ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Διδακτικής της Τεχνολογίας και Ψηφιακών Συστημάτων

Αναλυτής αρχείων καταγραφής για τείχη ηλεκτρονικής προστασίας στο λειτουργικό σύστημα Linux

Μιχαήλ Γ. Λαγός

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σεπτέμβριος 2006

Περίληψη

Ένας αναλυτής αρχείων καταγραφής (log analyzer) έχει ως λειτουργίες να παρακολουθεί την κυκλοφορία στο δίκτυο, τις επισκέψεις σε ιστοσελίδες, αλλά και την ποσότητα της πληροφορίας που προσπελάζεται. Με βάση τα δεδομένα που συλλέγει παρέχει πληροφορίες για την αποτελεσματική διαχείριση του εύρους ζώνης του δικτύου, την ανακάλυψη διαφόρων ειδών επιθέσεων και υιών, όπως και άλλα στοιχεία που μπορεί να είναι χρήσιμα σε έναν οργανισμό. Ο αναλυτής που παρουσιάζεται σε αυτή την διπλωματική εργασία είναι υπεύθυνος για τη συλλογή, ανάλυση και εξαγωγή πληροφορίας για κάποιο πληρεξούσιο εξυπηρετητή (proxy server), ή για ένα τείχος ηλεκτρονικής προστασίας (firewall) ή γενικότερα για κάποιο εξυπηρετητή. Η εφαρμογή τρέχει μέσω ενός τερματικού σε πλατφόρμες που είναι βασισμένες στο λειτουργικό σύστημα Linux.

Η παρούσα εργασία χωρίζεται σε πέντε ενότητες. Στην πρώτη ενότητα γίνεται μία εισαγωγή στον τρόπο ανάπτυξης ενός αναλυτή αρχείων καταγραφής αλλά και στη μεθοδολογία που ακολουθήθηκε για την ανάπτυξη του συγκεκριμένου αναλυτή. Στη δεύτερη ενότητα, γίνεται μία σύντομη παρουσίαση των τειχών ηλεκτρονικής προστασίας Διαδικτύου (Internet firewalls) και περιγράφεται ο τρόπος λειτουργίας τους. Επίσης, περιγράφεται η αρχιτεκτονική του δικτύου που χρησιμοποιήθηκε για τους σκοπούς αυτής της εργασίας. Πάνω σε αυτή την αρχιτεκτονική στηρίχτηκε τόσο η ανάπτυξη, όσο και η μορφοποίηση του τείχους ηλεκτρονικής προστασίας και του πληρεξούσιου εξυπηρετητή. Τέλος, γίνεται μία αναφορά στη σημασία και σπουδαιότητα μίας πολιτικής ασφαλείας (security policy) για έναν οργανισμό.

Η τρίτη ενότητα ασχολείται κυρίως με θέματα πληρεξούσιων συστημάτων (proxy systems). Συγκεκριμένα, γίνεται μία ανάλυση των πλεονεκτημάτων και των μειονεκτημάτων του πληρεξούσιου εξυπηρετητή, περιγράφεται ο τρόπος λειτουργίας του

πληρεξούσιου πελάτη και εξυπηρετητή, και παρουσιάζονται οι διάφοροι τύποι των πληρεξούσιων εξυπηρετητών.

Στην τέταρτη ενότητα παρουσιάζεται ο πληρεξούσιος εξυπηρετητής που χρησιμοποιήθηκε στην παρούσα εργασία, ο οποίος είναι ο squid. Ο σκοπός αυτής της ενότητας είναι να καθοδηγήσει τον αναγνώστη σε θέματα εγκατάστασης, μορφοποίησης, και εκκίνησης του squid στο λειτουργικό σύστημα Linux. Ιδιαίτερη έμφαση δίνεται στο αρχείο καταγραφής του squid, που αποτελεί και τη βασική πηγή συγκέντρωσης της πληροφορίας για την ανάπτυξη του αναλυτή αρχείου καταγραφής του πληρεξούσιου εξυπηρετητή.

Στην πέμπτη ενότητα παρουσιάζεται αναλυτικά η διαδικασία ανάπτυξης του αναλυτή αρχείου καταγραφής πληρεξούσιου εξυπηρετητή. Γίνεται μία ανάλυση του αλγορίθμου που αναπτύχθηκε τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο και παρουσιάζεται η ανάπτυξη της εφαρμογής σε μορφή ψευδοκώδικα. Η έκτη και τελευταία ενότητα περιγράφει τον μηχανισμό φιλτραρίσματος πακέτων (packet filtering mechanism) που ελέγχει την ροή των δεδομένων από και προς το δίκτυό μας. Στην ενότητα αυτή αναλύεται διεξοδικά το εργαλείο iptables καθώς και το αρχείο καταγραφής του τείχους ηλεκτρονικής προστασίας το οποίο αποτελεί τη βασική πηγή πληροφόρησης για την ανάπτυξη του αναλυτή αρχείου καταγραφής του τείχους ηλεκτρονικής προστασίας. Επίσης, παρουσιάζεται αναλυτικά η διαδικασία ανάπτυξης του δεύτερου αναλυτή, η οποία είναι παρόμοια με αυτή που ακολουθήθηκε και στην ανάπτυξη του πληρεξούσιου εξυπηρετητή.

Από την ανάπτυξη του αναλυτή καταγραφής αρχείου για τον πληρεξούσιο εξυπηρετητή αλλά και για το τείχος ηλεκτρονικής προστασίας προκύπτει ότι με μικρές τροποποιήσεις είναι εύκολη η ανάπτυξη οποιουδήποτε άλλου αναλυτή αρχείου καταγραφής, όπως για παράδειγμα ενός αρχείου καταγραφής ηλεκτρονικού ταχυδρομείου. Ο αλγόριθμος που αναπτύχθηκε είναι αρκετά γενικός, ώστε να καθίσταται δυνατή η γρήγορη προσαρμογή του σε οποιοδήποτε αρχείο καταγραφής. Αξίζει να σημειωθεί, ότι σύμφωνα με μία

έρευνα που έκανα στο Διαδίκτυο, δεν υπάρχει κάποιος αναλυτής καταγραφής αρχείου σε μορφή ανοιχτού κώδικα (open source) που να παρέχει τη λειτουργία που μας ενδιαφέρει σε τόσο συγκεντρωτική μορφή. Υπάρχουν μόνο μερικά εμπορικά προϊόντα, όπου η ανάλυση των αρχείων καταγραφής αποτελεί τμήμα ενός υπερκείμενου ολοκληρωμένου συστήματος ασφαλείας.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

Πίνακας Περιεχομένων

ΠΕΡΙΛΗΨΗ ..... **II**

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ ..... **V**

1. INTRODUCTION **1**

2. INTERNET FIREWALL **3**

2.1 SECURITY POLICY 5
2.2 FIREWALL ARCHITECTURES 6

3. PROXY SYSTEMS **9**

3.1 PROXY ADVANTAGES 11
3.2 DISADVANTAGES OF PROXY 12
3.3 HOW PROXY SERVERS WORKS 13
3.4 HOW CLIENT SIDE WORKS 15
3.5 TYPES OF PROXY SERVERS 17
3.5.1 Application-Level versus Circuit-Level Proxies *17*
3.5.2 Intelligent Proxy Servers *18*

4. SQUID **19**

4.1 INTRODUCTION 19
4.2 HARDWARE AND OPERATING SYSTEM REQUIREMENTS 22
4.3 SETUP CONFIGURATION AND STARTUP 23
4.3.1 Setup *24*
4.3.2 Server Configuration *26*
4.3.2.1 Most Basic Settings *28*
4.3.2.1.1 User IDs *28*
4.3.2.1.2 Port Numbers *30*
4.3.2.1.3 Access Controls *30*
4.3.2.1.4 Other Parameters *33*
4.3.3 Startup *35*
4.4 LOG FILES 39
4.3.1 access.log *41*
4.3.2 Configuration Directives that Affect access.log *43*

5. PROXY LOG ANALYZER IMPLEMENTATION **46**

5.1 PSEUDOCODE 56
5.2 LOG FILE ANALYZER INSTALLATION 62
5.2.1 For the Impatiens *64*

6. PACKET FILTERING **66**

6.1 WHY PACKET FILTERING? 69
6.2 PACKET FILTERING UNDER LINUX 71
6.2.1 Netfilter/iptables *71*
6.2.1.1 Setup iptables *77*
6.2.1.2 iptables Startup *78*
6.2.1.3 Netfilter/iptables Configuration *79*
6.2.1.3.1 Kernel Setup *80*
6.2.1.3.2 Userland Setup *84*
6.2.1.3.3 State Engine *92*

6.2.1.3.4 Specifying an Interface *92*
6.2.1.4 Network Address Translation (NAT) *93*
6.2.1.5 iptables Logging *97*
6.2.1.5.1 Netfilter Log Format (firewall.log) *99*
6.3 FIREWALL LOGGING IMPLEMENTATION 105
6.3.1 Pseudocode *107*
6.2 FIREWALL LOG ANALYZER INSTALLATION 112
6.2.1 For the Impatiens *113*

APPENDIXES 116

A. TCP/IP EXAMPLE IN A REAL PARADIGM 116
B. PROXY LOG ANALYZER CODE 121
C. FIREWALL LOG ANALYZER CODE 122

REFERENCES 123

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

1. Introduction

This log analyzer is a log analysis tool that collects analyses and reports information on enterprise-wide firewalls, proxy servers, and radius servers. It will help you to audit traffic, monitor web site visits, control the amount of information (per-byte) accessed and manage your network bandwidth efficiently, and also help you to ensure appropriate usage of networks by employees (legal issues, information exchange with competitors, etc). The program runs in a Linux-based system via a terminal.

To begin with, I looked into some general issues concerning firewall, proxy and security, in order to enhanced my knowledge in the field that I was about to get into. After a first examination of some basic issues about file I/O in C (I was not very comfortable on this issue), I moved on to the second part of the actual implementation. Firstly, I tried to find the best way to parse the access log file of squid proxy server (and the other log files such as firewall log and mail log). I started with the log file of the proxy server because it is the most complicated of all the others. By implementing the access log file, it is much easier to implement all other log files (*mail.log*, *firewall.log*, etc). However, the main problem is the performance. Initially, I attempted to parse the file by using arrays. I made data structures for each column in the *access.log*. This technique seems to be very slow, because the log file is a number of *Gigabytes* in most cases (especially when we refer to an enterprise-wide applications). After an extended investigation on the above issues, I found that the best way to deal with the above is a combination of *linked lists* and *hash tables*.

Linked lists consist of a number of elements grouped, or *linked*, together in a specific order. They are useful in maintaining collections of data, similar to the way that arrays are often used. However, linked lists offer important advantages over arrays in many cases. Specifically, linked lists are considerably more efficient in performing insertions and deletions. Linked lists also make use of dynamically allocated storage, which is

storage allocated at runtime. Since in many applications the size of the data is not known at compile time, this can be a nice attribute as well (Parlante, 2001).

On the other hand, *hash tables* support one of the most efficient types of searching: *hashing*. Fundamentally, a hash table consists of an array in which data is accessed via a special index called a *key*. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a *hash function*. A hash function accepts a key and returns its *hash coding*, or *hash value*.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

2. Internet Firewall

A *firewall* is the most effective way to connect a network to the Internet and still protect that network. In building construction, a firewall is designed to keep a fire from spreading from one part of the building to another. In theory, an Internet firewall serves a similar purpose: it prevents the dangers of the Internet from spreading to your internal network. An Internet firewall is most often installed at the point where your protected internal network connects to the Internet (see *figure 2-1*). All traffic coming from the Internet or going out from your internal network passes through the firewall. Because the traffic passes through it, the firewall has the opportunity to make sure that this traffic is acceptable or not (Zwicky, 2000). When designing your firewall, keep this in mind. You can't stop everything, but you can keep the risks within your range of acceptance.

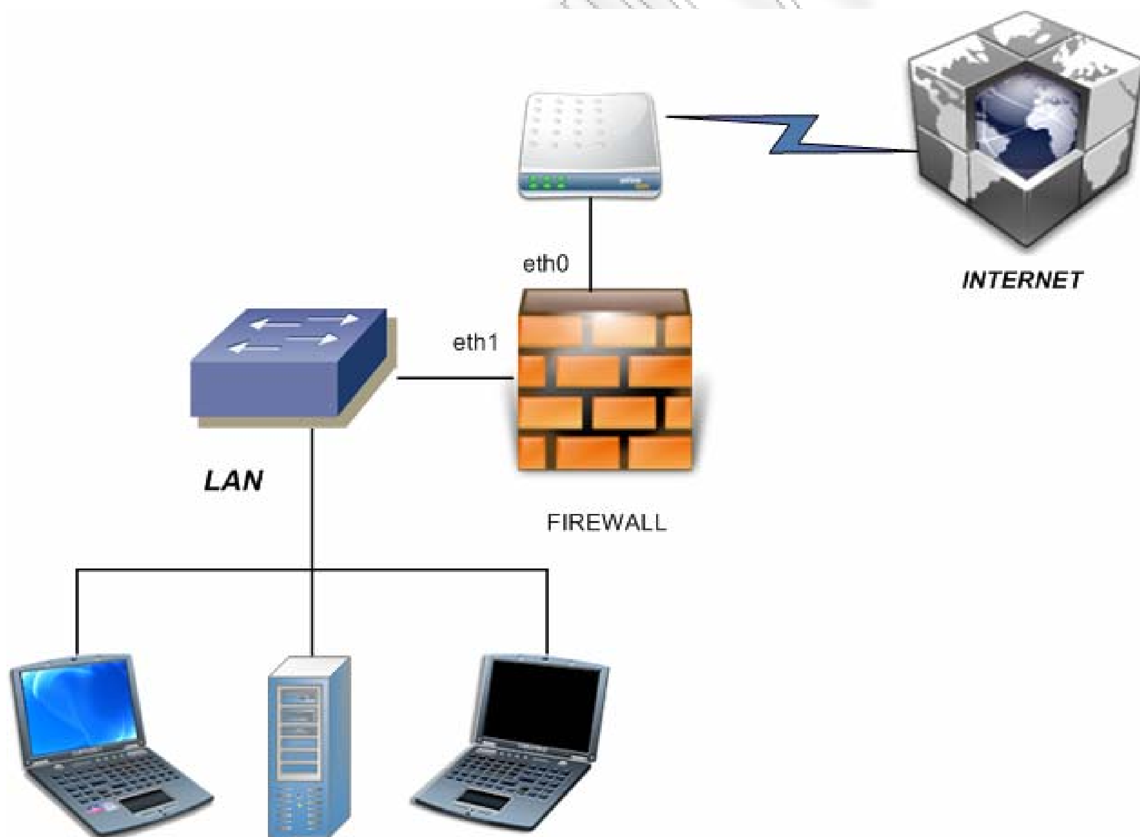


Figure 2-1.Internet firewall.

Logically, a *firewall is a separator, a restricter, an analyzer*. The physical implementation of the firewall varies from site to site. Most often, a firewall is a set of hardware components - a router, a host computer, or some combination of routers, computers, and networks with appropriate software. There are various ways to configure this equipment; the configuration will depend upon a site's particular security policy, budget, and overall operations.

A firewall is the traffic cop of the insecure Internet services. It enforces the site's security policy allowing only the “approved” services to pass through and those only within the rules setup for them. Because a firewall is like a choke point, all traffic in and out must pass through this single, narrow choke point. So, it gives you an enormous amount of leverage for network security because it lets you concentrate your security measures on this choke point. Moreover, the firewall provides a good place to collect information about system and network use-and misuse. As a single point of access, the firewall can record what occurs between the protected network and the external network. Also, it is able to limit your exposure (limits the damage that a network security problem can to the overall network).

Firewalls offer excellent protection against network threats, but they aren't a complete security solution. Certain threats are outside the control of the firewall. It can't protect you against malicious insiders, connections that don't go through it, new threats and viruses. You need to figure out other ways to protect against these threats by *incorporating physical security, host security, and user education into your overall security plan* (Zwicky, 2000).

Nevertheless, the above issues have no meaning if you don't have a security policy at your site. *Your starting point must be the security policy*. You can't just do without a policy because a firewall is an enforcement device; if you didn't have a policy before, you do once you have a firewall in place, and it may not be a policy that meets your needs.

2.1 Security Policy

Implementing a successful security policy in an organization, however, is not a straightforward task and depends on many factors. It is known that many organizations are failing to consistently provide the high quality information resources that their managers require, because of unacceptably high levels of security breaches experienced. For example, in the UK, it has recently been found that 'the number of security incidents continues to rise ', with 74% of businesses reporting a security breach in 2004, as compared with only 44% in 2000. In a similar vein, in the United States security breaches affect 90% of all businesses every year, and cost some $17 billion (Doherty, 2005). One important mechanism for protecting corporate information, in an attempt to detect, prevent and respond to security breaches is through the formulation and application of an information security policy.

There is a growing consensus both within the academic and practitioner communities that the information security policy is the basis for the dissemination and enforcement of sound security practices, within the organizational context. It is well known, at least among true security professionals, that formal policy is a prerequisite of security. The primary reason that the information security policy has become the "prerequisite" or "foundation" of effective security practices is the following: without a policy, security practices will be developed without clear demarcation of objectives and responsibilities (Doherty, 2005).

The security policy needs to describe what you're trying to protect and why; it doesn't necessarily need to describe the details of how. It's much more useful to have a 1-page document that describes what and why in terms that everyone in your organization can understand than a 100-page document that describes how, but that nobody except your most senior technical staff can understand.

2.2 Firewall Architectures

Isolation or compartmentalization is the cornerstone of security and what firewalls are really good at implementing. The principle is simple – secure an asset so that it cannot be used to break into another asset regardless of how compromised that asset might have become. An example of this would be two servers, one on either side of a firewall, that have no means of communicating with each other through the firewall. In that sense, those servers are isolated and compartmentalized from one another, so if one is broken into, it cannot be used to break into the other. Another use of compartmentalization is through the use of internal network firewalls, isolating elements of an organization's networks from one another. An example would be firewalling off the accounting department from the rest of the company to prevent unauthorized users on the company's network from accessing the accounting network (Shinn, 2005).

There exists lot of architectures that let you put firewall components together. The simplest one are the *Single-Box Architectures* that have a single object acts as the firewall (screening router for example). Another one is the *Screened Subnet Architectures* that adds an extra layer of security by adding a perimeter network that further isolates the internal network from the Internet. Moreover, another important architecture is the *Multiple Screened Subnet* that has two additional perimeter networks.

The firewall architecture that I decided to construct for the purposes of this project is the one that is depicted in *figure 2-2* that follows. This configuration assumes that you have a front-end firewall with a DMZ segment and a LAN segment. The DMZ segment contains an FTP server, a WEB server and a MAIL server. The firewall has three interfaces. One, eth0, will be the Internet reachable interface; eth1, the internal interface; and eth2 will be the DMZ. Also, the IP address in the external interface is static. The firewall protects an internal network, 10.10.10.0/24, and a DMZ network, 192.168.1.0/24, which it contains a web server with IP address 192.168.1.80, a mail server with IP address 192.168.1.25 and an FTP server with IP address 192.168.1.21. The DMZ is configured in a way that it

cannot access the internal network. All HTTP, FTP and SMTP traffic destined to the firewalls external IP address, 141.29.35.31, is DNATed to the DMZ_WEB_SERVER (IP=192.168.1.80), DMZ_FTP_SERVER (IP=192.168.1.21) and DMZ_MAIL_SERVER (IP=192.168.1.25) correspondingly. In addition, traffic from the internal network forward through the firewall, and appears to come from the 141.29.35.31 address (SNAT). SNAT and DNAT are discussed in *Section 6.2.1.4*. In addition, I use *Squid proxy server* on the firewall as a local process.

DMZ, which stands for *De-Militarized Zone* (named after the zone separating North and South Korea) is dedicated to some specific tasks such as hosting corporate web, mail, DNS, FTP and so on. A DMZ network would be used to provide limited access to/from those systems from other networks for the purpose of isolating those systems from the internal network and the Internet. The intent is not only protecting those servers, but also to protect the internal network from them. After all, they are exposed to some untrusted network, the Internet perhaps, and those servers have a higher probability of being broken into. But because they are isolated on a DMZ network, those servers cannot be used to break into the internal network from some other network (Shinn, 2005).

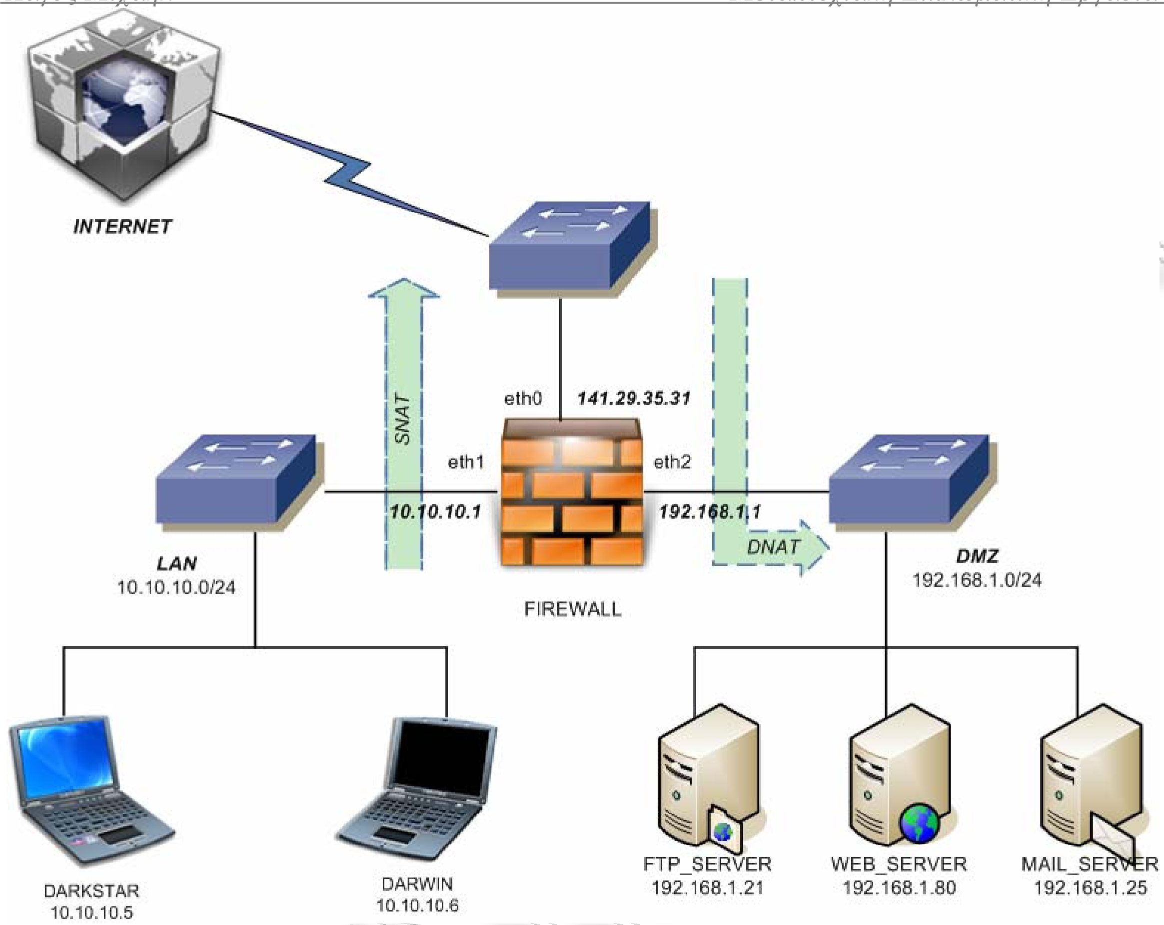


Figure 2-2.Firewall architecture.

3. Proxy Systems

There are certain risks associated with allowing people from inside an intranet to directly contact Internet servers and resources. An intranet user might obtain a file from the Internet that could damage the files on their computer and the entire intranet. Additionally, when intranet users are allowed unfettered access to the Internet, it is difficult for intranet administrators to guard against intruders who attempt to take over an intranet computer or server. A common way to block this kind of access is to use *proxy servers*. These servers sit inside a firewall, frequently on a *bastion host* (a computer system that must be highly secured because it vulnerable to attack, usually because it is exposed to the Internet and is a main point of contact for users of internal networks). They balance the two functions of providing intranet users with easy access to the Internet and keeping the network secure. When someone inside the intranet wants to contact the Internet to get information or a resource-for example, to visit a Web page-they don't actually contact the Internet directly. Instead, they contact a proxy server inside an intranet firewall, and the proxy server contacts the Internet (in this instance, a Web server). The Web server sends the proxy server the page, and the proxy server then sends that page to the requester on the intranet. The operation of the proxy server is depicted in *figure 3-1*:

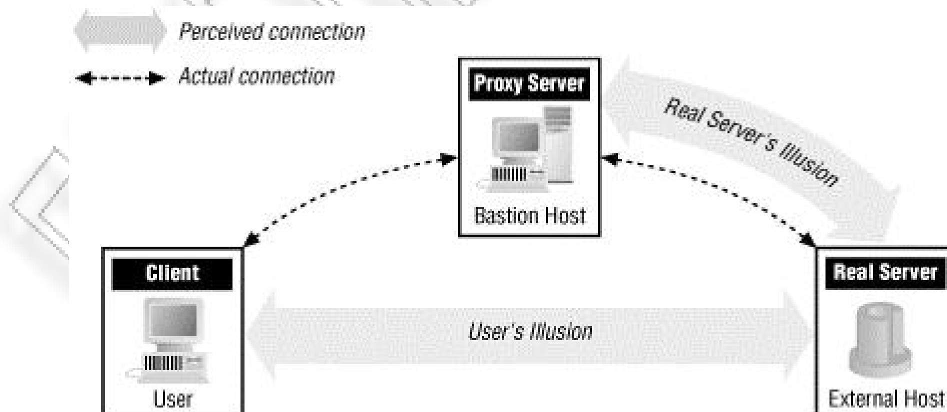


Figure 3-1.Proxy reality and illusion.

Basically, an application proxy is an application that runs on your firewall that relays traffic between you and your destination. The added advantage here is that the traffic is being sent / received between both endpoints by a third-party application, meaning you can enforce very specific guidelines on the way the traffic is crafted between both points.

In this project, when we are talking about proxy services, we are specifically talking about proxies run for security purposes, which are run on a firewall host: either a *dual-homed host* with an interface on the internal network and one on the external network, or some other *bastion host* that has access to the Internet and is accessible from the internal machines. Proxy servers can log all actions they take so that intranet administrators can check for attacks.

You will also run into proxies that are primarily designed for network efficiency instead of for security; these are *caching proxies*, which keep copies of the information for each request that they proxy. The advantage of a caching proxy is that if multiple internal hosts request the same data, the data can be provided directly by the proxy. Caching proxies can significantly reduce the load on network connections. There are proxy servers that provide both security and caching (Zwicky, 2000).

