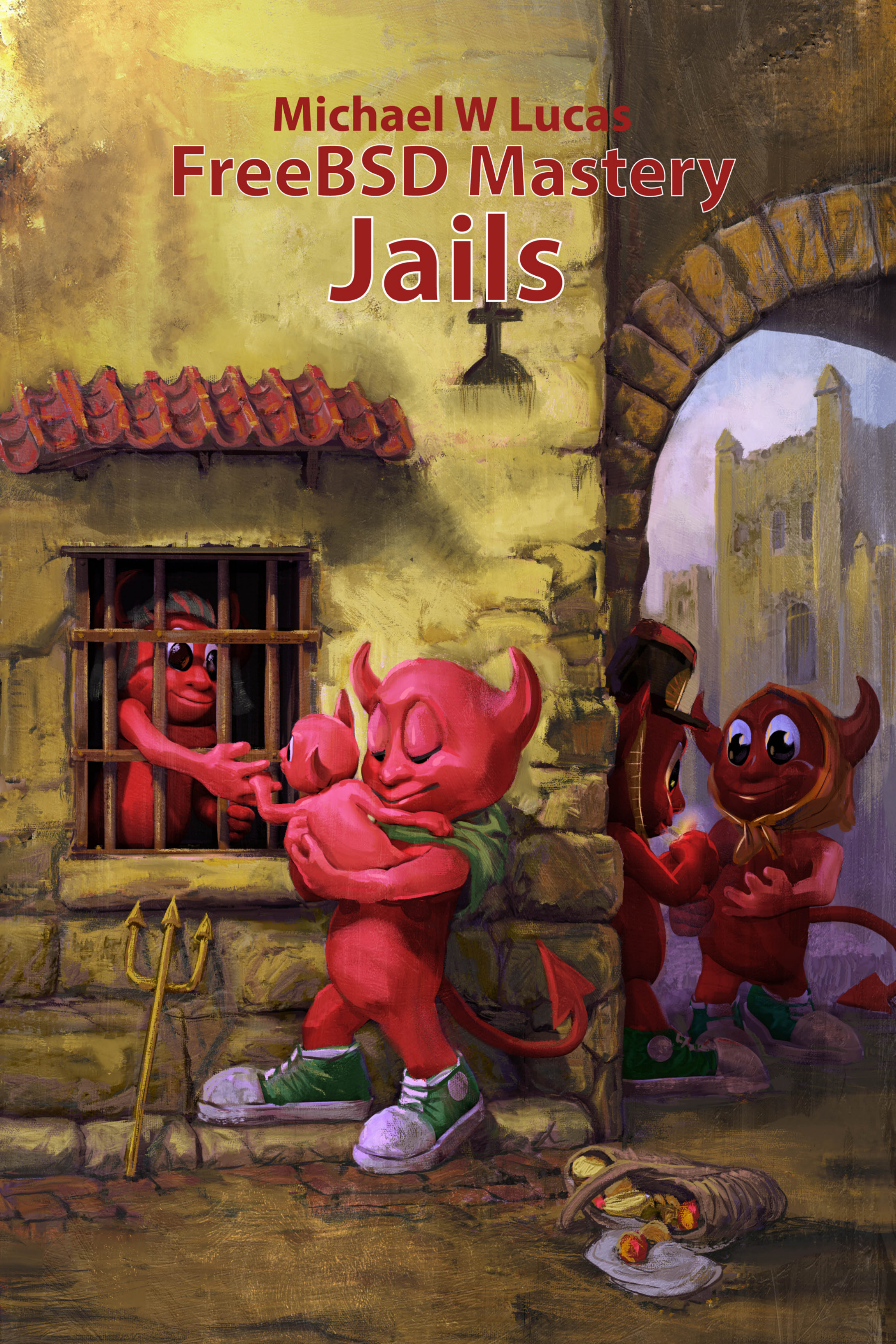
Michael W Lucas

FreeBSD Mastery Jails



FreeBSD Mastery

Jails

Michael W Lucas

Tilted Windmill Press

Authors: Michael W Lucas
Copyediting: Amanda Robinson
Cover art: Eddie Sharam, "The Beastie Prison,"
after John Phillip's "The Prison Window," 1857.

ISBN-13: 978-1-64235-023-4 (paperback)
ISBN-13: 978-1-64235-024-1 (hardcover)

Tilted Windmill Press
https://www.tiltedwindmillpress.com

FreeBSD Mastery: Jails

Michael W Lucas

Brief Contents

Complete Contents

Acknowledgements

I am immensely indebted to the entire FreeBSD community. All the folks who left bread crumbs in twenty years of blogs, forums, and mailing list archives provided valuable trails I followed to assemble this book.

