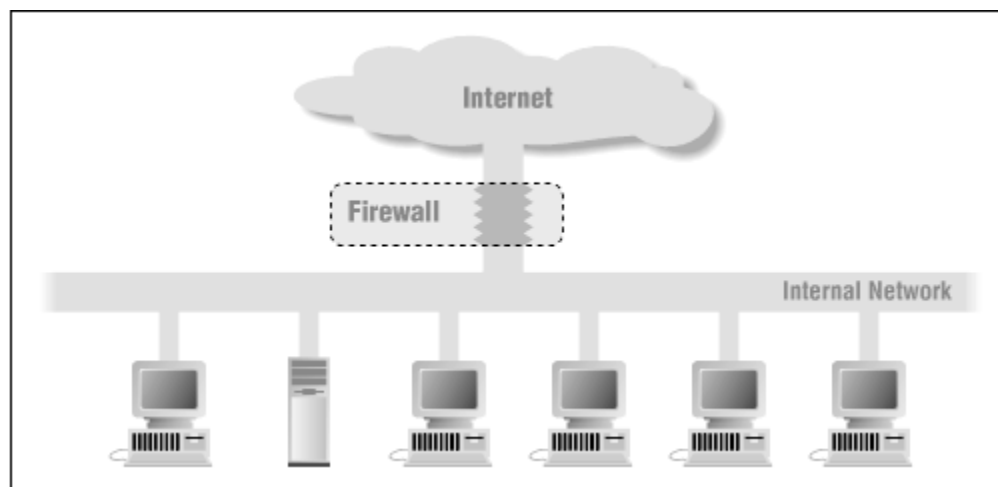


Fig. 1.1 A typical firewall architecture

Some of the different types and characteristic architectures of firewalls are detailed below.

1.3.1 PACKET FILTERING

Packet filtering is a network security mechanism employed by many Internet firewalls that works by controlling what data can flow to and from a network.

To transfer information across a network, the information has to be broken up into small pieces, each of which is sent separately. Breaking the information into pieces allows many systems to share the network, each sending pieces in turn. In IP networking, those small pieces of data are called **Packets**. All data transfer across IP networks happens in the form of packets. Packets traversing an internetwork travel from router to router until they reach their destination. A router has to make a routing decision about each packet it receives; it has to decide how to send that packet on towards its ultimate destination.

Packet filtering lets the network administrator control data transfer based on:

- The address the data is coming from.
- The address the data is going to.
- The session and application protocols being used to transfer the data.
- Information regarding various transfer stored parameters in the packet.

1.3.2 APPLICATION GATEWAYS

Application gateways act as relays of application-level traffic. Users contact the gateway using TCP/IP applications like Telnet and FTP. The gateway asks the user for the name of the remote host to be accessed, and relays the requested information. This process by which the application gateway intermediates and monitors network data transfer, is called proxying.

Some of the advantages of application-level gateways are as follows.

- Application gateways are more secure than packet filters.
- They can be simpler in their design and implementation.
- Logging and auditing is also easier at the application level.

The important disadvantages of such systems are given below.

- Additional processing overhead over each connection.
- Firewalling is implemented at the application level, instead of the kernel level, and hence can be subverted.

1.3.3 CIRCUIT GATEWAYS

Circuit gateways differ from application gateways in that they do not permit end-to-end traffic. Rather, the gateway sets up two separate connections for data transfer, one from a gateway to the client, and another from the gateway to the remote host or server. The gateway then performs the task of performing of copying allowed data between the two connections. Thus, circuit gateways operate in a non-transparent fashion with respect to the host and the client.

A typical use of such circuit-level gateways is a situation in which the system administrator trusts the internal users. The gateway can be configured to support application-level or proxy service on inbound connections and circuit-level functions.

1.3.4 BASTION HOSTS

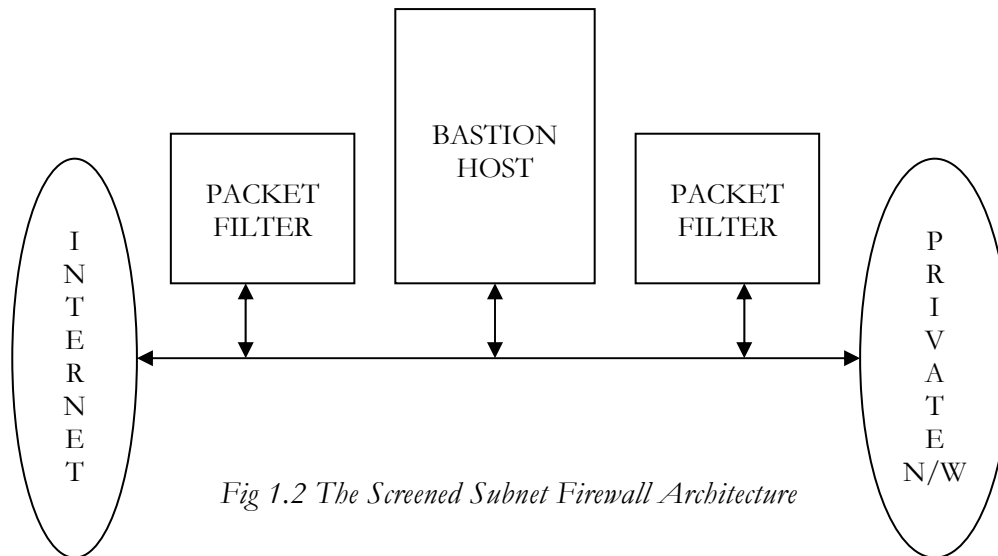
A bastion host is a system identified by the firewall administrator as a critical strong point in the network's security. Typically, a bastion host serves as a platform for deployment of an application-level or circuit-level gateway. Some of the common characteristics of bastion hosts are as below.

- Bastion hosts run a trusted Operating System version.
- They run only essential services designated by the administrator.
- The bastion host provides for secure and trusted authentication service for its network.
- The security of the entire system is dependent on the security of the Bastion Host.

1.4 FIREWALL CONFIGURATIONS

Firewalls can be deployed in a variety of configurations, ranging from the simple to the complex. Some of these are detailed below.

- **Screened Host Firewall, Single Homed Bastion** – In this configuration, the firewall consists of two systems, the packet filtering router, and the bastion host. The router is configured so that only traffic destined for the bastion host is allowed in, and only traffic originating from the bastion host is allowed out.
- **Screened Host Firewall, Dual Homed Bastion** – This configuration provides additional security as compared to the single homed configuration, by physically separating the external and internal networks.
- **Screen Subnet Firewall** – This configuration is the most secure of all configurations, in that it deploys two packet-filtering routers on both sides of the bastion host, one for filtering incoming traffic, and another for outgoing traffic. Such a configuration distributes the network load efficiently, and provides robustness.



1.5 DISADVANTAGES OF FIREWALLS

Some of the key limitations of firewall systems are listed below.

- Firewalls cannot protect against attacks that bypass the firewall through other mediums of transmission.
- Firewalls cannot protect against internal threats to the network.
- Firewalls cannot prevent or block viruses, worms, Trojans and other harmful programs from entering or leaving the private network.
- Many commonly used protocols including **rcp**, **rlogin**, **rdist**, **rsh** and RPC based protocols are not well suited to packet filtering and firewalling.
- Not all kinds of policies can be enforced on a packet filtering mechanism.
- Currently available firewall systems are not perfect; Moreover, building a foolproof security policy is a non-trivial task, especially for large networks with a variety of traffic.

1.6 NETWORK INTRUSION DETECTION

A set of attempts to compromise a computer or a computer network resource security is regarded as an **intrusion**. Intrusion detection systems are a second line of defense in network security solutions. They work in conjunction with firewalls and monitor attacks and intrusions into a network that get past the firewall subsystem. A strong need for such network intrusion detection systems has emerged due to the fact that firstly, system audit information can no longer be processed directly by a person, and secondly, complexity and variety of intrusions has dramatically increased.

Intrusion detection systems are based on the key assumption that the typical behavior of an attacker is different from a normal user of a network. The intrusion detection mechanism looks for this anomalous behavior in the network, and alerts the network administrator on its occurrence.

Furthermore, network intrusion detection systems can be built with intelligent deduction capabilities, which scan large amounts of network traffic, and identify potentially subversive activities for the network administrator to deal with. Such systems can greatly ease the task of the administrator, in addition to making networks very secure and resilient to attacks.

Some of the different approaches to designing anomaly detection systems are now detailed below.

1.6.1 AUDIT RECORDS

Audit records are records of ongoing activity in the network maintained by the firewall subsystem, or the intrusion detection system. These records, also known as logs, can be effectively used to detect potential intrusions into the system. Some of the information stored in such records is given below.

- **Subject** – The person or process initiating an operation.

- **Action** – The specific action performed by the subject.
- **Object** – The receptors of the action being performed.
- **Resource Usage** – A record of the system resources used by the operation.
- **Time Stamp** – Unique date-and-time stamp identifying when the operation took place.

1.6.2 STATISTICAL ANOMALY DETECTION

Statistical anomaly detection involves the collection of data relating to the behavior of legitimate users over a period of time, which is then used in statistical tests applied to observed behavior to determine whether or not it is legitimate. Such methods employ the techniques given below.

- **Threshold Detection** – This approach involves defining thresholds, independent of users, for the frequency of occurrence of various events. Such detection tends to be crude in their estimation. Thresholds must be frequently updated in order to reflect user activity. Such techniques are used in conjunction with more sophisticated techniques.
- **Profile based Detection** – A profile of activity of each user is developed and used to detect changes in the behavior of individual accounts. Past behavior of users is thus employed to detect significant deviations. A profile usually consists of a collection of detection parameters to reduce error probabilities.

1.6.3 RULE-BASED INTRUSION DETECTION

Rule-based intrusion detection techniques work by observing events in the systems and applying a set of rules that lead to a decision regarding whether a given pattern of activity is or is not suspicious.

- **Rule-based Anomaly Detection** – In such systems, rules represent past behavior patterns of users and programs. Current observed behavior is validated against these codified rules, and anomalous behavior is reported.
- **Rule-based Penetration Identification** – The key feature of such systems is the use of rules for identifying known penetrations. Such rules are compiled using previous experiences of system administrators and other experts.

1.6.4 DISTRIBUTED INTRUSION DETECTION

Modern intrusion detection systems employ distributed computing resources to defend a distributed collection of hosts supported by a LAN or an internetwork. They provide a more effective defense mechanism by coordination and cooperation among intrusion detection mechanisms deployed across the network.

Some important design issues in such systems are given below.

- **Heterogeneous Environments** - Distributed intrusion detection systems need to deal with a variety of audit record formats and system architectures.
- **Secure Inter-node communication** – Transmission of audit data between the nodes must be secured and authenticated, in order to prevent tampering by intruders.
- **Centralized vs. Decentralized architecture** – Centralized architectures feature a central node for audit data collection and processing. Such systems are easier to design but are subject to frequent bottlenecks in the internal network. Decentralized architectures, on the other hand perform processing separately at multiple nodes, which must be coordinated.

1.7 THE BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

This project develops on the concepts of Internet firewalls and anomaly detection systems. The final product developed in the course of this project is a bootable Compact Disc that provides a complete network security solution to the user. This CD has the following key features and characteristics.

- It employs the Linux Operating System as the Firewalling OS.
- The OS runs a specialized Linux kernel tailored for firewall and routing operations.
- It does not require a fixed hard disk for normal operation.
- Physical security of the firewall gateway system is provided by secure administrator authentication using passwords.
- The bootable Linux CD comes bundled with the following software components:
 - ❑ Network Interface Card detection and configuration tools
 - ❑ Iptables Firewalling software
 - ❑ DNS server software and related tools
 - ❑ Apache web server software and related tools
 - ❑ sendmail MTA software
 - ❑ Network monitoring and reporting tools
 - ❑ A text-based, command-line User interface
 - ❑ Menu based interfaces for configuring the above services

In addition to the above software components, the also bundled with the CD is an intelligent anomaly detection system or IADS, designed and implemented by us.

IADS is a rule-based network intrusion detection system that provides the network administrator with the ability to better monitor the traffic on the network and detect attacks on it. IADS allows the administrator to easily extract useful information regarding anomalous network activity from large volumes of firewall audit logs. It does this by employing a inferencing engine that applies a set of rule constraints on a set of audit records and come up with a report of potential anomalies and intrusions into the network.

Some of the key features of IADS are listed below.

- It works in conjunction with the iptables firewall architecture.
- It analyzes iptables logs to extract information about anomalous behavior.
- IADS runs as a background daemon and periodically analyzes log entries made by incoming traffic.
- IADS can currently work on IP, TCP, UDP and ICMP traffic.
- It supports a XML based rule-specification language.
- An IADS rule construct consists of a name, an impact type, an action and a arbitrary complex Boolean expression involving IP, TCP, UDP and ICMP header field values.
- IADS prepares intrusion alert reports in the IETF IDMEF (Intrusion Detection Message Exchange Format) standard.
- IADS generates log reports for reporting its operational status.
- IADS is compliant with the POSIX 1.1 OS standard.
- It can be deployed on Linux systems running kernel versions 2.4.x.

CHAPTER 2 - REQUIREMENTS SPECIFICATION

The overall process of development of the bootable firewall CD and the intelligent anomaly detection system has been supported and documented using the **Rational RequisitePro** software. This application is a part of the **Rational Suite of CASE tools** developed by Rational Software Inc.

Rational RequisitePro enables the software developer to create and manipulate the requirements document for a project, and model it through the various stages of its evolution. RequisitePro identifies the following types of requirements in a software development project.

- User Features
- Glossary Requirements
- Use Case Requirements
- Software Requirements

Furthermore, RequisitePro also provides the developer with the following features:

- Ability to manipulate a large set of attributes for each requirement, including, priority, status, difficulty level etc.
- Traceability trees and matrices to trace software requirements to user features.
- Management of interdependencies and hierarchical structures within the requirements set.
- Facility to generate reports and comprehensive documents in various formats.

The requirements document for the bootable firewall CD and the intelligent anomaly detection system developed using Rational RequisitePro is given below. The document has been divided into two sections. The first section deals with the set of features as formulated from the perspective of a hypothetical user of the software system. The second section deals with the software requirements of the system developed from the given set of user features.

2.1 FEATURE SPECIFICATION

FEAT1: Firewall OS

The firewall system must run on the Linux operating system.

Priority : Must

Status : Implemented

FEAT2: The Operating System Kernel

A kernel image on a removable disk should be able to initialize the firewall.

Priority : Must

Status : Implemented

FEAT3: The Ramdisk Filesystem

The System should use a RAM disk to load and store the file system.

Priority : Should

Status : Implemented

FEAT4: Network hardware discovery

The system should perform network discovery procedures to detect and configure the current network hardware architecture.

Priority : Must

Status : Implemented

FEAT5 : User authentication

The system should provide a suitable user identification and authorization protocol.

Priority : Must

Status : Implemented

FEAT6 : System services Configuration Interface

The System should provide user interface features for configuring the system services.

Priority : Must

Status : Implemented

FEAT7 : Networking Monitoring

The system should provide for firewall administration and status monitoring.

Priority : Must

Status : Proposed

FEAT8: Support for loadable modules

The bootable system should provide support for loading additional functionality on demand.

Priority : Must

Status : Implemented

FEAT9: Rescue capability

The Bootable Disk should also be able to act as a rescue disk.

Priority : Should

Status : Proposed

FEAT10: System Logging

The system should maintain records of people and their activities on the system.

Priority : Must

Status : Proposed

FEAT11: Firewall Location

A decision should be made regarding the position of the firewall in the system either internet, intranet or extranet.

Priority : Should

Status : Proposed

FEAT12: Access Control

System should provide selective access to authorized administrators.

Priority : Must

Status : Implemented

FEAT13: High availability

System should be online for most of the required time.

Priority : Should

Status : Proposed

FEAT14: Network load

The firewall should be able to accommodate increased network load.

Priority : Should

Status : Implemented

FEAT15: Anomaly and Intrusion Detection

An additional tool should be designed, which provides the administrator with the ability to probe the network traffic for anomalous activities and common intrusion signatures.

Priority : Should

Status : Implemented

FEAT15.1: Configurable Alerts

This detection system should provide administrator with configurable alerts about detected anomalous network activity.

Priority : Should

Status : Implemented

FEAT16: User interface

The administrator should be able to interact with the software through a centralized user interface.

Priority : Should

Status : Implemented

2.2 SOFTWARE REQUIREMENTS SPECIFICATION

Listed below is the software requirements specification for the firewall system. The software requirements specification has been developed after evaluating the features requirements of the system as listed above. A many-to-many traceability relationship is maintained between the user requested features and the software requirements of the system.

The requirements document given below is divided into two sections. The first section lists the software requirements of the bootable Firewall CD, and the second section lists the requirements of the anomaly detection system to be developed as part of this project.

2.2.1 BOOTABLE FIREWALL CD SYSTEM REQUIREMENTS

SR1: Bootable Disk

The firewall should be a self-contained Linux operating system with a bootable image on a CD or Floppy

Priority : Must

Status : Implemented

SR1.1: Linux Architecture

The Linux OS manages memory required for the system operation.

Priority : Must

Status : Implemented

SR1.2: OS Process Management

The OS creates, runs and terminates processes on the system.

Priority : Must

Status : Implemented

SR1.3: OS device management

The OS loads the drivers for the network devices.

Priority : Must

Status : Implemented

SR1.4: OS Kernel Filesystem

The kernel filesystem stores the bootable kernel image.

Priority : Should

Status : Implemented

SR1.5: OS Kernel Size

The size of the kernel should be small enough to fit on a floppy.

Priority : Should

Status : Implemented

SR1.6: Support for modular OS kernel

The kernel should provide for automatic or on-demand loading of external kernel modules that incorporate additional functionality into the system.

Priority : Must

Status : Implemented

SR2: System administrator authentication

The system should provide for safe authentication mechanism during initialization.

Priority : Must

Status : Implemented

SR2.1: Authentication mechanism

The authentication script provides for encrypting, storing and validating user passwords using the DES algorithm with 56 bit keys.

Priority : Should

Status : Implemented

SR3: Network discovery process

System must provide tools for discovering the local network hardware architecture.

Priority : Must

Status : Implemented

SR4: Network status display and monitoring facilities

The system should display the status of the local network.

Priority : Should

Status : Implemented

SR5: Packet filtering policy

The filter should accept the packets based on the policy selected by the administrator.

Priority : Must

Status : Implemented

SR5.1: Network address translation

The firewall must provide capability for source and destination NAT.

Priority : Should

Status : Implemented

SR5.2: TCP and UDP traffic management

The firewall can provide grant / block privileges to transport layer data.

Priority : Should

Status : Implemented

SR5.3: Packet filter logging facilities

The packet filter rules provide targets to log packet hits to select log locations. The log contents can be specified as a part of the rule specification.

Priority : Should

Status : Implemented

SR6: Application Protocol Specific Control

The system provides application level access control to resources.

Priority : Should

Status : Implemented

SR7: Proxying

Proxy server should record and redirect all the data to and from the outside networks.

Priority : Should

Status : Implemented

SR8: Packet Encryption policy

The firewall must decide whether to allow encrypted packages to enter the network.

Priority : Should

Status : Implemented

SR9: The Root file system

The Root File System stores the files required by the firewall system.

Priority : Must

Status : Implemented

SR9.1: Root filesystem type

The filesystem type is of the linux ext2 format. This filesystem contains a core directory structure which is then populated.

Priority : Should

Status : Implemented

SR9.2: Library dependencies

The root filesystem should include all the libraries to satisfy dependencies of the firewall layer.

Priority : Must

Status : Implemented

SR9.3: Configuration scripts

The root filesystem stores configuration scripts required for the firewall.

Priority : Must

Status : Implemented

SR9.4: Recovery utilities

The root filesystem should also contain utilities for repairing and restoring the system to normal operation in case of failures.

Priority : Should

Status : Implemented

SR10: Dynamic loader and linker

A loader software must be incorporated for loading elf scripts into memory

Priority : Must

Status : Implemented

SR11: System Configuration script

The script allows the administrator to configure various services in the order given below.

Priority : Should

Status : Implemented

SR11.1: Configuration of network interfaces

The various network interfaces on the system can be configured. If the network interface is not configured, the script does not initialize any other services.

Priority : Should

Status : Implemented

SR11.2: DNS Configuration

The DNS Configuration tool can be used to configure Name Services on the system.

Priority : Should

Status : Implemented

SR11.3: Firewall Configuration

The script provides tools for configuring the firewall services.

Priority : Should

Status : Implemented

SR11.4: Apache Configuration

The script provides tools for configuring Apache Web Server.

Priority : Should

Status : Implemented

SR11.5: Sendmail Configuration

The script provides tools for configuring sendmail.