There may be multiple proxy servers on a single intranet. There may be separate proxy servers for the *Web*, *Telnet*, *FTP*, and other *Internet services*. Often on an intranet, some services will require a proxy server, while others will not. For example, this includes anything involving Telnet or FTP, because they involve file transferring, and they would be likely to be on a proxy server. When a new Internet resource is first made available, such as streaming multimedia files, proxy servers usually can't be used because proxy server technology has not yet been developed for it. The intranet administrator will have to decide whether to block those services completely or let them be used until proxy software catches up to the new technology.

3.1 Proxy Advantages

Proxying has many advantages. One of the most important is the scope that is primarily designed which is network efficiency (*caching*). Since all requests are passing through the proxy service anyway, the proxy can provide caching, keeping local copies of the requested data. If the number of repeat requests is significant, caching can significantly increase performance and reduce the load on network links (Zwicky, 2000).

Another important advantage is the *logging* (the main purpose of this project). Because proxy servers can understand the application protocol, they can allow logging to be performed in a particularly effective way.

Moreover, proxy can do *intelligent filtering*. Since a proxy service is looking at specific connections, it is frequently able to do filtering more intelligently than a packet filter. For instance, proxy services are much more capable of filtering HTTP by content type (for instance, to remove Java or JavaScript) and better at virus detection than packet filtering systems.

Furthermore, proxy systems automatically provide *protection for weak or faulty IP implementations*. As a proxy system sits between a client and the Internet, it generates completely new IP packets for the client. It can therefore protect clients from deliberately malformed IP packets.

Also, proxy systems can *perform user-level authentication*. Because a proxy system is actively involved in the connection, it is easy for it to do user authentication and to take actions that depend on the user involved. Although this is possible with packet filtering systems, it is much more difficult (Zwicky, 2000).

3.2 Disadvantages of Proxy

There are also some disadvantages of Proxy systems. One of the most important is the *lag behind nonproxied services*. When a new Internet resource is first made available, such as streaming multimedia files, proxy servers usually can't be used because proxy server technology has not yet been developed for it. The intranet administrator will have to decide whether to block those services completely or let them be used until proxy software catches up to the new technology.

Moreover, another disadvantage of the proxy is that may *require different servers for each service*. You may need a different proxy server for each protocol, because the proxy server may need to understand the protocol in order to determine what to allow and disallow, and in order to masquerade as a client to the real server and as the real server to the proxy client. Collecting, installing, and configuring all these various servers can be a lot of work. Again, you may be able to use a generic proxy, but generic proxies provide only the same sorts of protection and functionality that you could get from packet filters (Zwicky, 2000).

In addition, Proxy systems usually *require modifications to clients, applications, or procedures*. Except for services designed for proxying, you will need to use modified clients, applications, and/or procedures. These modifications can have drawbacks; people can't always use the readily available tools with their normal instructions. Because of these modifications, proxied applications don't always work as well as nonproxied applications. They tend to bend protocol specifications, and some clients and servers are less flexible than others (Zwicky, 2000).

3.3 How Proxy Servers Works

Well, in much the same way a proxy for voting in your next Chairman would work. Assuming you are eligible to vote, but on this particular day (that the elections are being held), you happen to be out of town. However, having anticipated this, you ask whether you can cast your vote by proxy. So, finding a person whom you trust, you secretly tell them your vote and they, on your behalf, vote for you in absentia. The Proxy server is just like this.

- ✓ When a computer on the intranet makes a request out to the Internet-such as to retrieve a Web page from a Web server-the internal computer actually contacts the proxy server, which in turn contacts the Internet server. The Internet server sends the Web page to the proxy server, which then forwards the page to the computer on the intranet.

- ✓ Proxy servers log all traffic between the Internet and the intranet. For example, a Telnet proxy server could track every single keystroke hit in every Telnet session on the intranet-and could also track how the external server on the Internet reacts to those keystrokes. Proxy servers can log every IP address, date and time of access; URL, number of bytes downloaded, and so on. This information can be used to analyze any attacks launched against the network. It can also help intranet administrators build better access and services for employees.

- ✓ Some proxy servers must work with special proxy clients. A more popular approach is to use off-the-shelf clients such as Netscape with proxy servers. When such an off-the-shelf package is used, it must be specially configured to work with proxy servers from a configuration menu. Then the intranet employee uses the client software as usual. The client software knows to go out to a proxy server to get the data, instead of to the Internet.

- ✓ Proxy servers can do more than relay requests back and forth between an intranet

and the Internet. They can also implement security schemes. For example, an FTP proxy server could be set up to allow files to be sent from the Internet to a computer on the intranet, but to block files from being sent from the corporate network out to the Internet-or vice versa. In this way, intranet administrators can block anyone outside the corporation from downloading vital corporate data. Or they can stop intranet users from downloading files which may contain viruses.

- ✓ Proxy servers can also be used to speed up the performance of some Internet services by caching data-keeping copies of the requested data. For example, a Web proxy server could cache many Web pages, so that whenever someone from the intranet wanted to get one of those Web pages, they could get it directly from the proxy server across high-speed intranet lines, instead of having to go out across the Internet and get the page at a lower speed from Internet lines.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

3.4 How Client Side Works

On the client side, it needs one of the following:

- ✓ **Proxy-aware application software**

  With this approach, the software must know how to contact the proxy server instead of the real server when a user makes a request (for example, for FTP or Telnet), and how to tell the proxy server what real server to connect to.

- ✓ **Proxy-aware operating system software**

  With this approach, the operating system that the client is running on is modified so that IP connections are checked to see if they should be sent to the proxy server. This mechanism usually depends on dynamic runtime linking (the ability to supply libraries when a program is run). This mechanism does not always work and can fail in ways that are not obvious to users.

- ✓ **Proxy-aware user procedures**

  With this approach, the user uses client software that doesn't understand proxying to talk to the proxy server and tells the proxy server to connect to the real server, instead of telling the client software to talk to the real server directly.

- ✓ **Proxy-aware router**

  With this approach, nothing on the client's end is modified, but a router intercepts the connection and redirects it to the proxy server or proxies the request. This requires an intelligent router in addition to the proxy software (although the routing and the proxying can co-exist on the same machine). In the ***figure 3-2*** that follows is depicted this operation.

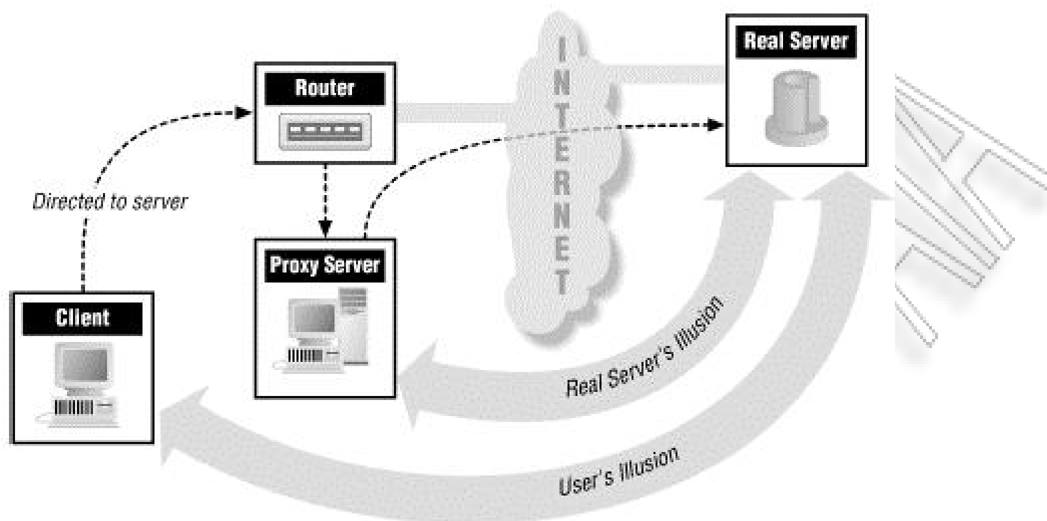


Figure 3-2.Proxy-aware router redirecting connections.

3.5 Types of Proxy Servers

There are different types of proxy servers. In this section is described a number of specific types of proxy servers.

3.5.1 Application-Level versus Circuit-Level Proxies

An *application-level proxy* is one that knows about the particular application it is providing proxy services for; it understands and interprets the commands in the application protocol. A *circuit-level proxy* is one that creates a circuit between the client and the server without interpreting the application protocol. The most extreme version of an application-level proxy is an application like *Sendmail*, which implements a store-and-forward protocol. The most extreme version of a circuit-level proxy is an application like *plug-gw*, which accepts all data that it receives and forwards it to another destination.

The *advantage of a circuit-level proxy* is that it provides service for a wide variety of different protocols. Most circuit-level proxy servers are also generic proxy servers; they can be adapted to serve almost any protocol. Not every protocol can easily be handled by a circuit-level proxy, however. Protocols like FTP, which communicate port data from the client to the server, require some protocol-level intervention, and thus some application-level knowledge. *The disadvantage of a circuit-level proxy server* is that it provides very little control over what happens through the proxy. Like a packet filter, it controls connections on the basis of their source and destination and can't easily determine whether the commands going through it are safe or even in the expected protocol. Circuit-level proxies are easily fooled by servers set up at the port numbers assigned to other services (Zwicky, 2000).

In general, circuit-level proxies are functionally equivalent to packet filters. They do provide extra protection against problems with packet headers (as opposed to the data within the packets). In addition, some kinds of protections (protection against packet fragmentation problems, for instance) are automatically provided by even the most trivial circuit-level proxies but are available only from high-end packet filters.

3.5.2 Intelligent Proxy Servers

A proxy server can do a great deal more than simply relay requests; one that does is an *intelligent proxy server*. For example, almost all HTTP proxy servers cache data, so that multiple requests for the same data don't go out across the Internet. Proxy servers (particularly application-level servers) can provide better logging and access controls than those achieved through other methods, although few existing proxy servers take full advantage of the opportunities. As proxy servers mature, their abilities are increasing rapidly. Now that there are multiple proxy suites that provide basic functionality, they're beginning to compete by adding features. It's easier for a dedicated, application-level proxy server to be intelligent; a circuit-level proxy has limited abilities.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

4. Squid

4.1 Introduction

Squid is a high-performance popular open source proxy caching server for web clients, supporting FTP, gopher, and HTTP data objects. With Squid, you can:

- ✓ Use less bandwidth on your Internet connection when surfing the Web
- ✓ Reduce the amount of time web pages take to load
- ✓ Protect the hosts on your internal network by proxying their web traffic
- ✓ Collect statistics about web traffic on your network
- ✓ Prevent users from visiting inappropriate web sites at work or school
- ✓ Ensure that only authorized users can surf the Internet
- ✓ Enhance your user's privacy by filtering sensitive information from web requests
- ✓ Reduce the load on your web server(s)
- ✓ Convert encrypted (HTTPS) requests on one side, to unencrypted (HTTP) requests on the other

Squid's job is to be both a *proxy* and a *cache*. As a proxy, Squid is an intermediary in a web transaction. It accepts a request from a client, processes that request, and then forwards the request to the origin server. The request may be logged, rejected, and even modified before forwarding. As a cache, Squid stores recently retrieved web content for possible reuse later. Subsequent requests for the same content may be served from the cache, rather than contacting the origin server again (Wessels, 2004). You can disable the caching part of Squid if you like, but the proxying part is essential and on this part based the project.

As *Figure 4-1* shows, Squid accepts HTTP (and HTTPS) requests from clients, and speaks a number of protocols to servers. In particular, Squid knows how to talk to HTTP,

FTP, and Gopher servers. Conceptually, Squid has two “sides”. The client-side talks to web clients (e. g., browsers and user-agents); the server-side talks to HTTP, FTP, and Gopher servers. These are called origin servers, because they are the origin location for the data they serve. Note that Squid's client-side understands only HTTP (and HTTP encrypted with SSL/TLS). This means, for example, that you can't make an FTP client talk to Squid (unless the FTP client is also an HTTP client). Furthermore, Squid can't proxy protocols for email (SMTP), instant messaging, or Internet Relay Chat.

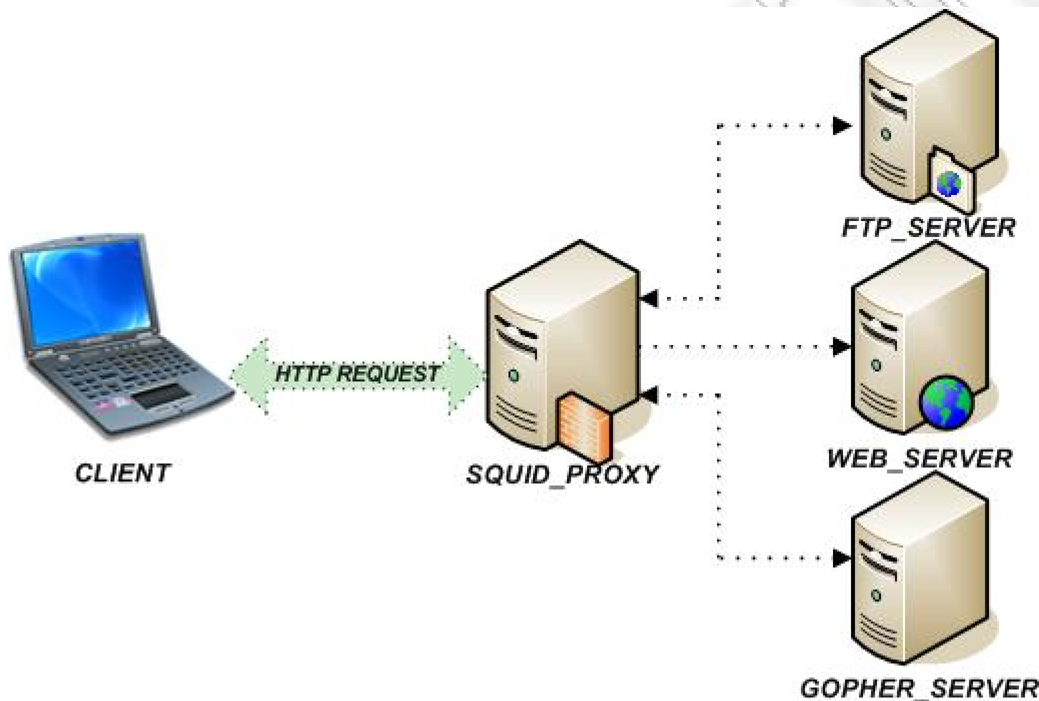


Figure 4-1.Squid sits between clients and server.

Unlike traditional caching software, Squid handles all requests in a single, non-blocking, I/O-driven process (Programs that use non-blocking I/O tend to follow the rule that every function has to return immediately, i.e. all the functions in such programs are non-blocking. Thus control passes very quickly from one routine to the next). Moreover, Squid keeps meta data and especially hot objects cached in RAM, caches DNS lookups, supports non-blocking DNS lookups, and implements negative caching of failed requests. Also, Squid supports SSL, extensive access controls, and full request logging. By using

the lightweight Internet Cache Protocol, Squid caches can be arranged in a hierarchy or mesh for additional bandwidth savings (Eisermann, 2002).

Squid consists of a main server program *squid*, a Domain Name System lookup program *dnsserver*, some optional programs for rewriting requests and performing authentication, and some management and client tools. When *squid* starts up, it spawns a configurable number of *dnsserver* processes, each of which can perform a single, blocking Domain Name System (DNS) lookup. This reduces the amount of time the cache waits for DNS lookups.

Internet object caching is a way to store requested Internet objects (i.e., data available via the HTTP, FTP, and gopher protocols) on a system closer to the requesting site than to the source. Web browsers can then use the local Squid cache as a proxy HTTP server, reducing access time as well as bandwidth consumption (Wessels, 2004).

4.2 Hardware and Operating System Requirements

Squid runs on all popular UNIX systems, as well as Microsoft Windows. Although Squid's Windows support is improving all the time, you may have an easier time with UNIX. If you have a favorite operating system, I'd suggest using that one. Otherwise, if you're looking for a recommendation, I really like *Redhat-like* systems (especially *Fedora Core 5*).

Squid's hardware requirements are generally modest. Memory is often the most important resource. A memory shortage causes a drastic degradation in performance. Disk space is, naturally, another important factor. More disk space means more cached objects and higher hit ratios. Fast disks and interfaces are also beneficial. SCSI performs better than ATA, if you can justify the higher costs. While fast CPUs are nice, they aren't critical to good performance.

Because Squid uses a small amount of memory for every cached response, there is a relationship between disk space and memory requirements. As a rule of thumb, you need 32 MB of memory for each GB of disk space. Thus, a system with 512 MB of RAM can support a 16-GB disk cache. Your mileage may vary, of course. Memory requirements depend on factors such as the mean object size, CPU architecture (32- or 64-bit), the number of concurrent users, and particular features that you use.

An often question for lot of people is the following: *"I have a network with X users. What kind of hardware do I need for Squid?"* This kind of question is difficult to answer for a number of reasons. In particular, it is hard to say how much traffic X users will generate. The easiest way is to look at bandwidth usage, and go from there. In this case you can build a system with enough disk space to hold 3-7 days worth of web traffic. For example, if your users consume 1 Mbps (HTTP and FTP traffic only) for 8 hours per day, that is about 3.5 GB per day. So, I would say you want between 10 and 25 GB of disk space for each Mbps of web traffic (Wessels, 2004).

4.3 Setup Configuration and Startup

Squid is normally distributed as source code. This means you will probably need to compile it. The installation process should be relatively painless. The developers put a lot of effort into making sure Squid compiles easily on all the popular operating systems.

You can also find precompiled binaries for some operating systems. Linux users can get Squid in one of the various package formats (e.g., RPM, Debian, etc.). The FreeBSD, NetBSD, and OpenBSD projects offer Squid *ports*. The BSD ports aren't binary distributions but rather a small set of files that know how to download, compile, and install the Squid source. While these precompiled or preconfigured packages may be easier to install, I recommend that you download and compile the source yourself.

While RPMs and precompiled packages may initially save you some time, they also have some drawbacks. Certain features must be enabled or disabled *before* you start compiling Squid. The precompiled package that you install may not have the particular feature you want. Furthermore, Squid's *./configure* script probes your operating system for certain parameters. These parameters may be configured differently on your machine on which Squid was compiled. Finally, if you want to apply a patch to Squid, you'll either have to wait for someone to build a new RPM/package or get the source and do it yourself. The computer that I work is a Redhat-like system (*Fedora Core 5*). So, I will present the setup and basic configuration procedure in this operating system as an RPM package.

4.3.1 Setup

In order to be able to setup Squid in *Fedora Core 5* you must do the following:

I. Login as root

II. Verify the Squid package has been installed. Give the following command:

rpm -qa | grep squid

If you have no results after giving the above command, then you need to install Squid.

III. Make sure that you have an internet connection. Also, you have to install the ***yum*** application in order to be able to download and install squid. Moreover, it is a requirement to have a web server installed in your machine. In my computer, I have an Apache Web Server that run as standalone daemon process (***httpd***). If you are under a proxy server, you need to set and export the proxy as:

*set http_proxy=*http://Proxy_Name:Proxy_Port/

*export http_proxy=*http://Proxy_Name:Proxy_Port/

IV. Then you can give the command:

yum install squid

In the following figure (*Figure 4-2*) is depicted the above operation in a screenshot, as I have tried in my computer. Once it has been verified that the packages are installed, the squid proxy server will need to be configured.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

```
lagosm@darkstar:~
File Edit View Terminal Tabs Help
[root@darkstar ~]# rpm -qa | grep squid
[root@darkstar ~]# set http_proxy="http://139.23.33.27:81/"
[root@darkstar ~]# export http_proxy="http://139.23.33.27:81/"
[root@darkstar ~]# yum install squid
Loading "installonlyn" plugin
Setting up Install Process
Setting up repositories
updates                                                                     [1/5]
livna                                                                       [2/5]
core                                                                        [3/5]
freshrpms                                                                   [4/5]
extras                                                                      [5/5]
Reading repository metadata in from local files
Parsing package install arguments
Resolving Dependencies
--> Populating transaction set with selected packages. Please wait.
---> Downloading header for squid to pack into transaction set.
squid-2.5.STABLE14-2.FC5. 100% |=========================| 137 kB    00:08
---> Package squid.i386 7:2.5.STABLE14-2.FC5 set to be updated
--> Running transaction check

Dependencies Resolved

=============================================================================
 Package                 Arch       Version          Repository        Size
=============================================================================
Installing:
 squid                   i386       7:2.5.STABLE14-2.FC5  updates           1.2
M

Transaction Summary
=============================================================================
Install      1 Package(s)
Update       0 Package(s)
Remove       0 Package(s)
Total download size: 1.2 M
Is this ok [y/N]: y
```

Figure 4-2.Squid setup procedure.

4.3.2 Server Configuration

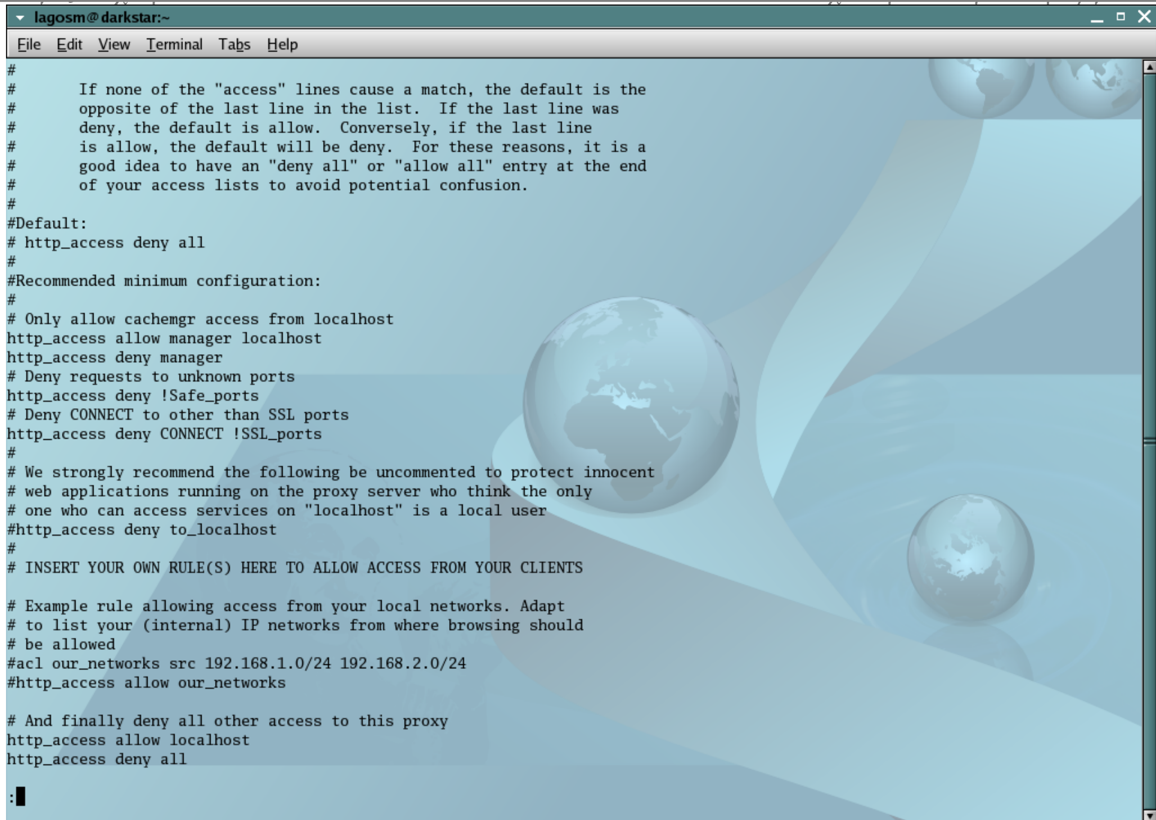
After installing Squid successfully, the next task is to delve into the configuration file. Open a terminal window as root and type the following:

joe /etc/squid/squid.conf

Note that any text editor can be used in place of *joe* (gedit, vi, etc). This is the configuration file for the squid proxy server. It contains a very good documentation about the available configuration options with examples in most cases and it is about 3500 rows. All the *squid.conf* directives have default values.

Squid's configuration file is relatively straightforward. It is similar in style to many other UNIX programs. Each line begins with a configuration directive, followed by some number of values and/or keywords. Squid ignores empty lines and comment lines (beginning with #) when reading the configuration file. In the following figure (*figure 4-3*) you can see some sample configuration lines:

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ




```
#
#       If none of the "access" lines cause a match, the default is the
#       opposite of the last line in the list.  If the last line was
#       deny, the default is allow.  Conversely, if the last line
#       is allow, the default will be deny.  For these reasons, it is a
#       good idea to have an "deny all" or "allow all" entry at the end
#       of your access lists to avoid potential confusion.
#
#Default:
# http_access deny all
#
#Recommended minimum configuration:
#
# Only allow cachemgr access from localhost
http_access allow manager localhost
http_access deny manager
# Deny requests to unknown ports
http_access deny !Safe_ports
# Deny CONNECT to other than SSL ports
http_access deny CONNECT !SSL_ports
#
# We strongly recommend the following be uncommented to protect innocent
# web applications running on the proxy server who think the only
# one who can access services on "localhost" is a local user
#http_access deny to_localhost
#
# INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS

# Example rule allowing access from your local networks. Adapt
# to list your (internal) IP networks from where browsing should
# be allowed
#acl our_networks src 192.168.1.0/24 192.168.2.0/24
#http_access allow our_networks

# And finally deny all other access to this proxy
http_access allow localhost
http_access deny all

:
```

Figure 4-3.Squid.conf sample.

Some directives take a single value. For these, repeating the directive with a different value overwrites the previous value. For example, there is only one ***shutdown_lifetime*** value. The first line in the following example has no effect because the second line overwrites it:

shutdown_lifetime 30 seconds
shutdown_lifetime 1 minute

On the other hand, some directives are actually lists of values. For these, each occurrence of the directive adds a new value to the list. The ***extension_methods*** directive works this way (Wessels, 2004):

extension_methods UNGET
extension_methods UNPUT
extension_methods UNPOST

For these list-based directives, you can also usually put multiple values on the same line:

extension_methods UNGET UNPUT UNPOST

In general, the configuration file directives may appear in any order. However, the order is important when one directive makes reference to something defined by another. Access controls are a good example. An *acl* must be defined before it can be used in an *http_access* rule:

acl all src 0.0.0.0/0.0.0.0
http_access allow all

Many things in *squid.conf* are case-sensitive, such as directive names. You can't write ACL instead of *acl*. Moreover, the default *squid.conf* file contains comments describing each directive, as well as the default values.

4.3.2.1 Most Basic Settings

4.3.2.1.1 User IDs

It is known that UNIX processes and files have user and group ownership attributes. You need to select a user and a group for Squid. This user and group must have read and write access to most of the Squid-related files and directories. It is a good practice creating a dedicated squid user and group. This minimizes the chance that someone can exploit Squid to read other files on the system. If more than one person has administrative authority over Squid, you can add them to the squid group (Wessels, 2004).

