Global Information Assurance Certification Paper

Iptables, Linux and You!
GSEC Practical Assignment v.1.4b (Option 1)

Woody Hughes
March 3, 2003

1. Summary

This document is intended to provide an analysis of **iptables**, the popular Open Source firewall, and I will attempt to discuss its developmental roots, namely Open Source, its internal workings within the kernel level, and the netfilter architecture. Network Address Translation (NAT), and connection tracking within netfilter will be discussed as well. Furthermore, it's backend requirements in a small office / home office setting will be touched upon briefly.

At the conclusion of this paper, the reader will have a concrete understanding of iptables, how it's theoretically implemented in a SOHO environment, and be armed with a basic understanding of why our "networked society" needs secure perimeter solutions to the entrance of their business.

2. An Introduction to Open Source Solutions

The whole "Linux" debate is rather interesting. And that's not because it's the big buzzword these days, but because of the way a person often responds when another person begins to defend its particular roots. It's not uncommon for one to witness rolling eyes, raised eyebrows, and a reluctance to pursue anything remotely related to Open Source.

Matter of fact, it's almost evil. So evil in fact, that The Register, a popular online news site, quoted Steve Ballmer, the CEO of Microsoft, as saying that, "Linux is a cancer that attaches itself in an intellectual property sense to everything it touches."[1]

But Open Source hasn't had a chance to really prove itself. Maybe it's the reluctance by large enterprises that has spurred this "evil" commentary. Or maybe it's simply the reluctance of individuals to learn new solutions, rather than relying on canned commercial alternatives.

Regardless, for the home user, expensive commercial options in the security industry need not apply. Most security professionals employing home-based solutions will look into various Open Source firewall solutions that will offer what I believe are four key features:

- Connection tracking.

[1] Greene, Thomas. "Ballmer: Linux is a cancer." The Register. 6 February 2001. URL: http://www.theregister.co.uk/content/4/19396.html (20 Feb. 2003).

- Packet filtering.
- Special protocols such as FTP and various gaming protocols.
- Network Address Translation.

In my opinion, these are the four main areas that are the most important for a home user and their perimeter security.

3. Why Iptables?

Iptables is currently one of many firewall solutions that successfully applies these four key areas. There are more alternatives in the Open Source arena than simply iptables and each product has its own advantage and disadvantage. However, defining each advantage and disadvantage of every particular firewall solution is beyond the scope of this paper.

The decision to choose iptables is a no-brainer at best. The Linux 2.4 kernel employs the *netfilter* architecture*,* a packet filtering architecture, described later in this paper, that iptables influences. It packages with every Linux distribution that supports the 2.4 kernel, and its functionality is already compiled into most stock kernels these days anyway. All it takes is an additional selection of features that need to be enabled in the kernel, and you're ready to roll.

