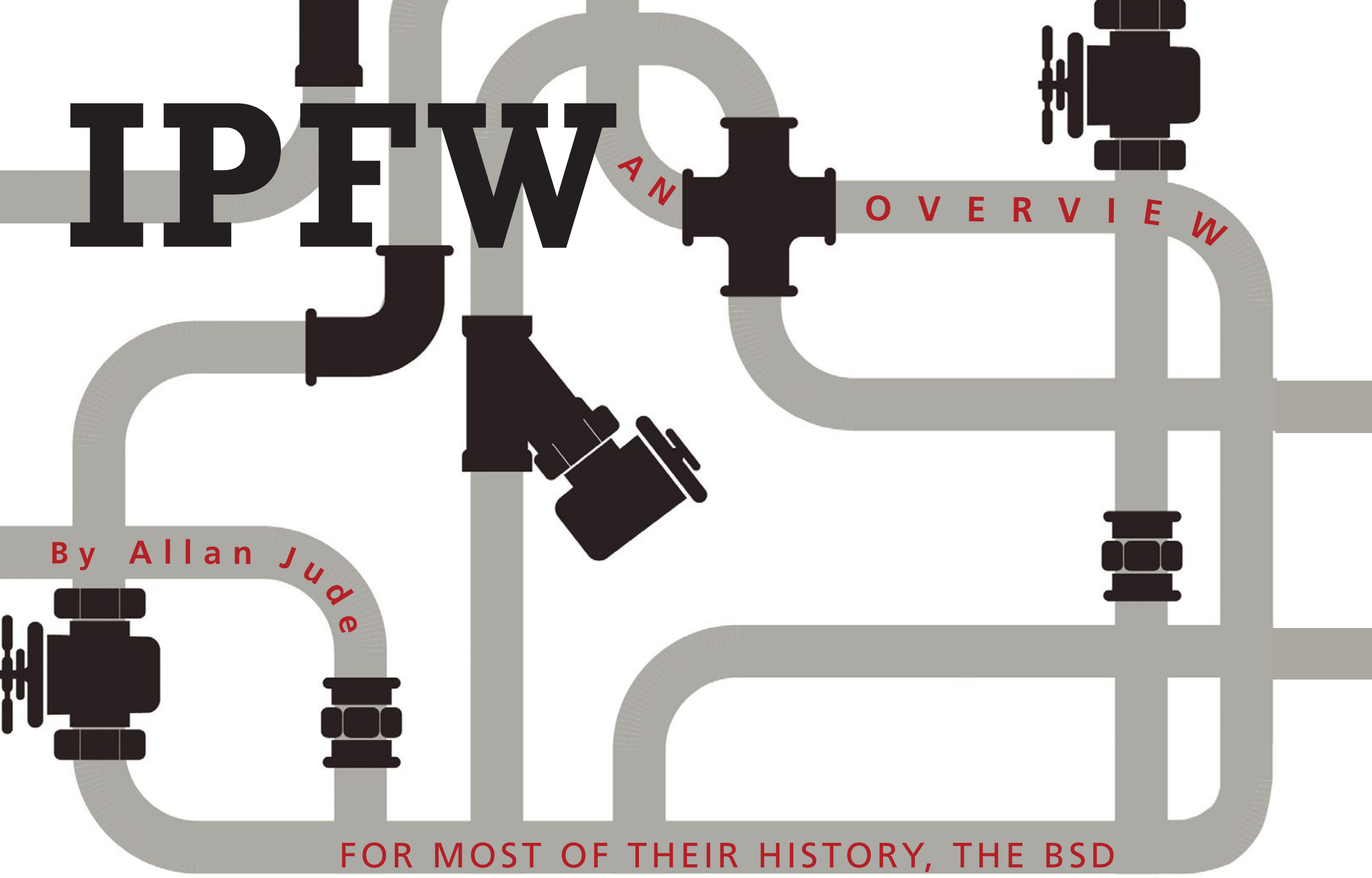


IPFW An Overview

By Allan Jude

FOR MOST OF THEIR HISTORY, THE BSD family of operating systems has been known for making great firewalls. IPFW receives less attention than the PF packet filter, but it is very well featured with many advantages. IPFW was first introduced with FreeBSD 2.0 in 1994, while dummynet functionality came along in 2.2.8 (1998).