UNIX processes inherit their parent process' ownership attributes. That is, if you start Squid as user *lagosm*, Squid also runs as user *lagosm*. If you don't want Squid to run as

lagosm, you need to change your user ID beforehand. This is typically accomplished with the *su* command. For example:

```
su - squid
/usr/local/squid/sbin/squid
```

Unfortunately, running Squid isn't always so simple. In some cases, you may need to start Squid as *root*, depending on your configuration. If you start Squid as root, it will change its effective/real UID/GID to the user specified below. The default is to change to UID to "squid". If you define *cache_effective_user*, but not *cache_effective_group*, Squid sets the GID to the effective user's default group ID (taken from the password file) and supplementary group list from the from groups membership of *cache_effective_user* (Wessels, 2004). For example, only *root* can bind a TCP socket to privileged ports like port 80. The following directive tells Squid which user to become after performing the tasks that require special privileges:

```
cache_effective_user squid
```

The name that you provide must be a valid user (i.e., in the */etc/passwd* file). Furthermore, note that this directive is used only when you start Squid as *root*. Only *root* has the ability to become another user. If you start Squid as *lagosm*, it can't switch to user *squid*.

You might be tempted to just run Squid as *root* without setting *cache_effective_user*. If you try, you will find that Squid refuses to run. This, again, is due to security concerns. If an outsider were somehow able to compromise or exploit Squid, he could gain full access to your system. Although we strive to make Squid secure and bug-free, this requirement provides some extra insurance, just in case (Fox, 2004).

If you start Squid as *root* without setting *cache_effective_user*, Squid uses nobody as the default value. Whatever user ID you choose for Squid, make sure it has read access to the

files installed in *$prefix/etc*, *$prefix/libexec*, and *$prefix/share*. The user ID must also have write access to the log files and cache directory (Wessels, 2004).

4.3.2.1.2 Port Numbers

The ***http_port*** directive is used to specify the socket addresses where Squid will listen for HTTP client requests (Fox, 2004). The best form for this tag (especially for security reasons) is the following:

http_port IP: port

The default port is 3128. You can instruct Squid to listen on multiple ports with additional ***http_port*** lines. This is often useful if you must support groups of clients that have been configured differently. For example, the browsers from one department may be sending requests to port 3128, while another department uses port 8080. Simply list both port numbers. Moreover, if you want Squid to listen to only to one interface, you can use the above described format. If we omit the IP declaration, then Squid will listen to every address in the server.

The *icp_port* directive specifies the port number that Squid listens for the communication with other proxy servers. ICP is a protocol used for communication among Squid caches. ICP is primarily used within a cache hierarchy to locate specific objects in sibling caches. The default value is 3130.

4.3.2.1.3 Access Controls

Squid's default configuration file denies every client request. You must place additional access control rules in *squid.conf* before anyone can use the proxy. The simplest approach is to define an Access Control List that corresponds to your user's IP addresses and an access rule that tells Squid to allow HTTP requests from those addresses. Squid has many different Access Control Lists types. The *src* type matches client IP addresses, and the *http_access* rules are checked for client HTTP requests. The format of this tag is:

acl aclname acltype parameters

For example, we have the following:

acl allpc src 0.0.0.0/0
acl mygroup src 10.0.0.0/8
http_access allow mygroup
http_access deny all

The first line declares that there exists a group of computers named "allpc". All the IP addresses are within this group. The second line declares that there exists a group of computers named "mygroup". In this category reside all that are in the subnet 10.0.0.0/8. In addition, the third line defines that the access is allowed to all that belong to mygroup and the last one deny the access to whichever pc. *If none of the "access" lines cause a match, the default is the opposite of the last line in the list.* If the last line was denied, as in the above example, then the default is allowed. Conversely, if the last line is allowed, the default will be denied. For these reasons, it is a good idea to have a "deny all" or "allow all" entry at the end of the access lists to avoid potential confusion. Also, do not forget that rules in the access list are read from top to bottom. The first rule matched will be used. Other rules won't be applied.

In *figure 4-4* that follows, we can see the recommended minimum configuration in the *squid.conf* file:

```
lagosm@darkstar:~
File  Edit  View  Terminal  Tabs  Help
#acl macaddress arp 09:00:2b:23:45:67
#acl myexample dst_as 1241
#acl password proxy_auth REQUIRED
#acl fileupload req_mime_type -i ^multipart/form-data$
#acl javascript rep_mime_type -i ^application/x-javascript$
#
#Recommended minimum configuration:
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl to_localhost dst 127.0.0.0/8
acl SSL_ports port 443 563
acl Safe_ports port 80          # http
acl Safe_ports port 21          # ftp
acl Safe_ports port 443 563     # https, snews
acl Safe_ports port 70          # gopher
acl Safe_ports port 210         # wais
acl Safe_ports port 1025-65535  # unregistered ports
acl Safe_ports port 280         # http-mgmt
acl Safe_ports port 488         # gss-http
acl Safe_ports port 591         # filemaker
acl Safe_ports port 777         # multiling http
acl CONNECT method CONNECT
:
```

Figure 4-4.Squid.conf ACL recommended configuration.

When you edit *squid.conf* for the first time, look for this comment:

 # INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS

Insert your new rules below this comment, and before the *http_access deny all* line (Lilliam, 2005).

4.3.2.1.4 Other Parameters

Some other basic parameters are the following:

cache_dir:

This parameter specifies especially the kind of storage system to use and the location where the cache swap files will be stored of the proxy cache. As an example you can see the following directive, which is the default in the squid configuration file (Wessels, 2004):

cache_dir aufs /var/spool/squid 1000 16 256

where *aufs* is the type of cache (uses the same storage format as the known "*ufs*", but utilizing POSIX-threads to avoid blocking the main Squid process on disk-I/O – that is the best storage scheme for Linux and Solaris), */var/spool/squid* is the top-level directory that the cache resides. It is important to say that it is a good practice to reside in a different disk or in a separate partition of the same disk. 1000 is the size of cache in Megabytes, and the last two parameters specify the first and second level of subdirectories which will be created under the top-level directory.

chroot:

This directive is a UNIX feature that gives a process a new root filesystem directory. It provides an extra level of security in the event that Squid is compromised. If an attacker somehow gains access to the operating system through Squid, she can only access files under the *chroot* filesystem. The other system files, outside of the *chroot* tree, remain inaccessible (Fox, 2004). The easiest way to run Squid in a chroot environment is by specifying the new root directory in the *squid.conf* file with the following directive:

chroot /new/root/directory

But it's a little bit tricky because you must replicate a number of files underneath the new root directory. For example, if the default configuration file is normally */etc/squid/squid.conf,* and you use the chroot directive, the file must be located at */new/root/directory/etc/squid/squid.conf*.

ftp_user (squid@server.com):

This directive can be used if you want the anonymous login password to be more informative. It is the e-mail address that Squid gives to the ftp servers that are connected with anonymous login.

shutdown_lifetime:

When you shut down the Squid process, some user requests will still be active. This directive specifies how long to wait until all client requests are complete. Squid finally exits when all client connections have been closed or when this timeout occurs (Fox, 2004).

cache_mgr (root):

Using this tag, we can specify the email address of the local cache manager who will receive mail, if the cache dies. The default is "root." In case squid dies, the mail will be sent to the given mail id.

forwarded_for (off):

If set, Squid will include your system's IP address or name in the HTTP requests it forwards. By default it looks like this:

X−Forwarded−For: 192.1.2.3

It is a good practice to have this directive disabled increased user's privacy. So it will appear as:

X−Forwarded−For: unknown

visible_hostname:

In most cases, you won't need to worry about the *visible_hostname* directive. However, you'll need to set it if Squid can't figure out the hostname of the machine on which it is running, such in my case. When I try to start Squid and verify that the *squid.conf* file makes sense, I take the following error:

"FATAL: Could not determine fully qualified hostname. Please set 'visible_hostname'"

So, after this, I edit the squid.conf by adding the directive:

visible_hostname darkstar

4.3.3 Startup

Now that the Squid is installed, and maybe even configured, its time to start running the Squid. First of all, I try to verify that the *squid.conf* file makes sense. I just run the following command:

squid -k parse

but a FATAL error about visible hostname is returned. I fix this problem by following the procedure that I have already described in the previous section (*4.3.2.1.4*), and run the above command again, with no output. So, the configuration file is valid.

Moreover, before running Squid for the first time, and whenever you add a new *cache_dir*, I try to initialize the cache directories with the following command:

squid -z

After the initialization of all swap directories, I try to run Squid in a terminal window with logging to stderr. This way, you can easily spot any errors or problems and make sure that Squid successfully starts.

squid -N -d1

The *-N* option keeps Squid in the foreground and the *-d1* option display level 1 debugging on *stderr*. After running the above command, Squid returns the following error during performing DNS tests:

FATAL: ipcache_init: DNS name lookup tests failed

Squid, makes a few DNS queries before starting. This ensures that DNS servers are reachable and functioning properly. Because of testing purposes, I try to disable the DNS testing by giving the following command (add option *-D* to the above one):

squid -D -N -d1

The output of the above command is depicted in the following figure (*figure 4-5*). Once the "*Ready to serve requests.*" message appears, then Squid is able to serve HTTP requests. I test it with the "*squidclient*" and I take the wanted results (*squidclient* http://www.squid-cache.org/). Now, you can interrupt the Squid process (*Ctrl-C*) and run Squid as a daemon.

```
lagosm@darkstar:~
File  Edit  View  Terminal  Tabs  Help
2006/10/02 00:10:26| Starting Squid Cache version 2.5.STABLE14 for i686-redhat-linux-gnu...
2006/10/02 00:10:26| Process ID 2883
2006/10/02 00:10:26| With 1024 file descriptors available
2006/10/02 00:10:26| DNS Socket created at 0.0.0.0, port 32768, FD 4
2006/10/02 00:10:26| Adding nameserver 141.29.38.62 from /etc/resolv.conf
2006/10/02 00:10:26| Adding nameserver 141.29.40.41 from /etc/resolv.conf
2006/10/02 00:10:26| User-Agent logging is disabled.
2006/10/02 00:10:26| Referer logging is disabled.
2006/10/02 00:10:26| Unlinkd pipe opened on FD 9
2006/10/02 00:10:26| Swap maxSize 102400 KB, estimated 7876 objects
2006/10/02 00:10:26| Target number of buckets: 393
2006/10/02 00:10:26| Using 8192 Store buckets
2006/10/02 00:10:26| Max Mem  size: 8192 KB
2006/10/02 00:10:26| Max Swap size: 102400 KB
2006/10/02 00:10:26| Local cache digest enabled; rebuild/rewrite every 3600/3600 sec
2006/10/02 00:10:26| Rebuilding storage in /var/spool/squid (CLEAN)
2006/10/02 00:10:26| Using Least Load store dir selection
2006/10/02 00:10:26| Set Current Directory to /var/spool/squid
2006/10/02 00:10:26| Loaded Icons.
2006/10/02 00:10:26| Accepting HTTP connections at 0.0.0.0, port 3128, FD 11.
2006/10/02 00:10:26| Accepting ICP messages at 0.0.0.0, port 3130, FD 12.
2006/10/02 00:10:26| WCCP Disabled.
2006/10/02 00:10:26| Ready to serve requests.
2006/10/02 00:10:26| Done reading /var/spool/squid swaplog (0 entries)
2006/10/02 00:10:26| Finished rebuilding storage from disk.
2006/10/02 00:10:26|         0 Entries scanned
2006/10/02 00:10:26|         0 Invalid entries.
2006/10/02 00:10:26|         0 With invalid flags.
2006/10/02 00:10:26|         0 Objects loaded.
2006/10/02 00:10:26|         0 Objects expired.
2006/10/02 00:10:26|         0 Objects cancelled.
2006/10/02 00:10:26|         0 Duplicate URLs purged.
2006/10/02 00:10:26|         0 Swapfile clashes avoided.
2006/10/02 00:10:26|   Took 0.6 seconds (   0.0 objects/sec).
2006/10/02 00:10:26| Beginning Validation Procedure
2006/10/02 00:10:26|   Completed Validation Procedure
2006/10/02 00:10:26|   Validated 0 Entries
2006/10/02 00:10:26|   store_swap_size = 0k
2006/10/02 00:10:27| storeLateRelease: released 0 objects
```

Figure 4-5.Squid startup process.

To start Squid as a daemon in a Redhat-like system (see *figure 4-6* below), you can just give the command:

/sbin/service squid start

If you want to run Squid on boot, you could give the following command:

chkconfig - - level 2345 squid on

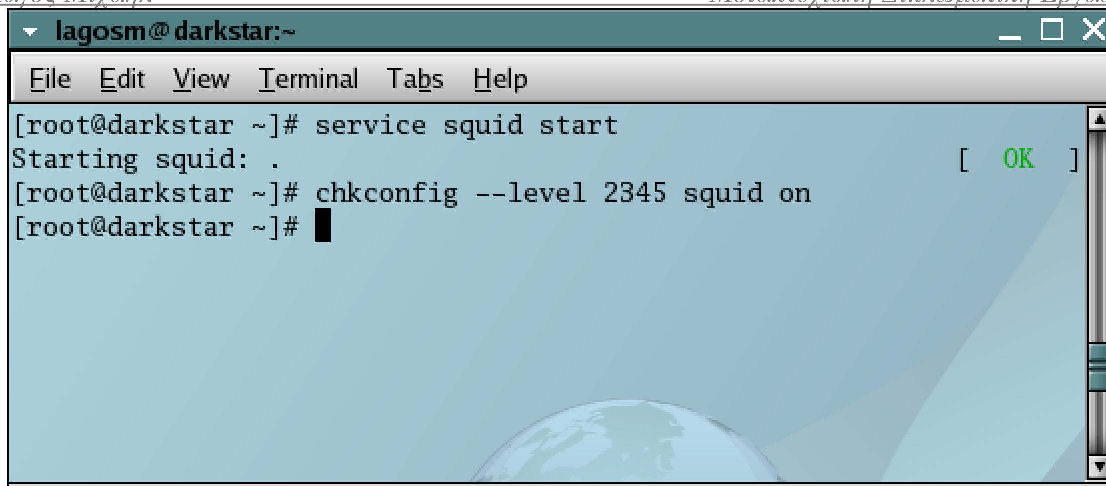


Figure 4-6.Start Squid as a daemon and enable on boot.

Correspondingly, you can stop *Squid* and disable it by running on boot if you replace in the above two commands the "start" with "stop" word and the "on" with the "off" word.

In addition, every time you make a change in the *squid.conf* file, in order to have the new settings take effect, you can either shutdown and restart Squid, or you can reconfigure Squid while it is running by giving the following command:

squid -k reconfigure

4.4 Log Files

Log files are the primary sources of persistent information about Squid's operation. In other words, they provide a record of what Squid has been doing. This includes URIs requested by users, objects that have been saved to disk, and various warnings and errors. When Squid appears to be malfunctioning, you will want to check the log files first.

Depending on the configuration, Squid maintains, at most, seven log files. The three primary files are: *cache.log*, *access.log*, and *store.log*. Two optional log files, *useragent.log* and *referer.log*, are similar to access.log but contain additional information. In addition, there exist two more files: the *swap.state* and *netdb_state*. These are databases, used by Squid when it restarts. Note that the filenames, such as access.log, are the default values. You can change most of the log file names with various squid.conf directives (Wessels, 2004).

Each of the above log files is described following:

cache.log:

This log file contains human-oriented, informational messages about Squid's operation. The filename is defined by the *cache_log* directive. Under normal conditions, the file grows by about 10-100 KB per day.

access.log:

This log file contains an entry for every HTTP and (optionally) ICP transaction made by Squid's clients. The filename is defined by the *cache_access_log* directive. It grows at a rate of 100-200 bytes per transaction.

store.log:

This log file contains low-level information about objects that enter and leave the cache. The filename is defined by the *cache_store_log* directive. It grows at a rate of about 150 bytes per transaction.

referer.log:

This optional log file contains HTTP referer headers for each client request. You must enable referer logging with the *—enable-referer-log* option when running *./configure*. The filename is defined by the *referer_log* directive. It grows at a rate of about 80 bytes per transaction (Wessels, 2004).

useragent.log:

This optional log file contains HTTP User-Agent headers for each client request. You must enable user-agent logging with the *—enable-useragent-log* option when running *./configure*. The filename is defined by the *useragent_log* directive. It grows at a rate of about 75 bytes per transaction (Wessels, 2004).

swap.state:

These files contain internal metadata about the objects stored on disk. Squid uses them to reconstruct the cache upon startup. By default, they are located in the *cache_dir* directories. However, you can change the location with the c*ache_swap_log* directive. They grow at a rate of 100 bytes per cache miss (Wessels, 2004).

netdb_state:

This file holds the contents of the Network Measurement Database. It is always located in the first *cache_dir* directory. Its size is determined by the *netdb_high* value (Wessels, 2004).

If Squid receives an error while writing a log file, it doesn't silently continue. Instead, it exits with a fatal error message to get your attention.

4.3.1 access.log

Squid saves key information about HTTP transactions in *access.log*. This file is line-based, such that each line corresponds to one client request. Squid records the *client IP address* (or hostname), *requested URI*, *response size*, and other information. Squid records all HTTP accesses in *access.log*, except for those that disconnect before sending any data. Squid also records all ICP (but not HTCP) transactions unless you disable them with the *log_icp_queries* directive. *Section 4.3.2* describes the other *squid.conf* directives that affect the *access log*.

The default *access.log* format contains 10 fields. Here are some examples, with long lines split and indented:

1146547018.954 448 192.168.100.112 TCP_REFRESH_HIT/304 393 GET http://www.opap.gr/Style.css - DIRECT/212.205.38.102 -
1146547019.382 382 192.168.100.112 TCP_REFRESH_HIT/304 392 GET http://www.opap.gr/Images/back.gif - DIRECT/212.205.38.102 -
1146547019.543 161 192.168.100.112 TCP_REFRESH_HIT/304 392 GET http://www.opap.gr/Images/tzoker2.gif - DIRECT/212.205.38.102 -
1146547019.575 504 192.168.100.112 TCP_MISS/304 393 GET http://www.opap.gr/Images/logo.jpg - DIRECT/212.205.38.102 -
1146547054.832 1335 192.168.100.112 TCP_MISS/304 322 GET http://www.alphahentai.com/ - DIRECT/70.85.72.250 -
1146547019.575 204 192.168.100.112 TCP_MISS/304 393 GET http://www.opap.gr/Images/logo.jpg - DIRECT/212.205.38.102 -
1146547019.575 204 192.168.100.112 TCP_MISS/304 393 GET http://www.test.gr/Images/logo.jpg - DIRECT/212.205.38.102 -
1146547063.549 540 192.168.100.112 TCP_MISS/304 322 GET http://www.alphahentai.com/titanimebnr03_468x80.jpg - DIRECT/70.85.72.250 -
1246547064.083 1690 192.168.100.112 TCP_MISS/200 64478 GET http://www.alphahentai.com/main.htm - DIRECT/70.85.72.250 text/html
1146547064.205 1015 192.168.100.112 TCP_MISS/200 7519 GET http://www.alphahentai.com/d3.jpg - DIRECT/70.85.72.250 image/jpeg
1146547426.753 245 192.168.100.104 TCP_MISS/302 425 GET http://toolbar.google.com/version3? - DIRECT/216.239.59.104 text/html
1146547427.083 330 192.168.100.104 TCP_MISS/200 296 GET http://www.google.com/tools/toolbar/service/update? - DIRECT/216.239.59.104 text/plain
1246547427.384 726 192.168.100.104 TCP_MISS/200 24517 GET http://www.whitepages.gr/gr/index.jsp - DIRECT/195.170.6.246 text/html
1146547427.964 202 192.168.100.104 TCP_MISS/304 166 GET http://www.whitepages.gr/css/site.css - DIRECT/195.170.6.246 -
1146547428.085 319 192.168.100.104 TCP_MISS/304 166 GET http://www.whitepages.gr/scripts/dw.js - DIRECT/195.170.6.246 -

1146547428.319 233 192.168.100.104 TCP_MISS/304 166 GET http://www.whitepages.gr/scripts/utils_gr.js - DIRECT/195.170.6.246 -
1146547430.920 1519 192.168.100.112 TCP_MISS/200 80398 GET http://www.alphahentai.com/dojins1/fgm_v135/024.jpg - DIRECT/70.85.72.250 image/jpeg
1146547443.070 1915 192.168.100.112 TCP_MISS/200 99092 GET http://www.alphahentai.com/dojins1/fgm_v135/027.jpg - DIRECT/70.85.72.250 image/jpeg
1246547482.536 2159 192.168.100.112 TCP_MISS/200 92196 GET http://www.in.gr/dojins1/fgm_v135/037.jpg - DIRECT/70.85.72.250 image/jpeg

I keep one sample entry in order to describe all the fields. The entry is the following:

1146547026.226 3102 192.168.100.112 TCP_MISS/200 29656 GET http://www.opap.gr/ - DIRECT/212.205.38.102 text/html

In the above line each field represents some piece of information that may be of interest to an administrator. They are as follows:

- ✓ *System time* in standard UNIX format. The time in seconds since 1970. There are many tools to convert this to human readable time.
- ✓ *Duration* or the Elapsed Time in milliseconds the transaction required.
- ✓ *Client Address* or the IP address of the requesting browser. Some configurations may lead to a masked entry here, so that this field is not specific to one IP, but instead reports a whole network IP.
- ✓ *Result Codes* provides two entries separated by a slash. The first position is one of several *result codes*, which provide information about how the request was resolved or wasn't resolved if there was a problem. The second field contains the *status code*, which comes from the subset of the standard HTTP status codes.
- ✓ *Bytes* is the size of the data delivered to the client in bytes. Headers and object data are counted towards this total. Failed requests will deliver an error page, the size of which will also be counted.
- ✓ *Request method* is the HTTP request method used to obtain an object. The most common method is, of course, GET, which is the standard method web browsers use to fetch objects.
- ✓ *URL* is the complete Uniform Resource Locator requested by the client.

- ✓ RFC931 is the indent lookup information for the requesting client, if indent lookups are enabled in your Squid. Because of the performance impact, indent lookups are not used by default, in which case this field will always contain "-".
- ✓ *Hierarchy code* consists of three items. The first is simply a prefix of TIMEOUT_ if all ICP requests timeout. The second (first if there is not TIMEOUT_ prepended) is the code that explains how the request was handled. This portion will be one of several hierarchy codes. This result is informative regardless of whether your cache is part of a cache hierarchy, and will explain how the request was served. The final portion of this field contains the name or IP of the host from which the object was retrieved. This could be the origin server, a parent, or any other peer.
- ✓ *Type* is simply the type of object that was requested. This will usually be a recognizable MIME type, but some objects have no type or are listed as ":".

From the above 10 fields, the most important for the scope of this project are the following: *system time*, *duration*, *client address*, *bytes* and *URL*.

4.3.2 Configuration Directives that Affect access.log

Following are some of the most important (for the purposes of this project) configuration file directives that affect the *access.log* in one way or another:

log_icp_queries:

This directive that is enabled by default causes Squid to log all ICP queries. If you are running a busy parent cache, this may make your *access.log* files huge. In order to save space, I disable this directive:

log_icp_queries off

emulate_httpd_log:

The *access.log* file has two formats: common and native. The common format is the same as most HTTP servers (e.g., Apache) use. It contains less information than Squid's native format. However, you might want to use the common log-file format if you use Squid as a surrogate. The common format may also be useful if you have log-file analysis tools that know how to parse it. In this case, because I am going to implement a log-file analysis tool, I use the following directive to enable the common format:

emulate_httpd_log on

log_fqdn:

By default, Squid puts client IP addresses in the *access.log*. You can record hostnames, when available, by enabling this directive: *log_fqdn on.*
This causes Squid to make reverse DNS lookups for the client's address when it receives a request. If an answer is available by the time the request is complete, Squid places it in the third field. In this project, I need only IP addresses, so this directive must be as:

log_fqdn off

strip_query_terms:

This directive is a privacy feature. It removes query terms from URIs before logging them. If your log files somehow fall into the wrong hands, they won't be able to find any usernames and passwords. When this directive is enabled, all characters after a question mark (?) are removed. For example, a URI like this:
http://agent.adman.gr/banner?MT=www.lagos.com&arch3&prov=&utf8
is logged like this:
http://agent.adman.gr/banner?
I have this directive enabled:

strip_query_terms on

uri_whitespace:

The related RFCs state that URIs must not contain whitespace, but in reality it happens all too often. The *uri_whitespace* directive dictates how Squid should handle such cases. The allowed settings are: *strip* (default), *deny*, *allow*, *encode*, and *chop*. Of these, strip, encode, and chop ensure that the URI field doesn't contain any whitespace (thus adding more fields to access.log).

The allow setting allows the request to pass through Squid unmodified. It is likely to cause trouble for redirectors and log file parsers. The deny setting, on the other hand, causes Squid to deny the request. The user receives an error message, but the request is still written to access.log with the whitespace characters.

If you set it to encode, Squid changes the whitespace characters to their RFC 1738 equivalents. This is probably what the user-agent should have done in the first place. The chop setting causes Squid to cut off the URI at the first whitespace character.

The default setting is strip, which makes Squid remove the whitespace characters from the URI. It ensures that your log-file parsers and redirectors will be happy, but it might break certain things, such as improperly encoded search engine queries. So, I have this directive enabled:

uri_whitespace strip

From the above it is obvious that the *access.log* file contains a wealth of information, much more than you can see by just browsing through it. In order to get the big picture view, a log-file analysis package is required. That is what exactly I implement. A fast and free log file analysis package. It produces highly detailed, easily configurable usage reports in text and HTML format. In the section that follows (*Section 5*), we can see an analysis of the real implementation of this tool (coding issues). At this point, it is important to note that the implemented log analyzer does not parse only the squid proxy *access.log* file, but with small modifications it can also parse the *iptables log file* (see *Section 6*) the *mail.log* file and more log files. It's a generic log analyzer.

5. Proxy Log Analyzer Implementation

As it is referred in the previous section, a part of the implemented log file parser is the squid traffic analyzer, designed to allow per-user scrutiny and analysis of squid log files. The program allows a non-technical user to extract information about web usage patterns, the type of information downloaded, the sites visited by users, the graphics downloaded, and the amount of information (per-byte) accessed. The program runs in a Linux-based system via a terminal.

As it has already referred in Section 1 (Introduction), the best way to deal with the above is a combination of *linked lists* and *hash tables*.

Linked lists consist of a number of elements grouped, or *linked*, together in a specific order. They are useful in maintaining collections of data, similar to the way that arrays are often used. However, linked lists offer important advantages over arrays in many cases. Specifically, linked lists are considerably more efficient in performing insertions and deletions. Linked lists also make use of dynamically allocated storage, which is storage allocated at runtime. Since in many applications the size of the data is not known at compile time, this can be a nice attribute as well (Parlante, 2001). Moreover, there are two categories of linked lists. The first one named *Singly-linked lists*, are composed of individual elements, each linked by a single pointer. Each element consists of two parts: a data member and a pointer, called the *next* pointer. The second category of linked lists named *Doubly-linked lists* is composed of elements linked by two pointers. Each element of a doubly-linked list consists of three parts: in addition to the data and the next pointer, each element includes a pointer to the previous element, called the *previous* pointer. A doubly-linked list is formed by composing a number of elements so that the *next* pointer of each element points to the element that follows it and the *previous* pointer points to the element preceding it (Loudon, 1999).