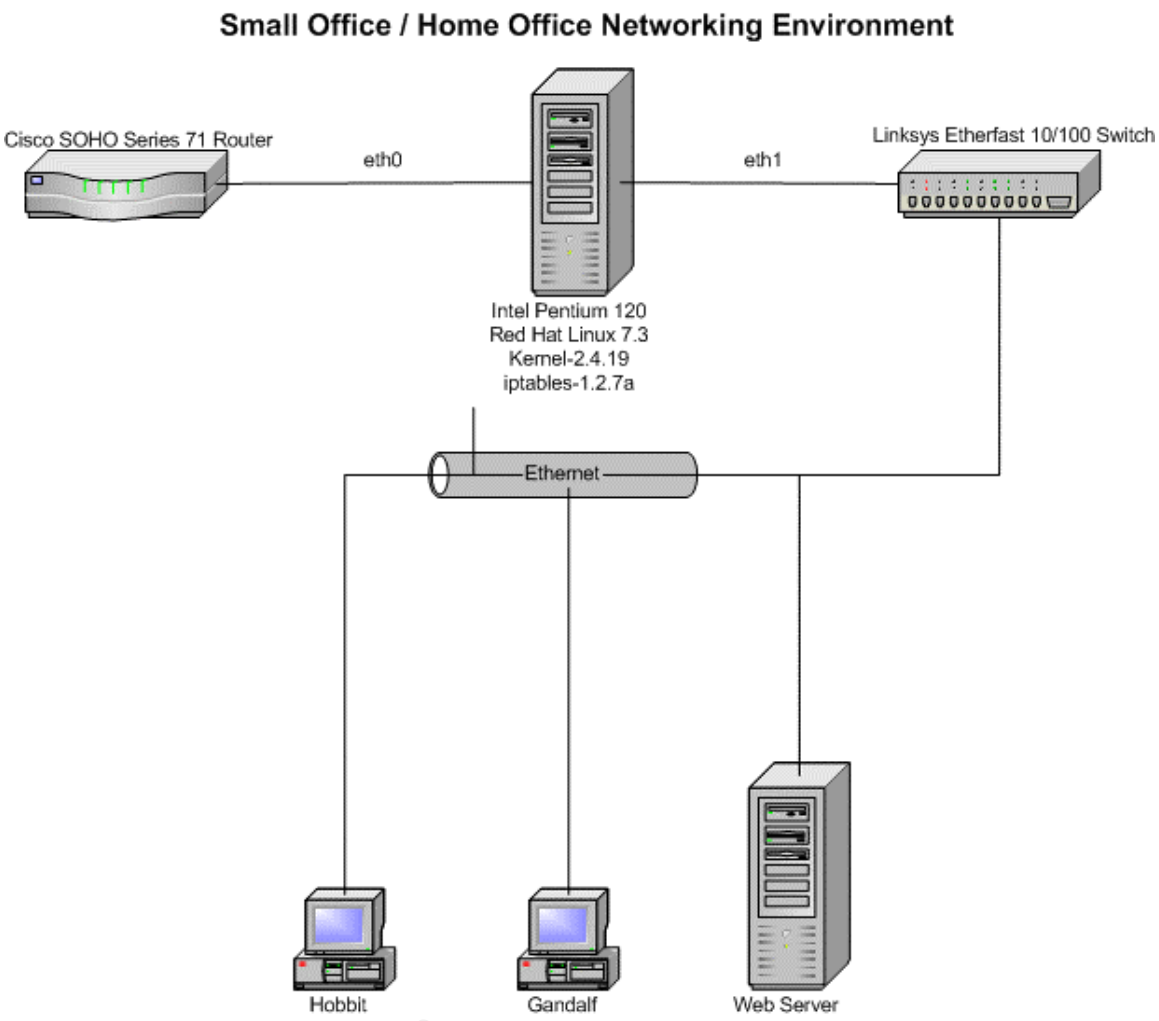
Furthermore, most people who have used its predecessor, **ipchains**, will have no trouble getting used to its slight change in syntax and its new features. For this reason, most security professionals will move to iptables and leave the older ipchains behind.

4. The Typical SOHO Environment

Like its predecessor's, ipchains and **ipfwadm**, running a Linux firewall doesn't require much of an overhead in system resources because of the strong reliance on kernel hooks. For this reason, and since the Linux kernel is efficient, most home users can get by with using a very low-end machine for their packet filtering needs.

As an example, I've worked with clients in small business situations implementing iptables on a Pentium platform. As a matter of fact, it was a Pentium 120! The client in question was on a shoestring budget, and happened to have an old system lying around. I quickly scooped the machine up, installed Red Hat Linux 7.3 on it, and configured iptables appropriately.

The following is a picture of what the typical SOHO setup and it's corresponding network segments represent:



As you can see, a typical small office setup would consist of a router, a firewall, a switch, and finally, the related networked hardware attached to the switch.

Often, I find that clients on a real shoestring budget may actually combine the Linux server into a router/firewall combo. The router portion of the server, in my opinion, doesn't act as a true router, but its functionality is satisfactory for a small office.

5. Server Security and Background Processes

One of the most important decisions that a person will make when building a custom Linux firewall server is what they explicitly enable and disable. From a strict security perspective, it's imperative that the individual involved in securing the server do so with a "whatever isn't needed, is explicitly denied" approach.

As Michael Warfield wrote[2], “You must strike the right balance in terms of who is allowed to do what, and when.”

