;login:

THE MAGAZINE OF USENIX & SAGE

June 2002 volume 27 • number 3

inside:

SYSADMIN

Using Jails in FreeBSD for Fun and Profit

by Paco Hope

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

using jails in freeBSD for fun and profit

by Paco Hope

Paco Hope has a M.C.S. from the University of Virginia, where he worked as the head system administrator in the Department of Computer Science. Hope is a UNIX and information security consultant currently consulting with Halyard Systems.

paco@paco.to

Wouldn't it be great if you could create a little sandbox for a bunch of users where they could play to their hearts' content, even have root privileges, but they couldn't actually take the box down? What about isolating that buggy or vulnerable piece of legacy software that you just can't phase out, putting it in its own little world where even if it is hacked it creates a minimal liability on your network? How would you like to install one box in your collocation facility that takes up only two units of rack space but creates the feeling of 100 virtual servers inside?

"Jails" are a relatively recent development[1] in OS technology available in FreeBSD, and they offer the potential applications outlined above. They are similar to the genie's description of his magical servitude in Disney's *Aladdin*: "Phenomenal, cosmic power! . . . Itty bitty living space."

Over the years, various techniques have been created to try to isolate processes, partition resources, or otherwise control the interactions between processes and system resources. These techniques have been motivated by desires to conserve on hardware, consolidate management activities, or isolate risks of harmful interactions between applications. They are most interesting to apply to software that serves some public function, like FTP or DNS.

Near one end of the isolation spectrum are operating system calls, like "chroot," that cause the OS to restrict a regular process to see only a subset of the actual file system. Toward the other end of the spectrum are virtual machine environments. In these environments multiple instances of operating systems run on virtualized hardware. In between these two points is the "jail" system call. This article will describe the existing solutions and their limitations and then explain in some detail what jails can do. That foundation of capabilities will provide the basis for several example applications. A few limitations of jails will be discussed, and then some practical commands for how to actually set up and use jails will be provided.

chroot(2)

A chroot environment is one in which a process's view of the file system is restricted to a specific part of the hierarchy. The process's file system has a virtual root. Probably the most readily available example of a commonly chrooted process is the FTP daemon. Most ftpd programs use a small, partial copy of the file system rooted at the virtual root of the FTP hierarchy. The directory /var/ftp is actually the root for the FTP daemon. That is why /bin and /etc and /lib directories are often found on public FTP sites. There is actually a /bin/ls program, which corresponds to the real file /var/ftp/bin/ls, and it is executed when an FTP daemon services a client's ls request. FTP is not the only process commonly chrooted these days. IMAP software and DNS software are both commonly chrooted as well.

The chroot solution is attractive for protecting certain kinds of vulnerable processes. It is usually used to guard against software that can be coerced to read or write files that it should not touch in normal operations. Such vulnerabilities are limited in scope to just the chroot area of the file system. Thus the files available to such a vulnerable process are dramatically reduced.

There are significant limitations to chroot's applicability. Though a process may be chrooted, it is not restricted from opening network sockets, creating special device files, or seeing other processes. A process running as "root" in a virtual file system is still a process with full administrative privileges and the ability to interfere with other running processes on the system. Furthermore, exploits for chroot exist (see *http://openbsd.org.br/ouah/chroot-break.html*) that enable programs to break out of their chrooted directory. Chroot is rarely used as a technique by itself but, instead, is combined with other best practices to create a safer environment.

Virtual operating systems are attractive in some contexts because they tend to offer very flexible configurations.

Virtual Operating Systems

Virtual operating systems present a virtualized view of the entire system hardware to allow multiple instances of operating systems to run simultaneously on the same hardware. When that emulation is done well, the operating systems cannot distinguish simulated hardware from actual hardware. A single server can function as many distinct servers, and each instance of an operating system can be fully partitioned from all other instances. All the running operating systems must be managed like separate servers, though, because they essentially are.

Virtual operating systems are attractive in some contexts because they tend to offer very flexible configurations. Each and every instance can be completely controlled separately and independently. In fact, different operating systems can peacefully run simultaneously on the same hardware. There are management advantages to having a single physical system running different operating systems. There can be cost savings to operating a single physical system instead of multiple systems.

The primary disadvantage is that simulating the hardware can be expensive in terms of system resources. Each instance of the OS must have its own disk resources for its file system. The hardware virtualization and mediation is not free; it takes CPU cycles away from the applications themselves. In some contexts a completely separate instance of the operating system is overkill for the level of isolation that is needed.