Priority : Should

Status : Implemented

SR11.6: Miscellaneous services

Miscellaneous services like date, time etc can be configured.

Priority : Should

Status : Implemented

2.2.2 INTELLIGENT ANOMALY DETECTION SYSTEM REQUIREMENTS

SR1

The system must have an intelligent anomaly detection system that analyzes the logs and prints out alerts

Priority : Should

Status : Implemented

SR2

The log entries made by the firewall must be written into a database from which the intelligent anomaly detection system reads the packet header information.

Priority : Should

Status : Implemented

SR3

The system should contain the Intrusion detection message exchange format library (libidmef).

Priority : Should

Status : Implemented

SR4

The system should contain a set of rules or signatures that the system uses to check if there is any anomaly.

Priority : Should

Status : Implemented

SR5

The rules for the intelligent anomaly detection system should be defined only in terms of the packet headers and must not involve any data checking.

Priority : Should

Status : Implemented

SR5.1

The rules for the intelligent anomaly detection system should be constructed as Boolean expressions of arbitrary length.

Priority : Should

Status : Implemented

SR5.2

Each of the rules must specify a target, which can be activating or deactivating a rule, dropping the packet or logging for the particular event that occurs.

Priority : Should

Status : Implemented

SR6

The IDMEF alert message must contain an indication of the possible impact of this event on the target

Priority : Should

Status : Implemented

SR6.1

The IDMEF alert message must provide information about the automatic actions taken by the analyzer in response to the event (if any).

Priority : Should

Status : Implemented

SR6.2

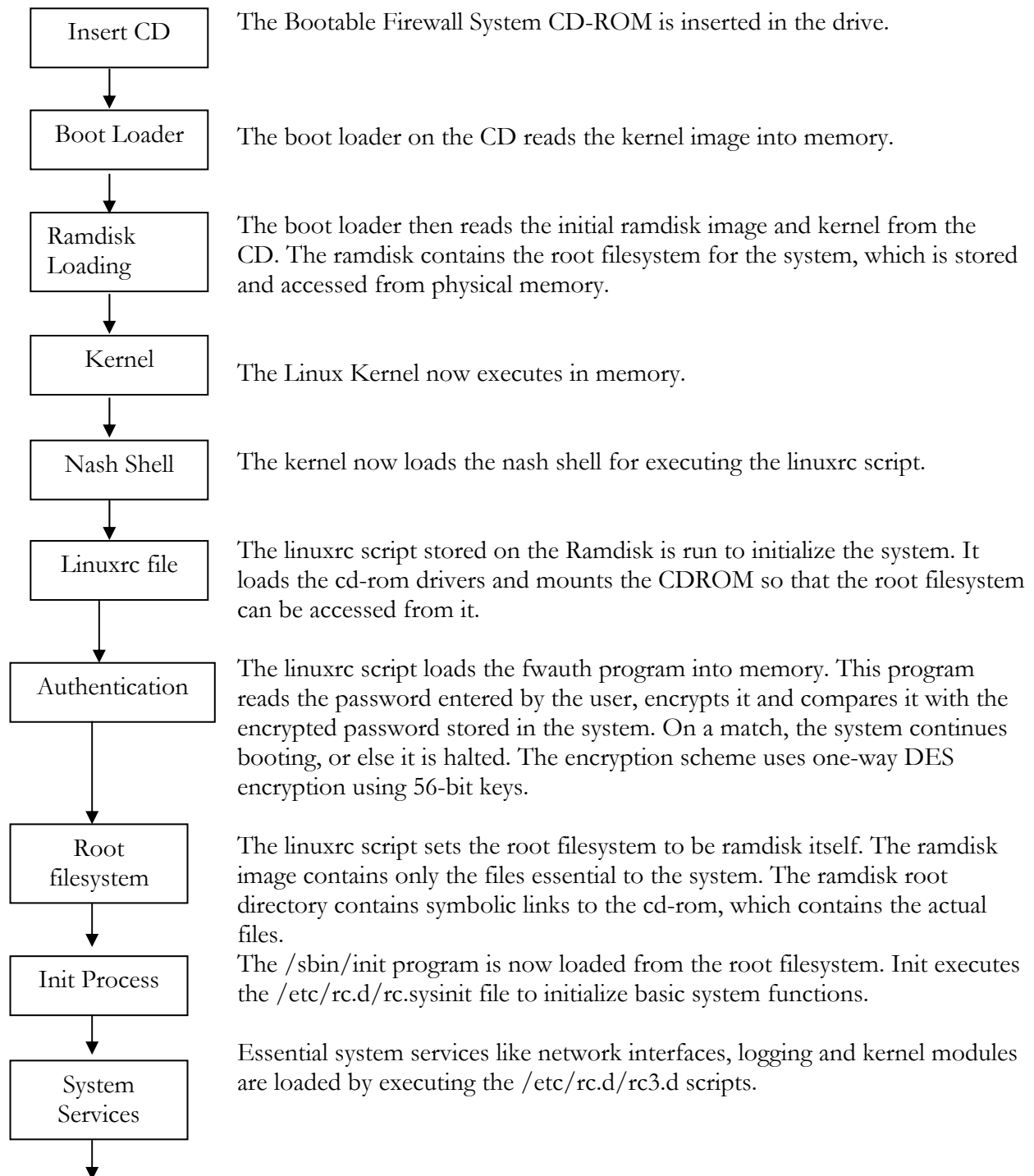
The IDMEF message must be uniquely identifiable in that it can be distinguished from other IDMEF messages.

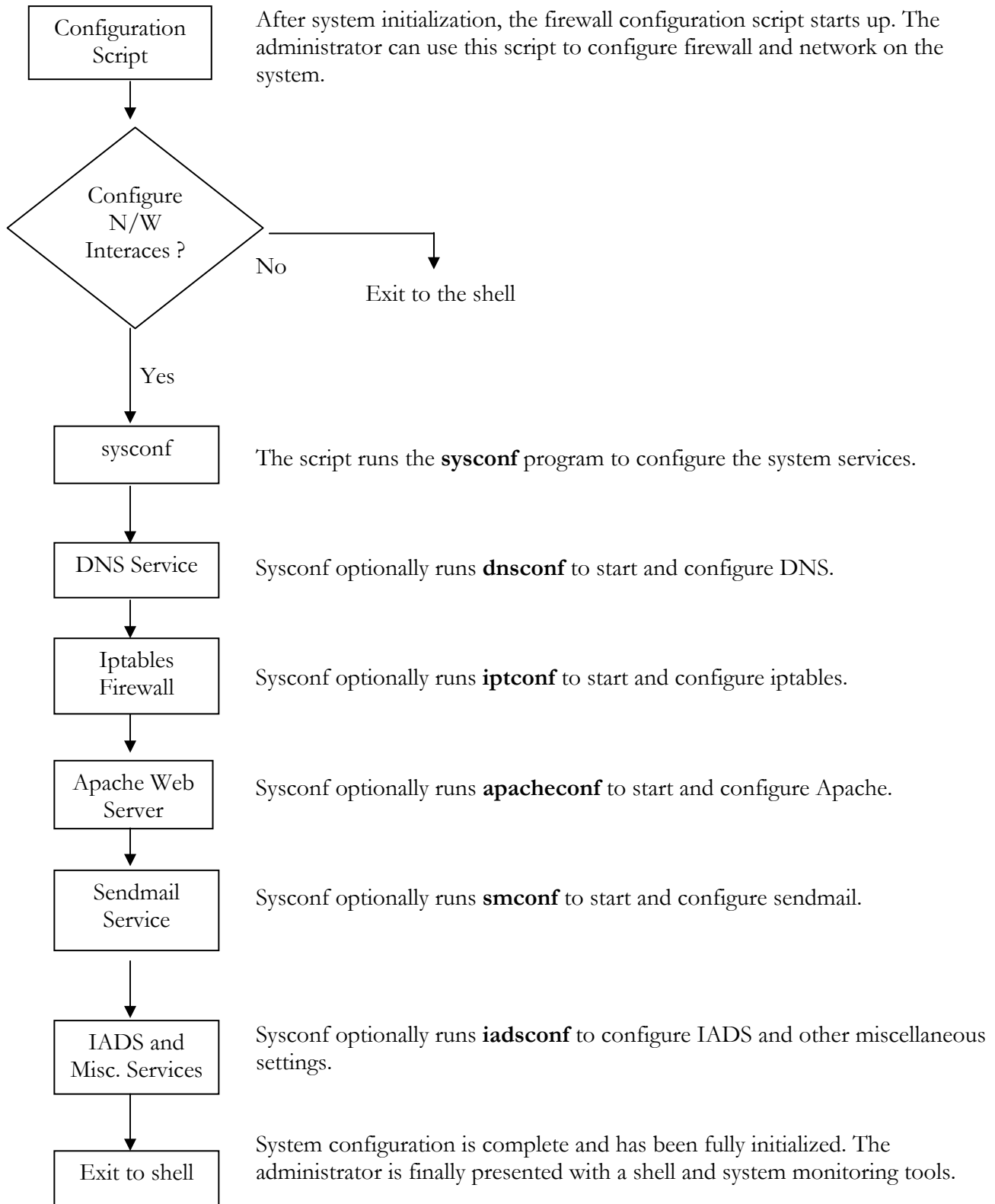
Priority : Should

Status : Implemented

CHAPTER 3 - DATA FLOW DIAGRAMS

3.1 FIREWALL CD BOOTUP SEQUENCE





3.2 INTELLIGENT ANOMALY DETECTION SYSTEM DATA FLOW DIAGRAM

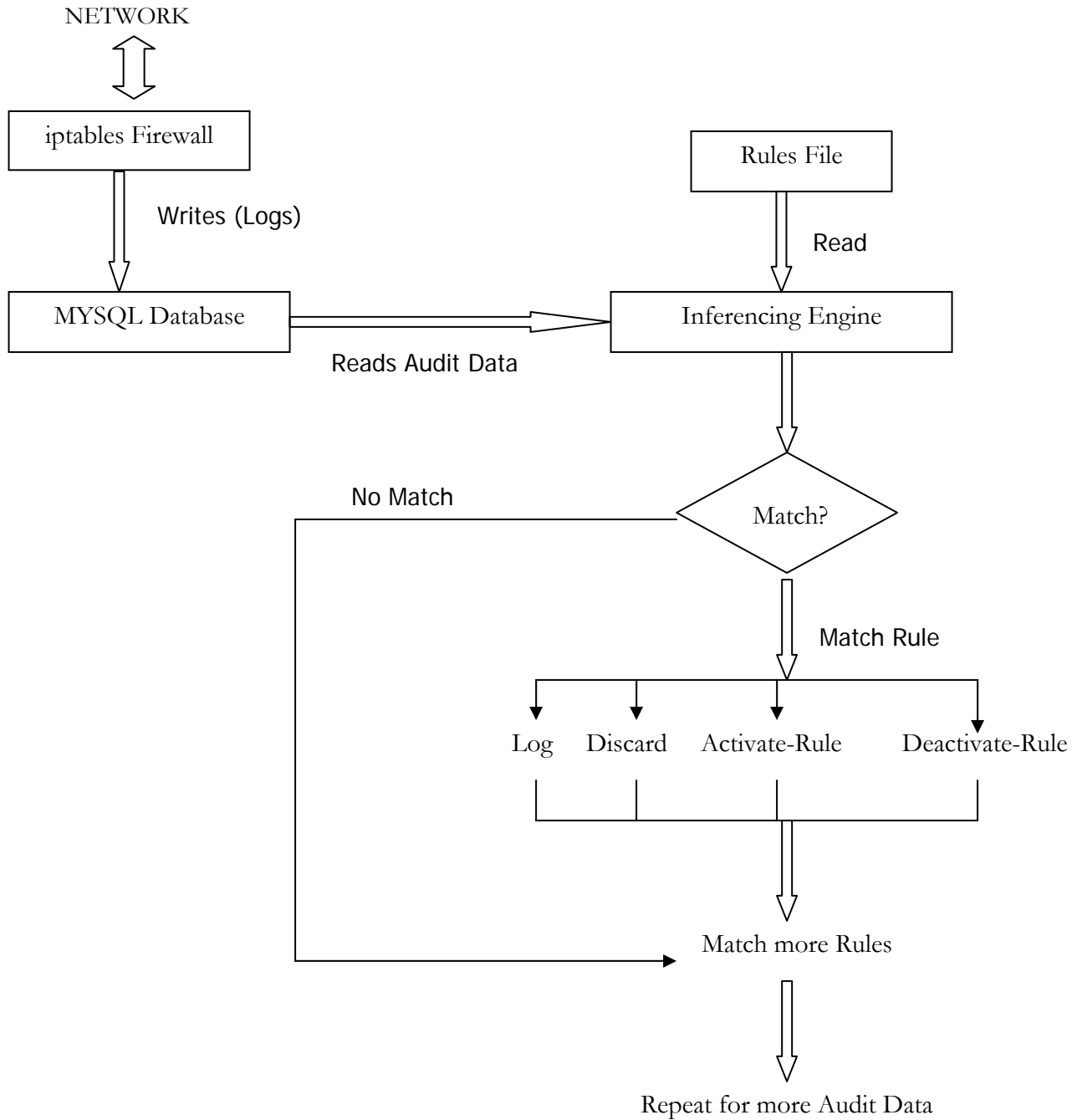


Fig 3.1 LADS Data Flow Diagram

CHAPTER 4 - DESIGN

The overall design process of the bootable CD firewall can be divided into two distinct phases, the creation of the CD-based firewall Linux system and the development of IADS. These two phases are elucidated in detail below.

4.1 DESIGN PHASE I - THE BOOTABLE FIREWALL CD

The primary goal of this design phase is to build a fully self-sufficient bootable CD containing the Linux OS architecture along with software implementing a Firewalling gateway and an Internet server system. The steps undertaken in creating such a system are explained in detail below.

Step 1 - Creation of a root filesystem

The first step in creating the bootable Firewall CD is the preparation of a root filesystem, which will eventually store all the files and software needed to the run the firewall and server components off the CD. This filesystem is first built on a normal fixed disk partition occupying about 650 MB of hard disk space; along with a 100 MB swap partition. In this project, RedHat Linux distribution Version 7.2 has been used to install a minimal filesystem consisting of the following package categories:

- Networking Workstation
- Firewalling and Routing
- Web and Mail Services
- Miscellaneous System Utilities

Step 2 – Customization of the filesystem

Once the basic root filesystem has been installed, it must be tailored to suit the firewall CD requirements. The steps for doing so are listed below.

- The **ntsysv** program is run to customize the system startup procedure, and all system services except **network** and **xinetd** are disabled. This ensures that unnecessary network services are not run at startup, and only essential system ports remain open.
- Additional packages and libraries required for functioning of the firewall and anomaly detection software are installed. These include the following.
 - The **netconf** network configuration utility.
 - The **ipmenu** iptables rules manipulation utility.
 - The **ulogd** logging module for iptables.
 - The **MySQL** database software for storing audit logs.
 - The **iptraf** network monitoring utility.
 - The **IADS** program and its library dependencies, **libxml** and **libidmef**.
 - The system configuration utilities, **sysconf**, **dnsconf**, **apacheconf** etc.
 - The **fwauth** authentication program.
- The **/etc/rc.d/rc.sysinit** file is modified to print the custom message “**Welcome to CD Firewall**” during boot-up. This file is responsible for initializing key system services and calling the **/etc/rc.d/rc** script, which in turn starts run-level specific services.
- The **/etc/rc.d/rc.local** script, loaded at the end of the initialization process, is modified so as to finally load the **/bin/sysconf** system configuration program.

- The **/etc/inittab** configuration script, which is used to control the behavior of the **init** program, is modified so as to directly load the **bash** shell program from **/bin** instead of the **login** program. Furthermore, inittab is altered to provide the system user with 4 terminal shells on **/dev/tty[1-4]**.
- The **/etc/rc.d/init.d/halt** script, which controls the system shutdown procedure is modified to adapt to a CD-ROM based filesystem.
- The swap partition is disabled by commenting out the entry in the **/etc/fstab** file.
- Additional space is recovered from files by cleaning up the directories in the **/var/log** subdirectory.

Step 3 – Development of the authentication program

The bootable firewall CD provides a secure mechanism for authenticating the administrator before the system services are initialized. The **fwauth** program developed in the course of this project serves this purpose. Some of the key features of this program are given below.

- It is an ELF (Extensible and Linkable Format) executable.
- It can be deployed on RedHat Linux (UNIX System V) compliant booting mechanisms.
- It is executed immediately after the kernel is initialized.
- It is loaded **before** any system services are initialized.
- It provides a **one-way encryption scheme** for password protection using the **DES algorithm** with 54-bit keys.

The procedure for creation and installation of the authentication script is as below.

- The **password creation program** is first used to create and the authentication password to be used. The source code for this program is given in the Source Code Appendix. (**fwpcrate.c**)
- This program accepts a confirmed password, and encrypts using the **crypt()** function.
- The encrypted password is stored in a simple text file on the root filesystem.
- The **fwauth** program is then set up to execute by specifying its path in the **linuxrc** Ramdisk initialization script.
- The **authentication program** works according to the following steps. The source code for this program is given in the Source Code Appendix. (**fwauth.c**)
- It interrupts the boot process and prompts for a password.
- It also prevents the typed password from echoing to the user output screen by altering the terminal setting.
- The accepted password is subjected to the one-way encryption scheme using the **crypt()** function. The **crypt()** library call is described in the Appendix.
- This encrypted password is compared with the one created and stored in the text file as described above.
- If the passwords match, the user is successfully authenticated.
- If authentication fails, the system is halted.

Step 4 – Building a custom kernel

An important step in the process of building the bootable firewall system is the building of a customized Linux kernel that can be used for booting the CD. This kernel incorporates options

for networking, firewalling and support loadable kernel modules for adding additional functionality. This kernel is built on another Linux system pre-installed with development packages and the kernel sources. The steps to be executed for building the kernel are given below.

- The system is booted into the Graphical X mode.
- At the console prompt, the kernel source directory is accessed at **/usr/src/linux**.
- The following command is then run to open up a Kernel configuration dialog.

```
# make xconfig
```

- A default kernel options configuration file is loaded from **/usr/src/linux/configs**.
- Kernel options pertaining to networking, firewalling and loadable modules are specifically selected.
- The configuration is saved, and the **xconfig** tool is exited.
- The following commands are executed to configure the kernel dependencies.

```
# make dep
```

```
# make clean
```

- The kernel image file is built using the command:

```
# make bzImage
```

- The compiled image is stored in the directory **/usr/src/linux/arch/i386/bzImage**. This file is copied into a temporary build directory using:

```
# mkdir /root/bootcd
```

```
# cp /usr/src/linux/arch/i386/bzImage /root/bootcd
```

- The external kernel modules are now compiled using:

```
# make modules
```

- The compiled modules are installed into `/lib/modules/<kernel-version>custom` using the command:

```
# make modules_install.
```

- The `/lib/modules/<kernel-version>custom` directory is copied into the customized root filesystem built previously:

Step 5 – Building the Ramdisk Image

The bootable Firewall CD uses a Ramdisk as a pseudo-root filesystem for its operation. A **Ramdisk** simulates a disk filesystem in memory. During the CD boot up process as illustrated in the flow diagram given previously, the kernel and an initial Ramdisk of size **16MB** are first loaded into memory and the kernel is executed.

The kernel then transfers control to the **linuxrc** script. This script runs on a simple shell called the **nash** shell stored on the Ramdisk filesystem, and mounts the CD-ROM and calls the authentication program at the end. After exiting the **linuxrc** script, the kernel loads and runs the **init** program, which completes the rest of the initialization and brings up the system services.

The components forming the structure of the Ramdisk image and the steps for building them is now detailed further.