On the other hand, ***hash tables*** support one of the most efficient types of searching: ***hashing***. Fundamentally, a hash table consists of an array in which data is accessed via a special index called a *key*. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a *hash function* (Unknown, 2005). A hash function accepts a key and returns its *hash coding*, or *hash value*. Keys vary in type, but hash coding is always integer. Each list forms a bucket in which we place all elements hashing to a specific position in the array (see *Figure 5-1*). To insert an element, we first pass its key to a hash function in a process called hashing the key. This tells us in which bucket the element belongs. We then insert the element at the head of the appropriate list. To look up or remove an element, we hash its key again to find its bucket, and then traverse the appropriate list until we find the element we are looking for. Because each bucket is a linked list, a chained hash table is not limited to a fixed number of elements. However, performance degrades if the table becomes too full (Loudon, 1999).

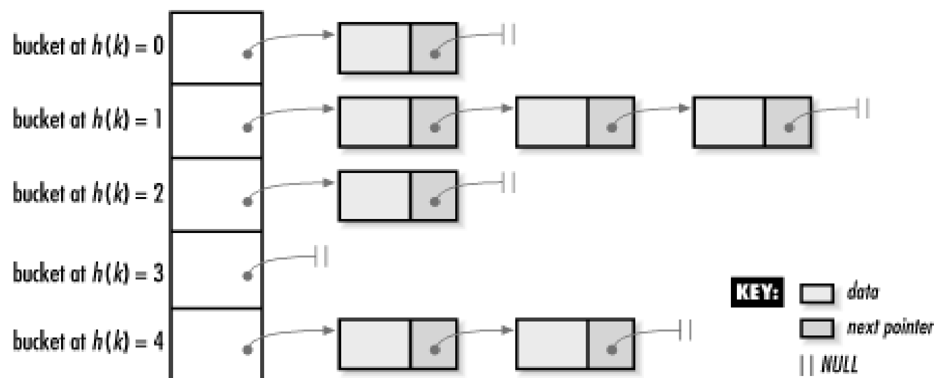


Figure 5-1.A chained hash table with five buckets containing a total of seven elements.

When two keys hash to the same position in a hash table, they collide. Chained hash tables have a simple solution for resolving collisions: elements are simply placed in the bucket where the collision occurs. One problem with this, however, is that if an excessive

number of collisions occur at a specific position, a bucket becomes longer and longer. Thus, accessing its elements takes more and more time (Loudon, 1999).

Ideally, we would like all buckets to grow at the same rate so that they remain nearly the same size and as small as possible. In other words, the goal is to distribute elements about the table in as uniform and random a manner as possible. This theoretically perfect situation is known as uniform hashing; however, in practice it usually can only be approximated (Jenkins). Following, in *figure 5-2* we can see a sample when two keys collide (k_2 and k_5) because they map to the same slot, and in *figure 5-3* is presented the collision resolution by chaining. Each hash-table slot *T[j]* contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$ (Loudon, 1999).

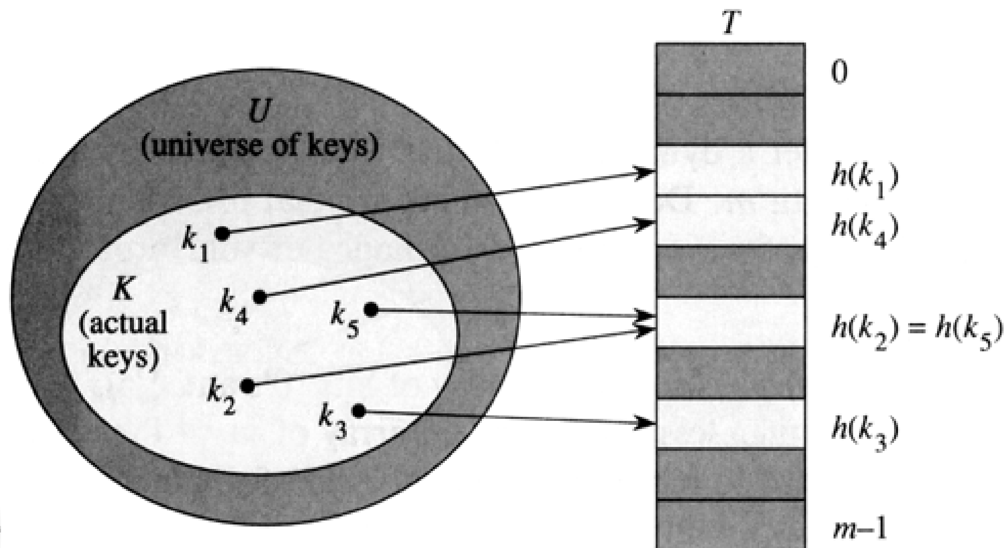


Figure 5-2.Using a hash function h to map keys to hash table slots. Keys k_2 and k_5 map to the same slot, so they collide.

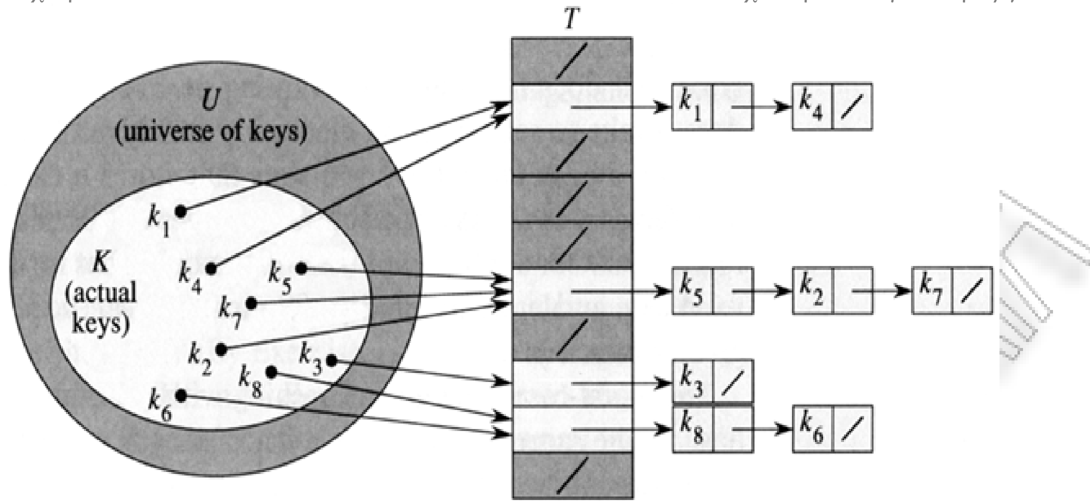


Figure 5-3.Collision resolution by chaining.

Even assuming uniform hashing, performance degrades significantly if we make the number of buckets in the table small relative to the number of elements we plan to insert. In this situation, all of the buckets become longer and longer. Thus, it is important to pay close attention to a hash table's load factor (Song, 2005). The load factor of a hash table is defined as:

$$a = n/ m$$

where n is the number of elements in the table and m is the number of positions into which elements may be hashed. The load factor of a chained hash table indicates the maximum number of elements we can expect to encounter in a bucket, assuming uniform hashing (Devi, 2004).

For example, in a chained hash table with m = 1699 buckets and a total of n = 3198 elements, the load factor of the table is a = 3198/1699 = 2. Therefore, in this case, we can expect to encounter no more than two elements while searching any one bucket. When the load factor of a table drops below 1, each position will probably contain no more than one element. Of course, since uniform hashing is only approximated, in actuality we end up encountering somewhat more or less than what the load factor suggests. How close we

come to uniform hashing ultimately depends on how well we select our hash function (Loudon, 1999).

The goal of a good hash function is to approximate uniform hashing, that is, to spread elements about a hash table in as uniform and random a manner as possible. A hash function h is a function we define to map a key k to some position x in a hash table. x is called the hash coding of k (Devi, 2004). Formally stated:

$$h(k) = x$$

Generally, most hashing methods assume k to be an integer so that it maybe easily altered mathematically to make h distribute elements throughout the table more uniformly. When k is not an integer (such in this implementation), we can usually coerce it into one without much difficulty. Precisely how to coerce a set of keys depends a great deal on the characteristics of the keys themselves. Therefore, it is important to gain as much of a qualitative understanding of them in a particular application as we can. In this project, the implemented hash function coerces a key (the key is the IP address, for example 192.168.100.108) into a permuted integer through a series of bit operations. The resulting integer is mapped using the division method. At this point it is important to refer the reason that the actual hash code used for accessing the table is the hash code modulo the table size. This transformation ensures that the hash coding does not position us past the end of the table. So, we are sure that the hash coding does not fall outside of the bounds of our table.

At this point, it is important to refer a problem that I had with the *access.log* file. Squid, writes to a number of log files unless you disable them in *squid.conf*. I thought that there was a need to periodically rotate the log files (more specifically the *access.log* file) to prevent this from consuming too much disk space. Also, it is more useful for an intranet administrator generates more helpful and easy conclusions in a daily basis, for example, log file, instead from a 5 days log file. I think that it is better to have the information as

small as possible. So, I try to make a function that checks the size of the log file. If it exceeds the 2GB then the file is renamed to a new one and a new *access.log* file will be generated from Squid. The implementation of the two functions is presented below:

```
/*********************************************************************/
/* Function   : getLogDirSize                                        */
/* Arguements : path   - the specified path
*/
/*
*/
/* This function is responsible to calculate the size of the        */
/* provisioned path that log files are located.                      */
/*                                                                   */
/*********************************************************************/

int getLogDirSize(char *path) {

        char cmd[BUF_LENGTH];
        char buf[BUF_LENGTH];
        FILE *ptrf;

   snprintf(cmd, sizeof cmd, "/usr/bin/du -ks %s", path);
   if ((ptrf = popen(cmd, "r")) != NULL) {

      while (fgets(buf, sizeof buf, ptrf) != NULL) {
         printf("%s\n", buf);
      }
      pclose(ptrf);
   }
        return (strtol(buf, (char **)NULL, 10));
}


/*********************************************************************/
/* Function  : backupFile                                            */
/* Arguments : srcName - the source file to be renamed               */
/*                                                                   */
/* This function will back up (move) the log file srcName to a file in
*/
/* the same directory with date extension ".YYYYMMDDHHMMSS".         */
/*                                                                   */
/*********************************************************************/
```

```
void backupFile(char *srcName) {

        time_t seconds;  /* an arithmetic type suitable to represent time */
        struct tm *tm;
        struct timeval tv;
        char destFile[BUF_LENGTH];

        /* get current time of day */
        gettimeofday(&tv, NULL);

        seconds = tv.tv_sec;

        /* break this into yy/mm/dd/hh/mm/ss */
        tm = localtime(&seconds);

        /* build filename string in format 'Filename.YYYYMMDDhhmmss' */
        snprintf(destFile, sizeof destFile, "%s.%04d%02d%02d%02d%02d%02d",
                        srcName,
                        tm->tm_year + 1900,
                        tm->tm_mon + 1,
                        tm->tm_mday,
                        tm->tm_hour,
                        tm->tm_min,
                        tm->tm_sec);

        printf("Dest file is: %s\n", destFile);

        printf("Created backup %s to %s", srcName, destFile);

        /* rename the current file */
        if(rename(srcName, destFile) != 0) {

                /* assume user has deleted the file so continue */
                return;
        }

        /* make target file read only */
        chmod(destFile, S_IRUSR);
}
```

As the project goes on, the above practice seems to be inappropriate for this purpose. After all, I find a very useful function of Squid. Squid places a lot of importance on log files and exits with an error message when it can't write to them.

Squid keeps only the last *logfile_rotate* versions of each log file. The older versions are simply removed during the renaming process. If you want to keep more copies, you need to increase the *logfile_rotate* limit or write some custom scripts that move the log files to a different location.

Because I need to keep the *access.log* file manageable and to keep some copies of this, I decide to rotate this in a daily basis. Squid has a built-in feature for rotating log files. You can invoke it with the following command in a *cron* job:

squid -k rotate

You then tell Squid how many old copies for each file to keep with the *logfile_rotate* directive. For example, if you set it to 7, you will have eight versions of each log file: the current file and seven old ones. Old log files are renamed with numeric extensions. For example, when you execute a rotation, Squid renames log.6 to log.7, then log.5 to log.6, and so on. The current log becomes log.0, and Squid creates a new, empty file named *log*. Each time the *squid -k rotate* command is executed, Squid rotates among others the *access.log* file.

Squid does not rotate the log files automatically. The best way to make it happen is with a daily cron job. For example:

*0 4 * * * /usr/local/squid/sbin/squid -k rotate*

This crontab entry rotates the logs every 24 hours, at 4 A.M. :

Nevertheless, I decide to write my own script to manage the *access.log* file. I just simply set the *logfile_rotate* directive to 0. Then, when I run *squid -k rotate*, Squid simply closes the current log files and opens new ones. This is very useful when the operating system

allows you to rename files opened by another process. The following shell script (Newham, 1998) illustrates the idea:

```
// Start of script
#!/bin/sh

set -e

yesterday_secs=`perl -e 'print time -43200'`

yesterday_date=`date -r $yesterday_secs +%Y%m%d`

archive_location=`/bin/mkdir -p /root/thorax/archive`

cd /var/log/squid

# rename the current log file without interrupting the logging process
mv access.log access.log.$yesterday_date

# tell Squid to close the current logs and open new ones
/usr/sbin/squid -k rotate

# give Squid some time to finish writing swap.state files
sleep 60

mv access.log.$yesterday_date $archive_location

gzip -9 $archive_location/access.log.$yesterday_date
// End of script
```

Furthermore, an important problem that is faced during the development is the following. When the incoming client address is the same with the existing in the memory in a specific slot, but the URL is the same with the one that has already inserted before the interception of other URL's, then I need to have a logic in the code that the list is traversed from the start in order to find a matching URL. If a matching URL was not found, then I need to create a new node that will contain the new URL. The above logic is described in the pseudo code in the following section (*Section 5.1*). In addition, a big problem is the case that we have collisions.

Moreover, another important issue is the response time of the log analyzer. Initially, after the first set of tests, the response time wasn't short enough for a professional tool like this. More specifically, for an *access.log* file that was 6.5MB (47173 requests) the response time was about 16 seconds. Because this program will be encapsulated into a product, the parsing response time must be very short. After a lot of modifications into the code (memory management, etc), the response time decreased and decreased. Finally, the response time for the same *access.log* file is decreased from 16 seconds to the following that is presented in the *figure 5-4*:

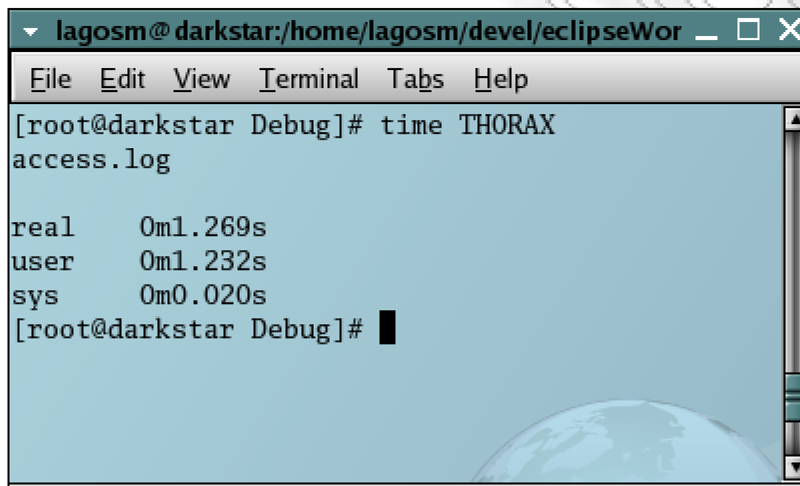


Figure 5-4.Response time of the proxy log analyzer.

Moreover, I tried to test the application with a larger log file, in order to observe the response time. So, with a size of 24MB (184353 requests) *access.log* file, the response time was approximately 3.5 seconds in the same working machine.

5.1 Pseudocode

Before the description of the pseudocode, I try to make a scheme of how to develop the parsing of the proxy log file. I mean that I make an initial guide of how to implement the algorithm. The logic behind the actual implementation is depicted in *figure 5-5*:

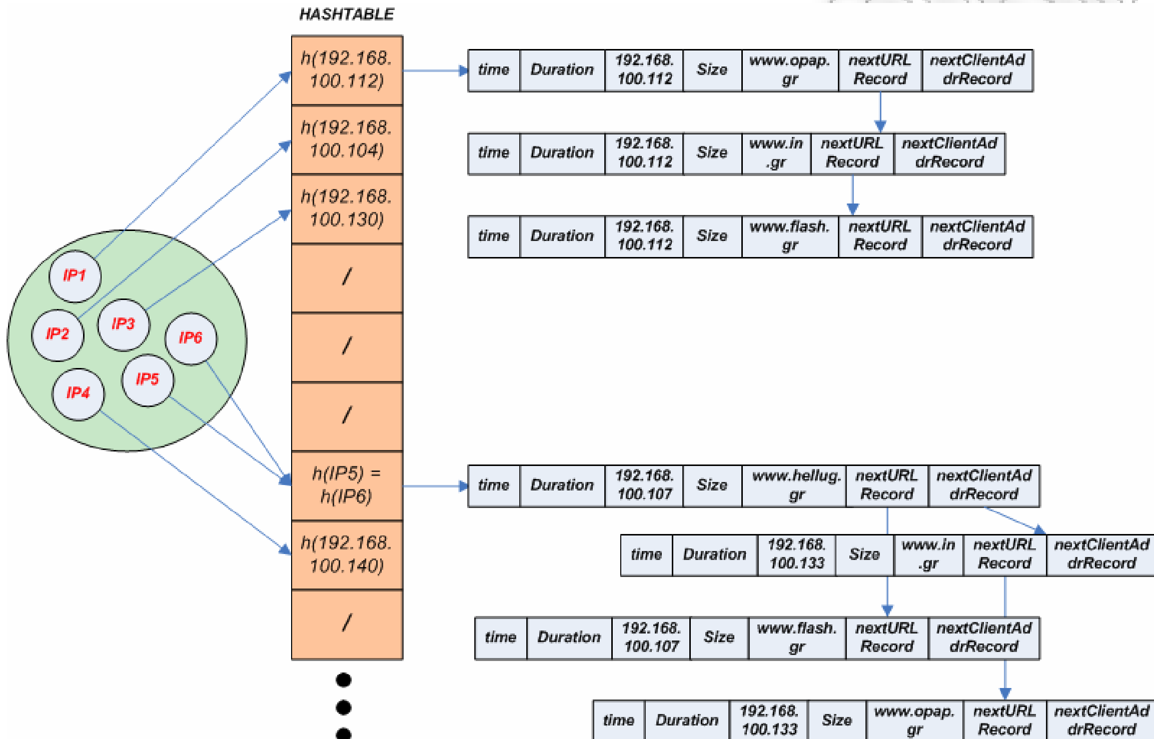


Figure 4-5.An example of the logic behind the algorithm.

The description of the above figure is provided in *Section 5*, in a more general manner. Following, is presented the next step that is the pseudocode.

############### Start Pseudocode ###############

Create a pseudorandom hash function that coerces the client IP address (the key) into a permuted integer through a series of bit operations.

Read the *thorax.conf* file that contains the specified paths.

Open the directory that contains the *access.log* file, check the permissions and read each entry.

If (the log file is a file and the suffix is "*.log*" that has suffix length 4) then open the *access.log* file and read the next input line from the *access.log* file into a character array buffer.

Define a structure for linked list elements and pointers to this structure that handle the ideal case (no collisions) and the case that we have collisions:

```
typedef struct sListElement {
        char unixTime[buffer_size];
        int duration;
        char clientAddr[buffer_size];
        int size;
        char URL[buffer_size];

        //Vertical expansion of list, if we have same ip and different URL
        struct sListElement* nextURLRecord;

        //Horizontal expansion of list in order to avoid collisions
        struct sListElement* nextClientAddrRecord;
} listElement;

listElement* startClientAddr[HASH_TABLE_SIZE];
listElement* nextClientAddr[HASH_TABLE_SIZE];
listElement* startURL[HASH_TABLE_SIZE];
listElement* nextURL[HASH_TABLE_SIZE];
listElement* hashTable;
```

Get each record in the *access.log* file and separate it into tokens. Then set the *key*, the *time* in human readable format, the *elapsed time*, the *IP*, the *bytes* and the *URL* into the parsing algorithm and keep this information in memory.

In order to set the data into memory, create the following algorithm:

```
If (nextURL[slot] == NULL) {
```

```
        //the defined slot in the hashtable is empty
        Create and fill the structure for the first time in the defined slot;
} else if (nextURL[slot] != NULL) {
        //the defined slot has already filled
        //Check the IP address for collisions
        if (clientsAddress is the same with the existing in the slot) {
                //Check the URL
                if (URL is the same as the existing) {
                        Add duration and size;
                } else {
                        initialize the nextURL[slot] to point to the same location as
                startURL[slot];
                        if (URL is the same as the existing) {
                                Add duration and size;
                        } else {
                                search the list for a matching URL. Case where you have
                                the same IP address that has already inserted previously,
                                before the interception of other web sites.
                                While (URL in each record != NULL) {
                                        if (matching URL found) {
                                                Add duration and size;
                                                break;
                                        }
                                }
                                if (did not pass into the while loop) {
                                        Create a new Vertical structure in the same slot but
                                        with different URL;
                                }
                        }
                }
```

```
    } else if (clientsAddress is different with the existing in the slot) {
        //We have collisions
        Create a new horizontal list;
        if (nextClientAddr[slot] == NULL) {
            Create a new list due to collisions;
        } else if (nextClientAddr[slot] != NULL) {
            //Check the IP address
            if (IP is the same with the existing in the slot) {
                //Check URL
                if (URL is the same) {
                    Add duration and size;
                } else if (URL is different) {
                    initialize the nextClientAddr[slot] to point to the
                same location as startClientAddr[slot];
                    if (URL is the same with the existing) {
                        Add duration and size;
                    } else if (URL is different) {
                        Search the linked list for a matching URL.
                    Case that we have the same IP address, and a
                    web site that has already inserted previously,
                before the interception of other web sites
            while (URL in each record != NULL) {
                            if (matching URL found) {
                                Add duration and size;
                                break;
                            }
                        }
                        if (did not pass into the while loop) {
                            Create a new vertical node to the
                        current list;
```


ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

```
                    }
                }
            }
        } else if (different IP address to the same slot) {
            Search the list in order to find a matching IP address;
            while (search the list until will be null) {
                if (match a client address) {
                    break;
                }
            }
            if (did not pass into the while loop) {
                Insert new additional list;
            } else {
                //a matching IP was found. Passed through the
                //while loop.
                If (URL is matching) {
                    Add size and duration;
                } else if (different URL) {
                    //Search the list for a matching URL
                    while (search the list until will be null) {
                        if (matching URL found) {
                            Add size and duration;
                            break;
                        }

                    }
                    if (did not pass through the loop) {
                        insert new node to the current
                        additional list;
                    }
```

```
                                        }
                                }
                        }
                }
        }
}
```

When the parsing of the *access.log* file has been completed and all the data resides into the memory, create a function that will be able to manipulate this information and to exclude the data into a file. The name of the generated file will be the current date (that will be taken from the system time).

```
For (go through the hash table and check each slot) {
        if (nextURL[slot] != NULL) {
                go to start of the list;
                while(nextURL[slot] != NULL) {
                        write the information contained in the slot into the file;
                }
                while (nextClientAddr[slot] != NULL) {
                        write the information contained in the slot into the file;
                }
        }
}
make the generated file read only;
```

Moreover, create a function that will be able to manipulate this information and to export the data in an HTML format.

############## End of Pseudocode ##############

5.2 Log File Analyzer Installation

First of all, this application is written in C programming language (ANSI) to be extremely fast and highly portable by using the Eclipse IDE version 3.1.2. The compiler is: *GCC version 4.1.1*. Moreover, my working computer is a Sony Vaio laptop with CPU Intel Pentium M 1.50 Ghz, HDD 60GB, RAM 512 MB and the operating system is *Linux Fedora Core 5*.

In order to be able to run the existing application in your computer you must do the following steps:

- ✓ Login as root in a Linux distribution and open a console.
- ✓ Make a directory named "thorax" under the /root path (/root/thorax). In the new directory, create another directory named "userData" (/root/thorax/userData). In this directory (/root/thorax/userData), you will have the generated file that includes the needed information from the access log file. The filename comes from the current date (the exact date that you run the application). The format of the date is: YYYY-MM-DD.hh.mm. So, a sample file will have the following format: "2006-08-31.00.00". Note that you need to have the computer date and time synchronized with the <<real>> date and time if you want to have the name of the generated file the same with the current date.
- ✓ Extract the attached tarball file (proxy.tar.gz). In the generated folder named "proxy" you can see the source code (fully comented).
- ✓ Put the "thorax.conf" file that you must take from the above generated folder (proxy) in the /root/thorax path (/root/thorax/thorax.conf).
- ✓ Moreover, put the access.log file under the /root/thorax path (/root/thorax/access.log).
- ✓ Go into the "proxy" directory and run the following command: make
- ✓ After the above steps, you can go to the /root/thorax/userData path and you can open the generated file with a text editor (for example *joe*). Note that any text editor can be used in place of *joe* (gedit, vi, etc). The generated file is read only.

So, you cannot change it. Moreover, if you use *gedit* as an editor, in order to have the best alignment between the data in the file, choose as font the Monospace and as size 10 (Edit->Preferences->Fonts & Colors->Editor font). Also, you can see the results with the less command (for example: *less 2006-09-04.21.24*), or with the joe editor (*joe 2006-09-04.21.24*).

- ✓ That's all!Have fun!

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

5.2.1 For the Impatiens

After you log on in a Linux-based distribution as root, put the attached tarball file *"proxy.tar.gz"* and the *"proxy.install"* file into the same directory. Open a terminal as root and give the command: proxy.*install* (See *figure 5-5* below). Then, the above described steps will be done automatically. Note that in the tarball file, there exists a sample *access.log* file for testing purposes. Also, in the extracted folder named *"proxy"* you can see the source code (fully commented) and the *Makefile*.

