;login:
THE MAGAZINE OF USENIX & SAGE
November 2001 • Volume 26 • Number 7

Special Focus Issue: Security
Guest Editor: Rik Farrow

inside:

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

a secure OS-based firewall

System administrators working with limited resources must be resourceful. Sometimes, however, this same limitation can force the system administrator to think thoroughly about the problem at hand in order to utilize the scarce resources effectively. In this article, I present an example of how a limitation in disk space led me to rethink the role of the firewall. I also show how to build a BSD-based firewall that fits on a floppy and runs on a PC without a hard disk. This type of configuration could be, in my opinion, more secure than some commercial products and most OS-based firewalls running on a PC with a hard disk drive.

by Oscar Bonilla

Oscar Bonilla is an instructor at Galileo University in Guatemala. As part of his consulting career, he has designed the networks of three major ISPs in Guatemala and El Salvador. His main interests include operating systems, computer networks, and Internet security.

obonilla@galileo.edu

For the past few years, I have been working as an instructor for the computer science department of Galileo University in Guatemala (*http://www.galileo.edu*). As is often the case with universities in undeveloped countries, the teaching staff has to take over other responsibilities usually given to administrative staff in universities with more resources. Since the subjects I teach are operating systems and computer networks, I have been implicitly expected to be the system administrator for the computing infrastructure of the university. Universities in third-world countries usually have very limited resources for computer infrastructure. For example, the primary firewall of the university I work at, a PC running FreeBSD and IP Filter, crashed a couple of months ago and needed to be replaced. I found a machine lying around that had most of the specifications needed for the job, except for a hard disk drive. I asked the supplies department for a 40GB hard disk drive and got a 1.44MB floppy.

I asked the supplies department for a 40GB hard disk drive and got a 1.44MB floppy.

This lack of resources forces system administrators to think carefully about how the available resources will be used. In the firewall example above, having limited space makes you to think very carefully about what programs you will be installing on the machine and the uses you will give them.

A firewall is essentially a discriminating router. It receives IP packets on one network interface, tries to match the packet with one of the rules, and takes an action which could be routing or dropping the packet. All of this is done inside the kernel. Thus, firewalls do not really need many programs to accomplish their job. All that is needed is the kernel, a program for installing the firewall rules, and a few commands for initializing the network interfaces, turning routing on, and checking how the firewall is doing.

Most operating systems come with many programs and services installed; many of these services are even enabled by default. This is not surprising since they are general purpose operating systems and have to be useful to a wide variety of users. In the next section we will see how we can choose a minimal subset of these programs that lets a firewall do its job.

The One-Floppy Firewall (using PicoBSD)

PicoBSD is a variant of FreeBSD that fits in a single floppy. It lets you create a boot floppy that contains a custom kernel and a memory file system in which you can install your own subset of the programs available in the FreeBSD distribution. You can also add your own programs as long as there is still space in the file system.

You must have a FreeBSD system with full sources in order to build a PicoBSD floppy. This is because PicoBSD utilizes a technology called "crunched binaries." This means that several programs (and libraries) are combined in a single statically linked binary, thus saving considerable amount of disk space. This statically linked binary uses its

The bigger you make the MFS, the bigger the kernel will be, and the less memory you'll have for running user processes.

own name (argv[0]) to know which function it must perform. You only need to hard link it to every program name you have "crunched" in it to have it execute the proper functions. In order to build the crunched binary, all the binary files must be recompiled using special flags. The reason we hard link the different program names to the crunched binary, as opposed to soft linking it, is that a hard link only uses an entry in the directory for storing another name for the same inode, whereas a soft link uses another inode with the path of the linked program. Since inodes take space in the file system, it is more efficient to use hard links.

In FreeBSD, all of the required sources for PicoBSD are in the directory /usr/src/release/picobsd. In this directory you'll find several examples of various one-floppy configurations that do things such as routing, dial-up serving, etc. However, we'll see how to build a custom PicoBSD floppy with only the programs you want.

A typical configuration directory has the following hierarchy:

```
floppy_name/
        PICOBSD
        config crunch.conf
        floppy.tree/
    etc/
        ...files that will go in /etc on the floppy...
    root/
        ...files that will go in /root on the floppy...
mfs_tree/
    etc/
        ...files that will go in /etc on the MFS...
```

In the picobsd directory there are several examples that you can copy and modify to suit your needs. Alternatively, you can create the configuration directory from scratch. I will explain here how to create the hierarchy from scratch.