- The following command is executed to create a default Ramdisk image that is then modified.

```
# cd /root/bootcd
```

```
# /sbin/mkinitrd -v -f initrd.img 2.4.7-10
```

- The Ramdisk now mounted onto a directory using the following commands:

```
# zcat initrd.img > initrd

# mkdir /mnt/initrd

# mount -o loop initrd /mnt/initrd
```

- In the Ramdisk, the system directories are created as symbolic links to the actual directories to be mounted and accessed from the CD-ROM. The commands for creating the links are as follows.

```
# cd /mnt/initrd

# mkdir -p mnt/cdrom

# cd mnt/cdrom

# mkdir bin sbin boot dev lib root usr

# cd /mnt/initrd

# ln -s mnt/cdrom/* .

# mkdir proc tmp
```

- The device files and kernel modules required for mounting the CD-ROM are copied to the Ramdisk.

```
# cp -dpR /dev/hd[bcd]* mnt/cdrom/dev

# cd /lib/modules/<kernel-version>/drivers/cdrom/

# cp -dpR cdrom.o ide-cd.o /mnt/initrd/lib
```

- A **static executable** program called **cdmount** is created for mounting the CD-ROM. The C code for creating **cdmount** is given in the Source Code Appendix (**cdmount.c**). The source code is compiled and the executable is copied to the Ramdisk.

```
# cd /mnt/bootcd  
  
# gcc -O2 -static -s -o cdmount cdmount.c  
  
# cp cdmount /mnt/initrd
```

- The **linuxrc** script is now created in Ramdisk filesystem. The code for the **linuxrc** script is given in the Source Code Appendix. The sequence of steps executed by this script is given below.
 1. The **nash** shell is invoked as the script interpreter.
 2. The CD-ROM driver modules are loaded into memory using **insmod**.
 3. The using the loaded CD-ROM driver, the **cdmount** program is called to mount the CD-ROM at the **/mnt/cdrom** subdirectory on the Ramdisk filesystem.
 4. The **/proc** filesystem is mounted.
 5. The Ramdisk filesystem is set to be the root filesystem, with symbolic links to the actual files stored on the CD.
 6. The **fwauth** authentication program is called to authenticate the user.
 7. The script exits and control is transferred to the **/sbin/init** program.
- Finally, the Ramdisk filesystem is unmounted and compressed and stored as a Ramdisk image file, using the commands below.

```
# umount /mnt/initrd  
  
# losetup -d /dev/loop0  
  
# cd /mnt/bootcd  
  
# gzip -9 initrd
```

Step 6 – Building the bootable image file

In this step of the creation process, a bootable image of the kernel and associated files is built on a floppy disk. This image will then be used for writing a bootable CD-ROM. The steps for creating this image file are given below.

- An empty 3-1/2 inch floppy disk is inserted into the drive, and the following command is executed to create a Linux ext2 filesystem on it. This mini-filesystem is called the **Kernel Filesystem**.

```
# mke2fs -i 8192 -m 0 /dev/fd0
```

- The floppy is now mounted, and the previously built custom kernel image, essential boot files, and the Ramdisk image are copied to the floppy.

```
# mount -t ext2 /dev/fd0 /mnt/floppy
```

```
# cd /mnt/bootcd
```

```
# cp bzImage initrd.img /mnt/floppy
```

```
# cd /mnt/floppy
```

```
# mkdir dev boot
```

```
# cp -dpR /dev/{fd0, null} dev
```

```
# cp -dpR /boot/{boot.b, map} boot
```

- Next, a configuration file for installing the LILO boot loader is created on the floppy, called **lilo.conf**. This file provides information to the LILO boot loader installer about what changes are to be made to the boot record of the floppy image. The contents of this file are given below.

```
# LILO.CONF FOR BOOT FLOPPY IMAGE #
```

```
boot          = /dev/fd0
install      = /boot/boot.b
```

```

map          = /boot/map
read-write
backup       = /dev/null
compact
image        = /bzImage
label        = firewall
root         = /dev/fd0
initrd       = /initrd.img
append       = "ramdisk_size=16384k"
    
```

- Now LILO is installed onto the boot sector of the floppy using the command:

```
# /sbin/lilo -v -C lilo.conf -r /mnt/floppy
```

- Finally, the bootable floppy image is created on the hard disk using the **dd** command.

```
# cd /mnt/bootcd

# dd if=/dev/fd0 of=boot.img bs=10k count=144
```

Step 7 – Creating the bootable CD ISO image

The specialized filesystem along with the bootable image created in the previous step are now combined to create a bootable CD image that will then be written onto a CD.

The Bootable Firewall System uses the **El Torito** standard for creating bootable CD-ROMs. The El Torito standard works by making the CD drive appear, through BIOS calls, to be a normal floppy drive. In the headers of the ISO filesystem, a pointer to this image stored for use during booting. The BIOS will then access this image from the CD, and for all purposes, will behave as if it were booting the system from the floppy drive. This allows a working LILO bootable disk, for example, to simply be used as is for creating bootable CD-ROMs.

The steps for building such an El-torito standard CD image are given below.

- The partition on which the firewall root filesystem was built is mounted onto a directory in some other Linux installation.

```
# mkdir /mnt/fw

# mount -t ext2 <device-file> /mnt/fw

# cp /root/bootcd/boot.img /mnt/fw/boot
```

- Now the bootable CD image is built using the **mkisofs** which builds an image file over an ISO9660 filesystem, which is a industry standard for CD filesystems.

```
# mkisofs -j -r -b /mnt/fw/boot.img -c /mnt/fw/boot.cat -o
bootcd.iso /mnt/fw
```

Step 8 – Writing the bootable CD

In the final step in the CD creation process, the bootable CD **bootcd.iso** image file created in the previous step is finally written onto an empty CD-R. The following commands are executed for doing so.

- The device ID of a pre-configured IDE/SCSI CD-Writer device is identified using the following command:

```
# cdrecord -scanbus
```

- The device ID noted above is used to write the bootable CD image onto the CD.

```
# cdrecord -v dev=<deviceID> speed=4 -data=bootcd.iso
```

The CD thus created can now be deployed as a **self-sufficient bootable firewall system**.

4.2 THE BOOTABLE FIREWALL CD USER INTERFACE

The bootable firewall CD system provides the user with a collection of interface programs for configuring the system services. These programs are loaded immediately after the system has finished booting to enable the administrator to readily configure the system services, and can be also be accessed at any time from the shell prompt. Given below is a brief description of the system configuration utilities bundled with the CD. The source files for these programs are included in the Source Code Appendix.

- **Sysconf** – This program is the central interface that provides links to the other service-specific configuration utilities described below.
- **Network configuraton** – **Sysconf** can be used to configure the network interfaces, set IP addresses etc.
- **Dnsconf** – Provides an interface for starting, stopping the DNS service, with options for directly editing the configuration files.
- **Iptconf** – Provides an interface for starting and stopping the iptables Firewall, and for manipulating iptables filtering and NAT rules.
- **Apacheconf** – This program allows the administrator to start and stop the apache web server, with options for modifying the server configuration, and viewing the server logs.
- **Smconf** – Provides an interface for starting and stopping the sendmail Mail Transfer Agent, and for modifying the sendmail configuration file.
- **Iadsconf** – Provides an interface for accessing the intelligent anomaly detection system. This utility is described further in future sections.
- **Miscellaneous configuration** – **sysconf** can also be used to configure the keyboard settings, timezone etc.

4.3 DESIGN PHASE II - THE INTELLIGENT ANOMALY DETECTION SYSTEM

The second part of the design process is the development of the Intelligent Anomaly Detection System, which is incorporated into the bootable Firewall CD system. The data flow diagram of the IADS system has been illustrated previously. The functional components of IADS are now explained in detail.

4.3.1 THE IPTABLES FIREWALL

Iptables is an extensible packet filtering system built over the **Netfilter** packet selection architecture. Netfilter provides a framework for packet selection and mangling, deployed by iptables to provide a flexible packet filtering mechanism.

The diagram below depicts how a packet traverses the iptables/netfilter architecture from the point when it enters the network interface. Packets with different destinations pass through different filtering chains, namely **Input**, **Forward** and **Output**.

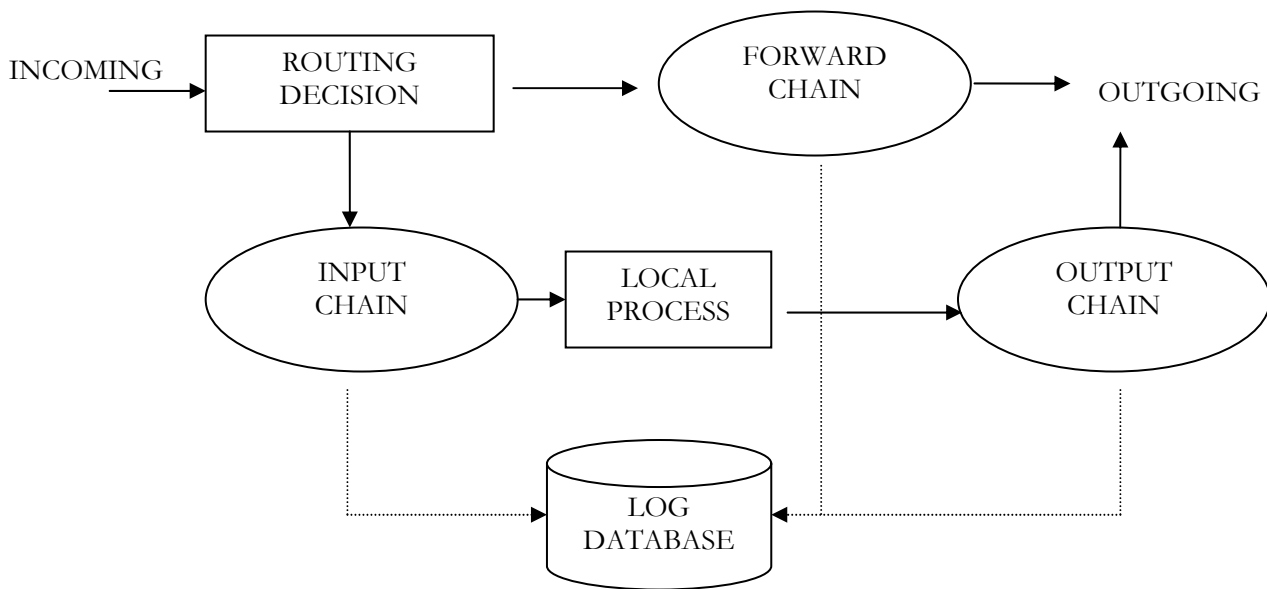


Fig 4.1 Iptables packet traversal

Rules in these chains specify **targets** that are executed whenever a packet matches the condition component of that rule. Iptables provides sophisticated userspace logging facilities collectively called **ULOG**, through the netfilter architecture, by which the targets of rules can log records stored at different types of configurable destinations. The ULOG target specifically provides options for logging to multicast groups and MySQL databases.

4.3.2 THE MYSQL DATABASE

The database logging facility provided by ULOG is used by IADS for recording netfilter packet hits to a specially constructed MySQL database. The description of the database table that stores the ULOG output is given below. The fields of the table described here store values of all the fields read from the packet IP, TCP, UDP and ICMP headers. This information is then used as input to the inferencing engine of IADS, which combines it with the specified signatures or rules to generate alerts.

FIELD	DATATYPE	DESCRIPTION
id	INT UNSIGNED AUTO_INCREMENT UNIQUE, KEY index_id (id)	The ID of the log entry
ip_saddr	INT UNSIGNED,	IP source address
ip_daddr	INT UNSIGNED,	IP destination address
ip_protocol	TINYINT UNSIGNED,	Transport protocol number
ip_tos	TINYINT UNSIGNED,	IP TOS field
ip_ttl	TINYINT UNSIGNED,	IP TTL field
ip_totlen	SMALLINT UNSIGNED,	Total IP packet length
ip_ihl	TINYINT UNSIGNED,	IP header length
ip_csum	SMALLINT UNSIGNED,	IP checksum
ip_id	SMALLINT UNSIGNED,	IP Identification field
ip_fragoff	SMALLINT UNSIGNED,	IP fragment offset field
tcp_sport	SMALLINT UNSIGNED,	TCP source port
tcp_dport	SMALLINT UNSIGNED,	TCP destination port
tcp_seq	INT UNSIGNED,	TCP sequence number
tcp_ackseq	INT UNSIGNED,	TCP acknowledgement number
tcp_window	SMALLINT UNSIGNED,	TCP window size
tcp_urg	TINYINT,	TCP URG flag
tcp_urgp	SMALLINT UNSIGNED,	TCP urgent data pointer
tcp_ack	TINYINT,	TCP ACK flag
tcp_psh	TINYINT,	TCP PSH flag
tcp_rst	TINYINT,	TCP ACK flag

tcp_syn	TINYINT,	TCP ACK flag
tcp_fin	TINYINT,	TCP ACK flag
udp_sport	SMALLINT UNSIGNED,	UDP source port
udp_dport	SMALLINT UNSIGNED,	UDP destination port
udp_len	SMALLINT UNSIGNED,	UDP packet length
icmp_type	TINYINT UNSIGNED,	ICMP type field
icmp_code	TINYINT UNSIGNED,	ICMP code field
icmp_echoid	SMALLINT UNSIGNED,	ICMP ECHO ID
icmp_echoseq	SMALLINT UNSIGNED,	ICMP ECHO sequence number
icmp_gateway	INT UNSIGNED,	ICMP gateway
icmp_fragmtu	SMALLINT UNSIGNED,	ICMP fragmentation MTU

Table 4.1 Database ULOGD - Table ULOG

IADS accesses this ULOG database table maintained by MySQL by connecting to the MySQL Server daemon through a Unix domain socket. The IADS MySQL client module uses the MySQL-to-C API to access the database and issue SQL queries. This API library is detailed further in the Appendix.

4.3.3 THE IADS RULE DATA MODEL

A rule encapsulates an apriori signature used by IADS inferencing engine to check whether a packet log entry in the database matches this signature. If a match occurs, an alert is raised, and the relevant actions corresponding to the rule are executed. This section describes the structure of the IADS rule construct and its fields in detail.

IADS defines an XML based rule specification language. The IADS XML Rule Data Type Definition or DTD file (**iads-rules.dtd**) defines the set of data elements and values that are legal in the XML rule file. This DTD thus is a specification of the syntax to be used for writing IADS rules. The administrator provides IADS with a valid rule file corresponding to this DTD, which is interpreted and collated by the IADS XML parser.

The set of XML elements forming a rule in a rule file are illustrated below in the IADS rule data model diagram. The complete IADS XML DTD is given in the Appendix.

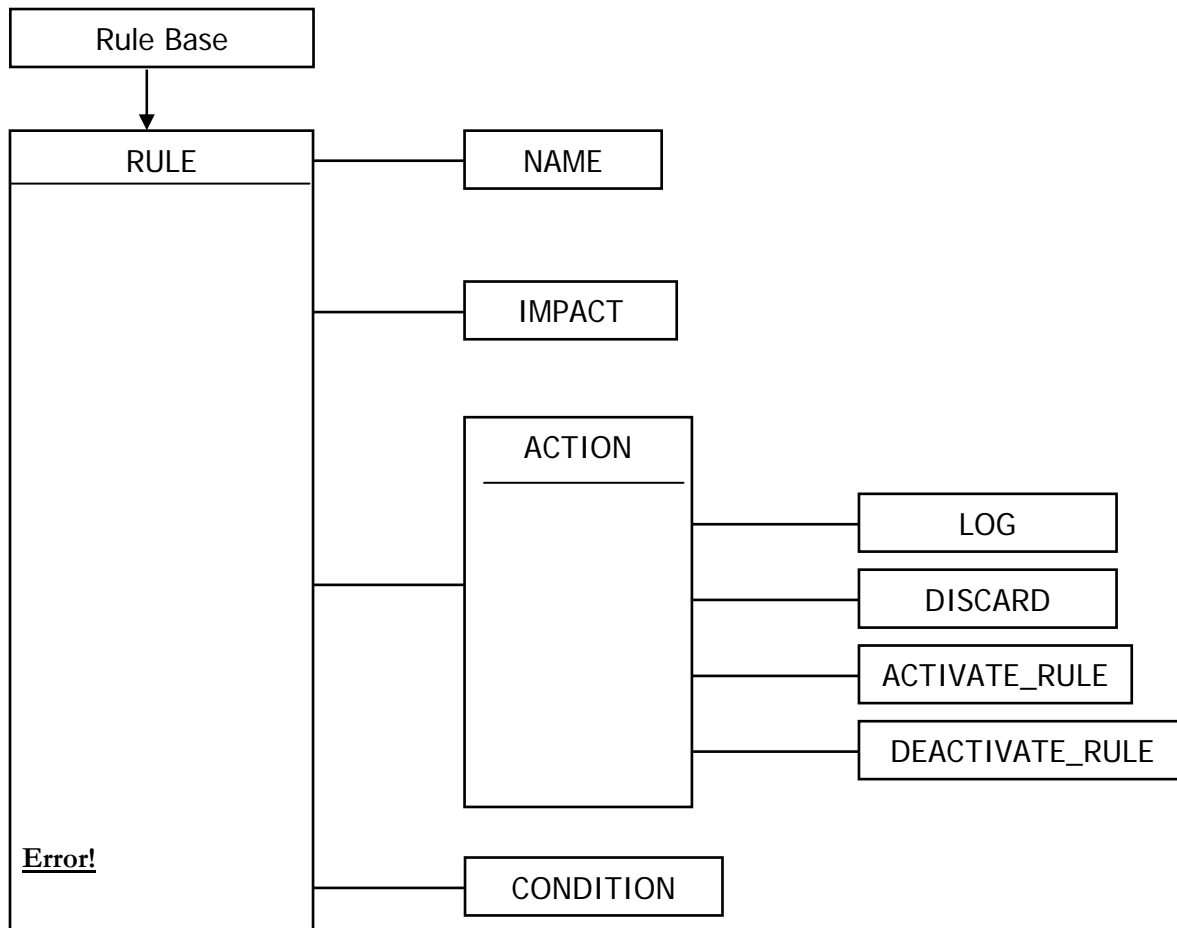


Fig 4.2 The LADS rule data model

The elements of the Rule data model as defined in the IADS XML DTD are explained below.

Rule base - The Rule Base is the top-level element in the data model. A Rule Base must have at least one rule.

Rule - A rule associates a condition with classification and identification information as well as an action that the system will perform on successful validation of the condition. A rule has two attributes: state and type.

1. **Type** (optional) - The type determines what kind of events the rules apply to. There are two values for type: **stateless** and **stream**. Rules of type **stateless** apply to packets independently of any correlations to saved information. Rules of type **stateless** must have a condition. This is the default value.
2. **State** (optional) - The state determines the initial state for the rule. There are two values for state: **active** and **inactive**. An active rule has its condition applied to the relevant event. It remains active until the system terminates or another rule deactivates it. This is the default value. An inactive rule does not have its condition tested. It remains inactive until another rule activates it.

Name - The name is the internal name of the rule. In absence of a formal classification using the reference element, the name will be used to classify the rule. The name must contain a string of arbitrary length.

Reference - The reference is a formal classification of a rule. It must have a rule name element. It may have a reference url element. A reference has one attribute: origin.

1. **Origin** (required) -The origin describes the origin of the reference information. Origin directly corresponds to the origin attribute of the Classification element in the IDMEF DTD. Origin has five values: "unknown", "bugtraqid", "cve", and "vendor-specific".

Reference name - The reference name is the reference source ID of the rule.

Reference url - The reference url is a url containing reference information for the rule. If none is provided, an appropriate url will be created using the reference origin and the name.

Impact - The impact describes both the severity and probable completion status of the event. Impact has three attributes, "type", "completion", and "severity".

1. **Type** (optional) - Type describes the type of event. There are six values for type, "admin," "dos," "file," "recon," "user," and "other". The value "other" is the default.

2. **Completion** (optional)- Completion describes the probable completion status of the event. There are three values for completion, "attempted," "successful," and "failed." The value "attempted" is the default.
3. **Severity** (optional) - Severity describes the severity of the event, there are three values, "low," "med," and "high." The value "high" is the default.

Action - The action element describes what action the system should perform when the rules condition is met.

1. **Log** - The log element specifies that IADS should log the event. Combining this element with the "drop" element in the same rule will cause the event not to be logged and is logically invalid.
2. **Drop** - The drop element specifies that the event should be ignored. No logging will occur, even if another rule matches the event.
3. **Activate rule** - The activate rule element contains the name of another rule that will be activated. The rule named does not have to succeed the rule referencing it in the rule document.
4. **Deactivate rule** - The deactivate rule element contains the name of another rule that will be deactivated. The rule named does not have to succeed the rule referencing it in the rule document.

Condition - The condition element is the root of an arbitrarily complex Boolean expression that must be satisfied for the actions specified in the rule to be executed. A condition may contain only one element that is either one of the Boolean elements or a match element. Conditions are only valid inside of stateless rules. The structure of the condition construct is illustrated in the diagram below.

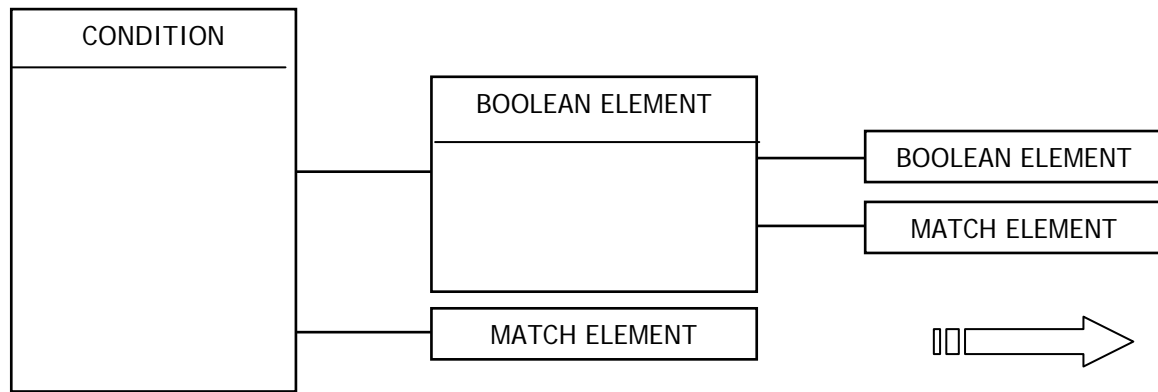


Fig 4.3 The LADS rule condition construct

The individual elements in the Condition construct are explained further.