Mr. Warfield goes on to say that securing a Linux box should be based on a couple principles, with one being the principle of minimum access, or, "That which is not explicitly permitted is denied." Furthermore he states, “This is one of the basic mantras of firewall design. It applies equally well to security in general.”

I couldn’t agree more.

5.1. Essential Services

When it comes to the actual services that run on our server, the question that most people will ask is, “Which ones must be activated?” You might think that some services must be enabled for the machine to run, and you’d be absolutely correct. The machine must operate and without certain services running, the server will effectively become worthless.

But securing a Linux server that’s acting as a firewall isn’t much different from writing the actual rules for the firewall. I think diligence comes into play here—essentially making sure that the correct services are allowed to operate, as this server will act as the front man to the entire environment. The server that the firewall resides upon must be locked down to the fullest extent.

The following services should be the only services running on a Red Hat Linux 7.3 server that’s acting as a firewall:

```
init
keventd
ksoftirqd_CPU0
kswapd
bdflush
kupdated
/sbin/pump -i eth0
syslogd -m 0
klogd -2
xinetd -stayalive -reuse -pidfile /var/run/xinetd.pid
crond
/sbin/mingetty tty2
/sbin/mingetty tty1
/usr/sbin/sshd
```

The first question that comes to mind after reading this list is why do I have the **sshd** service running? The reason is simple—to allow the security or systems administrator to log in securely via an encrypted tunnel using Secure Shell.

[2] Warfield, Michael. Securing Linux, Part 1. 28 May 1999. URL: http://www.linuxworld.com/linuxworld/lw-1999-05/lw-05-ramparts.html (23 Feb. 2003).

However, I can take a step forward with all of this and ask myself if I really want to allow an avenue of compromise. Recently, SSH has come under scrutiny within the security industry because of vulnerabilities that have surfaced, primarily in the SSH version 1 protocol. It's also well known that ettercap, a popular sniffing utility can actually sniff version 1 traffic[3].

In the end, I'd want to remove SSH from the list of "essential services". To me, SSH is a "convenience" service and should probably be removed.

So, going forward, this leaves the system with the only service that is absolutely necessary for it to operate efficiently and with full system logging and **cron** job capabilities.

In this example, I've also allowed **xinetd** to run, so I must peruse the **/etc/xinetd.d** directory and make sure that each and every file contained in that directory has the '*disable=yes*' line present. This will effectively disable the corresponding service at startup.

Another important consideration when securing a Linux firewall is redundant logging. Firewall logs do indeed get large and a redundant logging server should have plenty of space with custom log rotation mechanisms in place. If redundant logging is important to you and you want to add that additional security, then you can enable it via the **/etc/syslog.conf** file.

For example, the line:

```
*            @longfish
```

allows us to log everything to our remote logging server, longfish. Our logging server, longfish, would have to have the syslog daemon running with the **–r** option to accept external connections to the syslog daemon.

5.2. Kernel Options

Since the firewall machine is now secured, I must make sure that the latest kernel is downloaded and the correct options are enabled within the kernel so that I have every possible scenario covered from an iptables point of view. In effect, I want to be able to tailor the kernel to my needs since I'm customizing and tweaking the firewall server to begin with.

With this mindset in place, it comes as no surprise that I'd want to compile everything related to netfilter in the kernel. "Network Packet Filtering" must be enabled within the kernel, and subsequently, I must select the "Netfilter Configuration" option to drill down to our available kernel options.

[3] URL: http://ettercap.sourceforge.net (28 Feb. 2003).

I've always recommended to my associates that they configure everything into the kernel. If you notice, I said "into the kernel". I'm not much of a module person. I never have been. And I figure that if I'm going to build a secure firewall then I should also make sure it's completely stable. My past experiences with modules being properly loaded all the time in Linux leaves something to be desired. In my opinion, compiling module support is really just an efficiency issue when it comes to Linux anyway—and since I'm building a pure firewall that's hooked directly into the kernel, then enabling netfilter options as a module within the kernel isn't a concern.