There are good reasons to use virtual operating systems for certain classes of applications, just like there are good reasons to use chroot environments. Jails, by comparison, fit neatly in between. Jails offer a very compelling cost-benefits ratio for certain classes of problems where a fully virtualized system is too much cost or trouble but chroot is not robust enough.

What Jails Do

Jails combine the virtual file system approach with a limited amount of resource mediation to achieve a middle ground. For instance, creating and managing a jail feels very much like creating and managing a chroot environment. However, the operating system restrictions on jails far exceed a chroot environment and feel more like a virtual machine. The kernel mediates access to global system information and network resources, controls the creation and use of special devices, and logically isolates jailed processes from the main system and from each other. This makes jails safer from a security point of view, and gives jails a more complete feeling of isolation.

What follows is a general explanation of what jails do, why they are interesting, and what they can do for a system administrator.[2] For purposes of this discussion, we will call the regular, unrestricted operating system the "host environment" and the restricted jail environment the "jail environment."

. . . packet sniffing is not possible within a jail.

Jails Limit TCP/IP Access

Processes in a jail are limited to a specific set of TCP/IP socket operations and a single IP address. Typically the host environment is "multi-homed," meaning that the single physical system uses multiple IP addresses.[3] One of these IP addresses is assigned to the jail when the jail is first started. The kernel ensures that every packet leaving the jail environment has this assigned address as its source address. Raw sockets are disabled inside a jail environment. Even ping does not function from within a jail. This means that no spoofed packets originate from a jail; all packets will have the correct source IP address.

Jailed processes are also restricted in the kinds of packets they can receive. In a normal UNIX environment, software that binds a socket on INADDR_ANY (i.e., 0.0.0.0) will receive packets for all legitimate IP addresses on the system. A jailed process, however, only receives packets that are destined for its IP address, no matter how the socket is bound. Furthermore, promiscuous mode for network devices is prohibited to jailed processes, meaning that packet sniffing is not possible within a jail. All of these restrictions add up to a significant improvement in network security. If a hacker should infiltrate a jail, they will find many of their tools inoperative or severely limited.

Similar to TCP/IP are the inter-process communication (IPC) functions, which are also limited by jails. If jailed processes are allowed IPC functions like msgsnd and semctl, they can conceivably affect other processes running on the system. In FreeBSD 4.5-RELEASE, there is a system-wide control for allowing all jails to either do all IPC functions or none. Robert Watson's jail NG work breaks these controls into a per-jail setting,[4] but IPC access is still a binary, all-or-nothing proposition. By default IPC is not permitted to jailed processes, which makes them safer from each other.

Most of these limitations do not interfere with normal operations of normal processes. Mail servers, Web servers, DNS servers, and almost everything else that uses TCP/IP can operate normally in a jail with little or no reconfiguration. They do, however, provide a barricade that is a significant and useful part of an overall security regime.

Jails Limit File System and Device Access

The kernel also prevents the creation of special-device files by jailed processes, and it moderates the use of the special-device files that are available (e.g., /dev/kmem). The administrator should be careful what /dev entries exist in a jail, since those that do exist can be used. Devices that do not exist in /dev, however, cannot be created by jailed processes. The kernel prevents jailed processes from executing the mknod() system call. To simplify things, however, the MAKEDEV script has a "jail"[5] option that makes only those devices which are appropriate and/or necessary in a jail. Because a process in a jail has no access to the direct disk devices, it cannot grope around on the raw disk for data outside its prescribed perimeter. This, too, improves the security of a jail and helps complete the barricade between it and its host environment.

Jails Mask Processes and ID Spaces

Jails are isolated from each other and from the host environment itself. The user IDs (UIDs) and group IDs (GIDs) used inside a jail are the same as those used in the host environment. However, the operating system considers the jail identity (JID) as well, for purposes of determining access and privileges. The superuser in a jail can start and stop processes, send signals, and do many things, but the superuser's effects are limited to processes with the same JID. Likewise, any processes running as root within a jail can affect other processes, but only those with the same JID.