Boolean Elements

1. **AND** - The AND element one of the four Boolean elements that can occur within the condition elements. All Boolean elements may contain themselves. There is no hard limit on the depth of Boolean expressions but expressions that are too deep will likely be inefficient. The AND element must have at least two children which may be any combination of Boolean or match elements. The AND condition evaluates true if all of its children evaluate true.
2. **OR** - The OR condition is one of the four Boolean elements and is similar to AND. It evaluates true if one or more than one of its children evaluate true.
3. **XOR** - The XOR condition is one of the four Boolean elements and is similar to AND. The XOR condition evaluates true if only one of its children evaluate true.
4. **NOT** - The NOT condition is one of the four Boolean elements and is similar to AND. It differs in that it may only have one child that may be either a match or Boolean element. It evaluates true if its child evaluates false.

Match Elements - The following match elements correlate to header values in IP, TCP, UDP, or ICMP packets. In most cases, ranges may be specified using the hyphen.

1. **ip_version** – The version of the IP protocol header, invariably IPv4.
2. **ip_header_length** – The length of the IP header.
3. **ip_tos** – The IP Type of Service field.
4. **ip_total_length** – The IP total packet length field.
5. **ip_identification** – The IP identification field.
6. **ip_df** – The IP DON'T FRAGMENT flag. This is a flag element and any value it contains will be ignored. It will evaluate true if the IP don't fragment flag is true. Like all match elements, it may be negated with the NOT Boolean element.
7. **ip_mf** - The IP MORE FRAGMENTS flag. This is a flag element similar to ip_df.
8. **ip_offset** – The IP fragment offset field.
9. **ip_ttl** – The IP Time To Live field.
10. **ip_protocol** – The IP transport layer protocol code. 1 for ICMP, 6 for TCP and 17 for UDP.
11. **ip_checksum** – The IP Checksum field.
12. **ip_source_address** – The IP address of the source of the packet. The ip_source_address element may take a range of values using a hyphen.
13. **ip_destination_address** – The IP address of the destination of the packet. Similar to ip_source_address.
14. **ip_address** - The value contained in the ip_address element will evaluate true if either the destination of source IP address match.
15. **tcp_source_port** – The TCP port of the source application.

16. **tcp_destination_port** – The TCP port of the destination application.
17. **tcp_port** - The value contained in the tcp_port element will evaluate true if either the TCP destination or source port match.
18. **tcp_sequence_number** – The sequence number field in the TCP header.
19. **tcp_acknowledge_number** - The TCP acknowledgement number field.
20. **tcp_header_length** – The TCP header length.
21. **tcp_urg** – The TCP URG flag bit.
22. **tcp_ack** – The TCP ACK flag bit.
23. **tcp_psh** – The TCP PSH flag bit.
24. **tcp_rst** – The TCP RST flag bit.
25. **tcp_syn** – The TCP SYN flag bit.
26. **tcp_fin** – The TCP FIN flag bit.
27. **tcp_window_size** – The TCP window size field.
28. **tcp_checksum** – The TCP checksum field.
29. **tcp_urgent_pointer** – The TCP urgent pointer field.
30. **udp_source_port** – The UDP source port field.
31. **udp_destination_port** – The TCP destination port field.
32. **udp_port** - The value contained in the udp_port element will evaluate true if either the udp destination or source port match.
33. **udp_length** – The UDP header length field.

34. **udp_checksum** – The UDP checksum field.

35. **icmp_type** – The ICMP type field.

36. **icmp_code** – The ICMP code field.

37. **icmp_checksum** - The ICMP checksum field.

Shown below is an example rule condition tree. The corresponding IADS rule construct that codifies this condition according IADS rule syntax is also described.

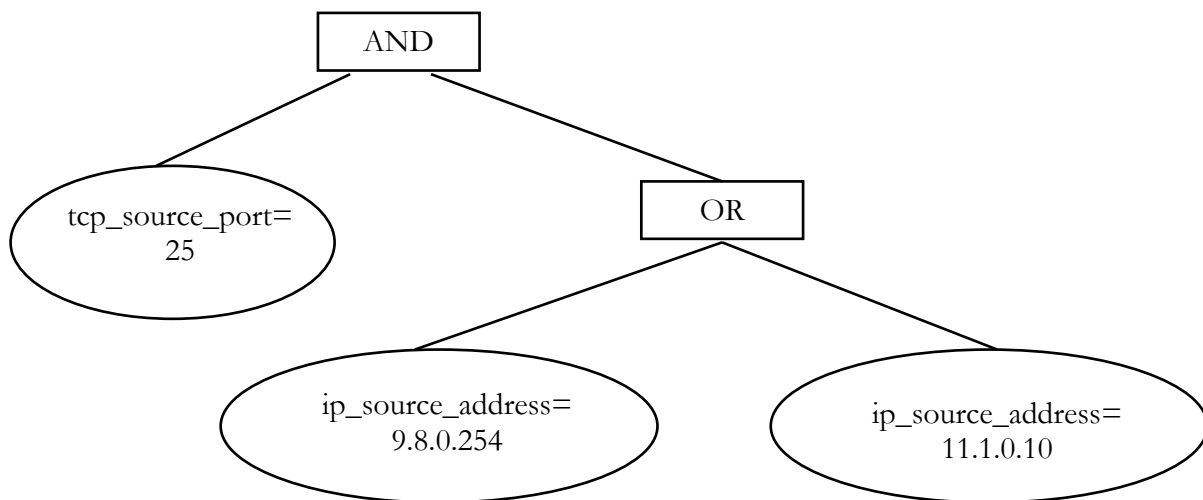


Fig 4.4 A example rule condition tree

```

<rule_base>
  <!-- BEGIN Example IADS Rule File -->
  <rule>
    <name>EXAMPLE</name>
    <impact type="admin" />
    <action>
      <log/>
    </action>
    <condition>
      <AND>
        <tcp_source_port>25</tcp_source_port>
        <OR>
          <ip_source_address>9.8.0.254</ip_source_address>
          <ip_source_address>11.1.0.10</ip_source_address>
        </OR>
      </AND>
    </condition>
  </rule>
</rule_base>
  
```

4.3.4 THE IADS ALERT FORMAT

IADS uses the **Intrusion Detection Message Exchange Format** or IDMEF standardized message format to report alerts to the administrator. The IDMEF XML library, **libidmef** provides the ability to create IDMEF XML messages from raw data. The raw data is created by IADS and molded together into IDMEF XML messages using the functions provided by libidmef. To facilitate explanation of the IADS IDMEF alert model, a brief introduction the IDMEF standard is provided below.

The Intrusion Detection Message Exchange Format (IDMEF) is a standard data format that automated intrusion detection systems can use to report alerts about events that they deem suspicious. The data model has been designed to provide a standard representation of alerts in an unambiguous fashion, and to permit the relationship between simple and complex alerts to be described. This standard format enables interoperability among a variety of intrusion detection systems, allowing users to combine the deployment of these systems to obtain an optimal implementation.

The IDMEF standard addresses the following important issues in its design.

- Alert information is inherently heterogeneous. Some alerts are defined with very little information, such as origin, destination, name, and time of the event. Other alerts provide much more information, such as ports or services, processes, user information, and so on. The data model that represents this information must be flexible to accommodate different needs.
- Intrusion detection environments are different. Some analyzers detect attacks by analyzing network traffic; others use operating system logs or application audit trail information. Alerts for the same attack, sent by analyzers with different information sources, will not contain the same information.

- Analyzer capabilities are different. The data model must allow for conversion to formats used by tools other than intrusion detection analyzers, for the purpose of further processing the alert information.
- Operating environments are different. Depending on the kind of network or operating system used, attacks will be observed and reported with different characteristics. The data model should accommodate these differences.
- Commercial vendor objectives are different. For various reasons, vendors may wish to deliver more or less information about certain types of attacks.

IDMEF uses the Extensible Markup Language (XML) to define its data model. XML is a metalanguage -- **a language for describing other languages** -- that enables an application to define its own markup. XML allows the definition of customized markup languages for different types of documents and different applications. XML allows applications to define set of identifier datatypes in the form of tags with name=value pairs.

The IDMEF standard employs XML to define a collection of pre-defined data structures in the IDMEF Document Type Definition (DTD) syntax. Specifically, the IDMEF DTD defines a set of datatypes including Integers, Real numbers, Characters, Strings, Bytes, and Enumerated types, Date-Time formats, NTP timestamps, Port Lists and Unique Identifiers. These datatypes are then used to define complex data structures, which anomaly detection applications can use to codify the information about an intrusion alert.

These IDMEF data structures, along with a defined hierarchy structure of the inter-relationships between them, form the IDMEF data model. Further details about IDMEF data model are provided in the Appendix.

The IADS application employs the IDMEF data model to represent the information about an anomaly alert generated when a packet log entry matches a rule condition. These alerts are stored in the IADS alert file.

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

A sample IDMEF alert entry generated by IADS is shown below. It contains key information about time of the alert, the source and target nodes, the network protocol etc.

```
<?xml version="1.0"?>
<!DOCTYPE IDMEF-Message PUBLIC "-//IETF//DTD RFC XXXX IDMEF v1.0//EN"
"./dtds/idmef-message.dtd">

<IDMEF-Message version="0.3">
  <Alert ident="1" impact="attempted-admin">
    <Analyzer analyzerid="IADS v1.0"/>

    <CreateTime ntpstamp="0xc0a1ac96.0x56250f84">2002-05-31T08:02:30Z
  </CreateTime>

    <Source>
      <Node>
        <Address category="ipv4-addr">
          <address>127.0.0.1</address>
        </Address>
      </Node>
      <Service>
        <port>1024</port>
        <protocol>UDP</protocol>
      </Service>
    </Source>

    <Target>
      <Node>
        <Address category="ipv4-addr">
          <address>127.0.0.1</address>
        </Address>
      </Node>
      <Service>
        <port>8000</port>
        <protocol>UDP</protocol>
      </Service>
    </Target>

    <Classification origin="unknown">
      <name>EXAMPLE</name>
      <url>none</url>
    </Classification>

  </Alert>
</IDMEF-Message>
```

4.4 IADS INTERNAL ARCHITECTURE

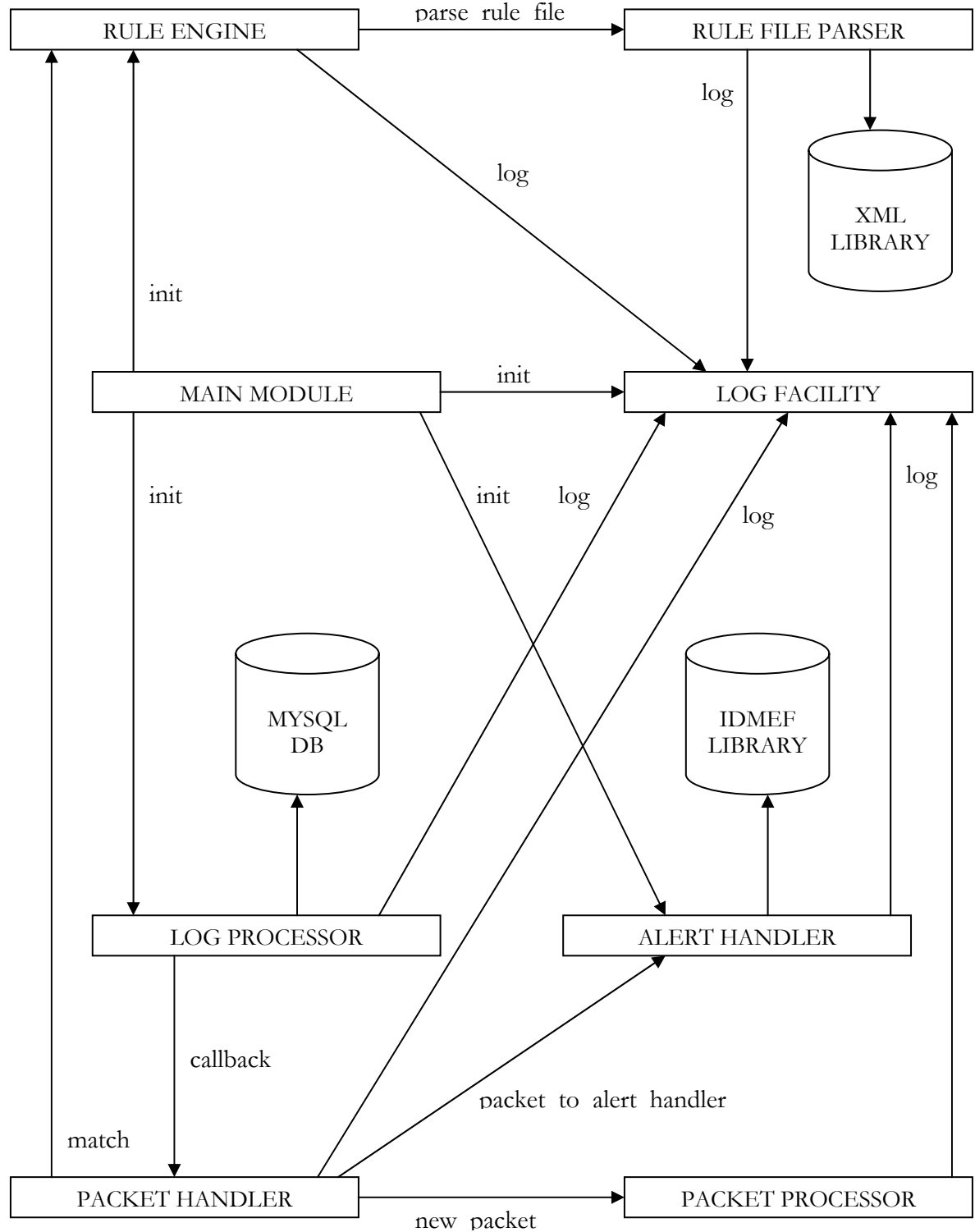


Fig 4.5 The LADS internal module diagram

The diagram above depicts the functional block components of IADS and the procedural dependencies between them. These components are now explained in detail.

The Rule Engine

The rule engine forms the core of the inferencing engine of IADS. When IADS is initialized, the rule engine calls upon the rule file parser component to read the XML format rules and builds a linked list data structure of these the rules. The condition expression of a rule is stored as a tree of arbitrary depth. During normal operation, the rule engine is called on to match the values in a packet log entry to the rule set. The rule engine then builds an array of rules whose condition trees match the packet and performs the actions specified by each rule definition.

The Rule File Parser

The rule file parser reads the IADS rule file and builds the rule data structure to be used by the rule engine. To parse the XML content of the rule file, the rule parser makes calls to the XML parser library, **libxml.so**. These calls validate the rule file against the IADS rule file DTD and return values of parsed XML data elements.

The Main Module

The main module contains the **main** function of the IADS application. This module initializes the IADS environment by processing any command line arguments and initializing all the other modules. The main module sets up the rule file parser, the alert handler document, the log processor MySQL client etc.

The Log Facility

The log facility component provides the IADS logging interface to the other components in the system. All the components make calls to the log facility to write log entries to the IADS log file stored on disk. The log facility writes formatted log records to disk, which can be used to monitor the functioning of IADS and diagnose any problems.

The Log Processor

The log processor component of IADS is responsible for extracting packet log entries from the MySQL ULOG database described above. The log processor employs the MySQL C API to make calls to the MySQL server. It connects to the MySQL server through a UNIX socket, opens the ULOGD database, and then queries it for all the new row entries in the log since the last time the query was made. The new log entries are then passed to the packet handler component for further processing. The IADS process then goes to sleep for 60 seconds, after which it repeats the database query and log processing all over again.

The Alert Handler

IDMEF alerts are generated by the alert handler component of IADS. When IADS is initialized, the alert handler sets up the default IDMEF XML alert document in memory. When a packet log matches a rule, the action for the rule is checked, and if it specifies a log action, the alert handler is called, and the packet header information is passed to it. The alert handler formats this raw data and combines it with timestamp information, rule impact information etc. to generate the complete IDMEF alert. To do so, the alert handler calls on the IDMEF XML library, **libidmef.so**. This library provides functions for manipulating the current IDMEF document initialized previously. The IDMEF alert thus created is finally written to the IADS alert file.

The Packet Handler

The log processor invokes the packet handler component as a callback routine when it reads a packet log entry. The packet header information is passed to the packet handler as a raw byte array. The packet handler then uses the packet processor component to generate a packet data structure containing all the relevant information about the packet. This structure is then sent to the rule engine component to check for rule matches. Finally, the packet handler calls the alert handler to generate the IDMEF alert.

The Packet Processor

The packet processor component is responsible for converting the raw byte data passed to it by the packet handler into a formatted packet data structure populated with data read from the

log. The packet processor also provides a library of routines for setting and getting header fields from a previously created packet data structure.

4.5 THE IADS USER INTERFACE

The following are the optional arguments that can be supplied to the IADS command-line application to configure its operation.

```
--help                display this help
--daemon              run as a daemon (default no)
--device <device>    use device (default all)
--alert-log <alert-log-file> write alerts to alert-log-file
--error-log <error-log-file> write errors to error-log-file
--rule-file <rule-file> read the rules from rule-file
--idmef-dtd-file <dtd-file> read IDMEF DTD file from dtd-file
--idmef-indent        indent IDMEF alerts
--user <user-name>   run as user user-name or uid
--group <group-name> run as group group-name or gid
```

The bootable Firewall CD system incorporates a user interface application, **iadsconf**, which provides easy access to all the features of the Intelligent Anomaly Detection System.

Some of the features of the **iadsconf** user interface program are listed below.

- Full-screen menu-based interface dialogs based on the **newt** library.
- Full featured interface for creation and deletion of IADS rule constructs.
- Provides options for starting and stopping the IADS daemon.
- Support for viewing IADS generated alerts and the IADS system log.
- Option for directly editing the IADS rule file.

CHAPTER 5 - IMPLEMENTATION

This section covers the implementation details of the Bootable CD Firewall system and the Intelligent Anomaly Detection System. Specifically, software and hardware dependency issues have been elaborated, and an overview of the source code structure of the IADS application has been discussed.

The system dependencies of the Bootable CD Firewall can be divided into two categories, namely Hardware and Software. These dependency categories are listed in the sections below.

5.1 HARDWARE DEPENDENCIES

The hardware requirements to be met in order to be able to deploy the Bootable CD Firewall system are listed below.

- An IDE CD-ROM drive of speed 32X or greater.
- A System BIOS that supports the ATAPI CD-ROM booting mechanism.
- A total system memory of 64 MB or greater.
- A VGA compatible display device.
- A US-International 101-key standard keyboard.
- Supported Networking Interface Cards for LAN connectivity.
- A secondary removable device drive for loading external modules.

5.2 SOFTWARE DEPENDENCIES

This section discusses the software and library dependencies of the Bootable CD Firewall system in terms of those of the individual components bundled on the CD-ROM.

- **Device Driver Modules** – The CD-ROM driver files, namely **cdrom.o** and **ide-cd.o** are required for mounting the root filesystem from the CD-ROM. Hence, these files are included as a part of the initial Ramdisk image
- **The fwauth authentication mechanism** – **fwauth** uses the **crypt** library routine to perform one-way DES encryption of the administrator password. It thus depends on the **libcrypt.so** library to provide this routine.
- **The system configuration utilities** – The collection of system configuration utilities explained previously, namely **sysconf**, **dnsconf**, **iptconf**, **smconf**, **apacheconf** and **iadsconf** use the **newt** user-interface API library stored in **libnewt.so** to provide the administrator with a menu-based configuration interface. This library is detailed further in the Appendix.
- **Iptables Logging** – The iptables logging facility requires the userspace logging plugin, ULOG for creating log records in MySQL databases.
- **The ipmenu iptables configuration tool** – The **ipmenu** utility bundled with the Firewall CD for manipulating iptables rules depends on the following libraries for its functioning.
 - **ncurses-ext** – An extension the ncurses terminal control API.
 - **iproute2** – A collection of advanced tools manage IP routing.
 - **Cursel** – A ncurses-based interface development library.
- **IADS dependencies** – IADS depends on the **libxml** and **libidmef** libraries for parsing rule files and creating IDMEF alerts respectively.

