

Using IPfilter to ringfence your datacentre

Security in the datacentre is a deeply contentious topic: many people recognise the importance of the requirement but unfortunately most companies shy away from the complexity of implementing adequate security even for their most critical servers.

As a simple illustration of the inadequacy of most installations, take a typical scenario within a large company: there are many servers spread around the corporate WAN, most of which occupies an important role in the running of an organisation. However - within this inventory, there is a small subset of business critical servers whose non-availability or compromise would cause significant financial, business and/or legal problems.

In our example company, there are several offices, each of which are interconnected via the corporate WAN. A compromise in any of these offices will give network access to critical servers, protected only by the weakest point of security on that server.

How difficult would it be for an individual with intent to gain access to the target company offices? It wouldn't be difficult: The individual does not have to be in a privileged position and may, for example, simply sign up with a local temping agency to engineer his way into the cleaner squad that clean the offices long after they have been vacated. Perhaps a savvy intruder could plant a wireless access point, allowing unrestricted access to the network from beyond the physical constraints of the building.

It is clear therefore, in the face of the sprawl of modern corporate [L|M|W]ANs that securing the entire network is a very difficult, and for most, a logistically unfeasible undertaking. It therefore makes sense to spend more time and effort protecting the critical resources upon which your company depends.

The traditional approach

A traditional solution would be to consolidate the datacentre onto a (relatively) small number of subnetworks, and have those subnetworks protected by a dedicated enterprise-class firewall solution (Checkpoint Firewall-1 for example), in much the same way as you might protect an external network connection.

This sounds like a good solution initially; however, a modern datacentre will generally contain many servers, each of which requires to communicate with peers and clients using a myriad of protocols and network paths. The resultant ruleset for our datacentre firewall is likely to be larger than the entire UK tax statutes. When we encounter such complexity, managing updates becomes a difficult and time-consuming task, vastly increasing the likelihood of human error (although this is not seen as an excuse for the Inland Revenue!)

The intrinsic approach

In the UK there was a television advert for a certain brand of wood preservative, whose slogan was the simple message that it "does exactly what it says on the tin". IPfilter certainly lives up to that slogan – it is a kernel module that is loaded in-between the hardware driver and the IP module that allows network traffic flowing to/from the unix kernel to be filtered, providing far superior access control to your servers than was previously possible.

Figure 1 illustrates how IPfilter integrates with the Solaris kernel modules that implement the hardware drivers and IP protocol handling. It provides an interface that can effectively block traffic travelling in either direction. It is implemented as two modules – the "pfil" STREAMS module, and the "ipfilter" module itself.

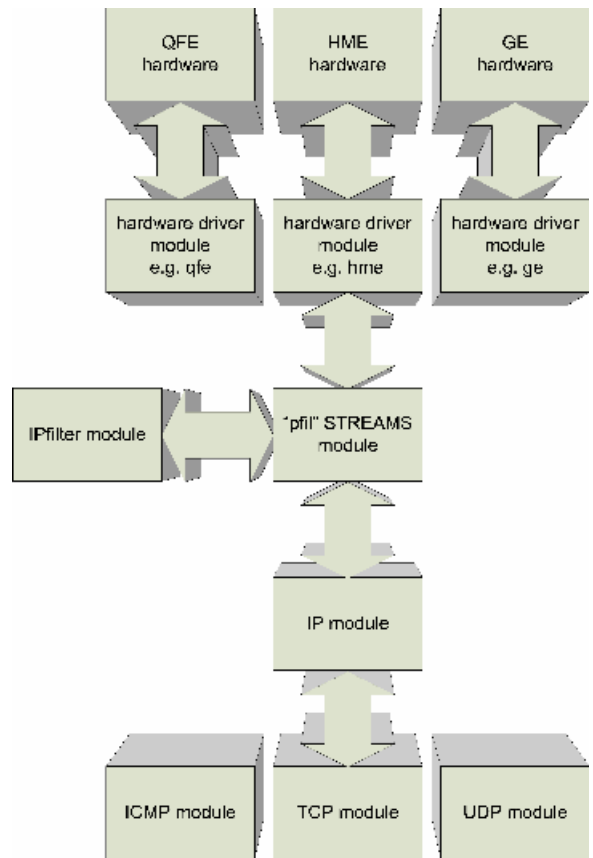


figure 1, solaris kernel

Most of the documentation available for IPfilter has been geared towards building dedicated firewall machines, but what appears to be glossed over in the currently available documentation¹, is the potential for using it to secure individual servers.