The quasi-isolation property of jails is the key distinguishing feature between jails and the two other approaches. A process running as root in a chroot environment is only limited in its view of the file system. It can still send signals to processes, reboot the system, open network sockets, etc. In a virtual operating system where an entire instance of the OS is running on simulated or arbitrated hardware, no instance of the OS is limited by the virtual environment. A compromised virtual OS is the same as any normal compromised system. Within a jail, however, a process running with "root" privileges actually has very limited abilities with respect to the rest of the host system. Jails offer just enough functionality to do a lot of legitimate work while isolating and limiting processes' access to unrelated resources.

Jails can allow a few physical machines to serve as many virtual servers.

Using Jails to Your Advantage

Jails can make sense on several different levels. They can save money, isolate risk, and offer an attractive virtualization technique. This section presents some of the benefits and some examples of how jails can be applied to specific problems.

JAILS CAN SAVE MONEY

Every IT department wants to save money, and jails might actually help. Jails can allow a few physical machines to serve as many virtual servers. Since most collocation facilities factor the physical dimensions of servers into their overall charges, fewer physical boxes will lower collocation fees. Even an organization that has no collocation charges to consider will appreciate the simplicity and cost-effectiveness of creating a new jail on an existing server, rather than purchasing new hardware when a new service must be provided.

Since jails are really a subset of the operating system, they can be upgraded en masse differently than real hosts. With some prior planning and automation, jail creation can be automated easily. This also means that jail upgrades can be rolled out easily, since taking down and restarting a jail is a very limited operation. This can translate into time and money savings by reducing management labor.

JAILS CAN BE A SECURITY TOOL

Implementing jails can offer another barricade in the network security battle. Every company must have some number of systems connected to the Internet. By isolating individual services in jails, the impact of a compromise or denial of service can be carefully managed. For instance, a machine with five jails might run a database, mail server, Web server, FTP server, and DNS server each in its own jail. A compromise in one jail would not necessarily lead to a compromise of the host environment or any other jails. The TCP/IP and socket restrictions limit what tools an attacker could use even if they got the foothold of "root" inside of a jail.

JAILS HELP LOGICALLY COMPARTMENTALIZE SYSTEMS

ISPs or IT departments that have very independent user bases may find the virtualization of jails to be an attractive way to partition administrative access. If the Web staff demands full and unfettered access to the Web server, they can have it – in a jail dedicated to the purpose. Now they do not need privileged access to a key server in order to operate just one of its many services. If there are junior administrators who need to manage a few services (for example, DNS and DHCP), those services can be put inside a jail where the junior administrators can have what access they need. Interestingly, all the existing techniques for compartmentalization and security still function within jails. So commands like sudo(8) or techniques like chroot can still be used inside a jail to further restrict access, or to impose finer-grained security. An ISP that wants to offer virtual hosting to advanced users can use jails to great effect. The user can appear to have "root" access inside their jail despite the fact that they actually have only limited access to the machine itself.

AN EXAMPLE USE OF JAILS: VIRTUAL HOSTING

Consider a system where an ISP wants to offer virtual hosting to its customers, but without allocating a full system and rack space to each customer. After establishing a baseline standard jail to serve as the per-customer virtual host, the ISP can add such customers quickly and easily.

The ISP allocates an IP address to one of its customer servers, and establishes its standard jail on one of its servers using that IP address. The jail runs sshd for secure login access, some kind of Web server, mail server, FTP server, and perhaps even a DNS server. The customer can login and have access to the real, live configuration files for all the important servers. If they pay the appropriate premiums and are sufficiently motivated (and competent), they can install new Web server modules, mail server configurations, FTP login IDs, or whatever especially suits their needs.

This approach to virtual hosting is interesting because giving a customer free rein in their jail does not interfere with any other customers at all. If the customer misconfigures their Web server such that it won't even start, all the other Web servers (in other jails) run unimpeded. If they have specific needs or specialized requirements, it is easy to provide for them in the context of their own jail.

As a variation on this theme, the ISP could install Webmin[6] in the jail and give selective access to selective subsystems in a controlled way. The user has the illusion of full control of the system, so they are happy. The IT staff have absolute control of the actual system, so they are happy.