5.3 IADS SOURCE CODE STRUCTURE

The sources of the IADS application are organized along the lines of its internal components described previously. These functional components and the files comprising them are described in the sections below.

5.3.1 THE RULE ENGINE

The rule engine consists of three files. The **rule-engine.c** contains the functions that form the inferencing engine of IADS. **rule-engine.h** has include files, type definitions and function declarations used in **rule-engine.c**. **rule-engine-priv.h** contains declarations private to the rule engine. These include private variables, pointers to the linked list of rules, private utility functions, rule engine state variable etc. Some of the important structures and functions part of the rule engine are listed below.

- The structure representing a IADS rule built from the rule file is:

```
typedef struct _ids_rule{
    u_char* name;
    rule_reference* reference;
    attack_impact* impact;

    u_int id;
    enum ids_rule_state state;
    enum ids_rule_type type;

    ids_action* action;

    rc_node* condition;
    rc_node* stream_condition;
    rc_node* stream;

    struct _ids_rule* next;
}ids_rule;
```

- The structure representing a rule action is:

```
typedef struct _ids_action{
    enum action_type type;
    enum action_capture_level capture_level;
    struct _ids_rule* subject_rule;
```

```

    u_int alerts_per_hour; /*rate limit logging*/
    struct _ids_action* next;
}ids_action;

```

- The following structure represents a condition tree node.

```

typedef struct _rc_node{
    enum node_type type;
    union{
        struct _one_child_node*   one_child;
        struct _two_child_node*   two_child;
        struct _field_match*      field_condition;
        struct _misc_match*       misc_condition;
    }object;
}rc_node;

```

- **Init_rule_engine** – Initializes the rule engine and calls **parse_rule_file** to build the rule data structures.
- **Destroy_rule_engine** – Deallocates memory used by rule data structure.
- **Reset_rule_engine** – Resets rule data structure and destroys information about previous rule matches.
- **Get_stateless_alert_rules** – Returns the list of all rules that have matched a packet since the last engine rule reset.
- **Match_stateless** – Matches a packet against the rule set, and executes the associated action on a successful match.
- **Match_condition_tree** – Matches a packet against the condition tree recursively.
- **Do_field_match** – Match a field in the packet data structure against the value specified in the rule signature.
- **New_ids_rule** – Creates a new rule.

5.3.2 THE RULE FILE PARSER

The rule file parser contains three files. The **rule-file-parser.c** file provides the XML rule file parsing functions. The **rule-file-parser.h** file provides the definitions of the XML data types and elements used by the parser. It includes the **libxml** header files used for parsing. The **rule-file-parser-priv.h** holds the parser state variables, and other data elements used for storing file delimiters etc. The following are some of the important functions defined by the rule file parser module.

- **Parse_rule_file** – Parses a rule file base.
- **Validate_rule_file** – validates a rule file against a specified DTD.
- **Make_rule, make_rule_action, make_rule_tree_node** – These routines create the various components forming an **ids_rule** data structure.

5.3.3 THE MAIN MODULE

The main module contains three files, **main.c**, **main.h** and **main-priv.h**. The **main.h** and **main-priv.h** file define data structures for holding the command line arguments, system includes and IADS state variables. The main module contains the following functions:

- **Main** – The main function for the IADS program.
- **Get_cmdg_args** – Processes and stores command line arguments.
- **Init_IDS** – Initializes the intrusion detection engine components.
- **Destroy_IDS** – Deinitializes the system before shutdown.
- **Become-daemon** – Makes the IADS process a background daemon.

5.3.4 THE LOG FACILITY

The log facility includes the **log-facility.c**, **log-facility.h** and **log-facility-priv.h** files. The important functions defined as a part of the log facility module are given below.

- **Init_error_log** – Initializes the logging facility.
- **Destroy_error_log** – Deinitializes the logging facility
- **Error_log_message** – Accepts a message, log level and errno to be logged.

5.3.5 THE LOG PROCESSOR

The log processor module is comprised of log-processor.c, log-processor.h and log-processor-priv.h. The following data elements and functions are defined in this module.

- **MYSQL** – Encapsulates information about a MySQL client connection.
- **MYSQL_RES** – Represents a MySQL result set.
- **MYSQL_ROW** – Represents a row of the database table.
- **PACKET_CALLBACK** – The callback routine used to handle a retrieved packet log entry.
- **Init_log_processor** – Initializes the log processor by connecting to the MySQL ULOGD database.
- **Build_packet** – Builds a raw byte string containing the packet header using data from the packet log entry.
- **Start_log_processor** – Initiates the log-processing loop.
- **Destroy_log_processor** – Closes the MySQL database connection.

5.3.6 THE ALERT HANDLER

The alert handler module includes the files, **alert-handler.c**, **alert-handler.h**, and **alert-handler-priv.h**. The important data structures and functions in this module are as follows.

- The information about a generated alert to be written to the IDMEF alert file is stored in the following structure.

```
typedef struct _alert_info{
    char src_address_str [IP_ADDR_STR_LEN];
    char src_port_str [PORT_STR_LEN];

    char dst_address_str [IP_ADDR_STR_LEN];
    char dst_port_str [PORT_STR_LEN];

    u_int8_t proto;
    char* proto_str;

    char icmp_name_str [ICMP_NAME_LEN];

    int header_capture_len;
    char* header_capture;

    int packet_capture_len;
    char* packet_capture;

}alert_info;
```

- **Init_alert_handler** – Initializes the alert handler module by making calls to the **libidmef** library to set up the current XML alert document using the IDMEF DTD.
- **Destroy_alert_handler** – Deinitializes the alert handler and frees up allocated memory.
- **Packet_to_alert_handler** – generates the `alert_info` structure from raw packet data.
- **Alert_handler** – Creates an IDMEF alert message and logs to the alert file.
- **Close_log_file** – Closes the IDMEF alert file
- **Idmef_new_message** – Creates a new IDMEF message structure.

5.3.7 THE PACKET HANDLER

The files **packet-handler.c**, **packet-handler.h** and **packet-handler-priv.h** comprise the packet handler component. The important function in this module is listed below.

- **Packet_handler** – This is the main callback routine in the packet handler module. It is called by the log processor and is passed a raw packet header byte string. This raw data is converted to a packet data structure by making calls to the packet processor module. The packet data structure is in turn sent to the rule engine to check for matches, and then to alert handler to log all the successful matches, if any.

5.3.8 THE PACKET PROCESSOR

The packet processor module consists of the files **packet-processor.c**, **packet-processor.h** and **packet-processor-priv.h**. The important structures and functions in this module are now described.

- The packet data structure used to hold the raw packet data is defined as follows.

```
typedef struct _packet{
    struct timeval capture_time;
    u_int8_t trans_proto;
    u_int8_t trans_offset;
    u_int8_t app_proto;
    u_int8_t app_offset;
    u_char*   dat;
    u_int32_t dat_len;
    u_int16_t dat_word_len;
    u_int     is_fragment;
}packet;
```

- **New_packet** – Creates a packet data structure from raw packet data.
- **IP, TCP, ICMP and UDP getters** – Routines to read field values from respective headers in a packet data structure.
- **IP, TCP, ICMP and UDP setters** – Routines to set field values in respective headers in a packet data structure.

CHAPTER 6 - TESTING

This section elaborates on the testing procedures employed to test the Bootable Firewall CD system and the IADS application. An important goal of the testing process elaborated in this section is to ensure that the software requirements of the system have been fulfilled. The results of these tests have been listed, along with the description of any software tools employed for the testing process.

6.1 TESTING PHASE I - THE BOOTABLE FIREWALL CD TEST PLAN

The testing process for the bootable firewall CD system involved testing the functioning of the system as a firewall and gateway machine once all the system initialization was complete. The steps executed in this process have been listed below.

Step 1: Network Interface Test - The first testing task undertaken was to check the functioning of the network interface hardware and software on the test machine on which the bootable firewall had been configured. This was accomplished by using the **ping** utility to test the LAN connectivity. Ping sends ICMP ECHO packets to specified hosts on the network, and waits for replies. The successful ICMP ECHO replies received from other machines on the LAN demonstrated that the firewall system was successfully connected to the LAN.

At the same time, network activity occurring at the interfaces configured on the test machine was monitored using the IPTraf network monitoring utility. IPTraf is a console-based network-monitoring program for Linux that displays information about IP traffic. This program can be used to determine the type of traffic on a network, and what kind of service is the most heavily used on what machines, among others. IPTraf works on Ethernet, FDDI, ISDN, PLIP, loopback, and SLIP/PPP interfaces.

IPTraf provides the following information about the network status:

- Current TCP connections
- UDP, ICMP, OSPF, and other types of IP packets
- Packet and byte counts on TCP connections
- IP, TCP, UDP, ICMP, non-IP, and other packet and byte counts
- TCP/UDP counts by ports
- Packet counts by packet sizes
- Packet and byte counts by IP address
- Interface activity
- Flag statuses on TCP packets
- LAN station statistics

Step 2: DNS Server Test – The Linux DNS Server, **named** was tested for proper functioning using the **nslookup** program. Nslookup provides a command-based interface that can be used to issue DNS queries and test the response from the DNS server. Successful DNS lookup replies demonstrated the proper operation of the DNS service.

Step 3: iptables Firewall Test – The iptables firewall was tested by generating network traffic using a traffic generator program. Iptables Rules were set up to perform packet filtering, NAT, masquerading etc., and the iptables log output was viewed to check their functioning.

The traffic generator program **traffic** was used to generate packets for testing iptables. This program has a server and a client part with different command parameters. There are however some common parameters. These parameters are:

--client/-c: Selects client mode. Takes no options.
 --server/-s: Selects server mode. Takes no options.
 --verbose/-v: Makes the program more verbose. Takes no options.
 --quiet/-q: Makes the program less verbose. Takes no options.
 --port/-p: Selects the port to listen on for server and the port to connect to for client.

If the --client/-c flag is specified the following parameters are legal:

--tcp/-t: The maximum sized TCP packet to be sent.
 --udp/-u: The maximum sized UDP packet to be sent.
 --maxcon/-m: The maximum number of simultaneous connections to be made to the server.
 --ip/-i: The IP address of the server

The log output generated by iptables in the ULOG packet log database was viewed to verify proper functioning of the packet filtering software.

Furthermore, the ability of the **ipmenu** iptables rule manipulation software to create, delete and modify iptables chains and rules was tested by creating example rules which were then used for testing the packet filter.

Step 4: Apache Web Server Test – The functioning of the apache web server was tested by running a browser on another workstation on the same network as the bootable firewall test system, and pointing it to the HTTP port at the IP address assigned to the test system. The browser displayed the Apache Web Server test page; this demonstrated proper operation of the web server software.

Step 5: Sendmail Service Test – The sendmail service was tested by using a mail client like **pine** or **elm** to author and send a test email message to another user on the network. The receipt of the email message showed that the sendmail MTA was functioning properly.

Step 6: SOCKS proxy test – The functioning of the SOCKS proxy server was tested by configuring a browser to use proxying and providing it with the IP address of the proxy server. The observation that the browser was able to display web pages hosted on various machines on the network showed that the SOCKS proxy server was functioning normally.

6.2 TESTING PHASE II - THE INTELLIGENT ANOMALY DETECTION SYSTEM TEST PLAN

The next part of the testing process was the testing of the IADS application. The sequence of operations conducted in the testing process is listed below.

Step 1: Network traffic generation – In order to test the IADS application, various kinds of network traffic needed to be simulated, and rules created to generate alerts in response to the traffic patterns. For this purpose, the network traffic generator program, **traffic** was used to generate TCP and UDP traffic. The **ping** program was also used to generate ICMP traffic. The various fields in the packet protocol headers were manipulated to simulate common types of anomalous network activity, and rules were created to check for the same.

Step 2: Rule creation – In order to test the functioning of IADS, example rules were written, which tested the inferencing engine’s capability to detect various kinds of anomalous packet characteristics, indicative of network intrusion events.

Given below is an example rule used to test for the common “**ping of death**” attack. This attack falls under the general category of **Denial of Service** attacks. This attack signature has been obtained from the **CERT Internet advisories**. This database, hosted by the **Central Emergency Response Team**, provides a centralized collection of common attack signatures. The CERT attack ID for the **ping of death** attack is **CERT-CA-1998-01**. Here, ICMP Ping ECHO REQUEST packets are sent to a host on a network, with the destination address of the packets being spoofed to be the **broadcast address** of that network. This causes a flood of ICMP packets and results in a crash. The IADS rule for testing for such an attack is as below. The broadcast address for the test network is 192.168.0.255.

```
<rule>
  <name>ATTACK PING OF DEATH</name>
  <reference origin="other">
    <reference_name>CA-1998-01</reference_name>
    <reference_url>www.cert.org/advisories/CA-1998-01.html</reference_url>
  </reference>
  <impact type="dos" />
```

```

    <action>
      <log/>
    </action>
    <condition>
      <AND>
        <ip_protocol>1</ip_protocol>
        <ip_destination_address>192.168.0.255</ip_destination_address>
      </AND>
    </condition>
  </rule>

```

Step 3: Record of generated IDMEF alerts – A the IDMEF alert generated by IADS which identifies such an attack and specifies the spoofed broadcast source address, 192.168.0.255, is shown below.

```

<?xml version="1.0"?>
<!DOCTYPE IDMEF-Message PUBLIC "-//IETF//DTD RFC XXXX IDMEF v1.0//EN"
"./dtds/idmef-message.dtd">
<IDMEF-Message version="0.3">
  <Alert ident="1" impact="dos">
    <Analyzer analyzerid="IADS v1.0"/>
    <CreateTime ntpstamp="0xc0alac96.0x56250f84">2002-05-31T08:02:30Z
  </CreateTime>
  <Source>
    <Node>
      <Address category="ipv4-addr">
        <address>192.168.0.9</address>
      </Address>
    </Node>
  </Source>
  <Target>
    <Node>
      <Address category="ipv4-addr">
        <address>192.168.0.255</address>
      </Address>
    </Node>
  </Target>
  <Classification origin="bugtraqid">
    <name>PING OF DEATH EXAMPLE</name>
    <url>www.cert.org/ca-1998-01.html</url>
  </Classification>
</Alert>
</IDMEF-Message>

```

This test case and other attack scenarios used to the test the IADS software sufficiently demonstrated its ability process the iptables log records to detection common network intrusions and generate quantitative alerts about these attacks.

CHAPTER 7 - CONCLUSION

In conclusion of this report, the important objectives achieved in the course of this project are listed below.

- A self-contained firewall and packet filtering architecture deployable solely from a bootable CD-ROM containing the Operating System and other support software has been designed, implemented and tested successfully.
- An assorted collection of network services including the Apache web server, sendmail and DNS have also been loaded from the CD-ROM filesystem, configured and tested for proper operation.
- The ability of the firewall system to function over standard system and network hardware architectures has also been demonstrated.
- In addition, an intelligent anomaly detection system, capable of detecting network intrusions and attacks using a set of signatures and firewall audit records, has been designed, implemented and tested.

7.1 FUTURE IMPROVEMENTS

Some of the enhancements that can be considered for implementation in the course of development of the bootable firewall and anomaly detection system are listed below.

- Increased support for varied hardware architectures, like SCSI CD-ROMs etc.
- Support for the creation of more sophisticated stateful rules structures to enable better detection of network attack events.
- User-friendlier reporting of IDMEF alerts, in the form of E-mail messages etc.

CHAPTER 8 - USER MANUAL

The Bootable Firewall CD User Manual provides a documentation of the functioning of the system and its interface, according to the perspective of a typical user of the system. In this respect, this section describes in detail the process of setting up a firewall gateway with intrusion detection capabilities, using the bootable firewall CD-ROM.

8.1 CONFIGURING THE SYSTEM BIOS

In order to enable the system to boot from the firewall CD, the BIOS settings need to be altered to set the IDE CD-ROM as the first boot device. The steps to be performed for setting this option vary according to the BIOS architecture. More information about this can be found the configuration manual accompanying the system.

8.2 BOOTING FROM THE CD-ROM

Once the BIOS finishes initialization, the LILO boot loader on the CD will be loaded, and will load and execute the kernel. After loading the Ramdisk, the system will prompt the user for the administrator password. Three retries are provided in case the user mistakenly enters a wrong password. If authentication fails, the system will halt and should be restarted using the Ctrl-Alt-Del combination. On successful authentication, bootup proceeds normally, and system services are initialized.

8.3 THE SYSCONF CONFIGURATION PROGRAM

Once the system has completed booting, the sysconf program is loaded. This program provides the user with an integrated collection of utilities that can be used to configure and manage all the system services.

Sysconf first prompts the user to detect and configure the network interface cards attached to the system. If the user opts to configure the interfaces, the NICs drivers are installed and IP addresses are assigned. On other hand, if the user chooses not to configure the interfaces, the **sysconf** exits to the command shell, since no other services can be configured without setting up the network. The sysconf program can then be run at a later time using the command:

```
# sysconf --hwdetect
```

8.4 CONFIGURATION OF SYSTEM SERVICES

After network discovery, **sysconf** presents the user with a list of services that can be configured to run on the system, including DNS, Apache, Sendmail, the Intelligent Anomaly Detection System, network configuration, and miscellaneous system settings. Depending on the options selected from the list, the corresponding service-specific configuration programs are loaded and executed. These details of these configuration utilities are given below.

8.4.1 THE DNSCONF DNS SERVICE CONFIGURATION PROGRAM

The **dnsconf** program allows the user to manage DNS server and client settings on the system. It provides the user with the following options.

- Edit the DNS server configuration stored in **/etc/named.conf**
- Edit the DNS client configuration stored in **/etc/resolv.conf**
- Start the DNS server
- Stop the DNS server

8.4.2 THE IPTCONF IPTABLES CONFIGURATION PROGRAM

The **iptconf** program allows the user to manage the **iptables** firewall installed on the system. It provides the user with the following options.

- Start the iptables service.
- Stop the iptables service.

Manage iptables rules using the **ipmenu** interface program. Ipmenu is console-based program that provides a powerful menu-based interface for creating, deleting and modifying iptables firewall rules and security policies.

8.4.3 THE APACHECONF WEB SERVER CONFIGURATION PROGRAM

The **apacheconf** program an interface to the apache web server software. It provides the following options:

- Edit the apache configuration from **`/var/httpd/conf/httpd.conf`**
- Start the Apache web server
- View the server access log
- Stop the Apache web server

8.4.4 THE SMCONF SENDMAIL CONFIGURATION PROGRAM

smconf provides the following options for configuring sendmail on the system.

- Start the sendmail service
- Edit the sendmail configuration from **`/etc/sendmail.cf`**
- Stop the sendmail service.

8.4.5 THE IADSCONF IADS CONFIGURATION PROGRAM

The **iadsconf** program provides an all-inclusive interface to the Intelligent Anomaly Detection System and allows the administrator to monitor its operation without having to use any shell commands. The menu options provided in IADS are given below.

- Start the IADS daemon, with the ULOGD and MySQL servers.
- Create a new IADS rule – When selected, this option opens another window which provides the user with a dialog based interface for creating an IADS rule. All the parameters to be provided for creating a rule can be entered through text fields and list boxes. At any time during the creation of a rule, the user can activate a help screen, which lists all the valid rule data elements, and the rule syntax with examples. After entering all the relevant data, the user can choose to save the rule, and **iadsconf** will append the new rule to the IADS rule file.
- Edit the IADS rule file directly from **/etc/iads/iads_rules.xml**
- View the IADS generated alerts from **/etc/iads/iads_alert.xml**
- View the IADS error log from **/var/log/iads_error.log**
- Stop the IADS server.

8.4.6 NETWORK CONFIGURATION

If the user selects to view and modify the configured network interfaces, the **netconf** program is activated, which provides a menu-based interface for doing so.

8.4.7 CONFIGURATION OF MISCELLANEOUS SERVICES

The user can also choose to configure the following settings on the system.

- Keyboard configuration using **kbdconfig**.
- Time configuration using **timeconfig**.

After completing the entire configuration process, the **sysconf** program will exit and the **bash** shell will be activated in 4 terminals from tty1 to tty4. These shells can be activated using the **Ctrl-Alt-[F1 ~ F4]** key combinations. Furthermore, when at the shell, the user can activate the following programs directly:

- The **sysconf** configuration program using the **Ctrl-Alt-F6** key combination.
- The **IPTRAF** networking monitoring utility.

BIBLIOGRAPHY

Some of the important literature, software and Internet resources used in the course of this project have been listed below.

