

Linux Firewall-related /proc Entries

Brian Hatch 2003-07-14

Most people, when creating a Linux firewall, concentrate solely on manipulating kernel network filters: the rulesets you create using userspace tools such as `iptables` (2.4 kernels,) `ipchains` (2.2 kernels,) or even `ipfwadm` (2.0 kernels).

However there are kernel variables -- independent of any kernel filtering rules -- that affect how the kernel handles network packets. This article will discuss these variables and the effect they have on the network security of your Linux host or firewall.

What is Linux's /proc directory?

There are many settings inside the Linux kernel that can vary from machine to machine. Traditionally, these were set at compile time, or sometimes were modifiable through oft-esoteric system calls. For example each machine has a host name which would be set at boot time using the `sethostname(2)` system call, while `iptables` reads and modifies your Netfilter rules using `getsockopt(2)` and `setsockopt(2)`, respectively.

Modern Linux kernels have many settings that can be changed. Providing or overloading a plethora of system calls becomes unwieldy, and forcing administrators to write C code to change them at run time is a pain. Instead, the `/proc` filesystem was created.^[1] `/proc` is a virtual filesystem -- it does not reside on any physical or remotely mounted disk -- that provides a view of the system configuration and runtime state.

The `/proc` filesystem can be navigated just like any filesystem. Entries all appear to be standard files, directories, and symlinks, but are actually views into the kernel information itself. Some of these can be modified by root, but most are read only. To view these files, `cat` and `more` are your friends:

```
# cd /proc
# ls -l version
-r--r--r-- 1 root root 0 Jun 20 18:30 /proc/version
# cat version
Linux version 2.4.21 (guru@example.com) (gcc version 2.95.4 20011002) ...
```

Note that the kernel fudges a bit the `ls` output - these files will appear to have content when viewed, but will always have a length of 0 bytes. Rather than waste time figuring out how much output would be produced if the file were viewed, the kernel just reports 0 for most statistics, and gives the current time for all timestamps.

/proc/sys

All the `/proc` entries that can be modified live inside the `/proc/sys` directory. You can modify these in two different ways, using standard unix commands and via `sysctl`. The following examples show how you can set the hostname using both methods:

Changing /proc pseudo-files manually

```
# ls -l /proc/sys/kernel/hostname
-r--r--r-- 1 root root 0 Jun 20 18:30 /proc/sys/kernel/hostname

# hostname
catinthehat

# cat /proc/sys/kernel/hostname
catinthehat

# echo 'redfishbluefish' > /proc/sys/kernel/hostname

# hostname
redfishbluefish
```

Changing /proc pseudo-files via sysctl

```
# hostname
redfishbluefish

# sysctl kernel.hostname
kernel.hostname = redfishbluefish
```

```
# sysctl -w kernel.hostname=hop-on-pop
kernel.hostname = hop-on-pop

# hostname
hop-on-pop
```

Note that the main difference between these two methods is that `sysctl` uses dots[2] as a separator instead of slashes, and the initial `proc.sys` is assumed. `sysctl` can be run with a file as an argument, in which case all variable modifications in that file are performed:

```
# hostname
hop-on-pop

# cat reset_hostname
kernel.hostname=butterbattlebook

# sysctl -p reset_hostname
; Set our hostname
kernel.hostname=butterbattlebook
;
; Turn on syncookies
net.ipv4.tcp_syncookies = 1

# hostname
butterbattlebook
```

If `-p` is used and no filename is provided, the file `/etc/sysctl.conf` will be read.

The changes you make to `/proc` variables affect only the currently running kernel - they will revert back to the compile-time defaults at the next reboot. If you wish your changes to be permanent, you can either create a startup script that sets variables to your liking, or you can create a `/etc/sysctl.conf` file. Most Linux distributions will run `sysctl -p` at some point during the normal bootup process.

Firewall-related /proc entries

While there are many different kernel variables you can tweak, this article will only discuss those specifically related to protecting your Linux machine from network attacks. Also, we'll restrict ourselves to the IPv4 version, rather than IPv6, since the latter inherits variables settings from the former where appropriate anyway.

If you're interested in learning about other kernel variables, read the `proc(5)` man page. There are also several files in the kernel source inside the Documentation directory that may provide more information, `/usr/src/linux/Documentation/filesystems/proc.txt` and `/usr/src/linux/Documentation/networking/ip-sysctl.txt` are good starting points.

Some kernel variables are integers, such as `kernel.random.entropy_avail` which contains the bytes of entropy available to the random number generator. Others are arbitrary strings, such as `fs.inode-state` which contains the number of allocated and free kernel inodes separated by spaces. However most of the firewall-related variables are simple binary values where of '1' means 'on' and '0' means off.

A Linux machine can have more than one interface, and you can set some variables on different interfaces independently. These are in the `/proc/sys/net/ipv4/conf` directory, which contains all the current interfaces available, such as `lo`, `eth0`, `eth1`, or `wav0`, and two other directories, `all` and `default`.