The floppy_name is the name of the directory that will hold all the configuration files for the floppy. Think of it as a project name. I chose to call it offw (One-Floppy FireWall).

The PICOBSD file is a FreeBSD kernel configuration file specifying which device drivers and options to install in the new kernel. It must start with the following lines:

```
# Line starting with #PicoBSD contains PicoBSD build parameters
#marker        def_sz   init     MFS_inodes      floppy_inodes
#PicoBSD       3500     init     4096            32768
options MD_ROOT_SIZE=3500     # same as def_sz
```

The first two lines are just comments. The third line is a comment for the kernel configuration program config(8). However, that third line tells the PicoBSD building script the parameters for building both the MFS and floppy file systems. The four parameters def_sz, init, MFS_inodes, and floppy_inodes specify the size of the MFS file system, what program to use as init, the number of inodes to use in the MFS, and the number of inodes to use in the floppy file system, respectively.

PicoBSD uses two file systems for operation: MFS and a floppy file system. MFS is a Memory File System that's patched into the kernel and loaded at boot time to the machine's RAM. Since MFS is patched into the kernel, it makes the kernel binary image bigger. The bigger you make the MFS, the bigger the kernel will be, and the less memory you'll have for running user processes.

The other file system used by PicoBSD is the floppy file system. This is the file system that remains on the floppy. In this file system, you'll have the kernel itself and some files in which modifications must persist between reboots. Another program that will be copied to the floppy file system is the kernel binary image itself. What this means is that as the kernel image gets bigger (because the MFS is bigger or because you have too many drivers in the kernel), the space available in the floppy file system will get smaller.

You have to achieve a balance between the size of the MFS, the drivers configured in the kernel, and the files you put in the floppy file system. The bigger the MFS, or the more drivers you have in your kernel, the bigger the kernel binary image that will be copied to the floppy file system, thus leaving less space for programs and files in the floppy file system.

So how do you go about choosing the right sizes? I basically put as much as I can on the MFS and leave the floppy file system only for configurations files and data that must persist between reboots. There are two reasons for doing this: (1) if you put files in the floppy that will be used a lot (like binaries), the floppy must be inserted and working in order to use those files, and (2) floppy access times are usually orders of magnitude slower than memory access times. So in the end, I've found that it works best to keep most of the files in the MFS and leave as little as possible in the floppy file system. Files that need to be modified easily (without recompiling the kernel image) should go in the floppy file system. For example, the firewall rules configuration file is a good candidate for the floppy file system.

Getting back to the PICOBSD kernel configuration file, the rest of the configuration lines will be what drivers you need and what options you want set in your kernel. You can find plenty of information on configuring FreeBSD kernels on the FreeBSD Handbook, available from the FreeBSD home page (*http://www.freebsd.org/*). Just remember to keep things to a minimum. Here's my complete kernel configuration file:

```
# Line starting with #PicoBSD contains PicoBSD build parameters
#marker         def_sz  init    MFS_inodes      floppy_inodes
#PicoBSD        3500    init    4096            32768
options MD_ROOT_SIZE=3500      # same as def_sz
# the machine architecture
Machine         i386
# the cpu type
cpu             I686_CPU
# an identifier for this kernel
ident           FIREWALL
# maxusers sets the static sizes of various structures inside the
# kernel, like maximum number of open files, etc.
maxusers        12

options         INET            #InterNETworking
options         FFS             #Berkeley Fast File System
options         FFS_ROOT        #FFS usable as root device [keep this!]
options         MFS             #Memory File System
options         MD_ROOT         #MFS as root
options         COMPAT_43       #Compatible with BSD 4.3 [KEEP THIS!]
options         PCI_QUIET

device          isa0            # ISA Bus
device          pci0            # PCI Bus
```