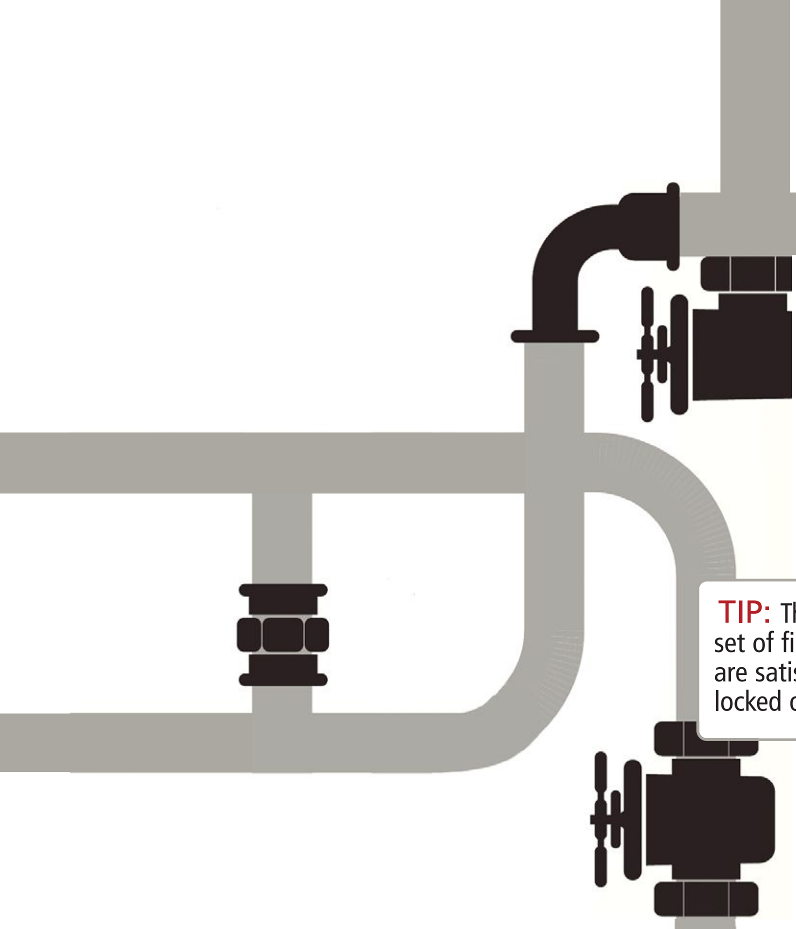
The current incarnation of IPFW, a complete rewrite dubbed IPFW2, was written and introduced in the summer of 2002. IPFW is remarkably fast and has very good SMP scalability.

IPFW is a "first match" firewall, meaning that each packet is compared against a numbered rules list, and once a rule matches, the search ends. This allows the administrator to write the rules in a specific order to achieve the greatest speed, and avoid comparing certain packets against more complex rules. Bandwidth and quality can be defined with pipes and queues, enforced with rules. IPFW also features an in-kernel NAT implementation which augments the existing user-space natd, full support for VIMAGE/VNET which creates a separate instance of the firewall in each VNET jail, multiple rule sets, dynamic rules, and tight integration with the operating system to provide features including rule matching against the user or jail which generated the packet.

This article covers the basics of enabling and configuring IPFW. It then discusses some advanced topics including rule numbering advice, simulating real world networks, traffic prioritization and shaping, and using the in-kernel NAT implementation, including configuring port forwarding, in conjunction with jails. The article then ends with an overview of some of the other features of IPFW.

LOADING THE FIREWALL

While IPFW can be compiled into a custom kernel, support is usually enabled by loading its kernel loadable module. Since the default policy is



to deny all traffic, one should also either create a custom rule set, or load a sample rule set. The built-in /etc/rc.firewall script contains the logic for a number of basic firewall rulesets. The available templates are: open, closed, simple, client, and workstation.

To enable IPFW, but not block any traffic, add the following lines to /etc/rc.conf:

```
firewall_enable="YES"
firewall_type="OPEN"
```

To create a simple stateful firewall:

```
firewall_enable="YES"
firewall_type="CLIENT"
firewall_client_net="192.168.0.0/24"  #Use the IP for your internal network
firewall_client_net_ipv6=""  #Specify the internal IPv6 net if you have one
```

Another option is to indicate the full path of a custom rule set file as the firewall_type, and IPFW will read that file and interpret each line as the arguments of the IPFW command. IPFW also supports using a preprocessor (such as m4) on the specified file, indicated by the -p flag. This allows an administrator to create a single template that, when preprocessed, generates the firewall rules specific to each different host.