I must especially thank my Patronizers, as well as all the fine folks who sponsored this specific book. You can see these folks at the end of the book, but with one hundred twenty-three backers this book has received more support than any other book I've written. Thank you all.

The members of the FreeBSD community who provided technical reviews before publication are owed a special thanks. If this book works for you, it's because of Fernando Apesteguía, Dave Cottlehuber, Lars Engels, Stefan Esser, Trix Farrar, Michael Gmelin, James Gritton, Poul-Henning Kamp, Marie Helene Kvello-Aune, Michael Dexter, Alexander Leidinger, Fuzz Leonard, Ed Maste, Edward Napierala, Colin Percival, Benedict Reuschling, Brandon Schneider, Matthew Seaman, Devin Teske, and Loganaden Velvindron from cyberstorm.mu. Any errors in this book are mine, and persist despite their valiant efforts.

Finally, I must thank Mateusz Guzik and Brooks Davis for eradicating iBCS2 support from the FreeBSD kernel. Without their timely rescue I would have needed to cover running Microsoft and/or SCO UNIX inside a jail, which would have transformed this from a happy little jails book into one of those madness-inducing Lovecraftian grimoires.

Without every one of these folks, this book would not exist.

Thank you all.

For Liz.

Chapter 0: Introduction

When I first became a system administrator, operating system installations were vital. Servers cost tens of thousands of dollars, and a medium-sized office had only one or two. Initial installations were rare events that senior sysadmins performed themselves—and with good reason, because everyone endured those decisions for five or ten years. We parsimoniously partitioned disks to balance performance and space utilization. We tuned or even rebuilt kernels to hoard every scrap of memory for our precious applications. What we'd today call routine updates and upgrades were tricky, dangerous events that required meticulous planning, several meetings, and hours or days of scheduled downtime.

Today, operating system installs are as common as sneezes and come in almost as many varieties. Virtualization has revolutionized system administration more than any other technology. We automatically deploy virtual systems to support increased load, and auto-destroy them once the need passes. The sheer number of operating system installs I'm responsible for would have horrified twentieth-century me.

Virtualization and Jails

Virtualization separates an operating system install from the underlying hardware, allowing you to stack operating system instances on top of one another. The operating system instance installed directly on the computer is called the *host*. The virtualization software running on the host provides virtual hardware for further operating system installs, called *virtual machines* or *guests*.

Full virtualization creates a virtual machine, from the BIOS or UEFI on up, for operating system installs. The virtualization system provides a keyboard, mouse, and remote console (usually over VNC or a similar protocol). The virtual machine gets a chunk of disk space, either on top of the host's filesystem or passed through to a disk partition. A subset of the host's memory gets set aside for the virtual machine. Full virtualization lets you run many wildly different operating systems simultaneously on one host. FreeBSD's full virtualization server, *bhyve*, lets you run anything from Linux to Microsoft Windows on a FreeBSD host.

Lightweight virtualization uses its own userland programs but runs on the host's kernel. Lightweight virtual machines don't have private disks, but instead use part of the host's filesystem. The host provides any networking. The host's kernel directs the lightweight virtual machine's memory, processor time, and networking. Lightweight virtual machines on the same host can share filesystems with one another. While the lightweight virtual machine cannot access the host, the host can usually access the lightweight virtual machine.

FreeBSD's jails were the first lightweight virtualization system in open source Unix. Their capabilities have been continuously enhanced since their introduction in 2000. Today's jails can run their own network stacks and packet filters, have private ZFS datasets, and do anything except run a different kernel. The modern FreeBSD kernel supports binary interfaces for FreeBSD versions back to FreeBSD 4, as well as certain versions of Linux. This compatibility is a great way to ditch that scary old hardware while keeping a vital application running.

Any time you see the word "host," it refers to the operating system install providing jail services. A host is the bottom layer of the stack of operating systems. The word "jail" refers to a lightweight virtual machine running on the host.

Jails versus Chroots

Jails have been called "chroots on steroids." That's not wrong, but it's not wholly correct either.

Chroot, or changed root, locks a process into a subset of a filesystem, restricting the files it can access. Chroots limit how much damage a deranged or damaged process can inflict on the host. At one time, chrooting servers to proactively contain intruders was a commonly recommended practice.[1] Today it's still an easy way to minimize potential damage, but not so widely recommended.

Chroot directories only contain files the chrooted program needs. Does the program need a shell? If not, don't put `/bin/sh` in the chroot. The same goes for every other program, shared library, and script on the host. An intruder who compromises a properly chrooted small program won't be able to do much with her access. Some servers, like OpenSSH's sshd(8), chroot themselves into an empty directory to thoroughly handicap potential intruders.