. . . it works best to keep most of the files in the MFS and leave as little as possible in the floppy file system.

```
# Floppy disk controller
device          fdc0    at      isa?    Port IO_FD1     irq 6     drq 2
# Floppy disk
device          fd0     at      fdc0    drive 0

# Keyboard controller
device          atkbdc0         at      isa?    port IO_KBD
# Keyboard
device          atkbd0          at      atkbdc? irq 1
# VGA display
device          vga0    at      isa?
# Console Driver
device          sc0     at      isa?
# Math Coprocessor [KEEP THIS!]
device          npx0    at      nexus?  port IO_NPX     irq 13
# Serial Port
device          sio0    at      isa?    port IO_COM1    flags 0x10   irq 4
# miibus is needed for certain Ethernet cards (like 3Com FastEthernet)
device          miibus
device          xl0             # 3Com FastEthernet Card
pseudo-device   loop            # local loop interface (lo0)
pseudo-device   ether           # Generic Ethernet Drivers
pseudo-device   pty     8       # Pseudo TTY's
pseudo-device   md              # memory disk
# Berkeley Packet Filter (not needed if you don't need tcpdump)
pseudo-device   bpf 4           # 4KB, for tcpdump
# IPFilter is the packet filter we'll use
options         IPFILTER
# Logging facility for IPFilter
options         IPFILTER_LOG
# Make IPFilter block all packets by default
options         IPFILTER_DEFAULT_BLOCK
options         PROCFS          #Process file system
```

The next file you need to create is config. This file is a configuration file that is sourced by the PicoBSD build script. It must contain only variable definitions. The one important variable that must be in this file is MY_DEVS, which tells the build script which devices to create in /dev inside the MFS. This is done passing each item in MY_DEVS to the standard FreeBSD MAKEDEV script usually found in /dev. This is what I have:

```
MY_DEVS="std vty10 fd0 pty0 cuaa0 bpf0 bpf1 ipl"
```

This example tells MAKEDEV to create standard devices (std), 10 tty's (vty10), a floppy disk drive device (fd0), the pseudo stty's (pty0), a serial port device (cuaa0), two Berkeley Packet Filter devices (bpf0 and bpf1), and an IPFilter logging device (ipl).

The ipl device is particularly important because it's the interface between IPFilter inside the kernel and the command line utilities that run in user space and are needed to load the firewall rules. The bpf devices can be left out if you don't need tcpdump in the firewall host.

The next file is called crunch.conf and contains the specifications needed to make the crunched binary.

The type of directives supported are (from the crunchgen(1) man page):

- srcdirs dirname: A list of source trees in which the source directories of the component programs can be found. These dirs are searched using the BSD <source-dir>/<progname>/ convention. Multiple srcdirs lines can be specified. The directories are searched in the order they are given.
- progs progname: A list of programs that make up the crunched binary. Multiple progs lines can be specified.
- libs libspec: A list of library specifications to be included in the crunched binary link. Multiple libs lines can be specified.
- buildopts : A list of build options to be added to every make target.
- ln progname linkname: Causes the crunched binary to invoke progname whenever linkname appears in argv[0]. This allows programs that change their behavior when run under different names to operate correctly.