- [1] Linux System Documentation
 - i. Firewall and NAT HOWTOs
 - ii. Kernel and Ramdisk HOWTOs
 - iii. Bootdisk HOWTO
 - iv. CD-Writing HOWTO
 - v. Linux System Manual Pages
- [2] Computer Networks – Andrew S. Tanenbaum
- [3] Internet Firewalls - D. Brent Chapman & Elizabeth D. Zwicky
- [4] **Netfilter** and **iptables** resources and documentation: www.netfilter.samba.org
- [5] The Linux XML library, **libxml**: www.xmlsoft.org
- [6] The Linux IDMEF library, **libidmef**: www.silicondefense.com/idwg/libidmef
- [7] Documentation on the IDMEF Standard: www.ietf.org
- [8] The CERT advisories on network intrusion signatures: www.cert.org
- [9] The IPMENU iptables rules editor: <http://users.pandora.be/stes>
- [10] The IPTRAF network monitoring utility: <http://cebu.mozcom.com/riker/iptraf>

APPENDICES

APPENDIX A - THE CRYPT() ENCRYPTION LIBRARY

The encryption used in the authentication program **fwauth** uses the **crypt()** function call provided by the UNIX system libraries. It is detailed below.

Syntax:

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

Description:

crypt is the password encryption function. It is based on the Data Encryption Standard algorithm with variations intended to discourage use of hardware implementations of a key search.

key is a user's typed password.

salt is a two-character string chosen from the set [aenzAenZ0en9./]. This string is used to perturb the algorithm in one of 4096 different ways.

By taking the lowest 7 bit of each character of the key, a 56-bit key is obtained. This 56-bit key is used to encrypt repeatedly a constant string, which usually consists of all zeros. The returned value points to the encrypted password, a series of 13 printable ASCII characters. The first two characters represent the salt itself. The return value points to static data whose content is overwritten by each call.

Return Value:

A pointer to the encrypted password is returned. On error, NULL is returned.

APPENDIX B - THE MYSQL C API LIBRARY

The MySQL Datatypes

MYSQL

This structure represents a handle to one database connection. It is used for almost all MySQL functions.

MYSQL_RES

This structure represents the result of a query that returns rows. The information returned from a query is called the **result set**.

MYSQL_ROW

This is a type-safe representation of one row of data. It is implemented as an array of counted byte strings.

MYSQL_FIELD

This structure contains information about a field, such as the field's name, type, and size.

MYSQL_FIELD_OFFSET

This is a type-safe representation of an offset into a MySQL field list. Offsets are field numbers within a row, beginning at zero.

The MySQL C API Library

mysql_affected_rows	Returns the number of rows affected by the last database query.
mysql_close	Closes a server connection.
mysql_connect	Connects to a MySQL server.
mysql_change_user	Changes user and database on an open connection.
mysql_create_db	Creates a database.
mysql_data_seek	Seeks to an arbitrary row in a query result set.
mysql_drop_db	Drops a database.
mysql_eof	Determines whether or not the last row of a result set has been read.
mysql_errno	Returns the error number for the most recently invoked MySQL function.
mysql_error	Returns the error message for the most recently invoked MySQL function.
mysql_fetch_field	Returns the type of the next table field.
mysql_fetch_field_direct	Returns the type of a table field, given a field number.
mysql_fetch_fields	Returns an array of all field structures.
mysql_fetch_lengths	Returns the lengths of all columns in the current row.
mysql_fetch_row	Fetches the next row from the result set.
mysql_field_seek	Puts the column cursor on a specified column.
mysql_field_count	Returns the number of result columns for the most recent query.

mysql_field_tell	Returns the position of the field cursor used for the last fetch.
mysql_free_result	Frees memory used by a result set.
mysql_init	Gets or initializes a `MYSQL' structure.
mysql_num_fields	Returns the number of columns in a result set.
mysql_num_rows	Returns the number of rows in a result set.
mysql_ping	Checks whether or not the connection to the server is working, reconnecting as necessary.
mysql_query	Executes a SQL query specified as a null-terminated string.
mysql_select_db	Selects a database.
mysql_shutdown	Shuts down the database server.
mysql_stat	Returns the server status as a string.
mysql_store_result	Retrieves a complete result set to the client.

APPENDIX C - THE NEWT WINDOWING SYSTEM

Introduction

The newt windowing system is a terminal-based window and widget library designed for writing applications with a simple, but user-friendly, interface. While newt is not intended to provide the rich feature set advanced applications may require, it has proven to be flexible enough for a wide range of applications.

Background

Newt was originally designed for use in the install code for Red Hat Linux. As this install code runs in an environment with limited resources (most importantly limited filesystem space), newt's size was immediately an issue. To help minimize its size, the following design decisions were made early in its implementation:

- newt does not use an event-driven architecture.
- newt is written in C, not C++.
- Windows must be created and destroyed as a stack (in other words, all newt windows behave as modal dialogs). This is probably the greatest functionality restriction of newt
- The tty keyboard is the only supported input device.
- Much behaviour, such as widget traversal order, is difficult or impossible to change.
- While newt provides a complete API, it does not handle the low-level screen drawing itself. Instead, newt is layered on top of the screen management capabilities of S-Lang library.

Components

Displayable items in newt are known as components, which are analogous to the widgets provided by most Unix widget sets. There are two main types of components in newt, forms and everything else. Forms logically group components into functional sets. When an application is ready to get input from a user, it "runs a form", which makes the form active and lets the user enter information into the components the form contains. A form may contain any other component, including other forms. Using subforms in this manner lets the application change the details of how the user tabs between components on the form, scroll regions of the screen, and control background colors for portions of windows.

Every component is of type `newtComponent`, which is an opaque type. It's guaranteed to be a pointer though, which lets applications move it through void pointers if the need arises. Variables of type `newtComponent` should never be directly manipulated -- they should only be passed to newt functions.

Basic Newt Functions

While most newt functions are concerned with widgets or groups of widgets (called grids and forms), some parts of the newt API deal with more global issues, such as initializing newt or writing to the root window.

Starting and Ending newt Services

There are three functions which nearly every newt application use. The first two are used to initialize the system.

```
int newtInit(void);
```

```
void newtCls(void);
```

`newtInit()` should be the first function called by every newt program. It initializes internal data structures and places the terminal in raw mode. Most applications invoke `newtCls()`

immediately after `newtInit()`, which causes the screen to be cleared. When a newt program is ready to exit, it should call `newtFinished()`.

```
int newtFinished(void);
```

`newtFinished()` restores the terminal to its appearance when `newtInit()` was called and places the terminal in its original input state. If this function isn't called, the terminal will probably need to be reset with the `reset` command before it can be used easily.

Drawing on the Root Window

The background of the terminal's display (the part without any windows covering it) is known as the root window (it's the parent of all windows, just like the system's root directory is the parent of all subdirectories). Normally, applications don't use the root window, instead drawing all of their text inside of windows newt doesn't require this though -- widgets may be placed directly on the root window without difficulty). It is often desirable to display some text, such as a program's name or copyright information, on the root window, however.

Newt provides two ways of displaying text on the root window. These functions may be called at any time. They are the only newt functions that are meant to write outside of the current window.

```
void newtDrawRootText(int left, int top, const char * text);
```

This function is straightforward. It displays the string text at the position indicated. If either the left or top is negative, the position is measured from the opposite side of the screen. The final measurement will seem to be off by one though.

As it's common to use the last line on the screen to display help information, Newt includes special support for doing exactly that. The last line on the display is known as the *help line*, and is treated as a stack. As the value of the help line normally relates to the window currently displayed, using the same structure for window order and the help line is very natural. Two functions are provided to manipulate the help line.

```
void newtPushHelpLine(const char * text);
```

```
void newtPopHelpLine(void);
```

The first function, `newtPushHelpLine()`, saves the current help line on a stack (which is independent of the window stack) and displays the new line. If text is `NULL`, `newt`'s default help line is displayed (which provides basic instructions on using `newt`). If text is a string of length 0, the help line is cleared. For all other values of text, the passed string is displayed at the bottom, left-hand corner of the display. The space between the ends of the displayed string the right-hand edge of the terminal is cleared. `newtPopHelpLine()` replaces the current help line with the one it replaced.

Suspending Newt Applications

If the application should suspend and continue like most user applications, it needs two `newt` functions.

```
void newtSuspend(void);
```

```
void newtResume(void);
```

`newtSuspend()` tells `newt` to return the terminal to its initial state. Once this is done, the application can suspend itself (by sending itself a `SIGTSTP`, fork a child program, or do whatever else it likes. When it wants to resume using the `newt` interface, it must call `newtResume()` before doing so.

Windows

While most `newt` applications do use windows, `newt`'s window support is actually extremely limited. Windows must be destroyed in the opposite order they were created, and only the topmost window may be active. Corollaries to this are:

- The user may not switch between windows.

- Only the top window may be destroyed.

While this is quite a severe limitation, adopting it greatly simplifies both writing newt applications and developing newt itself, as it separates newt from the world of event-driven programming.

Creating Windows

```
int newtCenteredWindow(int width, int height, const char * title);
```

```
int newtOpenWindow(int left, int top, int width, int height, const char * title);
```

The first of these functions open a centered window of the specified size. The title is optional -- if it is NULL, then no title is used. `newtOpenWindow()` is similar, but it requires a specific location for the upper left-hand corner of the window.

Destroying Windows

All windows are destroyed in the same manner, no matter how the windows were originally created.

```
void newtPopWindow(void);
```

This function removes the top window from the display, and redraws the display areas that the window overwrote.

Components

Components are the basic user interface element newt provides. A single component may be (for example) a listbox, push button checkbox, a collection of other components. Most components are used to display information in a window, provide a place for the user to enter data, or a combination of these two functions.

Forms, however, are a component whose primary purpose is not noticed by the user at all. Forms are collections of components (a form may contain another form) that logically relate the components to one another. Once a form is created and had all of its constituent components added to it, applications normally then run the form. This gives control of the application to the form, which then lets the user enter data onto the form. When the user is done (a number of different events qualify as "done"), the form returns control to the part of the application that invoked it. The application may then read the information the user provided and continue appropriately.

All newt components are stored in a common data type, a newtComponent. We start off with a brief introduction to forms.

Introduction to Forms

Forms are simply collections of components. As only one form can be active (or running) at a time, every component that the user should be able to access must be on the running form (or on a subform of the running form). A form is itself a component, which means forms are stored in newtComponent data structures.

```
newtComponent newtForm(newtComponent vertBar, const char * help, int flags);
```

To create a form, call newtForm(). The first parameter is a vertical scrollbar that should be associated with the form. For now, that should always be NULL. The second parameter, help, is currently unused and should always be NULL. The flags are normally 0.

```
newtComponent myForm;
```

```
myForm = newtForm(NULL, NULL, 0);
```

After a form is created, components need to be added to it --- after all, an empty form isn't terribly useful. There are two functions that add components to a form.

```
void newtFormAddComponent(newtComponent form, newtComponent co);
```

```
void newtFormAddComponents(newtComponent form, ...);
```

The first function, `newtFormAddComponent()`, adds a single component to the form which is passed as the first parameter. The second function is simply a convenience function. After passing the form to `newtFormAddComponents()`, an arbitrary number of components is then passed, followed by `NULL`. Every component passed is added to the form. Once a form has been created and components have been added to it, it's time to run the form.

```
newtComponent newtRunForm(newtComponent form);
```

This function runs the form passed to it, and returns the component that caused the form to stop running. When an application is done with a form, it destroys the form and all of the components the form contains.

```
void newtFormDestroy(newtComponent form);
```

This function frees the memory resources used by the form and all of the components that have been added to the form (including those components which are on subforms). Once a form has been destroyed, none of the form's components can be used.

Components

Non-form components are the most important user-interface component for users. They determine how users interact with `newt` and how information is presented to them.

Buttons

Nearly all forms contain at least one button. `Newt` buttons come in two flavors, full buttons and compact buttons. Full buttons take up quite a bit of screen space, but look much better than the single-row compact buttons. Other than their size, both button styles behave identically. Different functions are used to create the two types of buttons.

```
newtComponent newtButton(int left, int top, const char * text);
```

```
newtComponent newtCompactButton(int left, int top, const char * text);
```

Both functions take identical parameters. The first two parameters are the location of the upper left corner of the button, and the final parameter is the text that should be displayed in the button (such as ``Ok" or ``Cancel").

Labels

Labels are newt's simplest component. They display some given text and don't allow any user input.

```
newtComponent newtLabel(int left, int top, const char * text);
```

```
void newtLabelSetText(newtComponent co, const char * text);
```

Creating a label is just like creating a button; just pass the location of the label and the text it should display. Unlike buttons, labels do let the application change the text in the label with `newtLabelSetText`. When the label's text is changed, the label automatically redraws itself. It does not clear out any old text which may be leftover from the previous time it was displayed, however, so be sure that the new text is at least as long as the old text.

Entry Boxes

Entry boxes allow the user to enter a text string into the form that the application can later retrieve.

```
typedef int (*newtEntryFilter)(newtComponent entry, void * data, int ch, int cursor);
```

```
newtComponent newtEntry(int left, int top, const char * initialValue, int width, char **
resultPtr, int flags);
```

```
void newtEntrySet(newtComponent co, const char * value, int cursorAtEnd);
```

```
char * newtEntryGetValue(newtComponent co);
```

```
void newtEntrySetFilter(newtComponent co, newtEntryFilter filter, void * data);
```

`newtEntry()` creates a new entry box. After the location of the entry box, the initial value for the entry box is passed, which may be NULL if the box should start off empty. Next, the width of the physical box is given. This width may or may not limit the length of the string the user is allowed to enter; that depends on the flags. The `resultPtr` must be the address of a `char *`. Until the entry box is destroyed by `newtFormDestroy()`, that `char *` will point to the current value of the entry box. It's important that applications make a copy of that value before destroying the form if they need to use it later. The `resultPtr` may be NULL, in which case the user must use the

`newtEntryGetValue()` function to get the value of the entry box.

Entry boxes support a number of flags:

`NEWT_ENTRY_SCROLL`

If this flag is not specified, the user cannot enter text into the entry box which is wider than the entry box itself. This flag removes this limitation, and lets the user enter data of an arbitrary length.

`NEWT_FLAG_HIDDEN`

If this flag is specified, the value of the entry box is not displayed. This is useful when the application needs to read a password, for example.

`NEWT_FLAG_RETURNEXIT`

When this flag is given, the entry box will cause the form to stop running if the user pressed return inside of the entry box. This can provide a nice shortcut for users.

Checkboxes

Most widget sets include checkboxes that toggle between two values (checked or not checked). Newt checkboxes are more flexible. When the user presses the space bar on a checkbox, the checkbox's value changes to the next value in an arbitrary sequence (which wraps). Most checkboxes have two items in that sequence, checked or not, but newt allows an arbitrary number of value. This is useful when the user must pick from a limited number of choices.

Each item in the sequence is a single character, and the sequence itself is represented as a string. The checkbox components display the character that currently represents its value the left of a text label, and returns the same character as its current value. The default sequence for checkboxes is " *", with ' ' indicating false and '*' true.

```
newtComponent newtCheckbox(int left, int top, const char * text, char defValue, const char
* seq, char * result);
```

```
char newtCheckboxGetValue(newtComponent co);
```

Like most components, the position of the checkbox is the first thing passed to the function that creates one. The next parameter, text, is the text that is displayed to the right of the area that is checked. The defValue is the initial value for the checkbox, and seq is the sequence which the checkbox should go through defValue must be in seq. seq may be NULL, in which case " *" is used. The final parameter, result, should point to a character that the checkbox should always record its current value in. If result is NULL, newtCheckboxGetValue() must be used to get the current value of the checkbox.

Textboxes

Textboxes display a block of text on the terminal, and is appropriate for display large amounts of text.

```
newtComponent newtTextbox(int left, int top, int width, int height, int flags);
```

```
void newtTextboxSetText(newtComponent co, const char * text);
```

`newtTextbox()` creates a new textbox, but does not fill it with data. The function is passed the location for the textbox on the screen, the width and height of the textbox (in characters), and zero or more of the following flags:

`NEWT_FLAG_WRAP`

All text in the textbox should be wrapped to fit the width of the textbox. If this flag is not specified, each newline-delimited line in the text is truncated if it is too long to fit. When `newt` wraps text, it tries not to break lines on spaces or tabs. Literal newline characters are respected, and may be used to force line breaks.

`NEWT_FLAG_SCROLL`

The text box should be scrollable. When this option is used, the scrollbar which is added increases the width of the area used by the textbox by 2 characters; that is the textbox is 2 characters wider than the width passed to `newtTextbox()`.

After a textbox has been created, text may be added to it through `newtTextboxSetText()`, which takes only the textbox and the new text as parameters. If the textbox already contained text, that text is replaced by the new text.

Listboxes

Listboxes are the most complicated components `newt` provides. They can allow a single selection or multiple selections, and are easy to update. Each entry in a listbox is an ordered pair of the text that should be displayed for that item and a key, which is a `void *` that uniquely identifies that listbox item. Many applications pass integers in as keys, but using arbitrary pointers makes many applications significantly easier to code.

```
newtComponent newtListbox(int left, int top, int height, int flags);
```

```
int newtListboxAppendEntry(newtComponent co, const char * text, const void * data);
```

```
void * newtListBoxGetCurrent(newtComponent co);
```

```
void newtListBoxSetWidth(newtComponent co, int width);
```

```
void newtListBoxSetCurrent(newtComponent co, int num);
```

```
void newtListBoxSetCurrentByKey(newtComponent co, void * key);
```

A listbox is created at a certain position and a given height. The Height is used for two things. First of all, it is the minimum height the listbox will use. If there are fewer items in the listbox than the height, the listbox will still take up that minimum amount of space. Secondly, if the listbox is set to be scrollable (by setting the `NEWT_FLAG_SCROLL` flag, the height is also the maximum height of the listbox. If the listbox may not scroll, it increases its height to display all of its items. Once a listbox has been created, items are added to it by invoking `newtListBoxAppendEntry()`, which adds new items to the end of the list. In addition to the listbox component, `newtListBoxAppendEntry()` needs both elements of the (text, key) ordered pair. For lists which only allow a single selection, `newtListBoxGetCurrent()` should be used to find out which listbox item is currently selected. It returns the key of the currently selected item.