With that said, the following options should be selected within the "Netfilter Configuration" section of the kernel menu:

```
Connection tracking (required for masq/NAT)
FTP protocol support
IRC protocol support
IP tables support (required for filtering/masq/NAT)
limit match support
MAC address match support
Packet type match support
netfilter MARK match support
Multiple port match support
TOS match support
ECN match support
DSCP match support
AH/ESP match support
LENGTH match support
TTL match support
tcpmss match support
Helper match support
Connection state match support
Connection tracking match support
Packet filtering
Full NAT
Packet mangling
LOG target support
ULOG target support
TCPMSS target support
ARP tables support
```

The reasoning behind compiling all of these options into the kernel is simple—I want to make sure that whatever I want to do with this firewall is available to me at a moment's notice. It's not so much an issue about modules and efficiency, rather than "availability".

Consider a situation where there's something seriously wrong with a web server in an office. A Systems Admin working in an office can't figure out the problem and money is being lost by the minute. However, a backup web server is located on another machine on the network. There is another problem, however. The "Full NAT" option isn't enabled in the kernel on the firewall, essentially nullifying any chance of quickly modifying packet headers and their respective destination

addresses using Destination NAT'ing. At this point, someone must log in via the console on the firewall server, select “Full NAT” from the list of Netfilter options, and spend the next thirty minutes recompiling the kernel. And remember, this just happens to be a Pentium 120! Guess how long the compile is going to take? Of course, the Systems Admin wouldn’t do that—he’d simply change the IP address on the backup web server to reflect the existing web server’s IP. But that’s not the point. That means that he’d have to bring the existing web server’s networking interface down, essentially dissolving any hopes of his consultant logging in remotely to fix whatever problem caused the whole mess to begin with in the first place! As far as he knows, his Apache process is down.

What’s my point? If everyone gets into the mindset of enabling everything that they need to in the kernel, then they’ll be ready from a security standpoint. And in that example, if Mr. Systems Admin at Large had everything enabled, he would’ve been able to mangle his incoming packets using NAT and redirect them to a different machine on a different port. Basically, the change could’ve been made in seconds—not minutes.

In the end, being ready for anything is one of the responsibilities of a firewall administrator—even if it’s not security-related and merely providing additional business flexibility.

6. Inside Iptables

6.1. Iptables vs. ipchains

Before I touch on the differences between iptables and ipchains, I must point out what they both have in common—the Linux kernel. The kernel essentially examines the packet header and at that point, relies on other mechanisms within it to decide what exactly to do with that packet. This is called *packet filtering* and this entire mechanism, other than the actual userspace application itself, is built into the kernel.[4]

The major difference between ipchains (kernel 2.2) and iptables (kernel 2.4) is when the 2.2 kernel processes packets, the packets themselves cross all three filter chains, that is, the INPUT, FORWARD, and OUTPUT chains. This is inefficient. With the 2.4 kernel, packets don’t cross all three chains. So if a rule is destined for the local machine, then I can filter out the INPUT chain. If the packet is entering eth0 and it’s destined for my LAN through eth1, then I write a rule for the FORWARD chain. Finally, if I want to match packets based on their destination from my machine, then I can filter out the OUTPUT chain. According

[4] Russell, Rusty. “So What’s A Packet Filter?” Linux iptables HOWTO. v.0.0.2. 29 September 1999. URL: http://www.linuxguruz.org/iptables/howto/iptables-HOWTO-3.html#ss3.2 (21 Feb. 2003).

to Rusty Russell, the creator of the netfilter architecture[5], "This means that for any given packet, there is one (and only one) possible place to filter it. This makes things much simpler for users than ipchains was."

There are some minor differences, but those are related to syntax issues such as the MASQ syntax, meaning "to masquerade". With iptables, it's now replaced with the full word—MASQUERADE. The built-in chains, that I discussed earlier, are now upper case. If you remember correctly, in ipchains, you could essentially use any case when referring to INPUT and OUTPUT. This was because packets traversed both chains (or actually ALL chains) regardless of their destination. However, since packets now only traverse those chains whenever they are delivered locally, or are originating locally, they must be capitalized[6].

6.2. Packet Traversal

So I know what's going on with the built-in chains in the kernel. But what exactly is going on when it comes to packets traversing through those chains?