Here's what I have in my crunch.conf file:

```
# We don't need PAM, NETGRAPH, IPSEC or INET6 (and we'll hint the
# sources that this is a RELEASE_CRUNCH
buildopts -DNOPAM -DRELEASE_CRUNCH -DNONETGRAPH -DNOIPSEC -\
        DNOINET6
# directories where to look for sources of various binaries
srcdirs /usr/src/bin
srcdirs /usr/src/sbin/i386
srcdirs /usr/src/sbin
srcdirs /usr/src/usr.bin
srcdirs /usr/src/gnu/usr.bin
srcdirs /usr/src/usr.sbin
srcdirs /usr/src/libexec
# Some programs are especially written for PicoBSD and reside here.
srcdirs /usr/src/release/picobsd/tinyware

# init is almost always necessary.
progs init # 4KB.
# Without ifconfig you wouldn't be able to configure IPs on your
# network interfaces.
progs ifconfig  # 4KB.
 # You need a shell.
progs sh      # 36KB.
ln sh -sh
# These are just some utilities I find useful.
progs echo      # 0KB.
progs pwd
progs mkdir rmdir
progs chmod chown
progs mv ln # 0KB.
progs mount
# minigzip is smaller than gzip.
progs minigzip # 0KB.
ln minigzip gzip
progs cp      # 0KB.
progs rm
progs ls
progs kill
progs df      # 0KB.
```

```
progs ps      # 4KB.
# ns is a lightweight version of netstat.
progs ns      # 4KB.
ln ns netstat
progs vm      # 0KB.
progs cat     # 0KB.
progs test    # 0KB.
ln test [
progs hostname   # 0KB.
progs login   # 4KB.
progs getty   # 4KB.
progs stty    # 4KB.
progs w       # 0KB.
# uptime gives you only the first line of w's output.
ln w uptime
# msg is a lightweight version of dmesg.
progs msg     # 0KB.
ln msg dmesg
progs kget    # 0KB.
progs reboot  # 0KB.
# less is smaller than more
progs less    # 36KB
ln less more

# sysctl is a program for changing kernel variables;
# it's needed, for instance, to enable IP Forwarding.
progs sysctl
progs swapon  # 0KB.
progs pwd_mkdb  # 0KB.
progs dev_mkdb  # 0KB.
progs umount
progs mount_std

progs route      # 8KB
# If you need an editor ee is as small as they get although it is
# at least debatable why you would need an editor in the firewall.
#progs ee       # 32KB.
#libs -lncurses

# It might be useful to have tcpdump for debugging purposes.
progs tcpdump    # 100KB.
special tcpdump srcdir /usr/src/usr.sbin/tcpdump/tcpdump

progs arp # 0KB.
# I wouldn't NFS mount anything on a firewall, but it can be done.
#progs mount_nfs    # 0KB.
#ln mount_nfs nfs
progs ping    # 4KB.
#progs routed  # 32KB.
progs traceroute  # 0KB.
ln mount_std procfs
ln mount_std mount_procfs

# It's nice to be able to ssh into your firewall and see how it's doing.
progs sshd    # includes ssh and scp

# These programs are needed to control IPFilter.
progs ipf ipfstat ipnat ipmon
```

```
# IPFilter logs using syslog which should be configured to log remotely to a
centralized log server.
progs syslogd

progs chflags

# Libraries Needed
libs -ledit -lutil -lmd -lcrypt -lmp -lgmp -lm -lkvm
libs -lmytinfo -lipx -lz -lpcap -lwrap
libs -ltermcap -lgnuregex -ltelnet
libs -lcrypto
```

The process of selecting the programs for the floppy is basically a trial and error procedure. You think of something you would like to have in the floppy, you add it, the image turns out to be too big, you think again if you really, really need it (or delete something else), and so on.

As you can see in the crunch.conf example, the firewall should have as few programs as possible to accomplish its job. You could strip this list even further, but don't make the system completely unusable. You should still be able to login to it and troubleshoot it.

The two directories mfs_tree and floppy.tree will be copied to the MFS and floppy file systems, respectively. You should have there the minimum set of configuration files needed for the system to function properly.

In my mfs_tree directory I only have a /etc directory containing a stripped down version of the following files:

disktab	host.conf	profile	services
fstab	hosts	protocols	shells
gettytab	login.conf	rc	termcap
group	motd	remote	ttys

The rc file is a special script in PicoBSD. While rc is called by init, just like in any other UNIX, the shell script overwrites itself during execution. The reason for this is that no file created in the MFS can be modified without recompiling the kernel. Although you can modify them in a running system, the changes will be lost if you reboot the machine. By having a minimal rc in the MFS that only copies the configuration files from the floppy and rewrites itself, we can achieve the most flexibility for system configuration. The real rc in the floppy file system can be modified by mounting the floppy on another machine, and it will not be necessary to rebuild either the kernel or the floppy binary image.

The rc script file that is in the MFS will be something like this:

```
#!/bin/sh
### Special setup for one floppy PICOBSD ###
# WARNING!!! We overwrite this file during execution with a new rc file.
# Awful things happen if this file's size is > 1024B

stty status '^T'
trap : 2
trap : 3

HOME=/; export HOME
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin
export PATH
dev="/dev/fd0c" #
trap "echo 'Reboot interrupted'; exit 1" 3
```

While rc is called by init . . . the shell script overwrites itself during execution.

```
# Copy from MFS version of the files, and then from FS version.
echo "Reading /etc from ${dev}..."
mount -o rdonly ${dev} /fd
cd /fd; cp -Rp etc root / ; cd / ; umount /fd
cd /etc
#rm files to stop overwrite warning
for i in *.gz; do
      if [ -f ${i%.gz} ]; then
      rm ${i%.gz}
      fi
done
gzip -d *.gz
pwd_mkdb -p ./master.passwd
echo "Ok. (Now you can remove ${dev} if you like)"
echo " "
. rc
exit 0
```

In the file system floppy, you should put the configuration files that you expect to change. For instance, you can put your firewall rules (ipf.conf) and your NAT rules (ipnat.conf) there if you're doing NAT. Another file that is usually there is the master.passwd file used to generate the passwd and db hashes used for authentication.