Of course, this technology is not new with Solaris 10 – packet filtering has been around for a very long time indeed. If you are familiar with Linux, you will most probably already be aware of iptables (or its' predecessor, ipchains), OpenBSD has "pf", and of course Darren Reed has been developing ipfilter for earlier versions of Solaris/FreeBSD/NetBSD for years.

¹ Publicly available documentation for Solaris 10 is still a bit thin at the time of writing, so this may change over time.

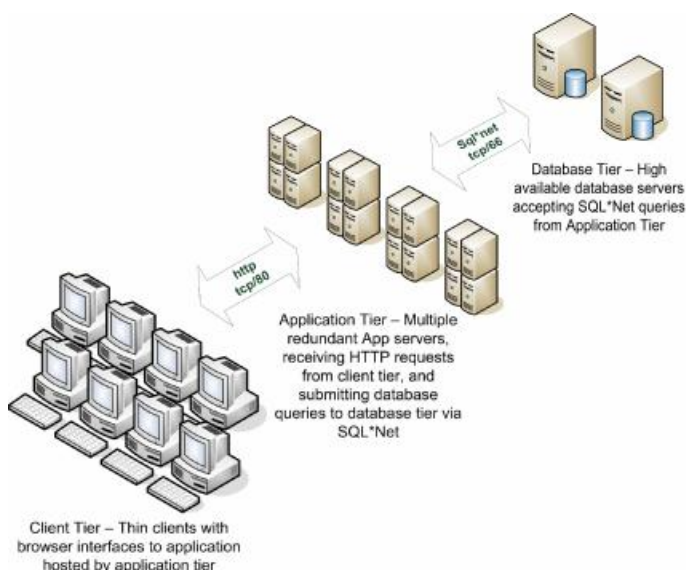


figure 2, three-tier application architecture

In our example company, every client (or indeed any network enabled device connected to the corporate network) has the ability to connect directly to the database servers and get a login prompt, (or any other network service that the database server is configured to offer), providing a potentially valuable point of entry to an intruder.

This situation can be improved by the use of IPfilter but, to do so, the data-paths of your application must be intimately known. This is an exercise that is relatively easy to implement during initial system installation, but to retrofit onto an existing service is a very difficult thing to do indeed, requiring many hours of analysis and documentation to ensure that all network services are fully understood before proceeding.

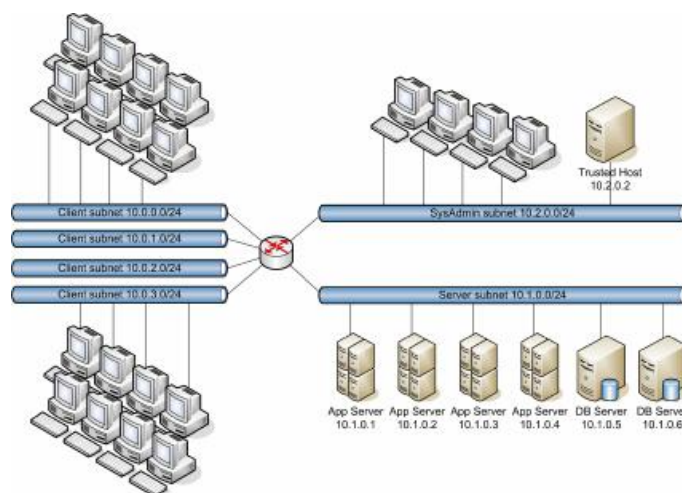
A secure implementation of IPfilter also has the advantage of helping to control your application set, and ensuring that datapaths and inter-system dependencies are documented. Quite often, departments that are responsible for deploying an application will neglect to document the data flow between systems, making outage negotiation difficult for the systems administrator. Strictly controlling the data flow in this manner helps to avoid this by ensuring that the application does not evolve beyond the systems administrators' understanding.