APPENDIX D - THE IADS XML RULE DTD

```

<?xml version="1.0" encoding="UTF-8"?>

<!--*****
      IADS Rule File Format XML DTD version 1.0
*****-->

<!--ELEMENTS-->

<!ELEMENT name (#PCDATA)>

<!ELEMENT ip_version (#PCDATA)>
<!ELEMENT ip_header_length (#PCDATA)>
<!ELEMENT ip_tos (#PCDATA)>
<!ELEMENT ip_total_length (#PCDATA)>
<!ELEMENT ip_identification (#PCDATA)>
<!ELEMENT ip_reserved (#PCDATA)>
<!ELEMENT ip_df EMPTY>
<!ELEMENT ip_mf EMPTY>
<!ELEMENT ip_offset (#PCDATA)>
<!ELEMENT ip_ttl (#PCDATA)>
<!ELEMENT ip_protocol (#PCDATA)>
<!ELEMENT ip_checksum (#PCDATA)>
<!ELEMENT ip_source_address (#PCDATA)>
<!ELEMENT ip_destination_address (#PCDATA)>
<!ELEMENT ip_address (#PCDATA)>

<!ELEMENT tcp_source_port (#PCDATA)>
<!ELEMENT tcp_destination_port (#PCDATA)>
<!ELEMENT tcp_port (#PCDATA)>
<!ELEMENT tcp_sequence_number (#PCDATA)>
<!ELEMENT tcp_acknowledge_number (#PCDATA)>
<!ELEMENT tcp_header_length (#PCDATA)>
<!ELEMENT tcp_reserved (#PCDATA)>
<!ELEMENT tcp_urg EMPTY>
<!ELEMENT tcp_ack EMPTY>
<!ELEMENT tcp_psh EMPTY>
<!ELEMENT tcp_rst EMPTY>
<!ELEMENT tcp_syn EMPTY>
<!ELEMENT tcp_fin EMPTY>
<!ELEMENT tcp_window_size (#PCDATA)>
<!ELEMENT tcp_checksum (#PCDATA)>
<!ELEMENT tcp_urgent_pointer (#PCDATA)>

<!ELEMENT udp_source_port (#PCDATA)>
<!ELEMENT udp_destination_port (#PCDATA)>
<!ELEMENT udp_port (#PCDATA)>
<!ELEMENT udp_length (#PCDATA)>
<!ELEMENT udp_checksum (#PCDATA)>

<!ELEMENT icmp_type (#PCDATA)>
<!ELEMENT icmp_code (#PCDATA)>

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

<!ELEMENT icmp_checksum (#PCDATA)>

<!ELEMENT hex (#PCDATA)>
<!ELEMENT pattern (#PCDATA | hex)*>
<!ELEMENT start_offset (#PCDATA)>
<!ELEMENT stop_offset (#PCDATA)>
<!ELEMENT data_size (#PCDATA)>

<!ELEMENT packet_data (pattern, start_offset?, stop_offset?)>
<!ELEMENT stream_data (pattern)>

<!ELEMENT AND (
ip_total_length?,          ip_version?, ip_header_length?, ip_tos?,
                             ip_identification?, ip_reserved?, ip_df?, ip_mf?,
ip_offset?,                ip_ttl?, ip_protocol?, ip_checksum?,
ip_source_address?,        ip_destination_address?, ip_address?,
                             (
                               (
tcp_port?,                  tcp_source_port?, tcp_destination_port?,
                             tcp_sequence_number?, tcp_acknowledge_number?,
                             tcp_header_length?, tcp_reserved?,
                             tcp_urg?, tcp_ack?, tcp_psh?, tcp_rst?,
tcp_syn?, tcp_fin?,        tcp_window_size?, tcp_checksum?,
tcp_urgent_pointer?
                               ) | (
udp_port?,                  udp_source_port?, udp_destination_port?,
                             udp_length?, udp_checksum?
                               ) | (
icmp_type?, icmp_code?, icmp_checksum?
                               )
                             )?,
packet_data*,
stream_data*,
data_size?,
AND*, OR*, XOR*, NOT*
)>

<!ELEMENT OR (
ip_total_length*,          ip_version*, ip_header_length*, ip_tos*,
                             ip_identification*, ip_reserved*, ip_df*, ip_mf*,
ip_offset*,                ip_ttl*, ip_protocol*, ip_checksum*,
ip_source_address*,        ip_destination_address*, ip_address*,
                             tcp_source_port*,
                             tcp_destination_port*, tcp_port*,
                             tcp_sequence_number*,
                             tcp_acknowledge_number*, tcp_header_length*,
                             tcp_reserved*,

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

tcp_urgent_pointer*, tcp_urg*, tcp_ack*, tcp_psh*, tcp_rst*, tcp_syn*,
tcp_fin*, tcp_window_size*, tcp_checksum*,
udp_source_port*, udp_destination_port*, udp_port*,
udp_length*, udp_checksum*, icmp_type*, icmp_code*,
icmp_checksum*, packet_data*,
stream_data*,
data_size*,
AND*, OR*, XOR*, NOT*
)>

```

```

<!ELEMENT XOR (
ip_total_length*, ip_version*, ip_header_length*, ip_tos*,
ip_offset*, ip_identification*, ip_reserved*, ip_df*, ip_mf*,
ip_source_address*, ip_ttl*, ip_protocol*, ip_checksum*,
tcp_source_port*, ip_destination_address*, ip_address*,
tcp_destination_port*, tcp_port*,
tcp_sequence_number*, tcp_acknowledge_number*, tcp_header_length*,
tcp_reserved*, tcp_urg*, tcp_ack*, tcp_psh*, tcp_rst*, tcp_syn*,
tcp_fin*, tcp_window_size*, tcp_checksum*,
tcp_urgent_pointer*, udp_source_port*, udp_destination_port*, udp_port*,
udp_length*, udp_checksum*, icmp_type*, icmp_code*,
icmp_checksum*, packet_data*,
stream_data*,
data_size*,
AND*, OR*, XOR*, NOT*
)>

```

```

<!ELEMENT NOT (
ip_total_length?, ( ip_version?, ip_header_length?, ip_tos?,
ip_offset?, ip_identification?, ip_reserved?, ip_df?, ip_mf?,
ip_source_address?, ip_ttl?, ip_protocol?, ip_checksum?,
ip_destination_address?, ip_address?,
(
(
tcp_source_port?, tcp_destination_port?,
tcp_port?,
tcp_sequence_number?, tcp_acknowledge_number?,
tcp_header_length?, tcp_reserved?,
tcp_urg?, tcp_ack?, tcp_psh?, tcp_rst?,
tcp_syn?, tcp_fin?,

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

        tcp_window_size?, tcp_checksum?,
tcp_urgent_pointer?
    ) | (
        udp_source_port?, udp_destination_port?,
udp_port?,
        udp_length?, udp_checksum?
    ) | (
        icmp_type?, icmp_code?, icmp_checksum?
    )
    )?,
    packet_data?,
    stream_data?,
    data_size?
) | AND | OR | XOR | NOT
)>

<!ELEMENT condition ( ( ip_version?, ip_header_length?, ip_tos?,
ip_total_length?,
        ip_identification?, ip_reserved?, ip_df?, ip_mf?,
ip_offset?,
        ip_ttl?, ip_protocol?, ip_checksum?,
ip_source_address?,
        ip_destination_address?, ip_address?,
(
    (
        tcp_source_port?, tcp_destination_port?,
tcp_port?,
        tcp_sequence_number?, tcp_acknowledge_number?,
        tcp_header_length?, tcp_reserved?,
        tcp_urg?, tcp_ack?, tcp_psh?, tcp_rst?,
tcp_syn?, tcp_fin?,
        tcp_window_size?, tcp_checksum?,
tcp_urgent_pointer?
    ) | (
        udp_source_port?, udp_destination_port?,
udp_port?,
        udp_length?, udp_checksum?
    ) | (
        icmp_type?, icmp_code?, icmp_checksum?
    )
    )?,
    packet_data?,
    data_size?
) | AND | OR | XOR | NOT
)>

<!ELEMENT stream ( ( ip_address?, (tcp_port|udp_port)?) | AND | OR | XOR |
NOT )>

<!ELEMENT stream_condition ( stream_data? | AND | OR | XOR | NOT )>

<!ELEMENT impact EMPTY>

<!ELEMENT log EMPTY>

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```
<!ELEMENT drop EMPTY>
<!ELEMENT activate_rule (#PCDATA)>
<!ELEMENT deactivate_rule (#PCDATA)>
<!ELEMENT action ( log | drop |
                  (log, (activate_rule, deactivate_rule)* ) |
                  (activate_rule+, deactivate_rule*) |
                  (activate_rule*, deactivate_rule+)
                )>

<!ELEMENT reference_name (#PCDATA)>
<!ELEMENT reference_url (#PCDATA)>
<!ELEMENT reference (reference_name, reference_url?)>

<!ELEMENT rule (name?, reference*, impact?, action, (condition| (stream,
stream_condition)) ) >

<!ELEMENT rule_base (rule+)>

<!--Attributes-->

<!ATTLIST rule state (active | inactive) "active"
               type (stateless | stream) "stateless"
               >

<!ATTLIST reference origin (unknown | bugtraqid | cve | vendor-specific)
"unknown">

<!ATTLIST impact type (admin | dos | file | recon | user | other)
"other"
               completion (attempted | successful | failed) "attempted"
               severity (low | med | high) "high"
               >

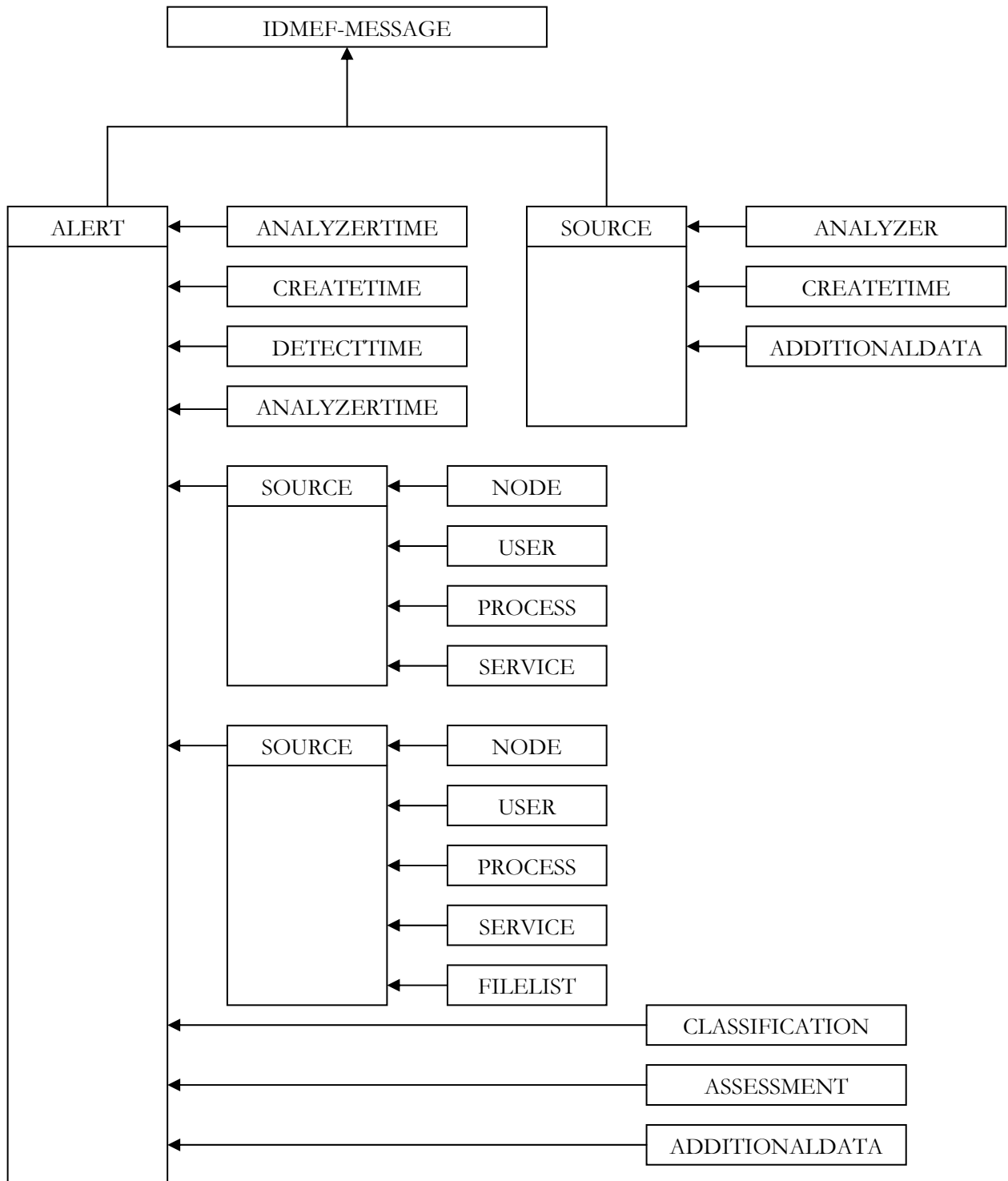
<!ATTLIST pattern encoding (hex | utf-8) "utf-8">

<!ATTLIST packet_data case (no | yes) "yes">

<!ATTLIST stream_data case (no | yes) "yes">

<!ATTLIST log capture (none | header | full) "none">
```

APPENDIX E - THE IDMEF DATA MODEL



APPENDIX F - SOURCE CODE**BOOTABLE FIREWALL SYSTEM SOURCES****fwpcreate.c**

```
#define _XOPEN_SOURCE
#define PWD_MAX 80

#include <stdio.h>
#include <unistd.h>

char pwd[PWD_MAX];
char cpwd[PWD_MAX];
char *epwd;
FILE *pfd;

int main() {

    printf("Enter password:");
    system("stty -echo");
    fgets(pwd, PWD_MAX, stdin);
    system("stty echo");

    printf("\nConfirm password:");
    system("stty -echo");
    fgets(cpwd, PWD_MAX, stdin);
    system("stty echo");

    if (strcmp(pwd, cpwd)) {
        printf ("Passwords do not match.");
        return 1;
    }

    epwd = crypt(pwd, "fw");

    pfd = fopen("/etc/pwf", "w");
    fprintf(pfd, "%s", epwd);
    fflush(pfd);
    fclose(pfd);

    printf("\nUsername and password created successfully.\n");
    return 0;
}
```

fwauth.c

```
#define _XOPEN_SOURCE
#define PWD_MAX 80

#include <stdio.h>
#include <unistd.h>
#include <signal.h>

char pwd[PWD_MAX];
char spwd[PWD_MAX];
char *epwd;
FILE *pfd;

int main() {

    sigset_t sigmask;
    int i;

    sigfillset(&sigmask);
    sigprocmask(SIG_BLOCK, &sigmask, NULL);

    printf("\n\tWelcome to the Bootable firewall authentication module.\n");
    printf("\tEnter the administrator password to continue.\n");

    for (i = 0; i < 3; i++) {
        printf("\n\tEnter password:");

        system("stty -echo");
        fgets(pwd, 80, stdin);
        system("stty echo");

        epwd = crypt(pwd, "fw");

        pfd = fopen("/etc/pwf", "r");
        fscanf(pfd, "%s", spwd);

        if (strcmp(epwd, spwd)) {
            printf("\n\tPassword authentication failure.\n");
        }
        else {
            printf("\n\tAuthentication successful.\n");
            printf("Continuing with boot-up sequence...\n\n");
            return 0;
        }
    }
    printf("System halted... Press Ctrl-Alt-Del to Reboot.\n\n");
    system("/sbin/init 0");
    return 0;
}
```

cdmount.c

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

int main(void)
{
    int data = 0;
    char root_dev[20], fs_type[20];

    printf("Trying /dev/hdb...\n");
    if (mount("/dev/hdb", "/mnt/cdrom", "iso9660", MS_RDONLY|MS_MGC_VAL,
&data) == 0) {
        printf("CD-ROM mounted successfully.\n");
        return 0;
    }

    printf("Trying /dev/hdc...\n");
    if (mount("/dev/hdc", "/mnt/cdrom", "iso9660", MS_RDONLY|MS_MGC_VAL,
&data) == 0) {
        printf("CD-ROM mounted successfully.\n");
        return 0;
    }

    printf("Trying /dev/hdd...\n");
    if (mount("/dev/hdd", "/mnt/cdrom", "iso9660", MS_RDONLY|MS_MGC_VAL,
&data) == 0) {
        printf("CD-ROM mounted successfully.\n");
        return 0;
    }

    else {
        printf("Unable to mount CD-ROM... Switching to failsafe mode.\nSpecify
root device: ");
        scanf("%s", root_dev);
        printf("Specify filesystem type: ");
        scanf("%s", fs_type);
        if (mount(root_dev, "/mnt/cdrom", fs_type, MS_RDONLY|MS_MGC_VAL, &data)
== 0)
            return 0;
    }

    return -1;
}
```

linuxrc script

```
#!/bin/nash

/bin/insmod /lib/cdrom.o
/bin/insmod /lib/ide-cd.o
echo 'Mounting Firewall CD-ROM...'
/cdmount
mount -t proc none /proc
echo 0x100 > /proc/sys/kernel/real-root-dev
umount /proc
/bin/cp -dpR /mnt/cdrom/etc /
/bin/cp -dpR /mnt/cdrom/var /
/bin/chmod -R u+w /etc
/bin/chmod -R u+w /var
/bin/fwauth
```

sysconf.c

```
#include<newt.h>
#include<stdlib.h>
#include<stdio.h>

int main(int argc, char **argv) {

    newtComponent form, button1, button2, check_ret, text, checkbox1,
checkbox2,checkbox3,checkbox4,checkbox5, checkbox6, checkbox7, checkbox8;
    char msg[200] = "This tool will help you configure and run the
firewall system.";
    char iptables,httpd,sendmail,dns,iads, network, kbd, time;
    int kudzu;

    kudzu = 0;

    if (argc > 1) {
        if (!strcmp(argv[1], "--hwdetect"))
            kudzu = 1;
    }

    system("clear");

    newtInit();
    newtCls();

    newtPushHelpLine(NULL);
    newtRefresh();

    newtDrawRootText(21,1,"### The Intelligent Firewall System ###");

    //start kudzu
    if (kudzu) {
```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

text = newtTextboxReflowed(2, 2, mesg, 38, 2, 2, 0);
button1 = newtButton(17, newtTextboxGetNumLines(text) + 4, "Ok");

newtCenteredWindow(40, newtTextboxGetNumLines(text) + 10, "Welcome");

form = newtForm(NULL, NULL, 0);
newtFormAddComponents(form, text, button1, NULL);

newtRunForm(form);

newtPopWindow();
newtFormDestroy(form);

text = newtTextboxReflowed(2, 2, "Do you want to detect and configure
network interfaces ?", 38, 2, 2, 0);
newtCenteredWindow(40, newtTextboxGetNumLines(text) + 8, "NIC
Configuration");
button1 = newtButton(7, newtTextboxGetNumLines(text) + 3, "Ok");
button2 = newtButton(25, newtTextboxGetNumLines(text) + 3, "Cancel");

form = newtForm(NULL, NULL, 0);
newtFormAddComponents(form, text, button1, button2, NULL);

newtPushHelpLine(" Press OK to configure network interfaces or
CANCEL to exit...");
newtRefresh();
check_ret=newtRunForm(form);

if(check_ret==button1) {
    newtSuspend();
    printf("Running hardware configuration utility. Please wait...");
    fflush(stdout);
    system("kudzu");
    newtResume();
}

else {
newtFinished();
system("clear");
printf("
Other services cannot be started without configuring the network
interfaces...
You can run sysconf to configure the network interfaces at any time using:

sysconf --hwdetect\n\n");
exit(-1);
}

newtPopWindow();
newtFormDestroy(form);

}

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

//choose which services to start
text = newtTextboxReflowed(3, 1, "Select the services to be
configured.", 38, 2, 2, 0);
newtCenteredWindow(42, 17, "System Services");
button1 = newtButton(18, 13, "Ok");

checkbox1 = newtCheckbox(8, 4, "Network Configuration ", ' ', " *",
&network);
checkbox2 = newtCheckbox(8, 5, "iptables Firewall ", ' ', " *",
&iptables);
checkbox3 = newtCheckbox(8, 6, "DNS Service ", ' ', " *",
&dns);
checkbox4 = newtCheckbox(8, 7, "Apache Web Server ", ' ', " *",
&httpd);
checkbox5 = newtCheckbox(8, 8, "Sendmail Service ", ' ', " *",
&sendmail);
checkbox6 = newtCheckbox(8, 9, "IADS ", ' ', " *",
&iads);
checkbox7 = newtCheckbox(8, 10, "Keyboard Configuration ", ' ', " *",
&kbd);
checkbox8 = newtCheckbox(8, 11, "Timezone Configuration ", ' ', " *",
&time);

form = newtForm(NULL, NULL, 0);
newtFormAddComponents(form, text, checkbox1, checkbox2, checkbox3,
checkbox4, checkbox5, checkbox6, checkbox7, checkbox8, button1, NULL);

newtPushHelpLine(" Use SPACE to select or deselect a service from the
list...");
newtRefresh();
check_ret=newtRunForm(form);

newtPopWindow();
newtFormDestroy(form);

//start netconf
if(network=='*') {
    newtSuspend();
    system("netconf");
    newtResume();
}

//start dns server
if(dns=='*') {
    newtSuspend();
    system("dnsconf");
    newtResume();
}

//start firewall
if(iptables=='*') {
    newtSuspend();

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```

        system("iptables");
        newtResume();
    }

    //start apache web server
    if(httpd=='*') {
        newtSuspend();
        system("apacheconf");
        newtResume();
    }

    //start sendmail
    if(sendmail=='*') {
        newtSuspend();
        system("smconf");
        newtResume();
    }

    //start IADS
    if(iads=='*') {
        newtSuspend();
        system("iadsconf");
        newtResume();
    }

    //configure keyboard
    if(kbd=='*') {
        newtSuspend();
        system("kbdconfig");
        newtResume();
    }

    //configure timezone
    if(time=='*') {
        newtSuspend();
        system("timeconfig");
        newtResume();
    }

    //finished message
    strcpy(msg, "System configuration is now complete.\n\nYou can monitor
network activity using the IPTRAF utility.\n\nAt the shell,\nPress
Ctrl+Alt+F5 at any time to run IPTRAF.\nPress Ctrl+Alt+F6 at any time to
run this configuration tool.");
    text = newtTextboxReflowed(2, 1, msg, 38, 2, 2, 0);
    button1 = newtButton(18, newtTextboxGetNumLines(text) + 2, "Ok");
    newtCenteredWindow(45, newtTextboxGetNumLines(text) + 6, "Finished");
    newtPushHelpLine("    Press OK to exit...");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, text, button1, NULL);

    newtRunForm(form);

```

```

newtPopWindow();
newtFormDestroy(form);

newtFinished();

system("clear");
}

```

apacheconf.c

```

#include<newt.h>
#include<stdlib.h>
#include<stdio.h>

newtComponent form, button1, button2, button3, button4, button5,
check_ret;

newtComponent init_screen() {
    newtComponent check;

    button1 = newtCompactButton(5, 2, " Configure Apache Server ");
    button2 = newtCompactButton(5, 4, "   Start Apache Server   ");
    button3 = newtCompactButton(5, 6, "     View Apache log     ");
    button4 = newtCompactButton(5, 8, "   Stop Apache Server   ");
    button5 = newtCompactButton(5, 10, "       Exit               ");

    newtCenteredWindow(40, 13, "ApacheConf");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, button1,
button2,button3,button4,button5, NULL);

    newtPushHelpLine("  Choose the operation to be performed by
pressing ENTER...");
    newtRefresh();

    check=newtRunForm(form);

    newtPopWindow();
    newtFormDestroy(form);

    return check;
}

void configure_server() {

    newtSuspend();
    system("clear");
    system("vim -Z /etc/httpd/conf/httpd.conf");
}