Tips and Tricks with Jails

JAILS AND FLAGS

Jails can combine with another BSD file system feature, flags, to make them administratively safer, cleaner, and more easily managed.[7] The files which make up the operating system (e.g., /bin, /usr/lib, /sbin, etc.) can be made "immutable" using the chflags(1) command. Even an inept administrator or a joyriding hacker vandal will have a hard time completely ruining the jail. Immutable files can be made such that root (even root in the host environment, if desired!) cannot modify them. To make the operating system honor immutable flags, the kernel's security level must be set higher than the default.[8]

HARD LINKS ARE YOUR FRIENDS

Every jail requires a copy of the operating system – or at least enough to run the necessary software properly. This requirement can make jails disk intensive. A minimal FreeBSD installation with a few interesting services is likely to use between 50 and 100MB. The naïve approach to building jails would create one copy of this hierarchy per jail. However, the vast majority of these files need not be unique to the jail. In an application where many jails are likely, hard links can save a lot of disk space.

To make a hard-linked copy, one must first copy the directory hierarchy, and then recurse the source hierarchy and use the link(2) system call (or the ln command, which is equivalent) to create hard links between all the files in the source and the corresponding destinations.

In a system that has many jails, the hard-linked hierarchy allows the inode and data cache in the kernel to work at optimum efficiency. Consider a system with 10 httpd processes in 10 jails. The naïve approach would cause 10 copies of the httpd binary and copies of the corresponding shared libraries to be loaded into separate process address spaces in RAM. However, if 10 httpd processes and all their shared libraries are all hard links to the same inodes, the operating system can be much more efficient. Only the data blocks for the one httpd binary are loaded in the kernel's disk cache. A single text segment can be shared in RAM by all the running httpds, since they are all loaded from the same inode. There are several other subtle ways in which this scheme allows for shortcuts and efficiency in the kernel. Most of these efficiencies, however, are only realized on an active system with many, many jails. A two-jail system, for instance, would benefit less noticeably.

Hard links create a security concern because they create resources that are shared between jails. Shared files cross the otherwise rigid boundaries and potentially allow one jailed process to write to a file that many other jailed processes might read. Using hard links for disk space efficiency almost demands using the BSD flags above for additional protection.

Some Limitations of Jails

METERING, MONITORING, MANAGING

Most operations in jails are not especially mediated by the operating system. It is not possible, for instance, to dedicate a CPU to a jail, or regulate CPU usage by particular jails. That lack of control cuts both ways. It means low management overhead so that more CPU cycles and RAM are available to the regular processes. It does make it hard, however, to account for resource usage on a per-jail basis on a server that has hundreds or thousands of jailed processes.

Starting and stopping individual jails can be scripted, but there are no extant management tools yet that make it easy. Starting a jail by running /etc/rc in it works fine, but it is not possible to stop a jail in the same way a system is normally shutdown (i.e., /etc/rc.shutdown). The ability to inject a new process into an already running jail is critical to most management functions, but is lacking in the current jail implementation. JailNG fixes many of these limitations and makes jails easier to manage.

JAILS SOAK UP IP ADDRESSES

Each jail needs its own unique IP address. So the example above of a server with five jails would use six IP addresses: one for each jail and one for the host environment. Inside a firewall where private IP addresses are plentiful, this does not pose a significant problem. It may be significant to some organizations, depending on how they arrange their IP address space. One way around this limitation is to use a NAT mapping at a firewall entry point. Jails can then be assigned private IP addresses. The NAT mapping can direct port 25 connections (email) to one jail's private IP, while directing port 80 connections (HTTP) to a different jail's private IP, and so on.

NO DISK QUOTAS

Limiting the disk resources used by a given jail is difficult, because the operating system's quota facilities cannot be brought to bear on the problem. It is sometimes desirable to impose disk quotas on given jails, or to be able to impose quotas on users inside jails. There are no easy cookie-cutter solutions to this problem. A search on the FreeBSD mailing lists will reveal some attempts at launching jails in virtual (vnode) file systems. A discussion of such file systems is beyond the scope of this article, but they involve creating a large file (e.g., 100MB) and treating that file as if it were a disk device. By formatting the contents of the file as if it were a disk, and using a special mount command, the file's contents can be mounted as a file system. Since the file's size is fixed (100MB in our example), that imposes a hard quota on the entire jail. It does not, however, impose quotas on individual users inside the jail. It's a very coarse measure. This method is also complex and awkward. Increasing the quota is possible, but tricky. Backing up and restoring such file systems is very difficult as well.