According to the netfilter hacking HOWTO[7], one of the key designs of the netfilter architecture is tables. These tables are the '**filter**', '**nat**', and '**mangle**' tables. The beauty of this setup is that if I need to filter a packet, it stays on the filter table. If I receive a packet on port 80, and I want it to go to port 1300, then I use iptables to change the packet through Network Address Translation. This is done on the 'nat' table. If I need to mangle a packet, the actual work will take place on the 'mangle' table. So unlike ipchains, I have one place to do my work on the packet. In other words, whatever gets filtered on the 'filter' table, or NAT'ed on the 'nat' table, stays on that respective table.

Essentially, a packet will enter through the device layer, then it'll enter the kernel via the correct device driver for the NIC and finally, it either stops locally, on the machine itself—destined for whatever application is going to use it, or it gets forwarded to another host.

6.2.1. The Mangle Table

From here I can decide what I want to do with this packet. Most "mangle" operations involve changing the Type of Service (TOS), modifying TTLs and finally, MARK'ing a packet. This occurs within the PREROUTING portion of the netfilter architecture.

---

[5] Russell, Rusty. "Packet Selection: IP Tables." Linux netfilter Hacking HOWTO. v.1.14. 2 July 2002. URL: http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3.html#ss3.2 (21 Feb. 2003).

[6] Russell, Rusty. "Differences Between iptables and ipchains." Linux 2.4 Packet Filtering HOWTO. v.1.26. 24 January 2002. URL: http://www.netfilter.org/documentation/HOWTO//packet-filtering-HOWTO-10.html (20 Feb. 2003).

[7] Russell, Rusty. "Packet Selection: IP Tables." Linux netfilter Hacking HOWTO. v.1.14. 2 July 2002. URL: http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3.html#ss3.2 (22 Feb. 2003).

A perfect example of where a home office user might use iptables to change TTL data is when their Internet Service Provider has a problem with its users running multiple computers off of a single shared IP. SOHOs generally have multiple computers behind a common gateway. Often, most office managers want all of their machines to access the Internet via a single gateway IP. Oskar Andreasson explains[8] a "workaround" quite well. Since each machine would send back packets with different TTL stamps, you could theoretically change the TTL of everything coming into your gateway to a single TTL value, and it would seem as though only one machine was using that IP. I don't necessarily condone this, especially if it violates your ISP's policies, but this is merely an excellent technical example of using iptables to mangle TTL data.

In the end, however, it's up to you to consult your ISP regarding their official policy on the IP addresses that they issue. It's my experience that the majority of ISPs don't have a problem with this—but some ISPs are indeed a little nosy in trying to find out whether or not you have multiple computers in a home network. They do this so that they can offer "additional packages" or additional static IPs to compliment your needs.

6.2.2. The NAT Table

The NAT table is where all network address translation takes place. It takes place only in the PREROUTING and POSTROUTING phases. In order to understand this basic concept, you have to understand the three basic types of NAT—Source NAT (SNAT), Destination NAT (DNAT), and MASQUERADE and how they apply to the PREROUTING and POSTROUTING sections. It's easy to get confused with Source NAT'ing and Masquerading. In Oskar Andreasson's iptables tutorial[9], he discusses the MASQUERADE target as a way to do Source NAT'ing for those IPs that are dynamic. Essentially, every time the target matches a packet, it will check the IP address. This is different from the SNAT target, because SNAT doesn't check the IP at all. So if I have a nice home office setup, and I own a static IP, then I should use the SNAT target. If I have a dynamic address, that is, my address always changes, then the MASQUERADE target is what I'd want to use.

As an example, I have a client whose home office is run off of a cable modem. So in his rules, he would add the following lines:

```
# We alter our header packets in order to masquerade private routable
# addresses.
#
```

[8] Andreasson, Oskar. "Mangle table." Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN565 (25 Feb. 2003).