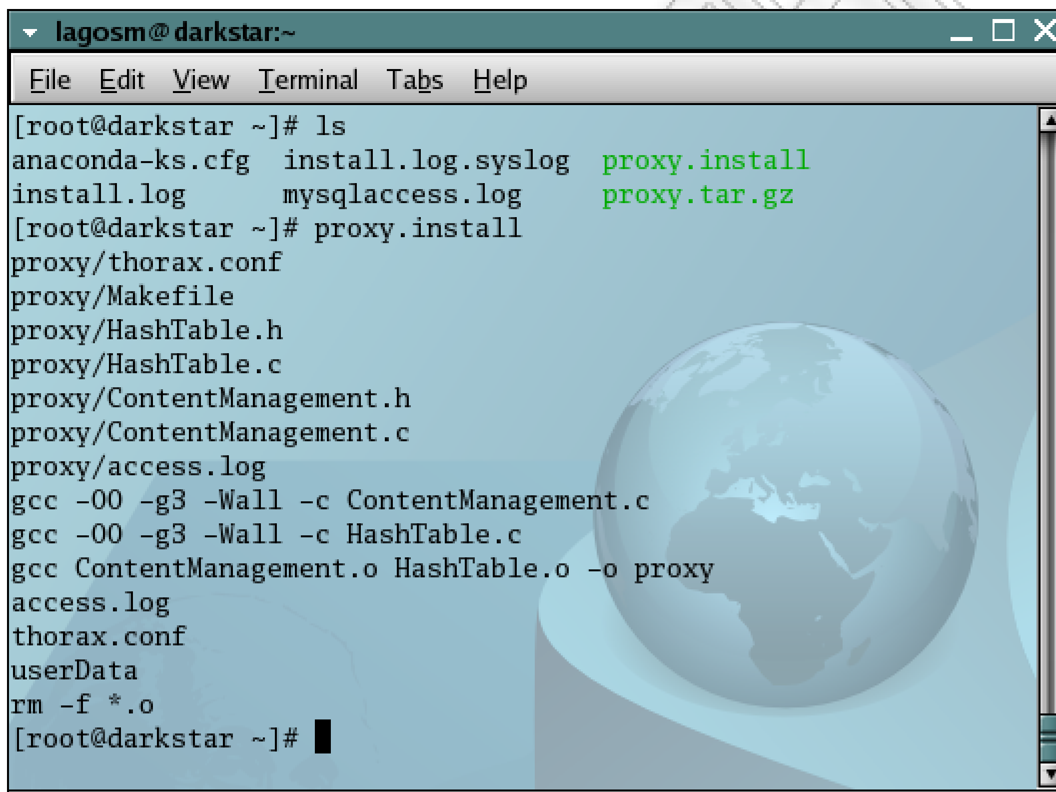


Figure 4-5.Proxy Log analyzer installation.

In *figure 5-6* that follows, one can see a sample screenshot after running the above steps and the file with users data has already generated (by giving the command: *less file_name*):

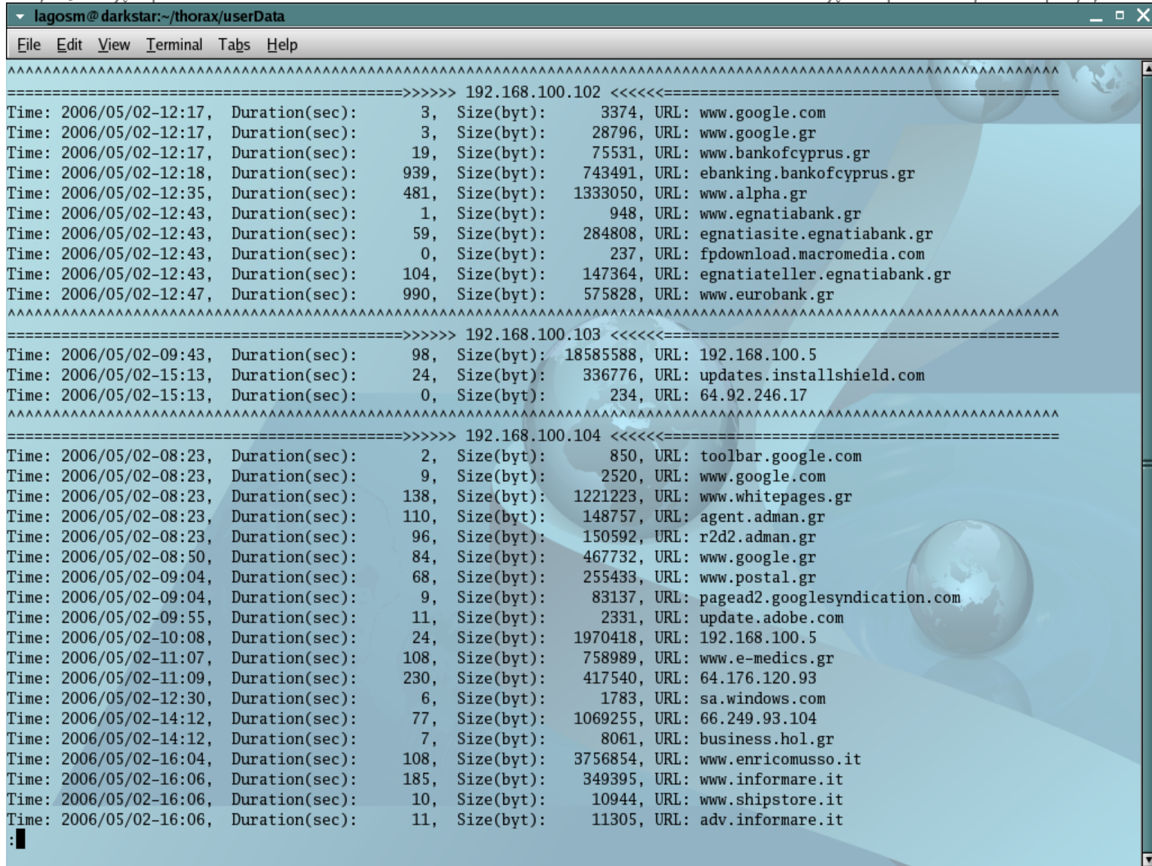


Figure 5-6.Sample screenshot with generated user data.

6. Packet Filtering

Firewall is just filters traffic, or at least it tries to, and we hope that it does. The golden rule with firewalls is always this: unless allowed, deny it. Never under circumstances attempt to build a firewall that is configured the other way around: allow everything; deny something. This is an approach doomed to fail. One additional advantage of the good approach, "unless allow, deny", is that you learn very quickly what short of traffic traversing your firewall. This might produce time-consuming work determining what to allow in and out of your firewall, but it's an excellent invaluable tool for determining what's going on. This approach gives you visibility into your network and users that you would otherwise never see if your firewall were built only to block certain things (Shinn, 2005).

Also, keep in mind that ports alone do not a good firewall policy make. Do not assume that just because you only allow port 80 traffic into a network that the traffic will only be HTTP if you are using the Linux kernels packet filtering capabilities. By default, the kernel has no way of inspecting the traffic to ensure that is HTTP traffic. Ports are no longer enough to define what the traffic is. If you want to filter the traffic into or out of a network by protocol, then you will need to use either an application proxy that understands the protocol (Squid) or utilize some of the iptables modules that can perform protocol inspection. A port is nothing but a number to an IP stack; it has no bearing on the protocol that might run over that port. Anything could be running on that port, not just the service that normally resides there.

Packet filtering is a network security mechanism that works by controlling what data can flow to and from a network. In other words, is a piece of software which looks at the header of packets as they pass through, and decides the fate of the entire packet in a way that reflects a site's own security policy (Russel, 2002), as shown in *figure 6-1*. It might decide to *DROP* the packet (i.e., discard the packet as if it had never received it – without

notifying the sender), *ACCEPT* the packet (i.e., let the packet go through), or *REJECT* the packet (i.e., refuse to forward it and return an error to the sender). Under Linux, *packet filtering* is built into the kernel (as a kernel module, or built right in), and there are a few trickier things we can do with packets, but the general principle of looking at the headers and deciding the fate of the packet is still there.

The basic device that interconnects IP networks is called a *router*. A *router* may be a dedicated piece of hardware that has no other purpose, or it may be a piece of software that runs on a general-purpose computer running UNIX or another operating system. Packets traversing an internetwork (a network of networks) 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. In general, a packet carries no information to help the router in this decision, other than the IP address of the packet's ultimate destination. The packet tells the router where it wants to go but not how to get there. Routers communicate with each other using routing protocols such as the *Routing Information Protocol (RIP)* and *Open Shortest Path First (OSPF)* to build routing tables in memory to determine how to get the packets to their destinations. When routing a packet, a router compares the packet's destination address to entries in the routing table and sends the packet onward as directed by the routing table (Zwicky, 2000). Often, there won't be a specific route for a particular destination and the router will use a default route; generally, such a route directs the packet towards smarter or better-connected routers (the default routes at most sites point towards the Internet).

In determining how to forward a packet towards its destination, a normal router looks only at a normal packet's destination address and asks only *"How can I forward this packet?"* A packet filtering router also considers the question *"Should I forward this packet?"* The packet filtering router answers that question according to the security policy programmed into the router via the packet filtering rules. Some machines do packet filtering without doing routing; that is, they may *accept* or *reject* packets destined

for them before they do further processing. The type of router used in a packet filtering firewall is known as a *screening router* (Zwicky, 2000).

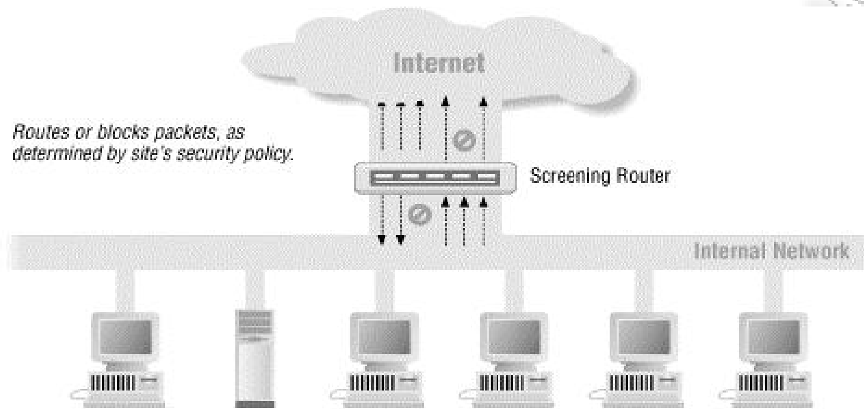


Figure 6-1.Using a screening router to do packet filtering.

6.1 Why Packet Filtering?

The reasons that the packet filtering is essential and a basic component of good firewall architecture are the following:

Control:

when you are using a Linux box to connect your internal network to another network (say, the Internet) you have an opportunity to allow certain types of traffic, and disallow others. For example, the header of packet contains the destination address of the packet, so you can prevent packets going to a certain part of the outside network. As another example, I use Mozilla Firefox to access the *contra.gr* athletic web site. There are advertisements from various sites on the page, and Firefox wastes my time by cheerfully downloading them. Telling the packet filter not to allow any packets to or from the address owned by these web sites solves that problem.

Security:

when your Linux box is the only thing between the chaos of the Internet and your nice, orderly network, it's nice to know you can restrict what comes tromping in your door. For example, you might allow anything to go out from your network, but you might be worried about the well-known 'Ping of Death' coming in from malicious outsiders. As another example, you might not want outsider's telnetting to your Linux box, even though all your accounts have passwords. Maybe you want (like most people) to be an observer on the Internet, and not a server (willing or otherwise). Simply don't let anyone connect in, by having the packet filter reject incoming packets used to setup connections (Russel, 2002).

Watchfulness:

sometimes a badly configured machine on the local network will decide to spew packets to the outside world. It's nice to tell the packet filter to let you know if anything abnormal occurs; maybe you can do something about it, or maybe you are just curious by nature (Russel, 2002).

Nevertheless, there are some disadvantages to using *packet filtering*. Currently filtering tools are *hard to configure*. Once configured, the packet filtering rules tend to be *hard to test*. Moreover, packet filtering reduces router performance because it adds *extra load* on it.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

6.2 Packet Filtering Under Linux

Linux kernels have had *packet filtering* since the 1.1 series. The first generation, based on *ipfw* from BSD, was ported by Alan Cox in late 1994. This was enhanced by Jos Vos and others for Linux 2.0; the userspace tool *"ipfwadm"* controlled the kernel filtering rules. In mid-1998, for Linux 2.2, Rusty Russell reworked the kernel quite heavily, with the help of Michael Neuling, and introduced the userspace tool *"ipchains"*. Finally, the fourth generation tool, *"iptables"*, and another kernel rewrite occurred in mid-1999 for Linux 2.4 (and beyond). It is this *iptables* which this project concentrates on (Russel, 2002).

You need a kernel which has the *netfilter* infrastructure in it: *netfilter* is a general framework inside the Linux kernel which other things (such as the iptables module) can plug into. Also, you need the tool *iptables* that talks to the kernel and tells it what packets to filter.

6.2.1 Netfilter/iptables

iptables is the replacement for the userspace tool ipchains in the Linux 2.4 kernel and beyond. It is part of the kernelspace Netfilter Project (http://www.netfilter.org). *iptables* has many more features than ipchains and is also structured more sensibly. A great advantage of *iptables* is that it can configure stateful firewalls (Marie). A stateful firewall is capable of assigning and remembering the state of connections made for sending or receiving packets. Also, it gives total control over firewall configuration and packet filtering and is free!

For a Linux system connected to a network, a firewall is the essential defense mechanism that allows only legitimate network traffic in and out of the system and disallows everything else. To determine whether the network traffic is legitimate or not, a firewall relies on a set of *rules* it contains that are predefined by a network or system administrator. These rules tell the firewall whether to consider as legitimate and what to do with the network traffic coming from a certain source, going to a certain destination, or having a certain protocol type. The term *"configuring the firewall"* refers to adding, modifying, and removing these rules (Bamdel, 2001).

Network traffic is made up of IP packets which are small chunks of data traveling in streams from a source system to a destination system. These packets have headers, i.e. bits of data prefixed to every packet that contain information about the packet's source, destination, and protocol types. Based on a set of rules, a firewall checks these headers to determine which packet to accept and which packet to reject. This process is known as *packet filtering*.

Iptables split the packet handling into three different tables, each of which contains a number of chains. The three tables are filter, nat and mangle. The first is quite obvious, and is used for packet filtering. Nat is used to provide packet modification capabilities, such as NAT/PAT and, of course, IP masquerading. The final table is the most obscure and is used for setting packet options, such as the Type of Service (TOS) field and marking packets for further filtering or routing (Coulson, 2003).

The *netfilter/iptables IP packet filtering system*, although referred to as a single entity, is actually made up of two components: *netfilter* and *iptables*. The netfilter component, also known as *kernelspace*, is the part of the kernel that consists of packet filtering tables containing sets of rules that are used by the kernel to control the process of packet filtering. The iptables component is a tool, also known as *userspace*, which facilitates inserting, modifying and removing rules in the packet filtering tables. By using userspace, you can build your own customized rules that are saved in packet filtering

tables in kernelspace (Stephens). These rules have *targets* that tell the kernel what to do with packets coming from certain sources, heading for certain destinations or have certain protocol types. If the rule doesn't match the packet, then the next rule in the chain is consulted. If a packet matches a rule, the packet can be allowed to pass through using target *ACCEPT*. A packet can also be blocked and killed using target *DROP* or *REJECT*. There are many more targets available for other actions that can be performed on packets.

The rules are grouped in chains, according to the types of packets they deal with. A chain is a checklist of rules. Rules dealing with incoming packets are added to the *INPUT chain*. Rules dealing with outgoing packets are added to the *OUTPUT chain*. And rules dealing with packets being forwarded are added to the *FORWARD chain*. These three chains are the main chains built-in by default inside basic packet-filtering tables. There are many other chain types available like *PREROUTING* and *POSTROUTING* and there is also provision for user-defined chains. Each chain can have a *policy* that defines "a default target", i.e. a default action to perform, if a packet doesn't match any rule in that chain. In a security-conscious system, this policy usually tells the kernel to *DROP* the packet (Coulson, 2001). In the following diagram (*figure 6-2*), we can see how packets traverse the different chains and in which order. When a packet first enters the firewall, it hits the hardware and then gets passed on the proper device driver in the kernel. Then the packet starts to go through a series of steps in the kernel before it is either sent to the correct application (locally), or forwarded to another host or whatever happens to it. In addition, as we talk about stateful firewalls, it is very important to understand the four states that the *netfilter state engine* recognizes. These are the states a packet is in based on the kernel's connection tracking capabilities (Shinn, 2005). At any given time a packet, when under the control of the state engine, is in one of the following four states:

NEW

A new connection. Only the first packet in a connection will meet this state. All subsequent packets, for that session, will not be considered NEW.

ESTABLISHED

A session that has been established and is being tracked by the state engine. All packets that follow after a packet labeled as NEW in a single session.

RELATED

A special state. A separate connection related to an ESTABLISHED session. This can happen when an ESTABLISHED connection spawns a new connection as part of its data communication process. An example of this is FTP-DATA connection spawned by an FTP connection. These types of connections are very complicated and almost always require a "helper" module that has been written to understand the underlying protocol. If there is no helper module to analyze a complex protocol that requires a related connection to function, then that connection will not function properly (Shinn, 2005).

INVALID

A packet that is otherwise not identified as having any state. It is always a good idea to DROP, not REJECT, this packet.

The specific piece of the netfilter code that handles all the above is called *conntrack*. Its job is to watch each session and determine if the packets are part of an existing session or a related one and to enforce the rules supplied by the userland tools around this information.

Related to the above, an important issue is the fragmentation (division of a large packet). Some packets simply lack enough context to tell the kernel enough about the connection it might be associated with. The fact that the kernel does this for us is a good thing. It makes the firewall more secure, the rules can be that much simpler because we don't have to worry about fragment anymore, and we can detect other problems with the packet before passing it on to a protected resource (Hall, 2000).

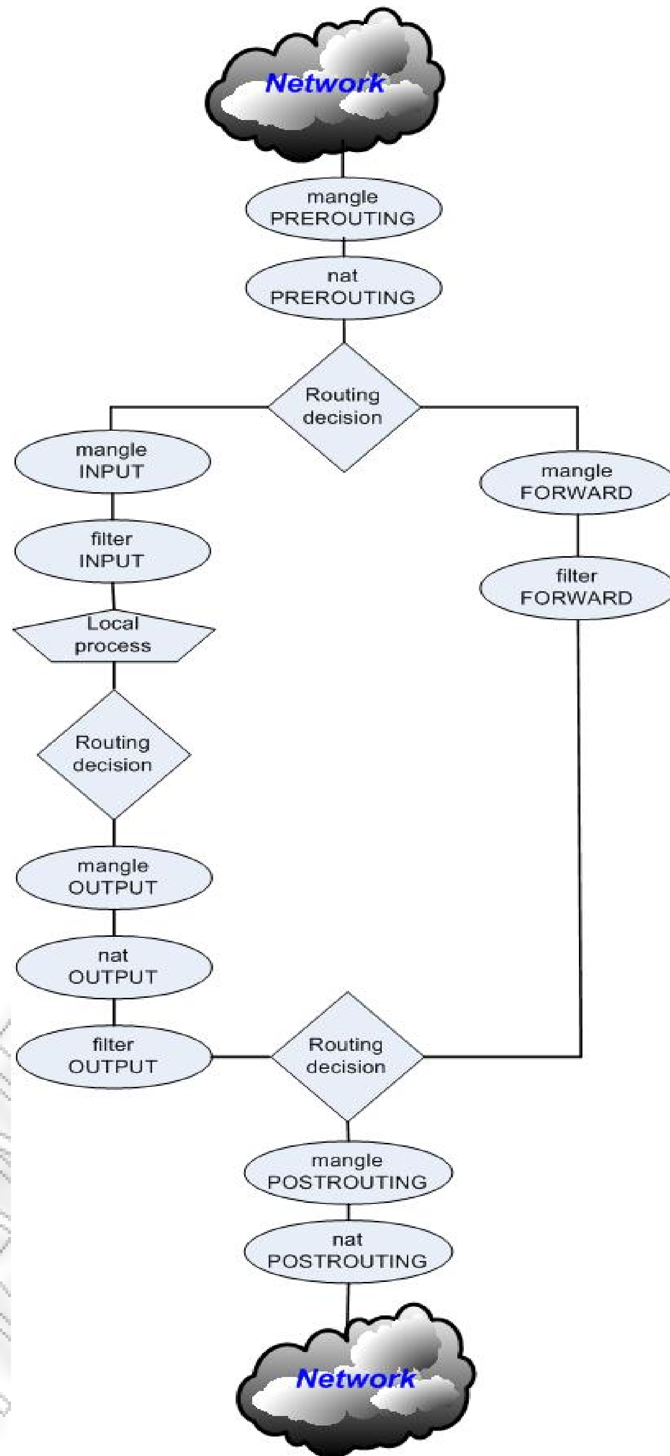


Figure 6-2.Traversing packets into different chains.

After the rules are built and chains are in place, the real work of packet filtering starts. Here is where the *kernelspace* takes over from *userspace*. When a packet reaches the firewall, the kernel first examines the header information of the packet, particularly the destination of the packet. This process is known as *routing* (Russel, 2002).

If the packet originated from outside and is destined for the system and the firewall is on, the kernel passes it on to the *INPUT* chain of the kernelspace packet filtering table. If the packet originated from inside the system or another source on an internal network the system is connected to and is destined for another outside system, the packet is passed on to the *OUTPUT* chain. Similarly, packets originating from outside systems and destined for outside systems are passed on to the *FORWARD* chain. Normally, we would write a rule something like this:

iptables [-t table] command [match] [target/jump]

The *table* option allows you to use any table other than the default (filter – is used for general packet filtering). The other tables are: *nat* and *mangle*. The *nat* table is used for packets to be forwarded and the *mangle* table is used if there are any changes to be made in packets and their headers (Coulson, 2001). The compulsory *command* section of the command above is the most important part of the *iptables*. It tells the *iptables* command what to do, for example, to insert a rule, to add a rule to the end of the chain, or to delete a rule. The optional *match* section specifies the characteristics that a packet should have to match the rule, such as *source* and *destination address*, *protocol*, etc. The matches are divided in two major categories: generic matches (can be used for packets having any protocol) and protocol-specific matches. And finally, the *target* option is the actions specified by rules to be performed on packets that match those rules (Andreasson, 2005).

Next the packet's header information is compared with each rule in the chain it is passed on to, unless it perfectly matches a rule. If a packet matches a rule, the kernel performs the action specified by the target of that rule on the packet. But if the packet doesn't match a rule, then it is compared to the next rule in the chain. Finally, if the packet

doesn't match to any rule in the chain, then the kernel consults the policy of that chain to decide what to do with the packet. Ideally the policy should tell the kernel to DROP that packet (Kenshi).

So, an entire network can be shielded from intrusions originating from other networks by the strategic placement of a rule-based firewall such as *iptables*. A single host on the same network can benefit from its own "line of defense" by running its own instance of *iptables*. This is particularly useful for hosts within a "*De-Militarized Zone*" (DMZ), where you need to open up service ports at the firewall/gateway to external access. DMZ's will often host web, ftp, DNS or email servers.

It is important to note that with the use of Squid on the firewall (such as in this case), the *FORWARD* rules will not be called, even though the firewall is technically forwarding packets. However, it's the Squid that is doing the actual forwarding. The kernel has no way of knowing this, so only the *INPUT* and *OUTPUT* rules will be tripped as Squid is just a local process on the firewall (Shinn, 2005).

I think its time to go on some more practical issues like setup and configuration of *iptables*.

6.2.1.1 Setup iptables

First of all, there is a need to have *iptables* installed on a computer. In order to install *iptables* in *Fedora Core 5* you must do the following (Lilliam, 2005):

- ✓ Login as root
- ✓ Verify the iptables package has been installed. Give the following command:

rpm -qa | grep iptables

If you have no results after giving the above command, then you need to install iptables.

- ✓ Make sure that you have an Internet connection. Also, you have to install the *yum* application in order to be able to download and install iptables. If you are under a proxy server, you need to set and export the proxy as:

*set http_proxy=*http://Proxy_Name:Proxy_Port/

*export http_proxy=*http://Proxy_Name:Proxy_Port/

- ✓ Then you can give the command:

yum install iptables

Once it has been verified that the packages are installed, iptables will need to be configured.

6.2.1.2 iptables Startup

Now that iptables package is installed on the computer, its time to start running it. To start iptables as a daemon in a Redhat-like system (see *figure 6-3* below), you can just give the command:

/sbin/service iptables start

If you want to run iptables on boot, you could give the following command:

chkconfig - - level 2345 iptables on

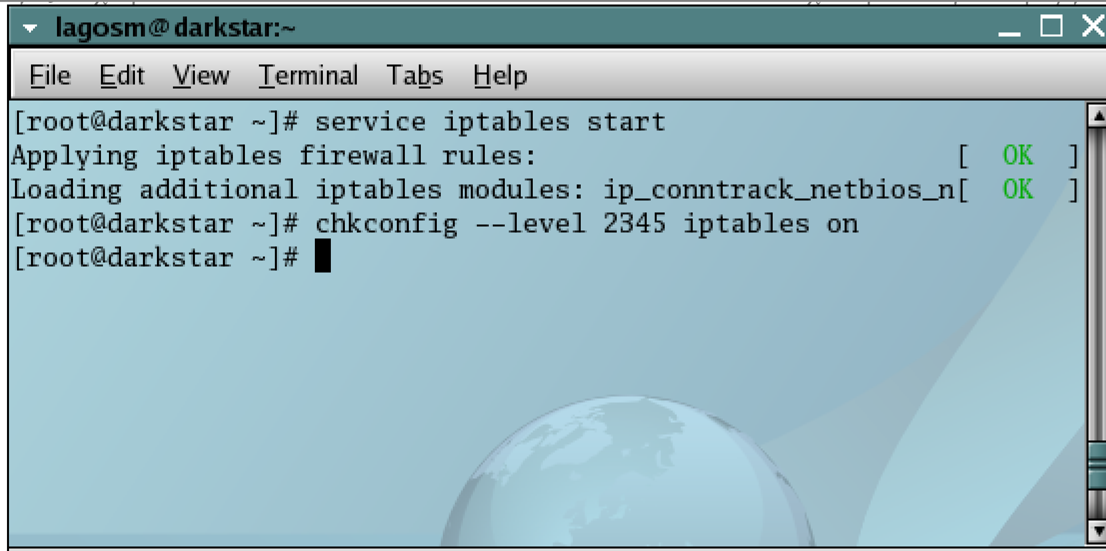


Figure 6-3.Start iptables as a daemon and enable on boot.

Correspondingly, you can stop *iptables* and disable it by running on boot if you replace in the above two commands the "start" with "stop" word and the "on" with the "off" word.

In order to check that iptables is up and running, I give the following command:

iptables -L #list the current ruleset

It responds so I am sure about the successful startup of *iptables*. After installing *iptables* successfully and is up and running, the next task is to delve into the configuration.

6.2.1.3 Netfilter/iptables Configuration

The first task to undertake when configuring the firewall ruleset is to turn on all the options you would like the kernel to use when processing IP packets and the next task is to configure the userland (iptables).