Figure 2 illustrates a somewhat over-simplistic example of a three-tier application architecture. Rarely will a setup be this easy - there will be data paths that have not been considered here (FTP uploads of batch reports, system and database backups, network printing, etc.). But to press on and to demonstrate the use of IPfilter, we shall assume that our application is indeed this simple.

We can define some network addresses for our application: (for the purposes of this illustration, we shall use CIDR address notation where the trailing number after the IP address is the number of significant bits in the subnet mask).

Client Subnets: 10.0.0/24 10.0.1/24 10.0.2/24 10.0.3/24	In our example, only four subnets are considered to be client subnets. This is good, because it will allow us to lock down access to the application server based on these criteria, preventing access from other parts of the corporate WAN who should not have any need to connect. This clearly doesn't apply in every case: for some applications users are deeply distributed. Our example would apply to a more restricted userbase, for example a call-centre environment.
---------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

App Servers: 10.1.0.1 10.1.0.2 10.1.0.3 10.1.0.4	There are a finite number of application servers, so it is easy to define exactly the well-known IP addresses of the servers.
Database Servers: 10.1.0.5 10.1.0.6	Similarly, the database servers are easy to define.
Administrative Network: 10.2.0/24	Once the network is locked down, we will need an administrative zone from which we can connect via SSH to administer the servers – there is no point in allowing the entire corporate WAN access to SSH login services.
Trusted Host: 10.2.0.2	For emergency use only, a trusted host should have unrestricted access to all network services. As this should be only a small number of machines with this access, we define the IP address itself, rather than a subnet address. NOTE: The trusted host itself should be highly secure.



Example network diagram

So how would we begin to configure this into Solaris? – In Solaris 10, the IPfilter packages are installed by default in the SUNWCall software cluster. To check you have the relevant packages, you can type:

```
$ pkginfo |grep SUNWipf
system      SUNWipfr      IP Filter utilities, (Root)
system      SUNWipfu      IP Filter utilities, (Usr)
$
```

To enable ipfilter on your machine, edit the "/etc/ipf/pfil.ap" file, to uncomment your network interface type and reboot the machine.

```
# IP Filter pfil autopush setup
#
# major minor lastminor modules
#le -1 0 pfil
#qe -1 0 pfil
#hme -1 0 pfil
#qfe -1 0 pfil
#eri -1 0 pfil
#ce -1 0 pfil
#bge -1 0 pfil
#be -1 0 pfil
```

```
#vge -1 0 pfil
#ge -1 0 pfil
#nf -1 0 pfil
#fa -1 0 pfil
#ci -1 0 pfil
#el -1 0 pfil
#ipdptp -1 0 pfil
#lane -1 0 pfil
```

/etc/ipf/pfil.ap, example for "hme" interface type

The ruleset itself is stored as "/etc/ipf/ipfilter.conf" and each time this file is changed the kernel module must be instructed with the new configuration. This is done by the following command:

```
# /etc/init.d/ipfboot reipf
```

sidenote : keeping state

Stateful inspection is a technique whereby only the first packet in a connection is passed through the entire ruleset. If a match is found, then an entry in a state table is made to record that fact and subsequent packets in the connection are allowed through without the need to consult the ruleset.

This has a couple of useful advantages: firstly we do not have to consider rules for the return path thereby simplifying our ruleset, and the performance of the kernel module improves (because it doesn't have to pass each and every packet through the ruleset).