[9] Andreasson, Oskar. "Nat table." Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN595 (25 Feb. 2003).

```
$IPT -t nat -A POSTROUTING -s 192.168.1.0/24 -o eth0 -j SNAT --to
1.2.3.4
```

In the preceding example, I'm taking any packets that are destined for the eth0 interface and changing their source IPs in the packet header to reflect the IP address of 1.2.3.4 of the rule. The **-t** switch is used to specify which table that I want to use for this rule. The **-A** switch defines which chain, and of course the **-s** switch is my source IP or network address and **-o** represents which interface my packets are traversing. Finally the **-j** switch means to "jump to" the target, **SNAT**, and finally, SNAT must have a **--to** switch followed by an IP, which in this case is my client's ISP assigned IP address.

To show an example of a DNAT rule, my client also has a few services running in his home office that he'd like to allow the outside world to access. He has a web server, and a mail server. The following rules are an example of changing the destination headers of the packets themselves so that they are "redirected" to wherever he wants them to go.

```
# We alter destination packets as they hit our interfaces so we can
# transmit them to specific points along our network.
#
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 80 -j DNAT
--to 192.168.20.20:80
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 25 -j DNAT
--to 192.168.20.20:25
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 110 -j DNAT
--to 192.168.20.20:110
```

The same definitions apply for our switches as in the previous SNAT examples. The difference here is that I know that my packets are transmitting into my primary interface, eth0. So I add the **-i** switch since it refers to the "incoming" interface. Also, I've added a **--dport** switch here for a reason. I'm most likely only going to be using the DNAT rule when I want to redirect traffic from one host to another. However, I have to use the **--dport** switch because I don't want to send **ALL** of my traffic that's heading to my 1.2.3.4 address to my 192.168.1.20 address. I only want to send data destined for specific ports. So in this case, all packets whose destination IPs reflects 1.2.3.4 port 80 will be NAT'ed to reflect a new destination IP of 192.168.1.20 port 80. This occurs for the other two rules as well, except that the ports are different.

So taking one step forward, my client simply registers his domain name for his website, sets his DNS server to assign his domain an address of 1.2.3.4, and his DNAT rule will take care of altering the destination IPs for him. Obviously, he must do this since the outside world can't see his internal address. And that's one reason why DNAT is so useful.

6.2.3. The Filter Table

Finally, the filter table is the table that's used when I want to match a packet after it's already been affected either by a mangling operation, or other actions that have been done in the PREROUTING and/or POSTROUTING phases.

Essentially, I can match a target based on whether I want to **ACCEPT** it or **DROP** it, and there's also a number of other filtering options that I can pursue, however, the two I just mentioned are probably the ones that are most used for a small office / home office.

However, I'd like to touch on a few other important targets during the filter phase that can be applied to packets.

First, I've already explained about ACCEPTing or DROPing a packet based on a specific condition, however, what if I want to **REJECT** it? And how is that different from DROPing it?

If I REJECT a packet, I actually end up sending a 'port-unreachable' response back to the originator of the packet. Otherwise, if I use the DROP target, I simply drop the packet. According to Oskar Andreasson[10], the default response from a REJECT is indeed the 'port-unreachable' message. However, I can elect to also send icmp-net-unreachable, icmp-host-unreachable, icmp-port-unreachable, icmp-proto-unreachable, icmp-net-prohibited and icmp-host-prohibited as well, which provides me with some interesting options depending on what I want to tell the host who sent the packet.

As an example of ACCEPT and DROP targets, here's a few rules for my client's POP3 service:

```
$IPT -A FORWARD -p tcp --dport 110 -j DROP
$IPT -A FORWARD -p tcp -s 2.3.4.22 --dport 110 -j ACCEPT
```

The beauty of iptables is that it's relatively easy to understand in terms of rules. In this example, I'm referencing my FORWARD chain because my packets are being forwarded through my firewall. They're not destined for the local machine, because I've already demonstrated that I'm DNAT'ing all port 110 traffic to my 192.168.1.20 machine. Remember, DNAT'ing operations take place either in PREROUTING or lastly at the POSTROUTING chain.

Looking at this example, I'm dropping all destination traffic that happens to be traversing my FORWARD chain with headers matching port 110. I've "explicitly denied" them. However, in the next line, I only allow traffic to port 110 if and only if it's source IP in the header matches IP address 2.3.4.22. As evident, this is "allowed" to do so via the target, **ACCEPT**.

[10] Andreasson, Oskar. "REJECT taget." Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN2442 (26 Feb. 2003).