Making Jails: Two Techniques

TECHNIQUE ONE: BUILD WORLD

This is what the man page recommends. It works, albeit a bit slowly. To summarize from the man page:

```
D=/here/is/the/jail
cd /usr/src
make world DESTDIR=$D
cd etc
make distribution DESTDIR=$D NO_MAKEDEV=yes
```

There are a few other options that will make the build proceed faster and will limit how many programs get built and installed in the jail. Several options can be enabled in /etc/defaults/make.conf such as NO_SENDMAIL, NO_LPR, and NO_BIND. They prevent large subsystems from being built, which will speed the build time and produce a smaller jail, assuming none of those subsystems is desired in the jail.

The disadvantage to this approach is the build time and disk space required to make "world" from sources. The advantage is the fine grain of control that's possible. To avoid installing compiler tools in the jail, for instance, (a prudent security measure), the directories that correspond to them can be removed from the /usr/src/gnu/usr.bin directory before building "world."

TECHNIQUE TWO: UNPACK DISTRIBUTIONS OFF THE CD

This method is simpler and faster by far, but has less fine-grained control. With the distribution CD mounted or copied into some file system location, each component of the operating system can be installed using its install.sh script. By setting the DESTDIR environment variable, each install.sh script will install its software in DESTDIR instead of overwriting the existing OS installation. A minimum jail should probably consist of the bin distribution and crypto distribution:

```
export DESTDIR=/here/is/the/jail
cd /cdrom    # or wherever your FreeBSD distribution lives
cd bin
sh install.sh
cd ../crypto
sh install.sh
```

The reason this technique has less control is that the entire bin distribution is installed, with compiler tools, sendmail, BIND, and many other programs. If for no other reason than space efficiency, a typical jail probably does not need most of what is installed in the bin distribution. From a security standpoint it is always best to only install exactly what is needed and nothing more. Installing from distributions will require some manual cleaning afterwards to remove unwanted software.

SETTING UP AFTERWARDS

After building a directory hierarchy using either technique above, there are routine chores to do in order to make the jail usable. The devices in /dev must be created, the root password should be set and a few other things must be set specially for a jail. Each command can be launched in the jail until the jail is able to run by itself:

```
cd $DESTDIR/dev
sh MAKEDEV jail
cd $DESTDIR
ln -sf dev/null kernel
touch etc/fstab

/usr/sbin/jail $DESTDIR jail-hostname 10.2.3.4 /usr/bin/passwd root
/usr/sbin/jail $DESTDIR jail-hostname 10.2.3.4 /usr/sbin/adduser
/usr/sbin/jail $DESTDIR jail-hostname 10.2.3.4 /bin/sh /etc/rc
```

The last command boots the jail. Assuming the jail's IP address is 10.2.3.4 and the host environment's networking is correct, it should now be possible to connect to it via SSH. Once logged in, it feels very much like a normal system. The man page for jail(8) discusses various modifications to /etc/rc.conf in the jail environment. It is also important to remove some programs from the jail that can leak information. There is no use for mount(8) or any of its related modules in a jail, and it's arguable whether any compiler tools belong in a jail. Once a good jail baseline is established, though, it's very easy to copy it to make more safe jails.

Conclusion

Jails enable system administrators to build relatively safe sandboxes that feel like virtual environments but have very low computational overhead. They are handy tools for system administrators to have in their toolboxes for certain classes of problems. While not a panacea, jails allow a number of configurations that have not been previously possible in the free UNIX operating systems and commercial desktop operating systems.

Adrian Filipi-Martin (*adrian@ubergeeks.com*) contributed to this article.

REFERENCES

1. P. Kamp and R. Watson, "Jails: Confining the Omnipotent Root," *Proceedings of the Second International System Administration and Networking Conference* (SANE), May 2000. *http://www.docs.freebsd.org/44doc/papers/jail/jail.html*

2. For a detailed discussion of how the operating system actually implements jails, see Kamp and Watson, "Jails"; E. Sarmiento, "Inside Jail," *Daemon News*, September 2001. *http://www.daemonnews.org/200109/jailint.html*

3. Multi-homing is accomplished through a technique commonly called IP aliasing. See the ifconfig command for more information on IP aliases.

4. R. Watson. "JailNG: From-Scratch Reimplementation of the Jail(2) Code on FreeBSD." *http://www.watson.org/~robert/freebsd/jailng/*

5. See Kamp and Watson, "Jails"; Sarmiento, "Inside Jail."

6. Webmin is a Web-based UNIX administration tool. *http://www.Webmin.com/*

7. See the man page for security(7) for more information on kernel security levels.

8. See *http://www.Webmin.com/*