The use of stateful inspection is controlled by the keywords "keep state" at the end of each configuration line (see listing 1)

We've discussed some of the broad terms associated with IPfilter; let's get to the meat of it and see an example:

```
01 # Database server - inbound SQL*Net traffic from specific hosts, plus admin access
02 #
03 # For simplicity, we will allow all outbound traffic
04 # However, for a production server, I would want to lockdown outbound traffic also
05 pass out quick all keep state
06
07 # By default, we shall block all inbound traffic.
08 # If you're not on the list - you're not getting in...
09 block in all keep state
10
11 # ICMP is useful to have (ping,traceroute,etc) - some sites may wish to block this
12 pass in quick proto icmp from any to any
13
14 # Allow access to all services from our trusted host
15 pass in quick from 10.2.0.32 to any flags S keep state
16 # Allow SSH access from all hosts on our SysAdmin subnet
17 pass in quick proto tcp from 10.2.0.0/24 to any port = 22 flags S keep state # SSH
18
19 # And, of course, SQL*Net connections from our application servers
20 pass in quick proto tcp from 10.1.0.1 to any port=66 flags S keep state # SQL*NET
21 pass in quick proto tcp from 10.1.0.2 to any port=66 flags S keep state # SQL*NET
22 pass in quick proto tcp from 10.1.0.3 to any port=66 flags S keep state # SQL*NET
23 pass in quick proto tcp from 10.1.0.4 to any port=66 flags S keep state # SQL*NET
```

Database server ruleset

For this ruleset, the rule on line #05 will allow all outbound traffic from the server by default (all of our rules here will "keep state"). The keyword "quick" ensures that when this rule is matched (by the criteria "out" and "all"), no further rules in the ruleset will be checked – processing stops (also called "first match and exit" behavior).

Line #09 sets the default policy for inbound traffic - notice the lack of the "quick" keyword. This allows the ruleset to be further processed to allow the opportunity for an explicit rule further down the ruleset to reverse that decision by the packet filter.

ICMP (line #12) is useful to have as it allows support staff to use ping and traceroute to diagnose typical connectivity problems (If desired this rule can be omitted to further reduce the network exposure of this machine). This rule allows any IP address to send an ICMP packet unrestricted to any² IP address that this machine has defined.

² (Note: the "to any" could be tightened to a specific IP address on the server, if that machine has multiple IP addresses defined, for example, virtual IP addresses or multiple network interface controllers).

Seems straightforward? – yes, it is indeed. Now onto the rules that do the application level filtering that we are interested in.. line #15 allows unrestricted access from the administrative trusted host. You're already familiar with the general format of the rule, so the new keyword here is "type S". This is an additional condition on the rule processing which will only match on packets with the "SYN" flag sent (the first packet in any TCP conversation) – we only need to match this packet because we are keeping state.

Line #17 defines SSH access to the server from only the administrative subnet (10.2.0.0/24), and then only to port tcp/22 (where SSHD listens for inbound connections).

Finally, the ruleset if finished off by lines #20-23, which allow access to the database servers' SQL*Net listener on port tcp/66.

The application server ruleset is similarly simple:

```
01 # Application server - inbound HTTP traffic from client subnets, plus admin access
02 #
03 # For simplicity, we will allow all outbound traffic
04 # However, for a production server, I would want to lockdown outbound traffic also
05 pass out quick all keep state
06
07 # By default, we shall block all inbound traffic.
08 # If you're not on the list - you're not getting in...
09 block in all keep state
10
11 # ICMP is useful to have (ping,traceroute,etc) - some sites may wish to block this
12 pass in quick proto icmp from any to any
13
14 # Allow access to all services from our trusted host
15 pass in quick from 10.2.0.32 to any flags S keep state
16 # Allow SSH access from all hosts on our SysAdmin subnet
17 pass in quick proto tcp from 10.2.0.0/24 to any port=22 flags S keep state # SSH
18
19 # Allow HTTP connections from our client subnets
20 pass in quick proto tcp from 10.0.0.0/24 to any port=80 keep state # HTTP
21 pass in quick proto tcp from 10.0.1.0/24 to any port=80 keep state # HTTP
22 pass in quick proto tcp from 10.0.2.0/24 to any port=80 keep state # HTTP
23 pass in quick proto tcp from 10.0.3.0/24 to any port=80 keep state # HTTP
```

Application server ruleset

Outbound filtering

In order to further secure the network stack, outbound filtering can be implemented also. For example, on the application servers, the following changes to the ruleset will ensure that only outbound traffic to the Database servers.