6.3. Exploring Connection Tracking[11]

There are, of course, more ways of working with packets within the netfilter architecture than simply doing standard filtering. A more, complex form of filtering can be achieved by filtering those packets that have a certain state.

Stateful firewalls, like iptables, are actually doing nothing more than establishing some form of connection tracking. In the end, that's what "being stateful" is all about—tracking a connection and verifying what kind of state it's in—whether it be **ESTABLISHED**, **NEW**, **RELATED**, or **INVALID**. These are the states that iptables recognizes.

An **ESTABLISHED** state is a state that's associated with a connection. A **NEW** state is a packet that has started a new connection, and has nothing to do with any other connection. An **INVALID** state would represent a packet that isn't associated with any other connection period, and finally, a **RELATED** state would be a packet that's related to the connection of another packet. For example, this could be FTP data traffic or traffic associated with an ICMP error.

A fine example of stateful filtering would be filtering for active FTP.

In this example[12], James Stephens describes active FTP best:

> The ftp client sends a port number over the ftp channel via a PORT command to the ftp server. The ftp server then connects from port 20 to this port to send data, such as a file, or the output from an ls command. The ftp-data connection is in the *opposite sense* from the original ftp connection.
>
> To allow active ftp without knowing the port number that has been passed we need a general rule which allows connections from port 20 on remote ftp servers to high ports (port numbers > 1023) on ftp clients. This is simply too general to ever be secure.
>
> Enter the *ip_conntrack_ftp* module. This module is able to recognize the PORT command and pick-out the port number. As such, the ftp-data connection can be classified as **RELATED** to the original outgoing connection to port 21 so we don't need **NEW** as a state match for the connection in the INPUT chain.

With that said, I would use the following rules in my firewall script to allow active FTP:

---

[11] Stephens, James. IPtables: Connection tracking. 2 October 2002. URL: http://www.sns.ias.edu/~jns/security/iptables/iptables_conntrack.html (26 Feb. 2003).
[12] Stephens, James. IPtables: Connection tracking. 2 October 2002. URL: http://www.sns.ias.edu/~jns/security/iptables/iptables_conntrack.html (26 Feb. 2003).

```
$IPT -A INPUT -p tcp --sport 20 -m state --state ESTABLISHED,RELATED -j
ACCEPT
$IPT -A OUTPUT -p tcp --dport 20 -m state --state ESTABLISHED -j ACCEPT
```

7. Applying Basic Rules To A SOHO Firewall

A basic set of rules for a small office need not be a complex lesson in firewall design. I think that iptables' syntax is relatively easy to understand, especially if you compare it to other, older alternatives like **IPF**.

Here's some rules that I've developed for a new client and his home office with their corresponding definitions and a small justification for their use:

```
$IPT -F
$IPT -t nat -P POSTROUTING DROP

# We alter our header packets in order to hide private routable
# addresses.

$IPT -t nat -A POSTROUTING -s 192.168.20.0/24 -o eth0 -j SNAT --to
1.2.3.4

# We alter destination packets as they hit our interfaces so we can
# send them to specific points along our network.

$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 80 -j DNAT
--to 192.168.20.22:80
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 25 -j DNAT
--to 192.168.20.22:25
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 110 -j DNAT
--to 192.168.20.22:110
$IPT -t nat -A PREROUTING -p tcp -i eth0 -d 1.2.3.4 --dport 21 -j DNAT
--to 192.168.20.22:21

# If any NetBIOS packets do not come from the local network, then we
# drop them.

$IPT -A INPUT -p tcp -s ! 192.168.20.0/24 --dport 137:139 -j DROP
$IPT -A INPUT -p udp -s ! 192.168.20.0/24 --dport 137:139 -j DROP

# We set up logging against all packets using –ULOG that pass through
# eth0 and are destined for port 21 and who are NOT originating from
# the local LAN—that is, 192.168.20.0. The "!" directive means to match
# anything that's the opposite of. The –ULOG directive allows us to
# dump logging to a userspace application. One such app is ulogd. It
# can be found at
# http://gnumonks.org/gnumonks/projects/project_details?p_id=1.

$IPT -A INPUT -i eth0 -p tcp -s ! 192.168.20.0/24 --dport 21 –j ULOG --
ulog-prefix "FTP connection attempt from: "

# Local Net
$IPT -I FORWARD -p tcp -s 192.168.20.0/24 --dport 21 -j ACCEPT
```

```
# Allow Joe User to have special access to FTP from his static IP
$IPT -I FORWARD -p tcp -s 1.1.1.1 --dport 21 -j ACCEPT

# EOF
```