6.2.1.3.1 Kernel Setup

Let's try to configure the firewall architecture that is described in *Section 2.2* from the kernel point of view. The first option is dependent on how your firewall gets its IP addresses. If one or more of the firewall's interfaces uses DHCP to configure its interfaces, then you need to turn the ip_dynaddr option on by passing it the integer value of 1:

```
echo 1 > /proc/sys/net/ipv4/ip_dynaddr
```

otherwise, if all of your firewall's interfaces are assigned static IP addresses, you will want to set this variable to 0:

```
echo 0 > /proc/sys/net/ipv4/ip_dynaddr
```

The following script disables source routing. Source routing is basically a way of dictating what route the traffic will take from the origin of the packet as dictated by the client. This means that a client can dictate the specific route to a destination, subverting much of your firewall's purpose.

```
if [ -e /proc/sys/net/ipv4/conf/all/accept_source_route ]; then
 for f in /proc/sys/net/ipv4/conf/*/accept_source_route
 do
  /bin/echo 0 > $f
 done
fi
```

The next two scripts instruct the kernel not to send/accept ICMP redirect requests. Some resource on the network can sent an ICMP redirect to your firewall telling it there is a new network route to some other device.

```
if [ -e /proc/sys/net/ipv4/conf/all/send_redirects ]; then
 for f in /proc/sys/net/ipv4/conf/*/send_redirects
 do
  /bin/echo 0 > $f
 done
fi

if [ -e /proc/sys/net/ipv4/conf/all/accept_redirects ]; then
 for f in /proc/sys/net/ipv4/conf/*/accept_redirects
 do
  /bin/echo 0 > $f
 done
fi
```

Spoof protection is something near to the hearts of many in the network security world, and fortunately Linux gives us the ability to limit spoofed packets to some extent. The kernel includes a neat little */proc* setting that basically tells the kernel not to respond to a packet out of a different interface than the interface from which it was received. This is basically a system for helping to prevent spoofed packets by looking at where, from the firewall's perspective, it expects the packet to come from. With this turned on, the firewall should reject packets that come from outside your protected network that pretend to come from inside. This will help to prevent an attacker from spoofing trusted systems in an attempt to circumvent your firewall rules.

```
if [ -e /proc/sys/net/ipv4/conf/all/rp_filter ]; then
 for f in /proc/sys/net/ipv4/conf/*/rp_filter
 do
  /bin/echo 1 > $f
```

```
  done
fi
```

In addition to detecting and stopping spoofed packets, the kernel can also detect what are known as "martian" addresses. These addresses are invalid IP addresses that the firewall should never see. If you tell the firewall to, it will log all those invalid addresses it sees. This can be very useful when tracking down an errant piece of network or equipment or some strange attack on your firewall or the devices it's protecting. You can turn this option on as follows:

```
/bin/echo 1 > /proc/sys/net/ipv4/conf/all/log_martians
```

The next setting allows you to set the system to not respond to ICMP echo messages set to a broadcast address, otherwise known as a "smurf" attack. Unless you need to be able to ping broadcast addresses on your firewall, which you probably do not, just use the setting below. Failure to do so may cause the firewall to be used by an attacker to "amplify" an ICMP flood on another host.

```
/bin/echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

Moreover, another important setting is to not response to ICMP pings as follows:

```
/bin/echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

Sometimes devices like routers will send to broadcast frames, which will be logged by the kernel. On a busy network this can quickly become annoying. Setting this to true will disable these messages from being logged.

```
/bin/echo 1 > /proc/sys/net/ipv4/icmp_ignore_bogus_error_responses
```

The final section to look at related to the kernel configuration is the network performance setting in */proc*. The first network setting of interest is the TCP FIN timeout setting. This defines the amount of time to hold a socket in the FIN-WAIT-2 status, if the socket was closed by our side. This is part of the final tear down part of TCP session where we send out FIN packet and are awaiting a response from the other side via an ACK response. If the host on the side is broken, is slow, or does not respond, the socket will be held open for a certain period of time to wait regardless of what happens. In my kernel (2.6), the value is 60 seconds. If you want to change to a shorter period of time, which means that the socket will be killed off if the remote host does not finish the tear down, then:

/bin/echo 40 > /proc/sys/net/ipv4/tcp_fin_timeout

A similar setting is for the UDP connection timeout value. This defines how long the connection tracking engine will consider a UDP connection to be alive. If nothing else comes back within this timeframe, the connection will be removed from the state engine. In my kernel the value of this option is 30. So, if you want to change it, then you can give the following command:

/bin/echo 50 > /proc/sys/net/ipv4/netfilter/ip_conntrack_udp_timeout

In addition, to prevent SYN floods attacks, you need to have the following option enables as:

/bin/echo 1 > /proc/sys/net/ipv4/tcp_syncookies

Moreover, you can enable the forwarding option in order to be able to forward packets as:

/bin/echo 1 > /proc/sys/net/ipv4/ip_forward

All the above settings form a basic initial configuration of the kernel. There are many other settings for the kernel that are not discussed because there are out of the scope of this project.

6.2.1.3.2 Userland Setup

After the kernel is configured as described in the previous section, its time to start loading *iptables* modules we might want to use. There are two ways to do this; explicitly, which is the safer and more secure manner of doing this than the one that loading all the modules we have configured for this kernel. So, if you want to load each one explicitly, then you can give the following commands:

/sbin/modprobe ip_tables
/sbin/modprobe ip_conntrack
/sbin/modprobe ip_conntrack_ftp
/sbin/modprobe iptable_filter
/sbin/modprobe iptable_mangle
/sbin/modprobe iptable_nat
/sbin/modprobe ipt_LOG
/sbin/modprobe ipt_limit
/sbin/modprobe ipt_state
/sbin/modprobe ip_nat_ftp
[...]

After the kernel is properly configured and all the necessary firewall modules are loaded, its time to start adding rules into the kernel.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

In case that *iptables* won't start, then you must do the following: the *iptables* startup script expects to find the "*/etc/sysconfig/iptables"* before it starts. If none exists, then symptoms include the firewall status always being stopped and the "*/etc/init.d/iptables"* script running without the typical [OK] or [FAILED] messages. If you have just installed iptables and have never applied a policy, then you will face this problem. Unfortunately, running the service iptables save command before restarting won't help either. You have to create this file.

There are two ways in adding rules. First of all, you could edit the "*/etc/rc.d/init.d/iptables"* script. This would have the bad effect that the rules would be deleted if you updated the iptables package or after a reboot. The second way of doing the setup is to make and write a *ruleset* in a file, or directly to iptables, that will meet you requirements, and use iptables-save command to save them, so that you don't have to retype them each time (Chunyan, 2004). Consequently, you can enter the command:

iptables-save > /etc/sysconfig/iptables

which would save the ruleset to the file "*/etc/sysconfig/iptables"*. But if you want to save the rules in a separate file other than the default then:

iptables-save > iptables-script
iptables-restore iptables-script

which all rules in the packet filtering tables are saved in the file *iptables-script* and then can be restored (after a reboot for example). In addition, if we want to have the ruleset in a startup script, we can create a new file called *rc.firewall* in the */etc/rc.d* directory and then open the file */etc/rc.d/rc.local* and add the line */etc/rc.d/rc.firewall*.

I suppose that its time to make an example. Note that it is beyond the scope of this project to document all the switches for iptables; however, a few brief pointers should help with uncovering the proper manner in which a command should be executed. Because I have

no real network connection, I decide to use the *"loopback"* interface (IP address 127.0.0.1). The scenario is to try to drop all ICMP packets coming from the *"loopback"* interface. So in this case, the conditions are that the protocol must be ICMP and that the source address must be 127.0.0.1. The target is *DROP*. I give the following command:

ping -c 1 127.0.0.1

and the result is that:

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.090 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.053 ms

Before going on the definition of rules, it is best practice of *flushing (-F)* all chains before loading rules. So,

$IPTABLES -F INPUT
$IPTABLES -F OUTPUT
$IPTABLES -F FORWARD

Aside from creating rules, you also can create your own chains with iptables by using the -N command:

$IPTABLES -N new_chain

And much like rules, you also can delete, rename or replace a chain. In order to delete all the existing custom tables and to zeroth all the packet counters (Kenshi), as following:

$IPTABLES -X

$IPTABLES -Z

and then we must define the default policy. This policy in my system is to accept everything. As I mentioned before, if we want to have a security-conscious system, the policy must be modified in order to *DROP everything*, if there is no rule in a chain for a specific packet (Chunyan, 2004). So,

$IPTABLES -P INPUT DROP

$IPTABLES -P OUTPUT DROP

$IPTABLES -P FORWARD DROP

Figure 6-4 that follows shows the default policy and the changed policy. So now, it's impossible to send or receive something.

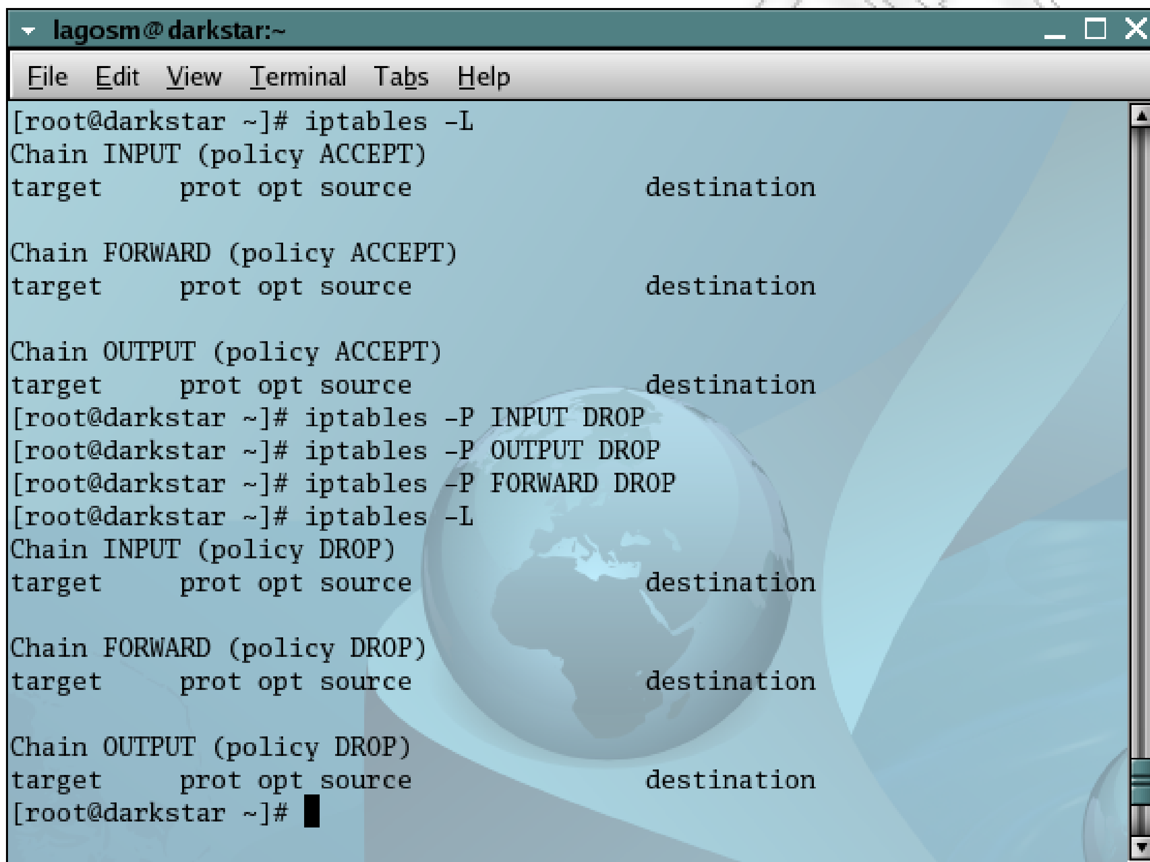


Figure 6-4.Default modified policy.

Then I append to the *INPUT* and *OUTPUT* chain, a rule specifying that for packets from source 127.0.0.1 with protocol ICMP we should jump to *ACCEPT*. Also, I need to log these actions. Note that the -A switch puts the new rule at the end of the chain, while the -I switch puts it at the beginning. So, I give the command:

$IPTABLES -A INPUT -p ICMP -s 127.0.0.1 -j ACCEPT
$IPTABLES -A INPUT -p ICMP -s 127.0.0.1 –j LOG –log-prefix "INPUT: "
$IPTABLES -A OUTPUT -p ICMP -s 127.0.0.1 -j ACCEPT
$IPTABLES -A OUTPUT -p ICMP -s 127.0.0.1 –j LOG –log-prefix "OUTPUT: "

The results are the willing and depicts in the following figures. *Figure 6-5* depicts the sent/received ICMP messages and *figure 6-6* shows the logging of these actions.

```
lagosm@darkstar:~
File  Edit  View  Terminal  Tabs  Help
[root@darkstar ~]# iptables -A OUTPUT -p ICMP -s 127.0.0.1 -j ACCEPT
[root@darkstar ~]# ping -c 3 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.164 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.156 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.163 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.156/0.161/0.164/0.003 ms
[root@darkstar ~]# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
LOG        icmp --  localhost.localdomain  anywhere            LOG level warning
ACCEPT     icmp --  localhost.localdomain  anywhere

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source               destination
LOG        icmp --  localhost.localdomain  anywhere            LOG level warning
ACCEPT     icmp --  localhost.localdomain  anywhere
[root@darkstar ~]#
```

Figure 6-5.Accept all ICMP messages from localhost.

```
lagosm@darkstar:~
File Edit View Terminal Tabs Help
~--> su
Password:
[root@darkstar lagosm]# cd
[root@darkstar ~]# tail -f /var/log/messages
Sep 12 20:54:09 darkstar kernel:   Type:   Direct-Access                      ANSI SCSI revision: 00
Sep 12 20:54:09 darkstar kernel:  0:0:0:0: Attached scsi generic sg0 type 0
Sep 12 20:54:09 darkstar kernel: SCSI device sda: 2052607 512-byte hdwr sectors (1051 MB)
Sep 12 20:54:09 darkstar kernel: sda: Write Protect is off
Sep 12 20:54:09 darkstar kernel: sda: assuming drive cache: write through
Sep 12 20:54:09 darkstar kernel: SCSI device sda: 2052607 512-byte hdwr sectors (1051 MB)
Sep 12 20:54:09 darkstar kernel: sda: Write Protect is off
Sep 12 20:54:09 darkstar kernel: sda: assuming drive cache: write through
Sep 12 20:54:09 darkstar kernel:  sda: sda1
Sep 12 20:54:09 darkstar kernel: sd 0:0:0:0: Attached scsi removable disk sda
Sep 12 20:55:55 darkstar kernel: ip_tables: (C) 2000-2006 Netfilter Core Team
Sep 12 21:05:00 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROT
O=ICMP TYPE=8 CODE=0 ID=5898 SEQ=1
Sep 12 21:05:00 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROTO=ICMP TYPE=8 CODE=0 ID=5898 SEQ=1
Sep 12 21:05:00 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52489 PRO
TO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=1
Sep 12 21:05:00 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52489 PROTO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=1
Sep 12 21:05:01 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROT
O=ICMP TYPE=8 CODE=0 ID=5898 SEQ=2
Sep 12 21:05:01 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROTO=ICMP TYPE=8 CODE=0 ID=5898 SEQ=2
Sep 12 21:05:01 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52490 PRO
TO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=2
Sep 12 21:05:01 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52490 PROTO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=2
Sep 12 21:05:02 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROT
O=ICMP TYPE=8 CODE=0 ID=5898 SEQ=3
Sep 12 21:05:02 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF PROTO=ICMP TYPE=8 CODE=0 ID=5898 SEQ=3
Sep 12 21:05:02 darkstar kernel: OUTPUT: IN= OUT=lo SRC=127.0.0.1 DST=127.0.0.1 LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52491 PRO
TO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=3
Sep 12 21:05:02 darkstar kernel: INPUT: IN=lo OUT= MAC=00:00:00:00:00:00:00:00:00:00:00:00:08:00 SRC=127.0.0.1 DST=127.0.0.1
LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=52491 PROTO=ICMP TYPE=0 CODE=0 ID=5898 SEQ=3
```

Figure 6-6.Logging of the actions.

Let's try to give some basic rules in order to have my firewall architecture works properly. There are paranoid people that they like to be more and safer. It is truth that it never hurts to add in a rule to catch any fragments and drop them if something were to go horribly wrong with all of this (Brockmeier, 2001). So, to catch any fragments that slip through, which, again, shouldn't occur if you are using the state engine, simply add the following rules to the top of your rule sets:

$IPTABLES -N NOFRAGS

$IPTABLES -A OUTPUT -p ip -f -j NOFRAGS

$IPTABLES -A INPUT -p ip -f -j NOFRAGS

$IPTABLES -A FORWARD -p ip -f -j NOFRAGS

*$IPTABLES -A NOFRAGS -m limit –limit 1/second -j LOG –log-level info –log-prefix *

"Fragment – DROP " --log-tcp-sequence –log-tcp-options –log-ip-options

$IPTABLES -A NOFRAGS -j DROP

Also, one more important setting is to define connection tracking rules as:

*$IPTABLES -A INPUT -m state --state ESTABLISHED,RELATED *

-j ACCEPT

*$IPTABLES -A OUTPUT -m state --state NEW,ESTABLISHED,RELATED *

-j ACCEPT

*$IPTABLES -A FORWARD -m state --state NEW,ESTABLISHED,RELATED *

-j ACCEPT

Also, we need to have TCP sequence numbers logged. In order to achieve this fact, you must have the following rule:

$IPTABLES –A INPUT –p TCP –j LOG –log-tcp-sequence

In addition, in order to configure the DMZ section of the firewall architecture you need to DNATed the HTTP, FTP and SMTP traffic as follows:

```
#
# HTTP traffic
#
$IPTABLES -A FORWARD -i $EXTERNAL -o $DMZ -p TCP --dport 80 \
-m state -state NEW, ESTABLISHED, RELATED -j ACCEPT
$IPTABLES -t nat -A PREROUTING -i $EXTERNAL -p TCP \
--dport 80 -j DNAT --to-destination $DMZ_WEB_SERVER

#
# SMTP traffic
#
$IPTABLES -A FORWARD -i $EXTERNAL -o $DMZ -p TCP --dport 25 \
```

-m state -state NEW, ESTABLISHED, RELATED -j ACCEPT

*$IPTABLES -t nat -A PREROUTING -i $EXTERNAL -p TCP *

--dport 25 -j DNAT --to-destination $DMZ_MAIL_SERVER

#

FTP traffic

#

*$IPTABLES -A FORWARD -i $EXTERNAL -o $DMZ -p TCP --dport 20:21 *

-m state -state NEW, ESTABLISHED, RELATED -j ACCEPT

*$IPTABLES -t nat -A PREROUTING -i $EXTERNAL -p TCP *

--dport 20:21 -j DNAT --to-destination $DMZ_FTP_SERVER

In order to have the users from internal network to be able to SSH into the firewall you just need to add the following rule:

$IPTABLES -A INPUT -p TCP --dport 22 -i $LAN -j ACCEPT

and in order to SNATed the traffic from the internal, then:

*$IPTABLES -t nat -A POSTROUTING -i $LAN -o $EXTERNAL -j SNAT *

--to-source $EXTERNAL_IP

Finally, I implement a catch all logging rule before the final DROP or REJECT rule. This is very helpful if you want to diagnose packets that are dropped by the firewall because of a missing rule, typo, or other mistake. This is also a useful security measure because the firewall will now log all the packets will be rejected as part of the "unless allow, deny" security philosophy. In this example, the number of packets is limited to 1 per second in order to prevent the firewall logging system from being overwhelmed by either too much traffic or by a deliberate attempt by an attacker to overwhelm the system. The rule that implement the above is:

```
$IPTABLES -A INPUT -p all -m limit --limit 1/second \
-j LOG --log-level info --log-prefix "##==>DROP==<##" \
--log-tcp-sequence --log-tcp-options --log-ip-options
```

6.2.1.3.3 State Engine

Section 6.2.1 describes a basic understanding of how the state engine works. Let's take a look at what its tracking. The first place to look is in the most powerful file system on a Linux system, */proc*. It's a virtual file system that lets you look at the devices in your system, to reconfigure some of them, and even to modify the kernels' and networks' behavior on the fly. This is an extraordinarily powerful feature of Linux (Russel & Welte, 2002).

As to the state engine, let's take a look at */proc/net/ip_conntrack*. This lists all the current connection the kernel is tracking and any information the system has on those connections. In my working computer, it has the following entry:

tcp 6 424763 ESTABLISHED src=127.0.0.1 dst=127.0.0.1 sport=51282 dport=6009 packets=93 bytes=255520 src=127.0.0.1 dst=127.0.0.1 sport=6009 dport=51282 packets=92 bytes=4912 [ASSURED] mark=0 use=1

6.2.1.3.4 Specifying an Interface

The '-i' (or *'--in-interface'*) and –o (or *'--out-interface'*) options specify the name of an *interface* to match. An interface is the physical device the packet came in ('-i') or is

going out on ('-o'). You can use the *ifconfig* command to list the interfaces which are "up". Packets traversing the *INPUT* chain don't have an output interface, so any rule using '-o' in this chain will never match. Similarly, packets traversing the *OUTPUT* chain don't have an input interface, so any rule using '-i' in this chain will never match. Only packets traversing the *FORWARD* chain have both an input and output interface.

It is perfectly legal to specify an interface that currently does not exist; the rule will not match anything until the interface comes up. This is extremely useful for dial-up PPP links (usually interface ppp0) and the like. As a special case, an interface name ending with a `+' will match all interfaces (whether they currently exist or not) which begin with that string. For example, to specify a rule which matches all PPP interfaces, the -i ppp+ option would be used. Moreover, the interface name can be preceded by a `!' to match a packet which does *NOT* match the specified interface(s).

I suggest that whenever possible you define all your rules and chains with their respective interfaces to add an extra layer of security into your security model. IP addresses can be spoofed, and do make mistakes with rules. By adding in an additional layer of specificity to your rules, by defining the interface that rule specifically applies to, you can help yourself with debugging and by making your firewall and network more secure.

6.2.1.4 Network Address Translation (NAT)

Normally, packets on a network travel from their source (such as your home computer) to their destination (such as www.fedora.org) through many different links. None of these links really alter your packet: they just send it onward. If one of these links were to do *NAT*, then they would alter the source or destinations of the packet as it passes through.

As you can imagine, this not how the system was designed to work, and hence *NAT* is always something a crock. Usually the link doing *NAT* will remember how it mangled a packet, and when a reply packet passes through the other way, it will do the reverse mangling on that reply packet, so everything works.

The main reasons that you need NAT are the following:

Modem Connections to the Internet

Most ISPs give a single IP address when you dial up to them. You can send out packets with any source address you want, but only replies to packets with this source IP address will return to you. If you want to use multiple different machines (such as a home network) to connect to the Internet through this one link, you will need NAT. This is by far the most common use of NAT today, commonly known as "masquerading" in the Linux world. This is usually called SNAT, because you change the source address of the first packet (Russel, 2002).

Multiple Servers

Sometimes you want to change where packets heading into your network will go. Frequently this is because (as above), you have only one IP address, but you want people to be able to get into the boxes behind the one with the "real" IP address. If you rewrite the destination of incoming packets, you can manage this. This type of NAT was called port-forwarding under previous versions of Linux (Russel, 2002).

Transparent Proxying

Sometimes you want to pretend that each packet which passes through your Linux box is destined for a program on the Linux box itself. This is used to make transparent proxies. The transparent part is because your network won't even know it's talking to a proxy, unless of course, the proxy doesn't work (Russel, 2002).

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

The NAT is divided into two different types: *Source NAT (SNAT)* and *Destination NAT (DNAT)*. *SNAT* is when you alter the source address of the first packet: i.e. you are changing where the connection is coming from. Source NAT is always done post-routing, just before the packet goes out onto the wire. Masquerading is a specialized form of SNAT. *Masquerading* is another name for what many call many to one NAT. In other words, traffic from all devices on one or more protected networks will appear as if it originated from a single IP address on the Internet side of the firewall. Note that the masquerade IP address always defaults to the IP address of the firewall's main interface. The advantage of this is that you never have to specify the NAT IP address. This makes it much easier to configure *iptables* NAT with DHCP. You can configure many to one NAT to an IP alias, using the *POSTROUTING* and not the *MASQUERADE* statement.

Keep in mind that *iptables* requires the *iptables_nat* module to be loaded with the *modprobe* command for the masquerade feature to work. Masquerading also depends on the Linux operating system being configured to support routing between the Internet and private network interfaces of the firewall. This is done by enabling IP forwarding or routing by giving the file "/proc/sys/net/ipv4/ip_forward" the value 1 as opposed to the default disabled value of 0. You need to enter:

```
/bin/echo "1" > /proc/sys/net/ipv4/ip_forward
```

Once masquerading has been achieved using the *POSTROUTING* chain of the *NAT* table, you will have to configure *iptables* to allow packets to flow between the two interfaces. To do this, use the *FORWARD* chain of the filter table. More specifically, packets related to *NEW* and *ESTABLISHED* connections will be allowed outbound to the Internet, but only packets related to *ESTABLISHED* connections will be allowed inbound. This helps to protect the home network from anyone trying to initiate connections from the Internet.

On the other hand, *DNAT* is when you alter the destination address of the first packet: i.e. you are changing where the connection is going to. Destination NAT is always done before routing, when the packet first comes off the wire. Port forwarding, load sharing, and transparent proxying are all forms of DNAT (Li, 2002).

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

6.2.1.5 iptables Logging

Nevertheless, another useful and very important thing is the *logging*. You track packets passing through the *iptables* list of rules using the *LOG* target. You should be aware that the *LOG* target:

- ✓ Logs all traffic that matches the *iptables* rule in which it is located.
- ✓ Automatically writes an entry to the "*/var/log/messages*" file and then executes the next rule.

If you want to log only unwanted traffic, therefore, you have to add a matching rule with a *DROP* target immediately after the *LOG* rule. If you don't, you will find yourself logging both desired and unwanted traffic with no way of discerning between the two, because by default *iptables* doesn't state why the packet was logged in its log message. The messages are stored in "/var/log/messages" and we can observe them with the command *"tail -f /var/log/messages"*. We can log all the incoming and outgoing tcp packets that were dropped as following:

```
iptables -A INPUT -p tcp -j LOG --log-prefix "iptables:IN-TCP DROPPED:"
iptables -A OUTPUT -p tcp -j LOG --log-prefix "iptables:OUT-TCP DROPPED:"
```

Also, we can log anything else that was dropped:

```
iptables -A INPUT -j LOG --log-prefix "iptables:INCOMING DROPPED:"
iptables -A OUTPUT -j LOG --log-prefix "iptables:OUTGOING DROPPED:"
```

The information that a computer exchanges with other computers on the Internet travels in units called *packets*. The packets contain the data intended to be communicated between two computers, as well as a header. The header contains information about the packet (e.g. the size of the packet, various flags, etc.), and routing information (such as

the source and the destination). The usual analogy is with a postal letter: if the letter itself is the data, the envelope (containing the "TO:" and the "FROM:" fields) is the header. Again, the packet header contains other information as well, and the exact fields in the header depend on the protocol (such as tcp, udp or icmp). We will be interested mainly in the source address/port and the destination address/port fields, the protocol of the packet, as well as some packet flags.

As is referred previously, the correct logic to design the firewall is this: drop everything by default, then punch holes into the firewall as needed, to let the "good" packets come through. Assume that all packets are bad, and single out the good ones, not vice versa. I cannot emphasize this enough. The DROP policy in the INPUT chain acts as a safety net. After we've set the policy to DROP, we punch holes to allow useful packets, such as ssh, ntp, http, smtp, ftp, and whatever the needs of your particular site require.

Logging is not mandatory for the proper operation of the firewall, but it is very useful. That's exact the aim of this log file analyzer that I have implemented. It can read the firewall log file and can extract useful information. There are obviously two extremes: *log nothing*, or *log everything*. If we log everything (or nearly everything), we must decide on the granularity of the logging. Specifically, the logs can be *coarse*, i.e. we can log irrespective of the nature of the packet (e.g. the same log prefix, or no log prefix at all for all packets), or *fine*: a special log prefix for each type of packet. In the latter case, we would have to accompany each "real" rule in the firewall, with a log rule, with a special log prefix. This way, we would be able to figure out in the logs, which rule was responsible for dropping/accepting a packet.