Once the rules are in place, "service ipfw start" will start the firewall and apply the rules. Remember to nohup the command if working remotely, otherwise the connection may be closed when the firewall module is loaded, before the rules that allow the connection are added.

TIP: The /usr/share/examples/ipfw change_rules.sh script will apply a new set of firewall rules, and then prompt the administrator to confirm that they are satisfied with the new rules. If the administrator is not satisfied, or is locked out by the new rules, the old rules are restored after 30 seconds.

SIMPLE RULES

IPFW rules are fairly straightforward and easy to write. The following is a brief walk-through of the basic administration commands for the OPEN firewall:

To show existing rules use 'ipfw list':

```
    # ipfw list
00100  5084973634  3455207967775 allow ip from any to any via lo0
00200           0              0 deny ip from any to 127.0.0.0/8
00300           0              0 deny ip from 127.0.0.0/8 to any
65000 94077116494 77774399268246 allow ip from any to any
65535           0              0 deny ip from any to any
```

The first column is the rule number. Each rule is assigned a number which determines the order in which packets are compared. It is possible to have multiple rules with the same number, but this makes it more difficult to manage those rules. The next field is the number of packets that have matched the rule, followed by the total number of bytes of those matching packets. These counters can be reset using the "zero" sub-command to "ipfw". The remainder of each line is the rule.

To add a basic rule which blocks incoming connections to port 25, use this command:

```
# ipfw add 5001 unreach port tcp from any to me dst-port 25
```

This creates rule number 5001. A rule is always created using the keyword "add", and takes effect immediately. In this rule, the action that will be applied to a matching packet is “unreach port”, which will generate an ICMP reply informing the remote host that the port is not accessible, as opposed to “deny” which will silently drop the packet. The body of the rule “tcp from any to me dst-port 25” determines which packets match. The first keyword is the protocol (ip, icmp, tcp, udp, etc). The next section of the body indicates the source and destination addresses of the packet; it allows keywords “any” and “me”, which will match any address assigned to any interface on the machine. Finally, additional options can be specified, including source and destination port (src-port and dst-port), direction (in or out), interface (via em0), attempts to create a new connection (keyword “setup”, packets with the SYN flag set, but the ACK flag not set), previously established connections (keyword “established”, packets with ACK or RST flags set), and most other IP and TCP protocol headers.

An example of a more advanced rule:

```
# ipfw add 5002 allow log logamount 50 ip from 192.168.0.0/24 to any dst-port 80 setup
```

This creates rule number 5002 (for consistency, and administrator sanity, example rules will be numbered up from 5000). This rule will match, log and allow attempts to establish a new connection from the specified subnet to any host on port 80. A log limit has been set to 50 entries: while additional packets will be allowed, only the first 50 packets that match this rule will be logged in order to prevent the logs from filling entries for allowed packets. If needed, the log counter can be reset to 0 with the “ipfw resetlog” command.

NAMING

Rules, pipes and queues each have a separate rule number space, ranging between 1 and 65535. This means it is possible to have a rule and a pipe with the same number, without them being related. Whatever numbering system works for you, keep in mind that having some kind of functional isolation in your numbering is beneficial if you are working on a firewall (inevitably remotely, and usually late). Isolating the numbering system (5000s for rules, 30000s for pipes, 40000s for queues) and potentially grouping into related numeric sets (5001, 30001, 40001), can assist with clarity and avoid confusion.

SIMULATING REAL-WORLD NETWORKS

It is often desirable to test an application under real world conditions, where the network is not a quiet LAN with no latency or congestion. IPFW has a feature to simulate the real world conditions of the public Internet called dummynet(4). Dummynet provides the ability to artificially limit, queue, delay or drop packets to create the desired simulated network conditions.

To enable dummynet functionality, the kernel module must be loaded. This can be done automatically by adding the following to /etc/rc.conf

```
dummynet_enable=”YES”
```

The most basic example of dummynet(4) is creating a pipe with limited bandwidth:

```
# ipfw pipe 30003 config bw 5Mbit/s
# ipfw add 5003 pipe 30003 ip from any to 192.168.0.101
```

The first command configures pipe 30003 with a bandwidth of 5 megabits per second. Pipe numbers are separate from rule numbers (for consistency, and administrator sanity, example pipes will be numbered in the 30000s, with the least significant bits matching the corresponding rule). The second command creates firewall rule 5003 using the “add” keyword. Rule 5003 directs all matching traffic (that has not matched a previous rule) through pipe 30003. This will effectively limit the downstream bandwidth of the host 192.168.0.101 through the firewall to 5 megabits per second.