```

```

    newtResume();
}

void start_server() {

    newtSuspend();
    system("clear");
    system("service httpd start");
    newtResume();
}

void view_log() {

    newtSuspend();
    system("clear");
    system("vim -Z /etc/httpd/logs/access_log");
    newtResume();
}

void stop_server() {

    newtSuspend();
    system("clear");
    system("service httpd stop");
    newtResume();
}

int main(void) {

    newtInit();
    newtCls();

    newtDrawRootText(21,1,"### The Apache configuration utility ###");

    for(;;) {

        check_ret=init_screen();

        if(check_ret == button5) {
            newtFinished();
            system("clear");
            exit(0);
        }

        if(check_ret == button1) {
            configure_server();
        }

        if(check_ret == button2) {
            start_server();
        }
    }
}

```

```

        if(check_ret == button3) {
            view_log();
        }

        if(check_ret == button4) {
            stop_server();
        }
    }
}

```

dnsconf.c

```

#include<newt.h>
#include<stdlib.h>
#include<stdio.h>

newtComponent form, button1, button2, button3, button4, button5,
check_ret;

newtComponent init_screen() {
    newtComponent check;

    button1 = newtCompactButton(6, 2, " Configure DNS Server ");
    button2 = newtCompactButton(6, 4, " Configure DNS Client ");
    button3 = newtCompactButton(6, 6, "   Start DNS Server   ");
    button4 = newtCompactButton(6, 8, "   Stop DNS Server   ");
    button5 = newtCompactButton(6, 10, "       Exit           ");

    newtCenteredWindow(40, 13, "DNSConf");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, button1,
button2,button3,button4,button5, NULL);

    newtPushHelpLine("   Choose the operation to be performed by
pressing ENTER...");
    newtRefresh();

    check=newtRunForm(form);

    newtPopWindow();
    newtFormDestroy(form);

    return check;
}

void configure_server() {

    newtSuspend();
    system("clear");
    system("vim -Z /etc/named.conf");
}

```

```

    newtResume();
}

void configure_client() {

    newtSuspend();
    system("clear");
    system("vim -Z /etc/resolv.conf");
    newtResume();
}

void start_server() {

    newtSuspend();
    system("clear");
    system("service named start");
    newtResume();
}

void stop_server() {

    newtSuspend();
    system("clear");
    system("service named stop");
    newtResume();
}

int main(void) {

    newtInit();
    newtCls();

    newtDrawRootText(21,1,"### The DNS configuration utility ###");

    for(;;) {

        check_ret=init_screen();

        if(check_ret == button5) {
            newtFinished();
            system("clear");
            exit(0);
        }

        if(check_ret == button1) {
            configure_server();
        }

        if(check_ret == button2) {
            configure_client();
        }
    }
}

```

```

        }

    if(check_ret == button3) {
        start_server();
    }

    if(check_ret == button4) {
        stop_server();
    }
}
}

```

iptconf.c

```

#include<newt.h>
#include<stdlib.h>
#include<stdio.h>

newtComponent form, button1, button2, button3, button4, check_ret;

newtComponent init_screen() {
    newtComponent check;

    button1 = newtCompactButton(4, 3, " Start iptables Firewall ");
    button2 = newtCompactButton(4, 5, " Configure iptables rules ");
    button3 = newtCompactButton(4, 7, " Stop iptables Firewall ");
    button4 = newtCompactButton(4, 9, " Exit ");

    newtCenteredWindow(40, 13, "IPTConf");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, button1, button2, button3, button4,
NULL);

    newtPushHelpLine(" Choose the operation to be performed by
pressing ENTER...");
    newtRefresh();

    check=newtRunForm(form);

    newtPopWindow();
    newtFormDestroy(form);

    return check;
}

void start_iptables() {

    newtSuspend();
    system("clear");
    system("service iptables start");
}

```

```

    newtResume();
}

void configure_iptables() {

    newtSuspend();
    system("clear");
    system("/usr/bin/ipmenu");
    newtResume();
}

void stop_iptables() {

    newtSuspend();
    system("clear");
    system("service iptables stop");
    newtResume();
}

int main(void) {

    newtInit();
    newtCls();

    newtDrawRootText(18,1,"### The iptables configuration utility ###");

    for(;;) {

        check_ret=init_screen();

        if(check_ret == button4) {
            newtFinished();
            system("clear");
            exit(0);
        }

        if(check_ret == button1) {
            start_iptables();
        }

        if(check_ret == button2) {
            configure_iptables();
        }

        if(check_ret == button3) {
            stop_iptables();
        }

    }
}

```

smconf.c

```

#include<newt.h>
#include<stdlib.h>
#include<stdio.h>

newtComponent form, button1, button2, button3, button4, check_ret;

newtComponent init_screen() {
    newtComponent check;

    button1 = newtCompactButton(7, 3, "  Configure Sendmail  ");
    button2 = newtCompactButton(7, 5, "    Start Sendmail    ");
    button3 = newtCompactButton(7, 7, "    Stop Sendmail    ");
    button4 = newtCompactButton(7, 9, "      Exit            ");

    newtCenteredWindow(40, 13, "SendmailConf");

    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, button1, button2, button3, button4,
NULL);

    newtPushHelpLine("  Choose the operation to be performed by
pressing ENTER...");
    newtRefresh();

    check=newtRunForm(form);

    newtPopWindow();
    newtFormDestroy(form);

    return check;
}

void start_sendmail() {

    newtSuspend();
    system("clear");
    system("service sendmail start");
    newtResume();
}

void configure_sendmail() {

    newtSuspend();
    system("clear");
    system("vim -Z /etc/sendmail.cf");
    newtResume();
}

void stop_sendmail() {

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```
        newtSuspend();
        system("clear");
        system("service sendmail stop");
        newtResume();
    }

int main(void) {

    newtInit();
    newtCls();

    newtDrawRootText(18,1,"### The Sendmail Configuration utility ###");

    for(;;) {

        check_ret=init_screen();

        if(check_ret == button4) {
            newtFinished();
            system("clear");
            exit(0);
        }

        if(check_ret == button1) {
            configure_sendmail();
        }

        if(check_ret == button2) {
            start_sendmail();
        }

        if(check_ret == button3) {
            stop_sendmail();
        }

    }
}
```

iadsconf.c

```
#include<newt.h>
#include<stdlib.h>
#include<stdio.h>
#include <time.h>
#include <signal.h>

#define RULE_FILE "/etc/iads/iads_rules.xml"
#define PID_FILE "/var/run/iads.pid"
```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```
newtComponent form, button1, button2, button3, button4, button5, button6,  
button7, check_ret;
```

```
char help[]="  
EXAMPLE RULES
```

```
    <rule>  
        <!-- Comment line -->  
        <name>EXAMPLE RULE</name>  
        <impact type=\"recon\" />  
        <action>  
            <log/>  
        </action>  
        <condition>  
            <AND>  
                <ip_protocol>6</ip_protocol>  
                <ip_source_address>0.0.0.0</ip_source_address>  
                <tcp_destination_port>0</tcp_destination_port>  
            </AND>  
        </condition>  
    </rule>
```

CONDITION

The condition element is the root of an arbitrarily complex boolean expression that must be satisfied for the actions specified in the rule to be executed. A condition may contain only one element which is either one of the boolean elements or a match element. Conditions are only valid inside of stateless rules.

BOOLEAN ELEMENTS

AND

The AND element one of the four boolean elements that can occur within the condition. All boolean elements may contain themselves. There is no hard limit on the depth of boolean expressions but expressions that are too deep will likely be very inefficient. The AND element must have at least two children which may be any combination of boolean or match elements. The AND condition evaluates true if all of its children evaluate true.

OR

The OR condition is one of the four boolean elements and is similar to AND. It evaluates true if one or more than one of its children evaluate true.

XOR

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

The XOR condition is one of the four boolean elements and is similar to AND. The XOR condition evaluates true if only one of its children evaluate true.

NOT

The NOT condition is one of the four boolean elements and is similar to AND. It differs in that it may only have one child which may be either a match or boolean element. It evaluates true if its child evaluates false.

MATCH ELEMENTS

The following match elements correlate to header values in IP, TCP, UDP, or ICMP packets. In most cases, ranges may be specified using the hyphen.

ip_version

ip_header_length

ip_tos

ip_total_length

ip_identification

ip_reserved

ip_df

This is a flag element and any value it contains will be ignored. Like all match elements, it may be negated with the NOT boolean element.

ip_mf

Flag element similar to ip_df...

ip_offset

ip_ttl

ip_protocol

ip_checksum

ip_source_address

The ip_source_address element may take a range of values using a hyphen but will also take a CIDR formatted address. (i.e 10.5.0.0/16).

ip_destination_address

Similar to ip_source_address.

ip_address

The value contained in the ip_address element will evaluate true if either the destination or source IP address match.

tcp_source_port

tcp_destination_port

tcp_port

The value contained in the tcp_port element will evaluate true if either the tcp destination or source port match.

tcp_sequence_number

tcp_acknowledge_number

tcp_header_length

tcp_reserved

tcp_urg

Flag element similar to ip_df.

tcp_ack

Flag element similar to ip_df.

tcp_psh

Flag element similar to ip_df.

tcp_rst

Flag element similar to ip_df.

tcp_syn

Flag element similar to ip_df.

tcp_fin

Flag element similar to ip_df.

tcp_window_size

tcp_checksum

tcp_urgent_pointer

udp_source_port

udp_destination_port

udp_port

The value contained in the udp_port element will evaluate true if either the udp destination or source port match.

udp_length

```

udp_checksum

icmp_type

icmp_code

icmp_checksum
";
newtComponent init_screen() {
    newtComponent check;

    button1 = newtCompactButton(10, 2, "    Start IADS    ");
    button2 = newtCompactButton(10, 4, " Create new Rule ");
        button3 = newtCompactButton(10, 6, " Open Rule File  ");
        button4 = newtCompactButton(10, 8, " Open Alert File ");
        button5 = newtCompactButton(10, 10, "  Open Log File  ");
    button6 = newtCompactButton(10, 12, "    Stop IADS    ");
        button7 = newtCompactButton(10, 14, "        Exit      ");

    newtCenteredWindow(40, 17, "Welcome to IADSConf");

        form = newtForm(NULL, NULL, 0);
        newtFormAddComponents(form, button1,
button2,button3,button4,button5, button6, button7, NULL);

        newtPushHelpLine("    Choose the operation to be performed by
pressing ENTER...");
        newtRefresh();

        check=newtRunForm(form);

        newtPopWindow();
        newtFormDestroy(form);

    return check;
}

char *rule_name, *ref_origin, *ref_name, *ref_url, *im_type, *im_comp,
*im_sever, *action,
*condition, *act_rule;

void append_to_rule_file(void) {

    FILE *rule_file;
    time_t curtime;
    char *ftime;
    char finstr[] = "</rule_base>";
    char buf[20];

    rule_file = fopen(RULE_FILE, "r+");

    if (rule_file == NULL)
        return;

```

```

fseek(rule_file, -strlen(finstr), SEEK_END);

while(1) {
    fgets(buf, strlen(finstr) + 1, rule_file);
    fseek(rule_file, -(strlen(finstr) + 1), SEEK_CUR);
    if (!strcmp(buf, finstr))
        break;
}

    fprintf(rule_file, "\n\n\t<rule>\n");

    curtime = time(NULL);
    ftime = ctime(&curtime);
    ftime[strlen(ftime) - 1] = '\0';

    fprintf(rule_file, "\t\t<!-- Rule created by IADSConf at: %s -->\n", ftime);
    fprintf(rule_file, "\t\t\t<name>%s</name>\n", rule_name);

    fprintf(rule_file, "\t\t\t<reference origin=\"%s\">\n", ref_origin);

    strcpy(ref_name, (strcmp(ref_name, "") ? ref_name : "unknown"));
    fprintf(rule_file, "\t\t\t\t<reference_name>%s</reference_name>\n", ref_name);

    strcpy(ref_url, (strcmp(ref_url, "") ? ref_url : "unknown"));
    fprintf(rule_file, "\t\t\t\t\t<reference_url>%s</reference_url>\n", ref_url);
    fprintf(rule_file, "\t\t\t</reference>\n");

    fprintf(rule_file, "\t\t\t<impact type=\"%s\" completion=\"%s\" severity=\"%s\" />\n", im_type, im_comp, im_sever);

    fprintf(rule_file, "\t\t\t<action>\n");

    if (!strcmp(action, "log"))
        fprintf(rule_file, "\t\t\t\t<log />\n");
    else if (!strcmp(action, "discard"))
        fprintf(rule_file, "\t\t\t\t\t<discard />\n");
    else if (!strcmp(action, "activate_rule"))
        fprintf(rule_file, "\t\t\t\t\t<activate_rule>%s</activate_rule>\n", act_rule);
    else if (!strcmp(action, "deactivate_rule"))
        fprintf(rule_file, "\t\t\t\t\t<deactivate_rule>%s</deactivate_rule>\n", act_rule);

    fprintf(rule_file, "\t\t\t</action>\n");

    fprintf(rule_file, "\t\t\t<condition>\n\t\t\t\t\t%s\n\t\t\t</condition>\n", condition);

    fprintf(rule_file, "\t</rule>\n");
    fprintf(rule_file, "</rule_base>\n");

```

```

    fflush(rule_file);
        fclose(rule_file);
}

void create_rule() {

    newtComponent formc, namelabel, referencelabel, condlabel,
act_rulelabel, ref_originlabel, ref_namelabel, ref_urllabel,
ref_nameentry, ref_urlentry, impactlabel, actionlabel, nameentry,
list_origin, im_typelabel, im_complabel, im_sevlabel, list_type,
list_comp, list_sever, list_action, button1, button2, button3, button4,
text1, text2, condentry, act_ruleentry, formhelp;

    newtCenteredWindow(75, 20, "Create a new rule");

    namelabel = newtLabel(1, 1, "Rule Name ");
    referencelabel = newtLabel(1, 3, "Reference ");
    impactlabel = newtLabel(1, 7, "Impact ");
    actionlabel = newtLabel(1, 11, "Action ");
    condlabel = newtLabel(1, 13, "Condition ");

    ref_namelabel = newtLabel(16, 4, "Name ");
    ref_urllabel = newtLabel(16, 5, "URL ");
    ref_originlabel = newtLabel(16, 3, "Origin ");
    act_rulelabel = newtLabel(36, 11, " Subject Rule ");

    im_typelabel = newtLabel(16, 7, "Type ");
    im_complabel = newtLabel(16, 8, "Completion");
    im_sevlabel = newtLabel(16, 9, "Severity");

    nameentry = newtEntry(16, 1, NULL, 30, &rule_name, NEWT_FLAG_SCROLL);
    ref_nameentry = newtEntry(30, 4, NULL, 20, &ref_name,
NEWT_FLAG_SCROLL);
    ref_urlentry = newtEntry(30, 5, NULL, 20, &ref_url, NEWT_FLAG_SCROLL);
        condentry = newtEntry(16, 13, NULL, 55, &condition,
NEWT_FLAG_SCROLL);
    act_ruleentry = newtEntry(50, 11, NULL, 21, &act_rule,
NEWT_FLAG_SCROLL);

    list_origin=newtListbox(30, 3, 1, NEWT_FLAG_SCROLL);
    newtListboxAppendEntry(list_origin, "unknown", "unknown");
    newtListboxAppendEntry(list_origin, "bugtraqid", "bugtraqid");
    newtListboxAppendEntry(list_origin, "cve", "cve");
    newtListboxAppendEntry(list_origin, "vendor-specific", "vendor-
specific");

    list_type=newtListbox(30, 7, 1, NEWT_FLAG_SCROLL);
    newtListboxAppendEntry(list_type, "other", "other");
    newtListboxAppendEntry(list_type, "admin", "admin");

```

```

newtListboxAppendEntry(list_type, "dos","dos");
newtListboxAppendEntry(list_type, "file","file");
newtListboxAppendEntry(list_type, "recon","recon");
newtListboxAppendEntry(list_type, "user","user");

list_comp=newtListbox(30, 8, 1, NEWT_FLAG_SCROLL);
newtListboxAppendEntry(list_comp, "attempted","attempted");
newtListboxAppendEntry(list_comp, "successful", "successful");
newtListboxAppendEntry(list_comp, "failed","failed");

list_sever=newtListbox(30, 9, 1, NEWT_FLAG_SCROLL);
newtListboxAppendEntry(list_sever, "high","high");
newtListboxAppendEntry(list_sever, "medium", "med");
newtListboxAppendEntry(list_sever, "low","low");

list_action=newtListbox(16, 11, 1, NEWT_FLAG_SCROLL);
newtListboxAppendEntry(list_action, "log","log");
newtListboxAppendEntry(list_action, "discard","discard");
newtListboxAppendEntry(list_action, "activate rule","activate_rule");
newtListboxAppendEntry(list_action, "deactivate
rule","deactivate_rule");

        button1 = newtButton(15, 15, "Help");
        button2 = newtButton(33, 15, "Save");
        button3 = newtButton(50, 15, "Cancel");

formc = newtForm(NULL, NULL, 0);
newtFormAddComponents(formc, namelabel, referencelabel, impactlabel,
actionlabel, act_rulelabel, condlabel, ref_namelabel, ref_urllabel,
ref_originlabel, im_typelabel, im_complabel, im_sevlabel, nameentry,
list_origin, ref_nameentry, ref_urlentry, list_type, list_comp,
list_sever, list_action, act_ruleentry, condentry, button1, button2,
button3, NULL);

newtPushHelpLine("Enter values for the fields... Press HELP to view
rule syntax");

for(;;) {
    check_ret=newtRunForm(formc);

    if(check_ret==button3) {
        //cancel creation of rule
        newtPopWindow();
        newtFormDestroy(formc);
        return;
    }

    if(check_ret==button1) {
        //help

        text2=newtTextbox(1, 1, 55, 15, NEWT_FLAG_SCROLL|NEWT_FLAG_WRAP);
        newtTextboxSetText(text2, help);
    }
}

```

```

newtCenteredWindow(60,21,"HELP!");

button4 = newtButton(27, 17, "OK");

    formhelp = newtForm(NULL, NULL, 0);
    newtFormAddComponents(formhelp, text2, button4, NULL);
    newtPushHelpLine("");
    newtRefresh();

newtRunForm(formhelp);
newtPopWindow();
newtFormDestroy(formhelp);

}

if(check_ret==button2) {
    newtPopWindow();

    //save condition to file here
    ref_origin=newtListboxGetCurrent(list_origin);
    im_type=newtListboxGetCurrent(list_type);
    im_comp=newtListboxGetCurrent(list_comp);
    im_sever=newtListboxGetCurrent(list_sever);
    action=newtListboxGetCurrent(list_action);

    append_to_rule_file();

    newtFormDestroy(formc);
    return;
}
}

}

void view_rule() {

    newtSuspend();
    system("clear");
    system("vim /etc/iads/iads_rules.xml");
    newtResume();
}

void view_alert() {

    newtSuspend();
    system("clear");
    system("vim /etc/iads/iads_alert.xml");
    newtResume();
}

```

BOOTABLE FIREWALL SYSTEM WITH INTELLIGENT ANOMALY DETECTION

```
void view_log() {
    newtSuspend();
    system("clear");
    system("vim -Z /var/log/iads_error.log ");
    newtResume();
}

void start_iads() {
    newtSuspend();
    system("clear");
    system("/usr/local/sbin/ulogd 2> /dev/null");
    system("iads --daemon");
    newtResume();
}

void stop_iads() {
    int fd;
    pid_t pid;
    char pidstr[10];
    FILE *pid_file;

    if ((pid_file = fopen(PID_FILE, "r")) != NULL) {
        fgets(pidstr, 10, pid_file);
        pid = strtoul(pidstr, NULL, 10);
        kill(pid, SIGTERM);
        close(fd);
    }
}

int main(void) {
    newtInit();
    newtCls();

    newtDrawRootText(18,1,"### The IADS configuration utility ###");

    for(;;) {
        check_ret=init_screen();

        if(check_ret == button1) {
            start_iads();
        }

        if(check_ret == button2) {
            create_rule();
        }

        if(check_ret == button3) {
```

```
        view_rule();
    }
    if(check_ret == button4) {
        view_alert();
    }
    if(check_ret == button5) {
        view_log();
    }
    if(check_ret == button6) {
        stop_iads();
    }
    if(check_ret == button7) {
        newtFinished();
        system("clear");
        exit(0);
    }
}
}
```

THE END