Delete lines #03-05, and insert the following:

```
# By default, we shall block all outbound traffic
block out all keep state
# Rules to explicitly allow outbound traffic to the database servers
pass out quick proto tcp from any to 10.1.0.4 port=66 flags S keep state # SQL*NET
pass out quick proto tcp from any to 10.1.0.5 port=66 flags S keep state # SQL*NET
```

Remember if you are considering this, you would also likely require rules to allow nameservice lookups (LDAP/DNS/NIS), plus any other services upon which the server depends.

As an example, here is a real-life ruleset for a Solaris 10 NFS server:

```
block in all keep state
block out all keep state

pass in quick proto icmp from any to any
pass out quick proto icmp from any to any

pass in quick proto tcp/udp from any to any port = 111 flags S keep state # sunrpc

pass in quick proto tcp from trustedhost to any .port = 514 flags S keep state # rsh
pass in quick proto tcp from backup1 to any port = 13782 flags S keep state # Netbackup
pass in quick proto tcp from backup2 to any port = 13782 flags S keep state # Netbackup
pass in quick proto tcp from backup3 to any port = 13782 flags S keep state # Netbackup
pass in quick proto tcp from backup4 to any port = 13782 flags S keep state # Netbackup
pass in quick proto tcp from 10.2.0/24 to any port = 22 flags S keep state # SSH - Admin network only

pass out quick proto tcp/udp from any to dns1 port = 53 flags S keep state # DNS
pass out quick proto tcp/udp from any to dns2 port = 53 flags S keep state # DNS
pass out quick proto tcp/udp from any to dns3 port = 53 flags S keep state # DNS

pass out quick proto tcp from any to mailhost port = 25 flags S keep state # SMTP
```

Of course, you may notice that neither of the NFS services appear in that configuration file – this is because they are RPC based, and are

prone to the port number changing when they are registered with rpcbnd.

This prevents these services being hardcoded in the configuration file, but can easily be worked around by the use of a small script, to be run after nfs.server has been run in /etc/rc3.d:

```
rpcrule() {
    print "# $1";
    rpcinfo -p |nawk -v service=$1 -v dest=$2 '($NF == service) {print
"pass in quick proto "$3" from a
ny to "dest" port = "$4" keep state"}' |uniq
}

rpcrule nfs    any | ipf -f - # Generate ipfilter rules, and feed as stdin to "ipf"
rpcrule mountd any | ipf -f - # Generate ipfilter rules, and feed as stdin to "ipf"
```

Conclusion

IPfilter is a highly configurable piece of software, and IP networking is a particularly broad topic, which makes it difficult to cover any more than the basics in the context of this article. Other things must be considered when implementing on a production server, for example:

- the size of the state table for servers with high numbers of simultaneous network connections;
- Monitoring and troubleshooting the IPfilter configuration
- Education of support personnel.

Servers are more usually hardened by the removal of services from /etc/inet/inetd.conf and by disabling unnecessary daemons from running. IPfilter takes this one step further by allowing the kernel to actually prevent the traffic from even being presented to the services. Even if those services are running – they are guaranteed to not receive any traffic.

We have presented a demonstration of how to secure servers using the IPfilter facility within Solaris 10. IPfilter has, of course been around for some time, available as freeware written by Darren Reed (packages for earlier versions of Solaris are available on <http://www.sunfreeware.com>).

If you haven't already, perhaps you should start planning your use of IPfilter.

Mike Scott is the director of Hindsight IT Ltd, a small Solaris consultancy based in Central Scotland. He has been working in the North East and the central belt for the last ten years, specialising in systems management with a keen interest in security and performance management. He can be contacted at sysadmin@hindsight.it

References

[1] Darren Reeds IPfilter homepage, available at <http://coombs.anu.edu.au/~avalon/>

[2] IPfilter HOWTO documentation, <http://www.obfuscation.org/ipf/>

[3] Amy Rich, "Securing Hosts with IP filter", available at <http://www.sun.com/bigadmin/features/articles/ipfilter.html>