This is useful for shaping traffic from specific hosts, or simulating slower links, but it does not provide an especially real simulation in terms of qualitative network variability. A better simulation would look like this:

```
# ipfw pipe 30003 config bw 5Mbit/s delay 150 burst 128k plr 0.001 queue 32KBytes noerror
```

This will reconfigure pipe 30003 with convenient options that can help to reproduce application behavior caused by network issues. The delay option will add 150ms of latency to each packet, simulating the latency between Los Angeles and London. The burst option will allow slightly more than the maximum amount of bandwidth to be used, if the pipe was not full beforehand. If the pipe is idle then the first 128kb of data is passed without being rate limited. The next option simulates a packet loss rate (plr) of 0.1%, causing an occasional retransmit, as might be expected on a less than ideal network. The queue option sets the maximum amount of excess data (in packets or KBytes) that will be accepted before additional packets are refused. This can be used to simulate buffer bloat, by setting a low rate limit with a large queue. The final option, noerror, causes the firewall not to return an error to the calling application when a packet is dropped.
Normally, if more data is trying to be sent than can be transmitted within the rate limit set, the firewall notifies the calling application with the same error that would be returned if the device queue was full on an unrestricted network. By suppressing this error, it simulates loss at an upstream router further along the path, where the application will be unaware that the packet has been dropped until it is not acknowledged on the other side of the connection.

CONTROLLING THE FLOW OF DATA

Pipes can also be allocated dynamically. For example, if there are many clients behind the firewall, each client can be limited to a 5Mbit/s flow by creating a dynamic pipe based on a mask (/24 in this case):

```
# ipfw pipe 30004 config bw 5Mbit/s mask src-ip 0x000000ff
# ipfw add 5004 pipe 30004 ip from any to 192.168.0.0/24
```

It is also possible to mask based on destination ip, source or destination port, or protocol.

One other option is to mask ‘all’ bits (source and destination IP, source and destination port, and protocol). This limits any single connection (flow) to 5Mbit/s, but allows each client multiple connections at this speed:

```
# ipfw pipe 30004 config bw 5Mbit/s mask all
```

Sometimes the goal of traffic shaping is not to limit the traffic of any one host, but to ensure that a set amount of bandwidth is shared equally among a set of hosts. In this case, rather than pushing all of the bandwidth directly through a pipe, the firewall can be used to create a number of queues with different priorities in order to classify the different types of traffic.

When a dynamic queue is created with a mask, each flow (in the following case, 1 per source ip address in the subnet) shares a parent pipe evenly. Create a pipe with limited bandwidth and then create a queue to use that pipe. Queue numbers are separate from pipe numbers (for consistency, and administrator sanity, example queues will be numbered up from 40000). Add a rule that matches the desired traffic. The queue rule will create a dynamic queue for each unique flow identifier, as determined by the specified mask. Each flow will have equal access to the limited pipe.

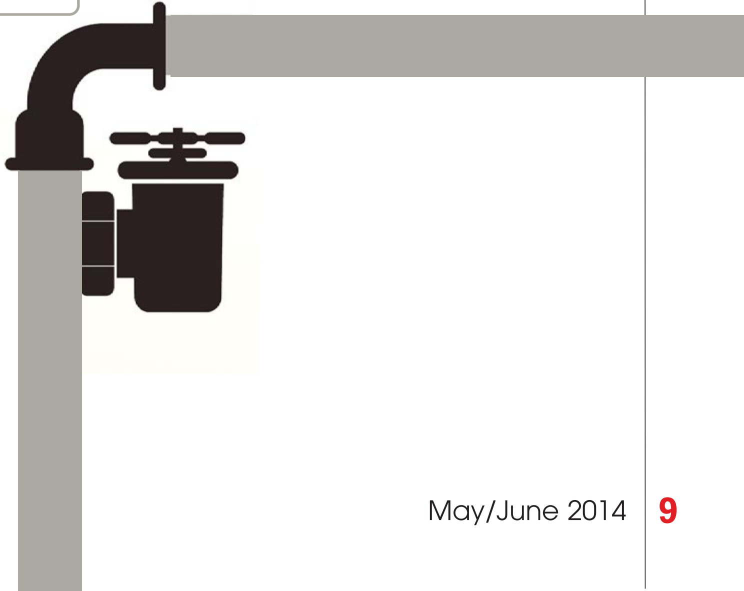
```
# ipfw pipe 30005 config bw 75Mbit/s
# ipfw queue 40005 pipe 30005 mask src-ip 0x000000ff
# ipfw add 5005 queue 40005 ip from any to 192.168.0.0/24
```

Contrast this to dynamic pipes, where each source ip address (flow identifier) had its own separate rate limit.