6.2.1.5.1 Netfilter Log Format (*firewall.log*)

When a packet is logged, it shows up in *"/var/log/messages"* in the below format (is a hypothetical log message generated by netfilter):

```
Feb 12 21:01:44 darkstar kernel: DROP IN=eth0 OUT=eth1
MAC=00:80:8c:1e:12:60:00:10:76:00:2f:c2:08:00
SRC=132.26.78.235 DST=128.192.234.143 LEN=73
TOS=0x00 PREC=0x00 TTL=64 ID=33345 CE DF MF FRAG=179
OPT(072728CBA404DFCBA40253CBA4032ECBA403A2CBA4033ECBA40
2C1180746EA18074C52892734A200)
PROTO=TCP SPT=33456 DPT=23456 SEQ=1168094040 ACK=0
WINDOW=34560 RES=0x00 URG ACK PSH RST SYN FIN URGP=0
OPT (020405B40402080A05E3F3C40000000001030300)
```

Like other *syslog* entries, each entry starts with a date, timestamp, hostname (darkstar in this case), and the fact that the kernel provided the log entry. The "DROP" entry is one chosen by the person creating the firewall; it could be any ascii string that provides clues as to why it is being logged. In this case, it simply refers to the fact that this packet is being caught by the generic "log and drop everything else" rules at the end of the firewall. Also, packets arriving at the system only have IN= set. Packets leaving the system have only OUT= set. Packets being forwarded through an iptables box acting as a router have both set. In addition, log prefixing is a feature that's so simple; people tend to overlook how powerful it can be. The prefixing capability allows you to define an *iptables* rule and specify a text string that should be recorded to the logs whenever that rule is matched. Let me give you an example. Assume eth1 is the internal interface and eth0 is the external interface:

```
iptables -A FORWARD -i eth1 -m state --state NEW -s 192.168.1.0/24 -d 0/0 \
-j LOG --log-prefix " DROP "
iptables -A FORWARD -i eth1 -m state --state NEW -s 192.168.1.0/24 -d 0/0 -j \
ACCEPT
```

The first rule states

For traffic received on the eth1 interface, match packets that are new connection requests originating from 192.168.1.0-192.168.1.255, going to any destination IP address. When a match is found, allow the connection request, log this packet and prefix the log entry with the text pattern " DROP ".

Here is an example log entry recorded by this pair of rules. Notice that the word DROP appears just before the packet specific information:

Oct 31 06:11:35 gw1 kernel: DROP IN=eth1 OUT=eth0 SRC=192.168.1.101 DST=204.152.189.116 LEN=52 TOS=0x00 PREC=0x00 TTL=127 ID=18437 DF PROTO=TCP SPT=32865 DPT=80 WINDOW=5840 RES=0x00 SYN URGP=0

In the following table (*table 1*), is presented an explanation of each field in the *netfilter* log format (Unknown, Netfilter Log Format).

Feb 12 21:01:44 darkstar kernel:	Syslog prefix.
DROP	Enabled with: --log-prefix **'prefix'** A user defined log prefix.
IN=eth0	Interface from which the packet arrived. Empty value for locally generated packets.
OUT=eth1	Interface through which the packet will leave the system.
MAC=00:80:8c:1e:12:60: 00:10:76:00:2f:c2: 08:00	Destination MAC=00:80:8c:1e:12:60, Source MAC=00:10:76:00:2f:c2, Type=08:00 (ethernet frame carried an IPv4 datagram)
SRC=132.26.78.235	The source IP address of the packet.
DST=128.192.234.143	The destination IP address of the packet.
LEN=73	Is the length of the total packet in bytes.
TOS=0x00	The Type of Service value specifies how an upper layer protocol (Layer 4+) should handle the packet and the importance of the packet. Its value is commonly 0 indicating no special priority handling needed.
PREC=0x00	Is for precedence.

TTL=64	Is set by the sender and then decremented by 1 each time a router forwards the datagram. In this case, the remaining Time To Live of this packet is 64 hops.
ID=33345	Unique ID of this IP datagram.
CE	The CE bit is located in the TOS field.
DF	If present, the Don't Fragment bit is set.
MF	"More Fragments following" flag.
FRAG=179	Fragment offset in units of "8-bytes". In this case the byte offset for data in this packet is 179*8=1432 bytes. Specifies what position the fragment is in relation to the beginning of the datagram.
OPT (0727..A200)	Enabled with: **--log-ip-options** IP options. This variable length field is rarely used. Certain IP options, f.e. source routing, are often disallowed by netadmins. Even harmless options like "Record Route" may only be allowed if the transport protocol is ICMP, or not at all.
PROTO=TCP	Protocol name or number. Netfilter uses names for TCP, UDP, ICMP, AH and ESP. Other protocols are identified by number. A list is in your */etc/protocols*.
SPT=33456	The source port (TCP and UDP) of the packet. A list of port numbers is in your */etc/services*. 0-65535 are the legal values.
DPT=23456	The destination port (TCP and UDP) of the packet, restricted to 0-65535.
SEQ=1168094040	Enabled with: **--log-tcp-sequence** Receive Sequence number. By cleverly chosing this number, a cryptographic "cookie" can be implemented while still satisfying TCP protocol requirements. These "SYN-cookies" defeat some types of SYN-flooding DoS attacks and should be enabled on all systems running public TCP servers. `echo 1 > /proc/sys/net/ipv4/tcp_syncookies`
ACK=0	Same as the Receive Sequence number, but for the other end of the TCP connection.
WINDOW=34560	The TCP Receive Window size. This may be scaled by bit-shifting left by a number of bits specified in the "Window Scale" TCP option. If the host supports ECN, then the TCP Receive Window size will also be controlled by that.
RES=0x00	Any value set in the TCP reserved bits in bytes 12 and 13
URG	Urgent flag.
ACK	Acknowledgment flag.

PSH	Push flag.
RST	Reset flag.
SYN	SYN flag, only exchanged at TCP connection establishment.
FIN	FIN flag, only exchanged at TCP disconnection.
URGP=0	The Urgent Pointer allows for urgent, "out of band" data transfer. Unfortunately not all protocol implementations agree, so this facility is hardly ever used.
`OPT (020…300)`	enabled with: **--log-tcp-options** TCP options. This variable length field gets a lot of use. Important options include: Window Scaling, Selective Acknowledgement and Explicit Congestion Notification.

Table 1.Explanation of netfilter log format.

One of the most important functions of the IP layer is *routing*. Every IP datagram contains a source and destination IP address. *Figure 6-7* shows the format of an IPv4 header. The most significant bit is numbered 0 at the left, and the least significant bit of a 32-bit value is numbered 31 on the right. The 4 bytes in the 32-bit value are transmitted in the order: bits 0-7 first, then bits 8-15, then 16-23, and bits 24-31 last. This is called *big endian* byte ordering, which is the byte ordering required for all binary integers in the TCP/IP headers as they traverse a network. This is called the *network byte order*. Machines that store binary integers in other formats, such as the *little endian* format, must convert the header values into the network byte order before transmitting the data (Stevens, 2003).

The header of the IP packet consists of 5 or more words of 32 bits (4 bytes) each. The minimum header length (no options) is therefore 20 bytes. The version field for the shown type of packet is 4 = IPv4 (Internet Protocol version 4). The header Length field is the header length in 32bit words; this would be 5 without options, and at most 15 with options. The Total Length is in bytes and includes the header. Data length can then be calculated from the supplied values.

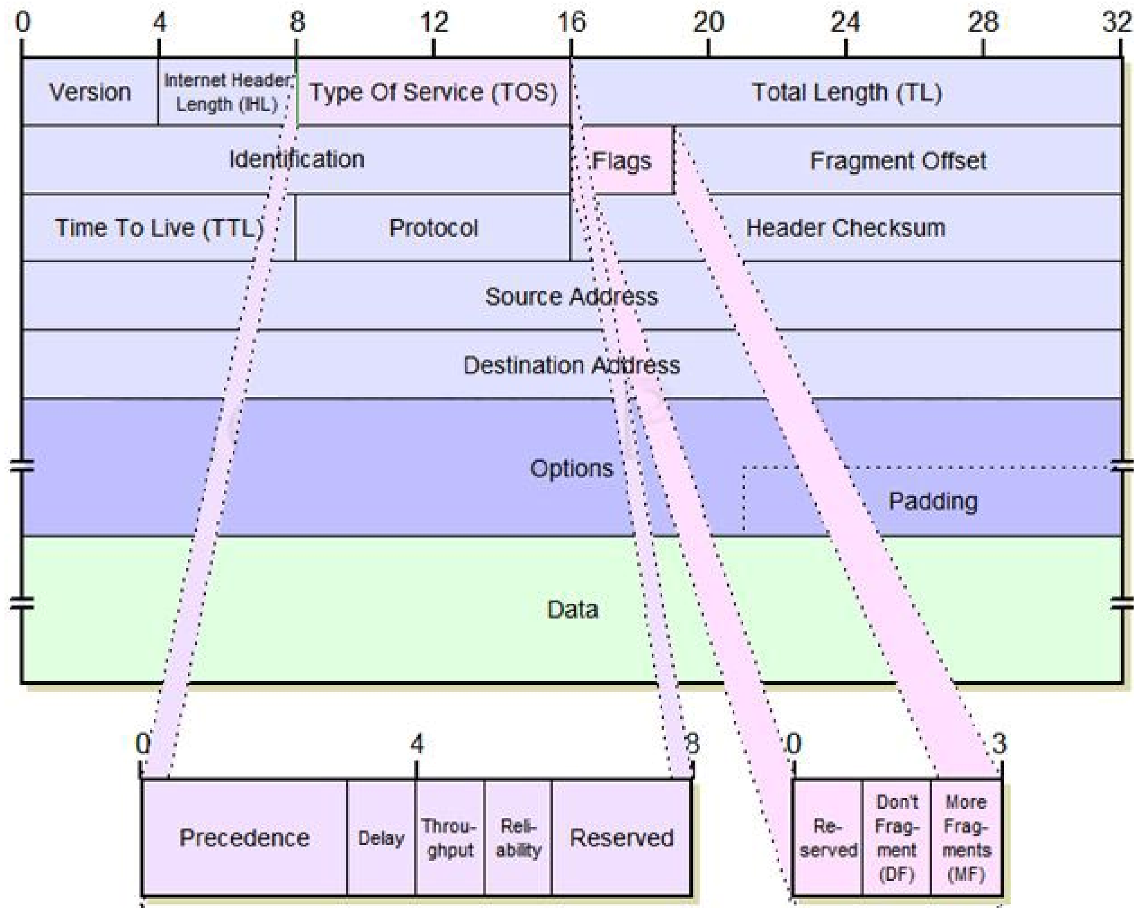


Figure 6-7.IP header format.

On the other hand, TCP is a connection-oriented service and handles the transfer of data, flow control, reliability, and multiplexing all in one protocol. TCP is a very robust protocol and can handle most error conditions with automated recovery. It is a good protocol for higher-level protocols, such as HTTP and SMTP, which do not have built in error recovery and flow control capabilities. But TCP is not ideal for protocols that handle this internally, such as VPN protocols, which are tunneling TCP connections within (Hall, 2000).

The header of a TCP packet consists of 5 or more words of 32 bits (4 bytes) each. The minimum header length (no options) is therefore 20 bytes. *Figure 6-8* shows the format of the TCP header.

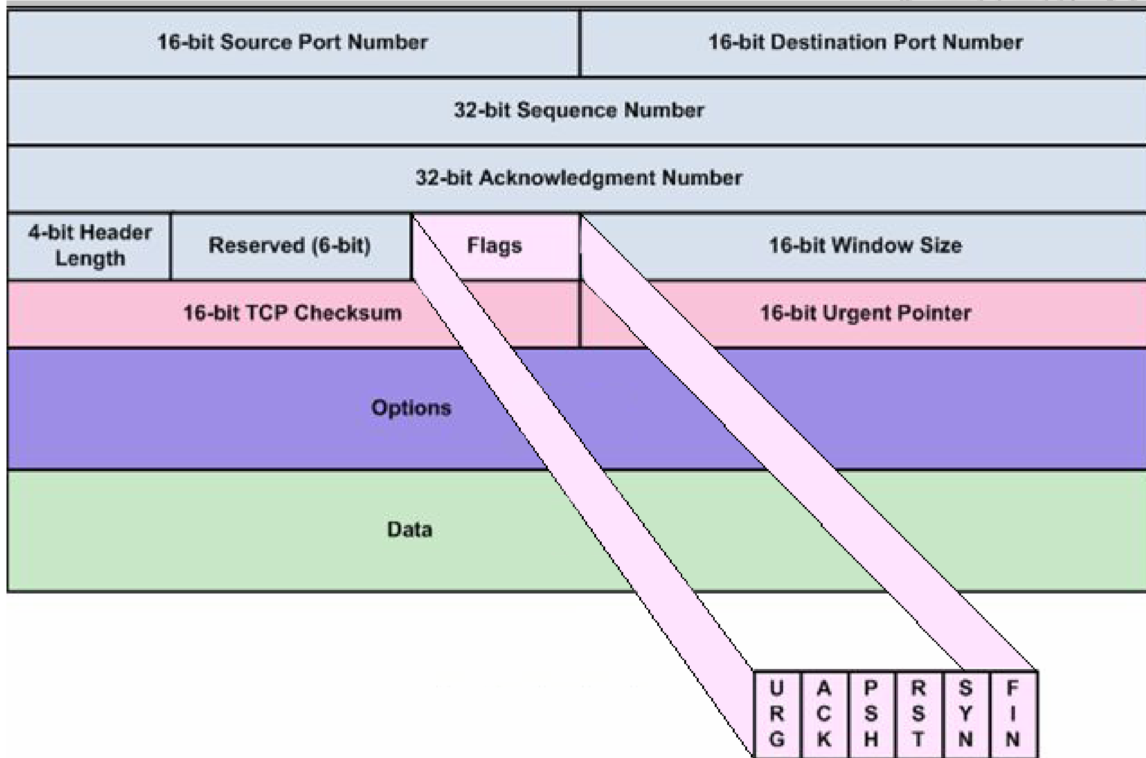


Figure 6-8.TCP header format.

In order to have a real paradigm of protocol functionality and mainly for the TCP/IP header, I analyze a *Voice Over IP* call that you can see in *Appendix A*.

6.3 Firewall Logging Implementation

Firewall logs reveal a lot of information on the nature of traffic coming in and going out of the firewall, allows you to plan your bandwidth requirement based on the bandwidth usage across the firewalls. Analyzing these firewall traffic logs is vital to understanding network and bandwidth usage and plays an important role in business risk assessment.

Firewall Analyzer uses the Linux syslog server (*syslogd*) that listens for exported firewall logs at the defined listener ports. The syslog server is a prerequisite for the right operation of Firewall Analyzer. The architecture of firewall log analyzer is shown in *figure 6-9*.

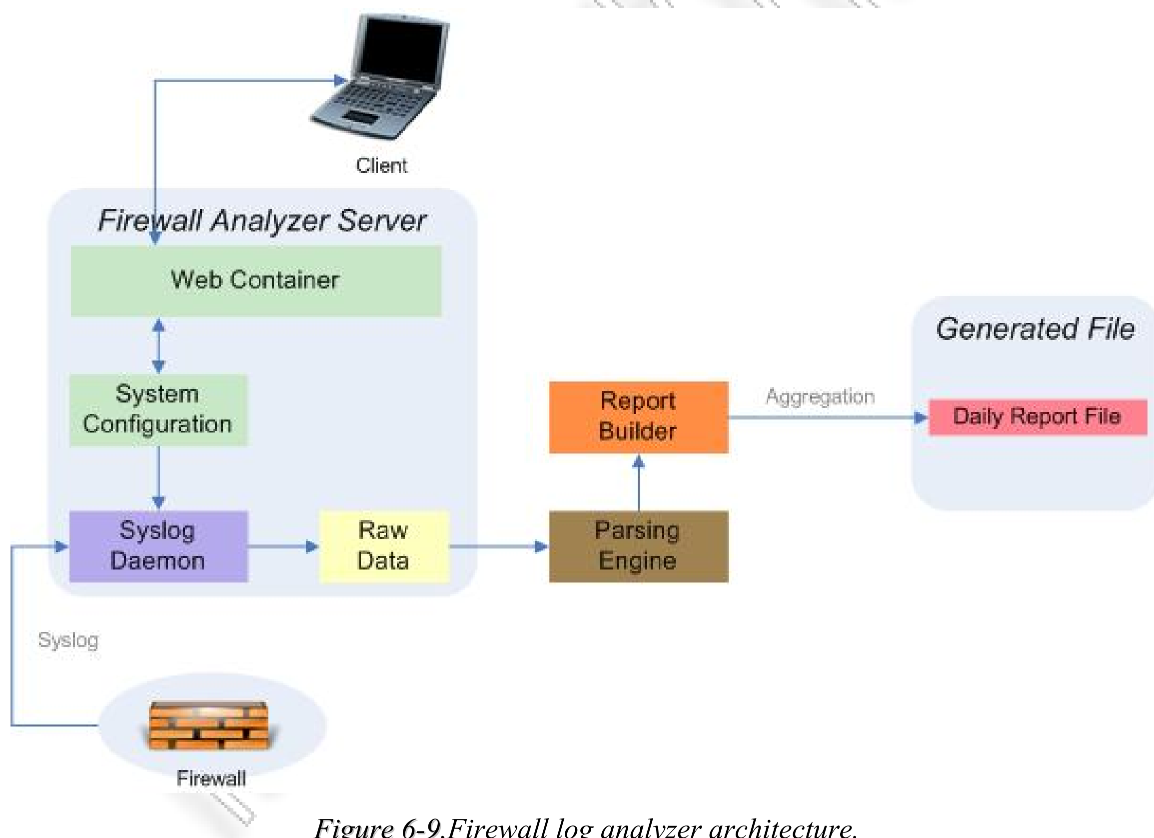


Figure 6-9.Firewall log analyzer architecture.

Firewall analyzer uses the Linux syslog daemon to store the firewall logs, and provides comprehensive reports on firewall traffic, security breaches, and more. This helps

network administrators to arrive at decisions on bandwidth management, monitor web site visits, audit traffic, and ensure appropriate usage of networks by employees.

The logic behind this implementation is exactly the same as the one that described in proxy log file implementation (*Section 5*). The actual implementation is much easier than the proxy log file, because the algorithm that implements the parsing is simpler. What you need is a sorting of information per source address. One can observe that the algorithm is generic and with few modifications are enough to parse any other log file (for example the *mail.log* file).

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

6.3.1 Pseudocode

Before the description of the pseudocode, I try to make a scheme of how to develop the parsing of the firewall log file. I mean that I make an initial guide of how to implement the algorithm. The logic behind the actual implementation is depicted in *figure 6-10*:

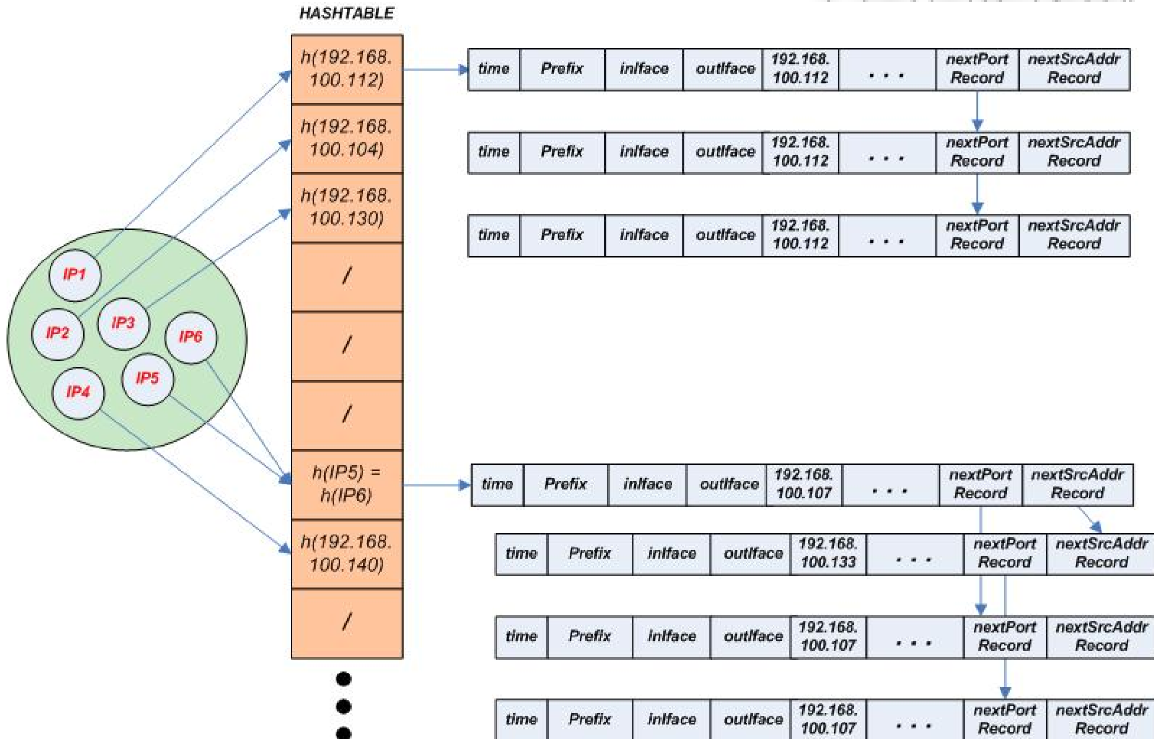


Figure 6-10.An example of the logic behind the algorithm.

The description of the above figure is provided in *Section 5*, in a more general manner. Following, is presented the next step that is the pseudocode.

############## Start Pseudocode ##############

Create a pseudorandom hash function that coerces the source IP address (the key) into a permuted integer through a series of bit operations (Lever, 200).

Read the *thorax.conf* file that contains the specified paths.

Open the directory that contains the *firewall.log* file, check the permissions and read each entry.

If (the log file is a file and the suffix is "*.log*" that has suffix length 4) then open the *firewall.log* file and read the next input line from the *firewall.log* file into a character array buffer.

Define a structure for linked list elements and pointers to this structure that handle the ideal case (no collisions) and the case that we have collisions:

```
/* Define a structure for linked list elements. */
typedef struct sListElement {
        char date[MIN_BUFFER_LEN];
        char prefix[MIN_BUFFER_LEN];
        char in[10];
        char out[10];
        char macAddress[MIN_BUFFER_LEN];
        char srcIP[19];
        char dstIP[19];
        char totalLength[MIN_BUFFER_LEN];
        char priority[10];
        char precedence[10];
        char hops[6];
        char uniqueId[15];
        char df[2];
        char protocol[15];
        char sourcePort[15];
        char dstPort[15];
        char seqNum[10];
```


ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

```
        /* Vertical expansion of list, if we have same ip and different URL */
        struct sListElement* nextPortRecord;

        /* Horizontal expansion of list in order to avoid collisions */
        struct sListElement* nextSrcAddrRecord;
} listElement;

listElement* startSrcAddr[HASH_TABLE_SIZE];
listElement* nextSrcAddr[HASH_TABLE_SIZE];
listElement* startPort[HASH_TABLE_SIZE];
listElement* nextPort[HASH_TABLE_SIZE];
listElement* hashTable;
```

Get each record in the *firewall.log* file and separate it into tokens. Then set the *key*, the *date*, the *log prefix,* the *IN* interface, the *OUT* interface, the *MAC* address, the source IP (*SRC*) and the destination IP (*DST*), the length of the packet (*LEN*), the type of service (*TOS*), the *PREC* flag, the *TTL*, *ID*, *DF* flag, *PROTO*, source port (*SPT*) and destination port (*DPT*) and sequence number (*SEQ*) into the parsing algorithm and keep this information in memory.

In order to set the data into memory, create the following algorithm:

```
If (nextPort[slot] == NULL) {
        //the defined slot in the hashtable is empty
        Create and fill the structure for the first time in the defined slot;
} else if (nextPort[slot] != NULL) {
        //the defined slot has already filled
        //Check the source address for collisions
        if (SRC is the same with the existing in the slot) {
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

```
        Create a new Vertical node in the same slot;
    } else if (SRC is different with the existing in the slot) {
        //We have collisions
        Create a new horizontal list;
        if (nextSrcAddr[slot] == NULL) {
            Create a new list due to collisions;
        } else if (nextSrcAddr[slot] != NULL)  {
            //Check the source IP address
            if (SRC is the same with the existing in the slot) {
                    Create a new vertical node to the current list;
            } else if (different SRC to the same slot) {
                Search the list in order to find a matching IP address;
                while (search the list until will be null) {
                    if (match a client address) {
                        break;
                    }
                }
                if (did not pass into the while loop) {
                    Insert new additional list;
                } else {
                    //a matching IP was found. Passed through the
                    //while loop.
                    if (did not pass through the loop) {
                        insert new node to the current
                        additional list;
                    }
                }
            }
        }
    }
```

}

When the parsing of the *firewall.log* file has been completed and all the data resides into the memory, create a function that will be able to manipulate this information and to exclude the data into a file. The name of the generated file will be the current date (that will be taken from the system time).