When you change variables in the `/proc/sys/net/ipv4/conf/all` directory, the variable for all interfaces and `default` will be changed as well. When you change variables in `/proc/sys/net/ipv4/conf/default`, all future interfaces will have the value you specify. This should only affect machines that can add interfaces at run time, such as laptops with PCMCIA cards, or machines that create new interfaces via VPNs or PPP, for example.

Proc files

Below are `/proc` settings that you can tweak to secure your network configuration. I've prepended each filename with either `enable` (1) or `disable` (0) to show you my suggested settings where applicable. You can actually use the following handy shell functions to set these in a startup script if you prefer:

```
enable () { for file in $@; do echo 1 > $file; done }
disable () { for file in $@; do echo 0 > $file; done }
```

```
enable /proc/sys/net/ipv4/icmp_echo_ignore_all
```

When enabled, ignore all ICMP ECHO REQUEST (ping) packets. Does nothing to actually increase security, but can hide you from ping sweeps, which may prevent you from being port scanned. Nmap, for example, will not scan unpingable hosts unless `-P0` is specified. This will prevent normal network connectivity tests, however.

```
enable /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

When enabled, ignore broadcast and multicast pings. It's a good idea to ignore these to prevent you from becoming an inadvertent participant in a distributed denial of service attack, such as Smurf.

```
disable /proc/sys/net/ipv4/conf/*/accept_source_route
```

When source routed packets are allowed, an attacker can forge the source IP address of connections by explicitly saying how a packet should be routed across the Internet. This could enable them to abuse trust relationships or get around TCP Wrapper-style access lists. There's no need for source routing on today's Internet.

```
enable /proc/sys/net/ipv4/conf/*/rp_filter
```

When enabled, if a packet comes in on one interface, but our response would go out a different interface, drop the packet. Unnecessary on hosts with only one interface, but remember, PPP and VPN connections usually have their own interface, so it's a good idea to enable it anyway. Can be a problem for routers on a network that has dynamically changing routes. However on firewall/routers that are the single connection between networks, this automatically provides spoofing protection without network ACLs.

```
disable /proc/sys/net/ipv4/conf/*/accept_redirects
```

When you send a packet destined to a remote machine you usually send it to a default router. If this machine sends an ICMP redirect, it lets you know that there is a different router to which you should address the packet for a better route, and your machine will send the packet there instead. A cracker can use ICMP redirects to trick you into sending your packets through a machine it controls to perform man-in-the-middle attacks. This should certainly never be enabled on a well configured router.

```
disable /proc/sys/net/ipv4/conf/*/secure_redirects
```

Honor ICMP redirects only when they come from a router that is currently set up as a default gateway. Should only be enabled if you have multiple routers on your network. If

your network is fairly static and stable, it's better to leave this disabled.

```
disable /proc/sys/net/ipv4/conf/*/send_redirects
```

If you're a router and there are alternate routes of which you should inform your clients (you have multiple routers on your networks), you'll want to enable this. If you have a stable network where hosts already have the correct routes set up, this should not be necessary, and it's never needed for non-routing hosts.

```
disable /proc/sys/net/ipv4/ip_forward
```

If you're a router this needs to be enabled. This applies to VPN interfaces as well. If you do need to forward packets from one interface to another, make sure you have appropriate kernel ACLs set to allow only the traffic you want to forward.

```
(integer) /proc/sys/net/ipv4/ipfrag_high_thresh
```

The kernel needs to allocate memory to be able to reassemble fragmented packets. Once this limit is reached, the kernel will start discarding fragmented packets. Setting this too low or high can leave you vulnerable to a denial of service attack. While under an attack of many fragmented packets, a value too low will cause legitimate fragmented packets to be dropped, a value too high can cause excessive memory and CPU use to defragment attack packets.

```
(integer) /proc/sys/net/ipv4/ipfrag_low_thresh
```

Similar to `ip_frag_high_thresh`, the minimum amount of memory you want to allow for fragment reassembly.

```
(integer) /proc/sys/net/ipv4/ipfrag_time
```

The number of seconds the kernel should keep IP fragments before discarding them. Thirty seconds is usually a good time. Decrease this if attackers are forging fragments and you'll be better able to service legitimate connections.

```
enable /proc/sys/net/ip_always_defrag
```

Always defragment fragmented packets before passing them along through the firewall. Linux 2.4 and later kernels do not have this `/proc` entry, defragmentation is turned on by default.