Here are the files I have in my floppy.tree /etc directory:

```
ipf.conf    ipnat.conf      rc      sshd_config
master.passwd     resolv.conf     syslog.conf
```

The interesting file here is rc. This rc script is the one that overwrites the rc in the MFS file system. This file is where I configure the network interfaces, load the firewall rules, load the NAT rules, start the appropriate daemons like SSH and syslogd, and rebuild the password files.

```
#!/bin/sh
mount -a -t nonfs
rm -rf /var/run/*
hostname firewall
ifconfig lo0 inet 127.0.0.1 netmask 0xff000000 up
ifconfig xl0 inet XX.XX.XX.XX netmask 0xffffff00 up
ifconfig xl1 inet YY.YY.YY.YY netmask 0xffffff00 up
route add default ZZ.ZZ.ZZ.ZZ
route add -net 192.168.0.0 192.168.0.1
ipf -Fa -f /etc/ipf.conf
ipnat -FC -f /etc/ipnat.conf
sysctl -w net.inet.ip.forwarding=1
(cd /var/run && { cp /dev/null utmp; chmod 644 utmp; })
sshd -f /etc/sshd_config
syslogd -s
dev_mkdb
cat /etc/motd
exit 0
```

Since this file is in the floppy file system, changing it is very easy. You only need to mount the floppy on any UNIX machine and make any modifications you want. Since the rc file in the MFS overwrites itself with this rc file, your modifications will have an effect on the firewall host.

Another trick is to put a /root directory in the floppy tree with a .ssh/ subdirectory. You can then store your SSH public keys in the floppy file system and configure SSHD to use only public key authentication.

After you have all files ready, it's only a matter of running the PicoBSD configuration script. The script is in the directory build/ and it's called picobsd. It has a menu which lets you modify various parameters and build the binary image of your floppy.

Once you have the floppy image done, you can use dd(1) to transfer it to a real floppy and boot your firewall from it. Remember to format the floppy first to make sure it does not have any bad blocks.

Once you have your one floppy firewall operational, there are some things you can try experimenting with. One of them is the kernel run levels in FreeBSD, and thus PicoBSD.

There is a variable in the FreeBSD kernel that specifies in which security context the kernel should operate. The name of the variable is kern.securelevel, and the default is -1 which means that no security is enabled. The possible values are:

-1 Permanently insecure mode – always run the system in level 0 mode. This is the default initial value.

0 Insecure mode – immutable and append-only flags may be turned off. All devices may be read or written subject to their permissions.

1 Secure mode – the system immutable and system append-only flags may not be turned off; disks for mounted file systems, /dev/mem, and /dev/kmem may not be opened for writing; kernel modules (see kld(4)) may not be loaded or unloaded.

2 Highly secure mode – same as secure mode, plus disks may not be opened for writing (except by mount(2)) whether mounted or not. This level precludes tampering with file systems by unmounting them, but also inhibits running newfs(8) while the system is multi-user.
In addition, kernel time changes are restricted to less than or equal to one second. Attempts to change the time by more than this will log the message "Time adjustment clamped to +1 second."

3 Network secure mode – same as highly secure mode, plus IP packet filter rules (see ipfw(8) and ipfirewall(4)) cannot be changed and dummynet(4) configuration cannot be adjusted.

The MFS and floppy file systems could be built with all files set as immutable, and right after loading the firewall rules you could switch to run level 3. The only disadvantage of this is that once the system is running you can no longer change anything, but I think that's as secure as you can get with a firewall.

Conclusion

Although firewalls are usually built using general purpose operating systems, they do not need all of the programs and utilities that come by default. All that firewalls really need are the kernel and a couple of programs for configuring network interfaces, loading the rules, etc. We have seen that all of these programs fit nicely in a single 1.44MB 3.5" floppy disk. There is no reason to have all of the extra unused programs in the disk, even if there is enough space for them. They make it possible to accidentally turn on an unwanted service. They also give a trespasser a rich development environment from which to launch further attacks. I believe that eliminating all of these unused programs makes a firewall more secure. In the case of a break-in, it gives the attacker an almost unusable system from which no further attack is possible.

. . . once the system is running you can no longer change anything, but I think that's as secure as you can get with a firewall.