While a chroot limits disk access, it does nothing to restrict network access. A chrooted program can access all of the host's network interfaces and IP addresses. It shares the same process and memory space with all the other processes. Security tools like privilege separation and dropping privileges can help, but a chrooted daemon can still access a whole bunch of resources that aren't the filesystem.

Jails unshare these resources.

1 Not a commonly *deployed* practice, mind you. But recommended.

What Is A Jail?

A jail is a collection of namespace transformations.

Uh, what?

Unix has many different named entities, each meant for a particular type of application. Each entity has a discrete namespace. Process IDs are a namespace—each PID is unique within it. The filesystem is a unique namespace, and each file path is unique within it. An IP address attached to the host is unique in the network namespace. Usernames are unique. Just as a single operating system install can't have two PID 100s, it can't have two files named */etc/passwd* or attach 203.0.113.1 to multiple interfaces. You can create two identical usernames, but it won't end well.

A jail creates new namespaces for each of these.

Much like a chroot, a jail has a root directory somewhere on the host's filesystem. You might assign */var/www1* as the root directory for the jail **www1**. Processes inside the jail think the contents of */var/www1* are in the root directory. While a FreeBSD install can only have one */etc/passwd* file, jailed processes see */var/www1/etc/passwd* as */etc/passwd*. The host knows of many *passwd* files across the filesystem, but a jailed process perceives only its own namespace.

A chroot-style transformation won't work on the network. A jail is granted limited access to the host's network stack, and can use only the IP addresses the host permits. The host might have the whole IP range 198.51.100.1 through 254 attached to a NIC, but if a jail is only allocated 198.51.100.99, running `ifconfig em0` in the jail displays only 198.51.100.99. The host, though, can use the jail's IP address. If the host sysadmin wants to send a ping from a jail's IP, she can hijack that IP. The jail has no such option. (You can allocate a jail a private network stack, as we'll see in Chapter 9, but even that's subordinate to the host.)

Rather like the network stack, process IDs are unique across the host. Each jail gets a slice of the process namespace. Jailed processes can affect only other processes within that jail's process namespace. The host can create, signal, terminate, and otherwise twiddle processes in any jail's namespace.

The **`root`** user inside a jail is not the same as the **`root`** user of the host. A jail's **`root`** account can start and stop that jail's processes, administer that jail's users, and even mount filesystems inside the jail if the host permits, but is helpless beyond the jail.

A jail is wholly subordinate to the host. Even if an intruder gains access to the jailed system, the host retains supervisory access and can intervene. Think of it like a literal jail cell. The inmates can freely use their own bed. The warden gets a one-way mirror that looks into the cell and can impose changes at will.

Most jail documentation assumes that you'll put a complete operating system install in a jail, but that's not necessary. You can populate a jail sparsely enough to make the most hard-core chroot fan nod approvingly.

Jail Security

Some people argue that jails are not secure. Others claim they are secure. Who's right?

To be simultaneously correct and useless: both of them.

Like all security matters, the answer is "it depends." Today's sysadmins practice in an environment where everything is a security risk. We can't trust our hardware, our operating systems, or our software. Every so often, someone figures out a way to escape hypervisors. Developers patch and we all move on. So jails are not secure.

The solution isn't to rely on bare metal. A FreeBSD security expert spotlighted a class of problems with Intel processors back in 2005 that recently reached critical mass. Even if your main CPU is solid, what about all the other hardware? Nowadays, many server components have their own operating system. You can attach a serial console to the proper pins on a hard drive and get a command prompt.

Nothing is secure. Everything is terrible.

Asking if a jail is secure means asking about your application, your environment, and your risks. Should you provide nuclear missile controls with a jail? No. You also shouldn't control nuclear missiles on a public web site. Can you provide customer web sites in a jail? Probably, unless you have the sort of customers that get kicked off of one provider after another. Consider the risks that you, your organization, and your customers face.

Even if you utterly trust the security provided by jails, they're only one component of a security strategy. You certainly need privilege management and properly applied encryption. You need packet filters, proxies, and host- and network-based intrusion detection. Really, the only optional component is the disturbingly large and muscular people with antisocial personality disorder and spiked clubs.[2]

If you don't trust the security of jails for your application, restraining applications via jails is still a great way to ensure that programs can't interfere with each other, satisfy conflicting software requirements, and complicate attackers' lives.

2 Coating the spikes in your choice of nasty bacteria doesn't help with the current intrusion, but does deter future attacks. Word gets around.

Documentation and History

BSD supported lightweight virtualization before any other mainstream operating system. The earliest jail-type virtualization code appeared sometime between the 18 July 1975 snapshot of UNIX v6 and the 7 May 1979 snapshot of UNIX v7. Poul-Henning Kamp created modern FreeBSD jails in 1998. Many people have improved and expanded them since, building an incredibly flexible lightweight virtualization platform. Features were added, refined, and refined again.