Sometimes, sharing equally is fine. However “all hosts are equal, but some hosts are more equal than others”. Queues can be weighted, allowing certain traffic to get a greater share of the available pipe:

```
# ipfw pipe 30006 config bw 75Mbit/s
# ipfw queue 40006 pipe 30006 mask src-ip 0x000000ff weight 5
# ipfw queue 40007 pipe 30006 mask src-ip 0x000000ff weight 25
# ipfw add 5006 queue 40006 ip from any to 192.168.0.0/24
# ipfw add 5007 queue 40007 ip from any to 192.168.1.0/24
```

Creating a pipe to allocate bandwidth and two queues with different weightings, with supporting



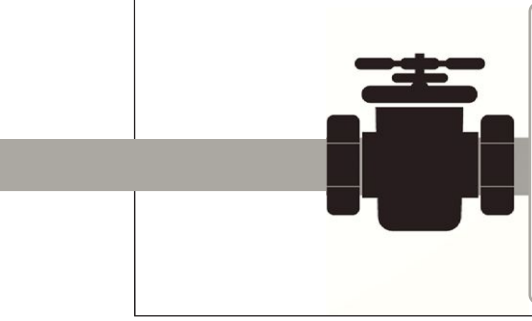
subnet rules, means that hosts in the second subnet get higher priority access to the allocated bandwidth.

This same style of traffic management can also be applied to specific applications and services. Set up a pipe, add two differently weighted queues, plug the queues into rules for the specific service, and add a final catch all:

```
# ipfw pipe 30008 config bw 50Mbit/s queue 20
# ipfw queue 40008 pipe 30008 mask all weight 100
# ipfw queue 40011 pipe 30008 mask all weight 10
# ipfw add 5008 queue 40008 ip from any to any dst-port 5060
# ipfw add 5009 queue 40008 ip from any to any src-port 5060
# ipfw add 5010 queue 40008 ip from any to any iptos lowdelay
# ipfw add 5011 queue 40011 ip from any to any
```

The above rule set creates a pipe with 50 megabits per second of bandwidth and a maximum queue of 20 packets. Two queues are then created, where the first has a weight 10 times higher than the second. Traffic is then classified into one of these two queues. Packets with a source or destination port of 5060 (SIP), or packets with the IPTOS_LOWDELAY flag go into the high priority queue (40008), and the rest of the traffic goes into the low priority queue (40011). This should help ensure that VoIP calls do not suffer during periods of peak network activity.

Network traffic can also be shaped based on criteria specific to the machine the firewall is on. IPFW can match traffic based on the user, group, or jail that generated the traffic:



```
# ipfw pipe 30014 config bw 100Mbit/s
# ipfw pipe 30015 config bw 5Mbit/s
# ipfw pipe 30016 config bw 10Mbit/s
# ipfw add 5012 allow ip from any to any uid root
# ipfw add 5013 allow ip from any to any gid wheel
# ipfw add 5014 pipe 30014 ip from any to any jail 4 in
# ipfw add 5015 pipe 30015 ip from any to any jail 4 out
# ipfw add 5016 pipe 30016 ip from any to any
```

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to annie.romas@emc.com.

EMC² ISILON

This set of rules will shape the traffic based on the user or jail that generated it. The first three commands configure pipes with specific amounts of available bandwidth. The next two rules allow all traffic generated by root, or members of the wheel group, to pass unshaped. The next pair of rules matches traffic flowing in and out of a specific jail, creating an asymmetric connection, limiting traffic to 100mbps inbound, but only 5mbps outbound. The final rule matches all other traffic (other users and jails) and limits them to 10mbps total (not per direction).

BASIC NAT FOR A JAIL

IPFW can be useful if you need to quickly setup basic NAT to allow a number of jails on a public facing machine to access the Internet, without each having a dedicated IP address. This example assumes the jails have internal IP addresses bound to lo0.