```
For (go through the hash table and check each slot) {
        if (nextPort[slot] != NULL) {
                go to start of the list;
                while(nextPort[slot] != NULL) {
                        write the information contained in the slot into the file;
                }
                while (nextSrcAddr[slot] != NULL) {
                        write the information contained in the slot into the file;
                }
        }
}
make the generated file read only;
```

Moreover, create a function that will be able to manipulate this information and to export the data in HTML format.

############## End of Pseudocode ##############

6.2 Firewall Log Analyzer Installation

The requirements are exactly the same with those of Proxy log analyzer in *Section 4.2*. First of all, this application is written in C programming language (ANSI) to be extremely fast and highly portable by using the Eclipse IDE version 3.1.2. The compiler is: *GCC version 4.1.1*. Moreover, my working computer is a Sony Vaio laptop with CPU Intel Pentium M 1.50 Ghz, HDD 60GB, RAM 512 MB and the operating system that is *Linux Fedora Core 5*.

In order to be able to run the existing application in your computer you must do the following steps:

- ✓ Login as root in a Linux distribution and open a console.
- ✓ Make a directory named “firewall” under the /root/thorax path (/root/thorax/firewall). In the new directory, create another directory named “userData” (/root/thorax/firewall/userData). In this directory (/root/thorax/firewall/userData), you will have the generated file that includes the needed information from the firewall log file. The filename comes from the current date (the exact date that you run the application). The format of the date is: YYYY-MM-DD.hh.mm. So, a sample file will have the following format: “2006-08-31.00.00”. Note that you need to have the computer date and time synchronized with the <<real>> date and time if you want to have the name of the generated file the same with the current date.
- ✓ Extract the attached tarball file (firewall.tar.gz). In the generated folder named “firewall” you can see the source code (fully comented).
- ✓ Put the “thorax.conf” file that you must take from the above generated folder (firewall) in the /root/thorax/firewall path (/root/thorax/firewall/thorax.conf).
- ✓ Moreover, put the firewall.log file under the /root/thorax/firewall path (/root/thorax/firewall/firewall.log).
- ✓ Go into the “firewall” directory and run the following command: make

- ✓ After the above steps, you can go to the /root/thorax/firewall/userData path and you can open the generated file with a text editor (for example *joe*). Note that any text editor can be used in place of *joe* (gedit, vi, etc). The generated file is read only. So, you cannot change it. You can see the results with the less command (for example: *less 2006-09-04.21.24*), or with the joe editor (*joe 2006-09-04.21.24*).
- ✓ That's all!Have fun!

6.2.1 For the Impatiens

After you log on in a Linux-based distribution as root, put the attached tarball file *"firewall.tar.gz"* and the *"firewall.install"* file into the same directory. Open a terminal as root and give the command: firewall*.install* (See *figure 6-11* below). Then, the above described steps will be done automatically. Note that in the tarball file, there exists a sample *firewall.log* file for testing purposes. Also, in the extracted folder named *"firewall"* you can see the source code (fully commented) and the *Makefile*.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

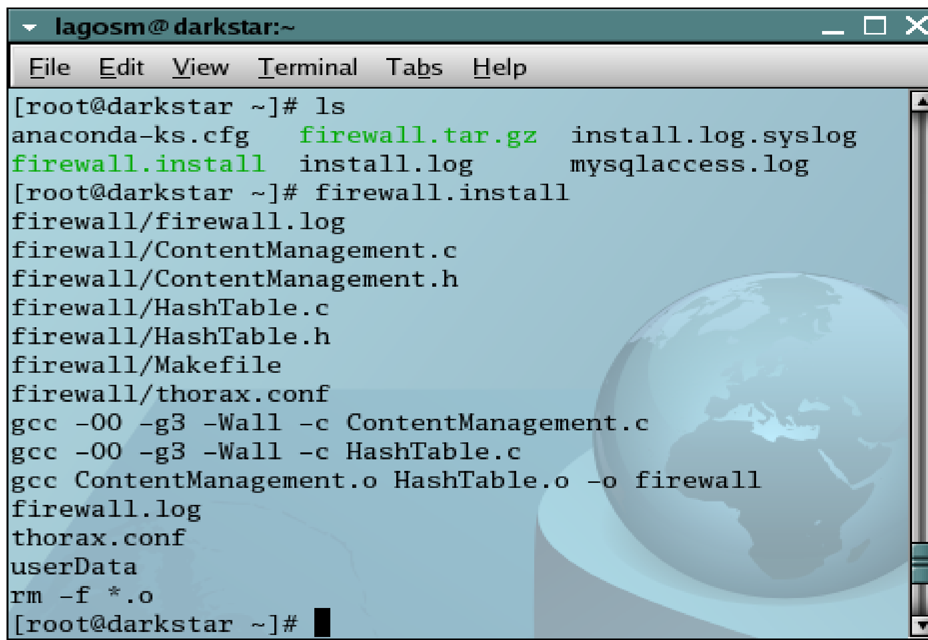


Figure 6-11.Firewall log Analyzer Installation.

In *figure 6-12* that follows, one can see a sample screenshot after running the above steps and the file with users data has already generated (by giving the command: *less file_name*):

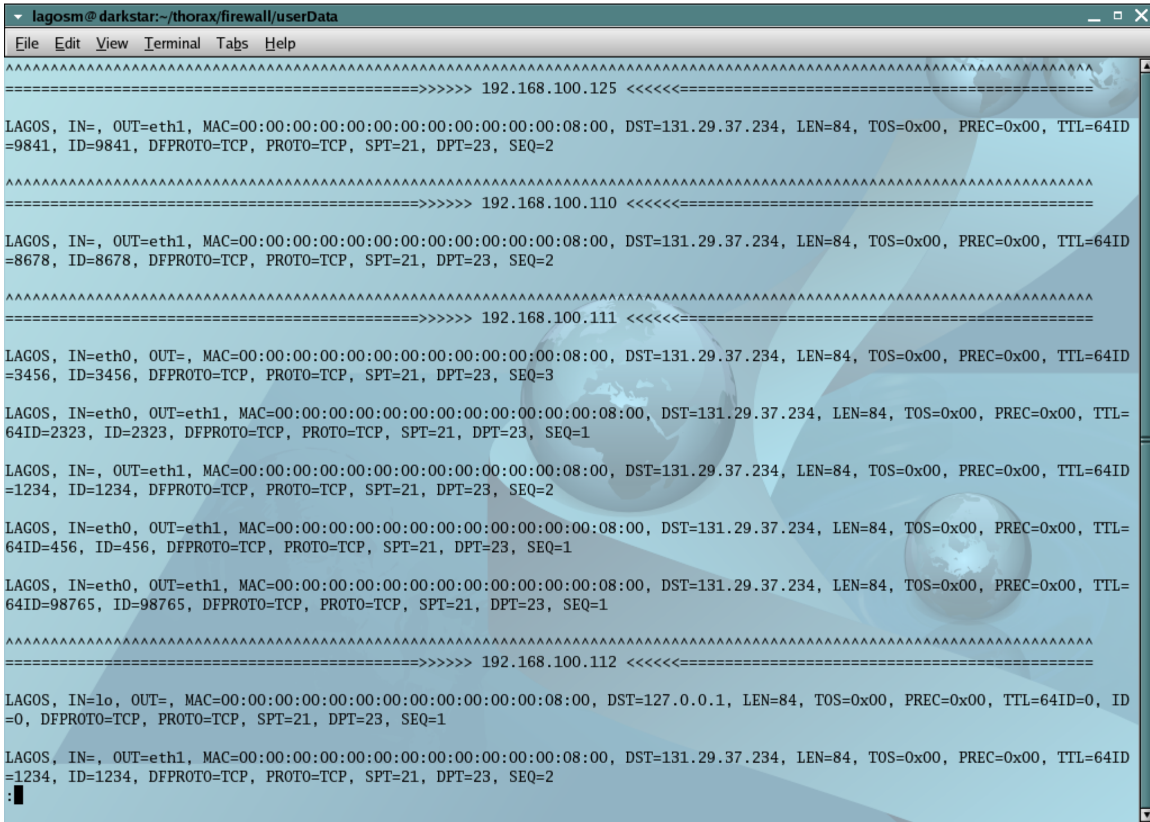


Figure 6-12.Sample screenshot with generated user's data.

Appendixes

A. TCP/IP Example in a Real Paradigm

In, *figure 6-13,* is depicted a screenshot from a network sniffer called *Wireshark* that captures a call that is generated from an *MGCP* (Media Gateway Protocol) endpoint, and via an *MGCP Signaling Manager* to a soft switch in order to route and translate the call to another end point, via the corresponding signaling manager (depends on the type of protocol – *ISUP*, *SIP*, *H.323*, *MEGACO*). More specifically, is a part of the call and not all frames. It is just an example from a real environment. Moreover, *figure 6-14* shows the initiation frame of the *MGCP* endpoint. Also, *figure 6-15* shows the IP header of a TCP frame and *figure 6-16* the corresponding TCP header of the same TCP frame.

CQ110958_MTA.cap - Wireshark

File Edit View Go Capture Analyze Statistics Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.224.1.9	10.224.6.71	MGCP	AUEP 89093 aaln/1@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
2	0.099643	10.224.1.9	10.224.6.70	MGCP	AUEP 89192 aaln/1@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
3	0.099825	10.224.6.19	10.224.1.9	ICMP	Destination unreachable (Host unreachable)
4	0.149875	10.224.1.9	10.224.6.71	MGCP	AUEP 89092 aaln/2@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
5	0.179403	10.224.1.9	10.224.6.70	MGCP	AUEP 89195 aaln/2@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
6	0.801554	10.224.6.17	224.0.0.18	VRRP	Announcement (v2)
7	0.880895	10.224.1.9	10.224.6.71	MGCP	AUEP 89095 aaln/1@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
8	0.881141	10.224.6.19	10.224.1.9	ICMP	Destination unreachable (Host unreachable)
9	0.890478	10.224.1.9	10.224.6.71	MGCP	AUEP 89090 aaln/2@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
10	0.890568	10.224.6.19	10.224.1.9	ICMP	Destination unreachable (Host unreachable)
11	0.939543	10.224.1.9	10.224.6.70	MGCP	AUEP 89195 aaln/2@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
12	0.969452	10.224.1.9	10.224.6.70	MGCP	AUEP 89194 aaln/1@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
13	1.219810	10.224.1.9	10.224.6.71	MGCP	AUEP 89094 aaln/2@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
14	1.220034	10.224.6.19	10.224.1.9	ICMP	Destination unreachable (Host unreachable)
15	1.329992	10.224.1.9	10.224.6.70	MGCP	AUEP 89193 aaln/2@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
16	1.399875	10.224.1.9	10.224.6.71	MGCP	AUEP 89095 aaln/1@0013117fcfe7.arris.mta.net MGCP 1.0 NCS 1.
17	1.569756	10.224.1.9	10.224.6.70	MGCP	AUEP 89190 aaln/1@0013117fd023.arris.mta.net MGCP 1.0 NCS 1.
18	1.570009	10.224.6.19	10.224.1.9	ICMP	Destination unreachable (Host unreachable)
19	1.658086	10.224.6.3	10.224.1.14	TCP	pktcable-cops > 52489 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0

```
Frame 1 (113 bytes on wire, 113 bytes captured)
Ethernet II, Src: Unispher_41:5b:3d (00:90:1a:41:5b:3d), Dst: ArrisInt_ab:53:85 (00:00:ca:ab:53:85)
802.1Q Virtual LAN
Internet Protocol, Src: 10.224.1.9 (10.224.1.9), Dst: 10.224.6.71 (10.224.6.71)
User Datagram Protocol, Src Port: mgcp-callagent (2727), Dst Port: mgcp-gateway (2427)
Media Gateway Control Protocol
```

```
0010  08 00 45 00 00 5f cb 6e  40 00 fc 11 96 0f 0a e0   ..E.._.n @.......
0020  01 09 0a e0 06 47 0a a7  09 7b 00 4b c0 6c 41 55   .....G.. .{.K.lAU
0030  45 50 20 38 39 30 39 33  20 61 61 6c 6e 2f 31 40   EP 89093  aaln/1@
0040  30 30 31 33 31 31 37 66  63 66 65 37 2e 61 72 72   0013117f cfe7.arr
```

Internet Protocol (ip), 20 bytes | P: 513 D: 513 M: 0

Figure 6-13.Frames analysis paradigm from a VoIP scenario.

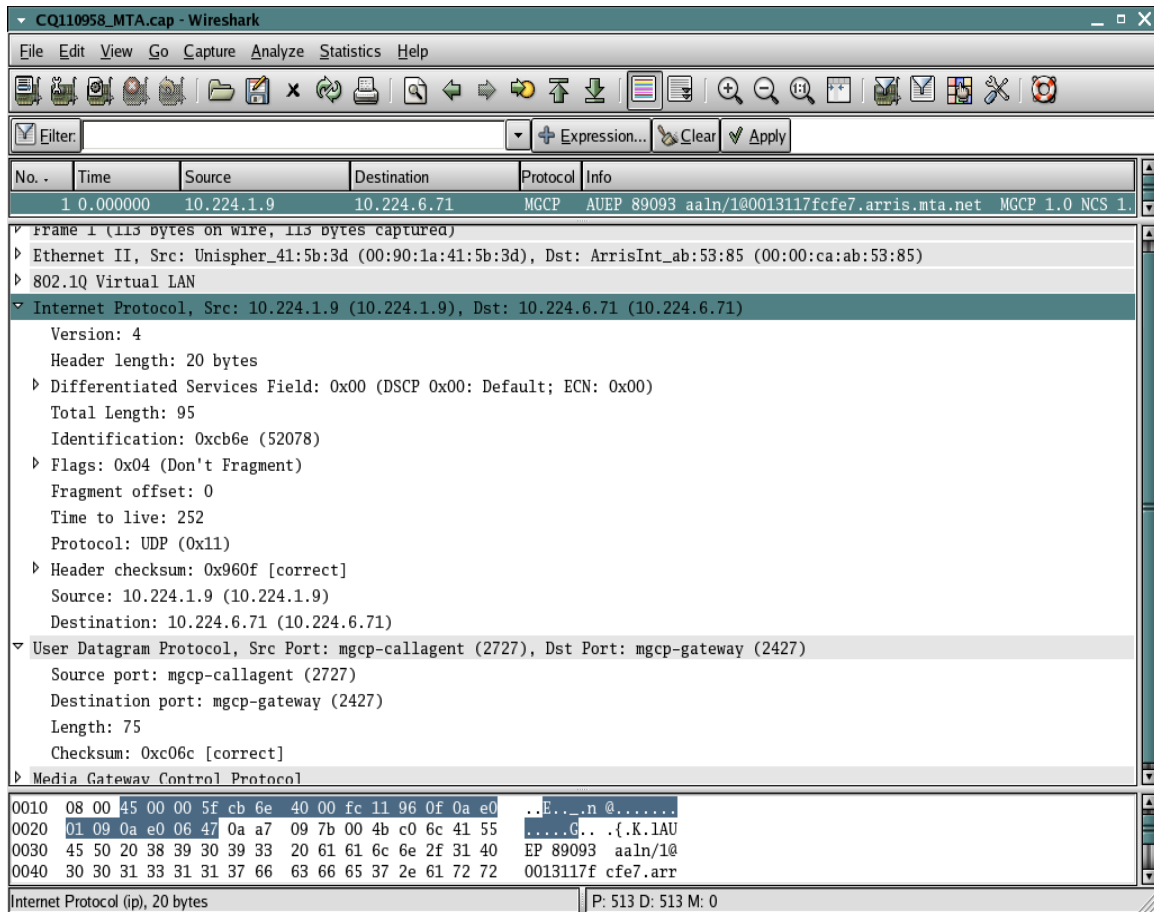


Figure 6-14.MGCP initiation frame IP header format.

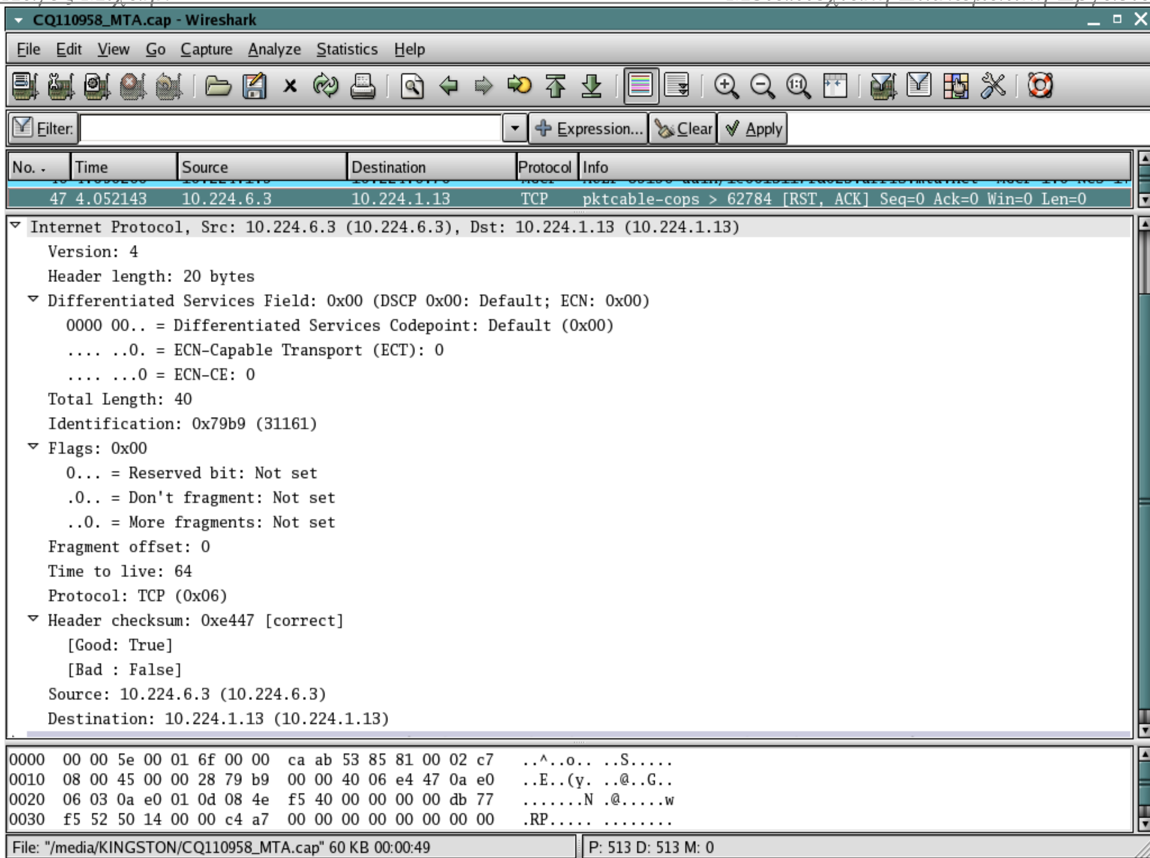


Figure 6-15.TCP frame IP header format.

```
CQ110958_MTA.cap - Wireshark
File  Edit  View  Go  Capture  Analyze  Statistics  Help
Filter:                                   Expression...  Clear  Apply

No. .  Time      Source       Destination   Protocol  Info
  47 4.052143   10.224.6.3   10.224.1.13   TCP       pktcable-cops > 62784 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0

▷ Frame 47 (64 bytes on wire, 64 bytes captured)
▷ Ethernet II, Src: ArrisInt_ab:53:85 (00:00:ca:ab:53:85), Dst: IETF-VRRP-virtual-router-VRID_6f (00:00:5e:00:01:6f)
▷ 802.1Q Virtual LAN
▷ Internet Protocol, Src: 10.224.6.3 (10.224.6.3), Dst: 10.224.1.13 (10.224.1.13)
▽ Transmission Control Protocol, Src Port: pktcable-cops (2126), Dst Port: 62784 (62784), Seq: 0, Ack: 0, Len: 0
    Source port: pktcable-cops (2126)
    Destination port: 62784 (62784)
    Sequence number: 0    (relative sequence number)
    Acknowledgement number: 0    (relative ack number)
    Header length: 20 bytes
  ▽ Flags: 0x0014 (RST, ACK)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...1 .... = Acknowledgment: Set
      .... 0... = Push: Not set
      .... .1.. = Reset: Set
      .... ..0. = Syn: Not set
      .... ...0 = Fin: Not set
    Window size: 0
    Checksum: 0xc4a7 [correct]

0010  08 00 45 00 00 28 79 b9  00 00 40 06 e4 47 0a e0   ..E..(y. ..@..G..
0020  06 03 0a e0 01 0d 08 4e  f5 40 00 00 00 00 db 77   .......N .@.....w
0030  f5 52 50 14 00 00 c4 a7  00 00 00 00 00 00 00 00   .RP..... ........

Flags (tcp.flags), 1 byte        P: 513 D: 513 M: 0
```

Figure 6-16.TCP frame TCP header format.

B. Proxy Log Analyzer Code

The source code for the proxy log analyzer is attached in the CD in a folder named proxyLogAnalyzer. Also, is attached a script for the rotation of the squid in a daily basis, as described in *Section 5*.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

C. Firewall Log Analyzer Code

The source code for the firewall log analyzer is attached in the CD in a folder named firewallLogAnalyzer. Moreover, is attached a script with iptables rules configuration for the firewall architecture that described in *Section 2.2*.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

References

Andreasson, O. (2005) *Iptables Tutorial 1.2.0*. Accessed 25/04/2006.
http://iptables-tutorial.frozentux.net/iptables-tutorial.html

Bamdel, D. (2001) *Taming the Wild Netfilter*. Accessed 25/04/2006.
http://www2.linuxjournal.com/article/4815

Brockmeier, J. (2001) *Filtering Packets with* `iptables`. Accessed 25/03/2006.
http://www.unixreview.com/documents/s=1237/urm0103c/0103c.htm

Brockmeier, J. (2001) *Using* `iptables`. Accessed 25/03/2006.
http://www.unixreview.com/documents/s=1236/urm0104l/0104l.htm

Chunyan, X. S. (2004) *Simple IPTables Tips*. Acecssed 12/05/2006.
http://www.nus.edu.sg/comcen/security/newsletter/Aug2004/simple_iptables_tips.htm

Coulson, D. (2001) *Mastering IPTables*. Accessed 26/04/2006.
http://davidcoulson.net/writing/lxf/14/iptables.pdf

Coulson, D. (2003) *Network Security IPTables*. Accessed 26/04/2006.
http://www.davidcoulson.net/writing/lxf/38/iptables.pdf

Devi, L. (2004) *Hash Tables*. Accessed 27/02/2006.
http://www.cs.unc.edu/~plaisted/comp122/13-hashing.ppt

Doherty, N. F. & Fulford, H. (2005) Aligning the Information Security Policy with the Strategic Information Systems Plan. *Computers & Security*, 25(1), pp. 55-63, 2006.

Eisermann, M. (2002) *Performance tests with the Microsoft Internet Security and Acceleration (ISA) Server*. Accessed 15/06/2006.
http://www.webperformanceinc.com/library/files/proxy_server_performance.pdf

Fox, T. (2004) *squid Proxy Server Configuration*. Accessed 15/06/2006.
http://www.linuxheadquarters.com/howto/networking/squid.shtml

Garfinkel, S., Schwartz, A. & Spafford, G. (2003) *Practical UNIX & Internet Security*. 3rd ed. United States of America: O'Reilly & Associates, Inc.

Hall, A. E. (2000) *Internet Core Protocols: The Definitive Guide*. United States of America: O'Reilly & Associates, Inc.

Jenkins, J. R. *Hash Functions for Hash Table Lookup*. Accessed 23/02/2006.
http://www.burtleburtle.net/bob/hash/evahash.html

Kenshi, P. *Help File Library: Iptables Basics*. Accessed 26/04/2006.
http://www.justlinux.com/nhf/Security/IPtables_Basics.html

Kernighan, W. B. & Ritchie, M. D. (1998) *The Ansi C Programming Language*. 2nd ed. Pearson Professional Education.

Kirch, O. & Dawson, T. (2000) *Linux Network Administrator's Guide*. 2nd ed. United States of America: O'Reilly & Associates, Inc.

Lever, C. (2000) *Linux Kernel Hash Table Behavior: Analysis and Improvements*. Accessed 24/02/2006.
http://www.citi.umich.edu/techreports/reports/citi-tr-00-1.pdf

Li, Y. (2002) *The Double NAT MINI-HOWTO*. Accessed 18/01/2006.

http://www.netfilter.org/documentation/HOWTO//netfilter-double-nat-HOWTO.html

Lilliam, K. P. (2005) *A simple guide to creating a firewall and squid proxy server*. Accessed 18/04/2006. https://www.redhat.com/apps/reseller_catalog/marketing/easy/network_services_linuxiseasy.pdf

Loudon, K. (1999) *Mastering Algorithms with C*. United States of America: O'Reilly & Associates, Inc.

Marie, F. *Netfilter Extensions HOWTO*. Accessed 18/01/2006. http://www.netfilter.org/documentation/HOWTO//netfilter-extensions-HOWTO.html

Messier, M. & Viega, J. (2003) *Secure Programming Cookbook: for C and C++*. United States of America: O'Reilly & Associates, Inc.

Newham, C. & Rosenblatt, B. (1998) *UNIX Shell Programming: Learning the Bash Shell*. 2nd ed. United States of America: O'Reilly & Associates, Inc.

Newstrom, H. (2002) *Using Linux Scripts to Monitor Security*. Accessed: 27/12/2005. http://www.sans.org/reading_room/whitepapers/linux/197.php

Oualline, S. (1997) *Practical C Programming*. 3rd ed. United States of America: O'Reilly & Associates, Inc.

Parlante, N. (2001) *Linked Lists Basics*. Accessed 23-02/2006. http://cslibrary.stanford.edu/103/LinkedListBasics.pdf

Prata, S. (2004) *C Primer Plus*. 5th ed. United States of America: Sams Publishing.

Russel, R. (2002) *Linux 2.4 Packet filtering HOWTO*. Accessed 18/01/2006.
http://www.netfilter.org/documentation/HOWTO//packet-filtering-HOWTO.html

Russel, R. (2001) *Linux Networking-Concepts HOWTO*. Accessed 18/01/2006.
http://www.netfilter.org/documentation/HOWTO//networking-concepts-HOWTO.html

Russel, R. (2002) *Linux 2.4 NAT HOWTO*. Accessed 18/01/2006.
http://www.netfilter.org/documentation/HOWTO//NAT-HOWTO.html

Russel, R. & Welte H. (2002) *Linux netfilter Hacking HOWTO*. Accessed 16/01/2006.
http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html

Shinn, M. & Shinn, S. (2005) *Troubleshooting Linux Firewalls*. Hagerston: Addison Wesley.

Stephens, J. *Iptables*. Accessed 23/04/2006.
http://www.sns.ias.edu/~jns/wp/iptables/

Song, H. et al. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. (2005) *In ACM sigcomm*, Philadelphia, Pennsylvania.

Spenneberg, R. (2005) *Analysis Tools for Firewall Logfiles: For the Record*. Accessed 15/05/2006.
https://www.linux-magazine.com/issue/50/Firewall_Logfile_Analyzers.pdf

Stevens, R. W., Fener, B. & Rudoff, M. A. (2003) *UNIX Network Programming: The Sockets Networking API*. 3rd ed. Addison Wesley.

Strebe, M. (2004) Network *Security Foundations*. United States of America: Sybex Inc.

Unknown. (2005) *Dictionaries*. Accessed 22/01/2006.
http://ww3.algorithmdesign.net/handouts/HashTables.pdf

Unknown. (2003) *Firewall Logging & Monitoring*. Accessed 18/04/2006.
http://www.loganalysis.org/sections/parsing/application-specific/firewall-logging.html

Unknown. *IP Datagram General Format*. Accessed 15/05/2006.
http://www.tcpipguide.com/free/t_IPDatagramGeneralFormat.htm

Unknown. *IPTABLES quick HOWTO*. Accessed 28/04/2006.
http://www.cse.msu.edu/~minutsil/iptables.html

Unknown. *Linked Lists, Trees, Hash Tables (Data Structures)*. Accessed 24/02/2006.
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial/lists.html

Unknown. *Netfilter Log Format*. Accessed 28/04/2006.
http://logi.cc/linux/netfilter-log-format.php3

Unknown. *Quick HOWTO : Ch14 : Linux Firewalls Using iptables*. Accessed 28/04/2006.
http://www.linuxhomenetworking.com/wiki/index.php/Quick_HOWTO_:_Ch14_:_Linux_Firewalls_Using_iptables

Unknown. *Using iptables*. Accessed 29/04/2006.
http://www.linuxguruz.com/iptables/howto/iptables-HOWTO-6.html

Wessels, D. (2004) *Squid: The Definitive Guide*. United States of America: O'Reilly & Associates, Inc.

Zwicky, D. E., Cooper, S. & Chapman, B. D. (2000) *Building Internet Firewalls*. 2nd ed. United States of America: O'Reilly & Associates, Inc.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

M I C H A I L   L A G O S

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