8. Conclusion

In conclusion, security without a doubt should be of utmost concern to every person, organization, and entity. It comes as no surprise that more than 40% of DDoS attacks that CERT reported from 1989 to 1995 were in the category of "Process Degradation"[13]—something that a firewall, over the long haul, can't necessarily provide adequate protection against. And this is an important point—you must use the correct tools for the job. And while iptables as a firewall, will provide adequate protection for most home offices, its efficiency at the enterprise level has yet to be defined, and it's recommended that a commercial alternative be addressed for those companies that need more robust solutions to deal with the growing threat of security-related incidents. Furthermore, firewalls, such as iptables, are just the tip of the iceberg when defining what a proper security architecture should be, whether it's for a home office, or large multi-million dollar enterprise.

However, iptables is both versatile, and efficient for home use. Its efficiency comes directly from the netfilter architecture within the Linux kernel. As I've demonstrated in this paper, once iptables' syntax and its various options are learned, it's relatively easier to understand compared to its counterpart, IPF, or the older, ipfwadm. In the end, iptables on Linux is an excellent solution for home offices and small businesses.

[13] Howard, John. "Chapter 11 Denial-of-Service Incidents." An Analysis Of Security Threats On The Internet 1989 –1995. 7 April 1997. URL: http://www.cert.org/research/JHThesis/Chapter11.html (28 Feb. 2003).

9. References

Greene, Thomas. “Ballmer: Linux is a cancer.” The Register. 6 February 2001. URL: http://www.theregister.co.uk/content/4/19396.html (20 Feb. 2003).

Warfield, Michael. Securing Linux, Part 1. 28 May 1999. URL: http://www.linuxworld.com/linuxworld/lw-1999-05/lw-05-ramparts.html (23 Feb. 2003).

URL: http://ettercap.sourceforge.net (28 Feb. 2003).

Russell, Rusty. “So What’s A Packet Filter?” Linux iptables HOWTO. v.0.0.2. 29 September 1999. URL: http://www.linuxguruz.org/iptables/howto/iptables-HOWTO-3.html#ss3.2 (21 Feb. 2003).

Russell, Rusty. “Packet Selection: IP Tables.” Linux netfilter Hacking HOWTO. v.1.14. 2 July 2002. URL: http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3.html#ss3.2 (21 Feb. 2003).

Russell, Rusty. “Differences Between iptables and ipchains.” Linux 2.4 Packet Filtering HOWTO. v.1.26. 24 January 2002. URL: http://www.netfilter.org/documentation/HOWTO//packet-filtering-HOWTO-10.html (20 Feb. 2003).

Russell, Rusty. “Packet Selection: IP Tables.” Linux netfilter Hacking HOWTO. v.1.14. 2 July 2002. URL: http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3.html#ss3.2 (22 Feb. 2003).

Andreasson, Oskar. “Mangle table.” Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN565 (25 Feb. 2003).

Andreasson, Oskar. “Nat table.” Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN595 (25 Feb. 2003).

Andreasson, Oskar. “REJECT taget.” Iptables Tutorial 1.1.11. v.1.1.9. 21 March 2002. URL: http://www.netfilter.org/documentation/tutorials/blueflux/iptables-tutorial.html#AEN2442 (26 Feb. 2003).

Stephens, James. IPtables: Connection tracking. 2 October 2002. URL: http://www.sns.ias.edu/~jns/security/iptables/iptables_conntrack.html (26 Feb. 2003).

Howard, John. "Chapter 11 Denial-of-Service Incidents." An Analysis Of Security Threats On The Internet 1989 –1995. 7 April 1997. URL: http://www.cert.org/research/JHThesis/Chapter11.html (28 Feb. 2003).