To enable NAT, add the following to /etc/rc.conf:

```
gateway_enable="YES"
firewall_enable="YES"
firewall_type="OPEN"
firewall_nat_enable="YES"
firewall_nat_interface="em0"                    #public interface
firewall_nat_flags="redirect_port tcp 10.99.0.2:80 80 redirect_port tcp 10.99.0.2:443 443"
```

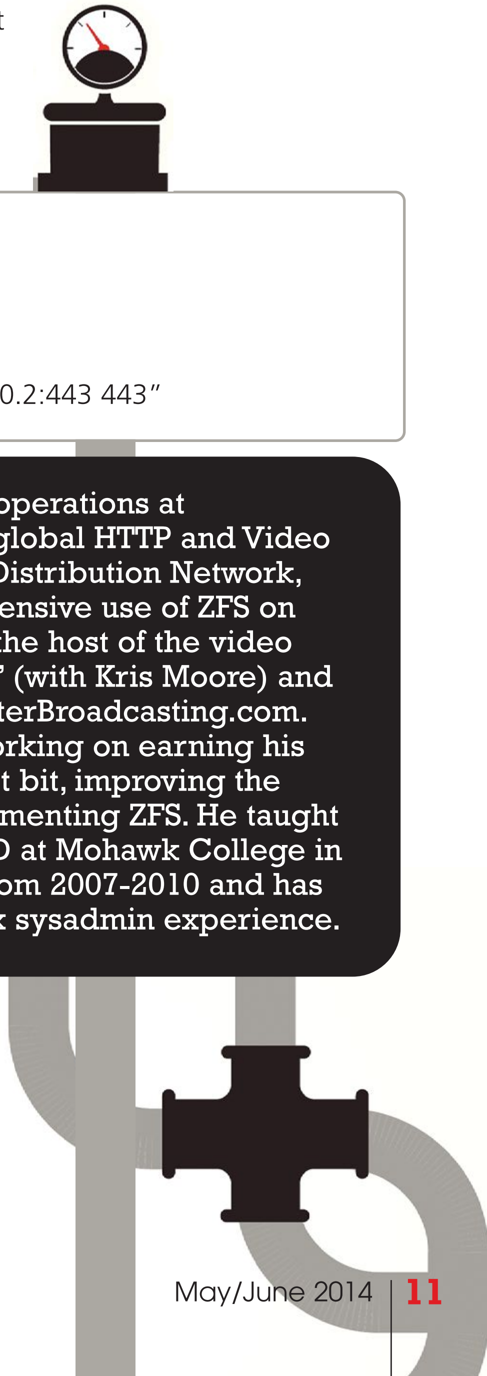
This will create an open firewall that will NAT outbound traffic via the IP address assigned to em0. It also configures port forwarding, to redirect inbound traffic on ports 80 and 443 to the private IP of the jail.

ADDITIONAL FUNCTIONALITY

IPFW has a number of other keywords that can be used to create advanced rule sets. The "prob" keyword, as part of the rule action, determines the probability that a packet will match the rule. Using this, the administrator can construct rules to direct portions of traffic in different ways, for split testing, load balancing, or simulating failure. Packets can also be "tagged" with numeric ID numbers to be used in later rules for things such as establishing trust relationships between interfaces. IPFW includes a forwarding capability; the "fwd" keyword will change the internal next-hop field of the packet as it passes through the firewall. This does not modify the headers of the packet, but changes how the kernel will route the packet and is ideal for IP- or port-based load balancing. IPFW can create a software monitor port with the "tee" keyword, which will send a copy of each matching packet to userland via a divert(4) socket. IPFW may also be used to mark packets with a specific FIB (Forwarding Information Base), causing matching packets to be routed using a specific kernel routing table.

CONCLUSION

This article only begins to scratch the surface of the capabilities and features of IPFW. The IPFW man page provides extensive documentation of each feature with plentiful examples. The FreeBSD handbook also includes a chapter on IPFW with additional explanation and examples. Users with questions are encouraged to address them to the freebsd-questions mailing list or post on the FreeBSD Forums. ●



Allan Jude is VP of operations at ScaleEngine Inc., a global HTTP and Video Streaming Content Distribution Network, where he makes extensive use of ZFS on FreeBSD. He is also the host of the video podcasts "BSD Now" (with Kris Moore) and "TechSNAP" on JupiterBroadcasting.com. Allan is currently working on earning his FreeBSD doc commit bit, improving the handbook and documenting ZFS. He taught FreeBSD and NetBSD at Mohawk College in Hamilton, Canada from 2007-2010 and has 12 years of BSD unix sysadmin experience.