```
(integer) /proc/sys/net/ipv4/tcp_max_orphans
```

The number of local sockets that are no longer attached to a process that will be maintained. These sockets are usually the result of failed network connections, such as the FIN-WAIT state where the remote end has not acknowledged the tear down of a TCP connection. After this limit has been reached, orphaned connections are removed from the kernel immediately. If your firewall is acting as a standard packet filter, this variable should not come into play, but it is helpful on connection endpoints such as Web servers. This variable is set at boot time to a value appropriate to the amount of memory on your system.

Other related variables that may be useful include `tcp_retries1` (how many TCP retries we send before giving up), `tcp_retries2` (how many TCP retries we send that are associated with an existing TCP connection before giving up), `tcp_orphan_retries` (how many retries to send for connections we've closed), `tcp_fin_timeout` (how long we'll maintain sockets in partially closed states before dropping them.) All of these parameters can be tweaked to fit the purpose of the machine, and are not purely security related.

```
(integer) /proc/sys/net/ipv4/icmp_ratelimit
```

```
(integer) /proc/sys/net/ipv4/icmp_ratemask
```

Together, these two variables allow you to limit how frequently specified ICMP packets are generated. `icmp_ratelimit` defines how many packets that match the `icmp_ratemask` per jiffie (a unit of time, a 1/100th of a second on most architectures) are allowed. The `ratemask` is a logical OR of all the ICMP codes you wish to rate limit. (See `/usr/include/linux/icmp.h` for the actual values.) The default mask includes destination unreachable, source quench, time exceeded and parameter problem. If you increase the limit, you can slow down or potentially confuse port scans, but you may inhibit legitimate network error indicators.

```
enable /proc/sys/net/ipv4/conf/*/log_martians
```

Have the kernel send syslog messages when packets are received with addresses that are illegal.

```
(integer) /proc/sys/net/ipv4/neigh/*/locktime
```

Reject ARP address changes if the existing entry is less than this many jiffies old. If an attacker on your LAN uses ARP poisoning to perform a man-in-the-middle attack, raising this variable can prevent ARP cache thrashing.

```
(integer) /proc/sys/net/ipv4/neigh/*/gc_stale_time
```

How often in seconds to clean out old ARP entries and make a new ARP request. Lower values will allow the server to more quickly adjust to a valid IP migration (good) or an ARP poisoning attack (bad).

```
disable /proc/sys/net/ipv4/conf/*/proxy_arp
```

Reply to ARP requests if we have a route to the host in question. This may be necessary in some firewall or VPN/router setups, but is generally a bad idea on hosts.

```
enable /proc/sys/net/ipv4/tcp_syncookies
```

A very popular denial of service attack involves a cracker sending many (possibly forged) SYN packets to your server, but never completing the TCP three way handshake. This quickly uses up slots in the kernel's half open queue, preventing legitimate connections from succeeding. Since a connection does not need to be completed, there need be no resources used on the attacking machine, so this is easy to perform and maintain.

If the `tcp_syncookies` variable is set (only available if your kernel was compiled with `CONFIG_SYNCOOKIES`) then the kernel handles TCP SYN packets normally until the queue is full, at which point the SYN cookie functionality kicks in.

SYN cookies work by not using a SYN queue at all. Instead the kernel will reply to any SYN packet with a SYN|ACK as normal, but it will present a specially-crafted TCP sequence number that encodes the source and destination IP address and port number and the time the packet was sent. An attacker performing the SYN flood would never have gotten this packet at all if they're spoofing, so they wouldn't respond. A legitimate connection attempt would send the third packet of the three way handshake which includes this sequence number, and the server can verify that it must be in response to a valid SYN cookie and allows the connection, even though there is no corresponding entry in the SYN queue.

Enabling SYN cookies is a very simple way to defeat SYN flood attacks while using only a bit more CPU time for the cookie creation and verification. Since the alternative is to reject all incoming connections, enabling SYN cookies is an obvious choice. For more information about the inner workings of SYN cookies, see <http://cr.yp.to/syncookies.html>

Summary

When creating a Linux firewall, or hardening a Linux host, there are many kernel variables that can be utilized to help secure the default networking stack. Coupled with more advanced rules, such as Netfilter (`iptables`) kernel ACLs, you can have a very secure machine with a minimum of fuss.

[Brian Hatch](#) is the author of [Hacking Linux Exposed, 2nd Edition](#), [Building Linux VPNs](#), and of the weekly [Linux Security: Tips, Tricks, and Hackery Newsletter](#). While he admits the `/proc` interface is extremely powerful, he prefers to change kernel variables by modifying `/dev/kmem` manually using `'dd if=/dev/random of=/dev/kmem bs=2 count=1 seek=...'`

Relevant Links

[1] Actually, kernel variables have been tweakable via the `_sysctl(2)` call since the olden days of the Linux kernel. Unfortunately, the actual kernel variable names change between versions, whereas the locations inside the `/proc` filesystem are more static, so `_sysctl(2)` is depreciated.

[2] `sysctl` can use slashes instead of dots, actually, but it is traditional/historical to use dots instead.

[Privacy Statement](#)

Copyright 2006, SecurityFocus