All this iterative development caused a documentation problem.

While the Internet is famed for obsolete tutorials, jails in particular suffer from old documentation. Twenty-year-old mailing list posts will claim to solve your problem, and the method given works—if you're running FreeBSD 4.8 or earlier. If you want to manage System V IPC facilities in a jail, documents claim it's impossible and there's a sysctl and there are per-jail settings. You'll find recommendations to configure jails in */etc/rc.conf* and */etc/jail.conf*. And entire dynasties of jail management tools have risen and fallen.

This book deals with all this history by ignoring it.

I present modern jail management, glossing over or outright skipping older methods. Some folks prefer the older methods, and while there's nothing wrong with that you might as well start off learning the current way. If you're researching a problem and find advice that contradicts this book, check the date. That heavily-cited post from 2005 is likely to be wrong. (If you're still trying to use this book in 2035: I'm sorry.)

Prerequisites

Unlike most of my other books, *FreeBSD Mastery: Jails* is not an introductory book. Yes, it introduces jails—but using jails effectively means understanding FreeBSD. You must understand ZFS snapshots and clones. You'll need a grip on devfs(5) and nullfs(5). I won't explain `pkg.conf` or `zfs send`.

Similarly, you must understand networking if you want to use jails' virtual networking features. You can run basic jails without this expertise, but once you get into jail isolation and firewalls you must understand bridges and routing tables. You want to set up a DHCP server for your jails, or use PF to redirect connections to your single public address to the dozens of jails tied to the loopback? Good for you! Not my problem.

You can learn more about all of these topics in the FreeBSD Handbook and FAQ, other online resources, or in other books I've written, like the newest edition of *Absolute FreeBSD* (2018, No Starch Press) and the *FreeBSD Mastery* books.

Jails require no special hardware support. When you're starting to learn, though, I recommend dedicating a separate IP address on your test system to each jail. It isn't strictly necessary, but will simplify the learning process. Chapters 9 and 11 discuss sharing a single IP address among multiple jails.

I specifically focus on FreeBSD 12 and later, because of vnet virtual networking support. Much of this book works on earlier versions of FreeBSD, but if you're installing a new jail host proceed directly to 12 or newer.

Jails and ZFS

ZFS has a whole bunch of features that enhance the power of jails, chief among them clones. Clones are wonderful. Clones are powerful. Unwatched clones can destroy your system. Remember: *Clones grow.*

Even when you don't do much with them.

If you deploy a jail by cloning another jail, the clone starts out using zero bytes. Every change to the filesystem increases the clone's size. Edit `/etc/rc.conf`? The clone's size increases by the size of `rc.conf`. Upgrade the jail to a new minor or major release? The clone uses enough space for all the new binaries. Testing a database upgrade and churning the entire storage backend adds the database size to the clone size. Logs increase the clone's size.

Clone growth isn't a big deal when you have three or four jails. With five hundred jails, that scripted `freebsd-update upgrade` might consume every sector of disk you've got and demand more. Clone users must carefully monitor disk space. Don't blindly churn jail filesystems just because the tooling makes it easy. Don't upgrade cloned jails across major FreeBSD versions: upgrading FreeBSD 12 to FreeBSD 13 burns up any space savings offered by cloning. Instead, install new jails for new FreeBSD versions.

When a storage pool fills, ZFS throws a tantrum. Recovering from a full pool is complicated. Sysadmins using ZFS should reserve twenty percent of each pool's space for an overflow dataset, to get unmistakable but easily recoverable warning of disk capacity issues. It's much easier to reduce that reservation to fifteen percent and order new disks than recover from a completely full pool.

When using cloned jails, space reservations are not a suggestion. They are utterly mandatory.

Jail Management Tools

Every jail management system is an add-on tool. Even the base system's `/etc/jail.conf` is an add-on tool. Jails are created and destroyed by running jail(8) with a whole bunch of command-line arguments. The jail(8) command has built-in support for parsing `jail.conf` and bootstrapping its own command-line arguments, but other tools can generate those commands.

Also, each additional jail increases system complexity. A minimal standard jail has three mount points (`/`, `/dev`, and `/dev/fd`). A dozen jails equals thirty-six mount points. Depending on the filesystem used, those might all be separate datasets or a mass of directories. Each base jail brings at least a *dozen* mount points. Each jail has its own packages and users.

With all of this complexity, jails demand management tools. While we'll cover FreeBSD's integrated jail management, if you intend to have many jails, consider an add-on tool.

This book uses the newest version of iocage (https://github.com/iocage/). Highly flexible, iocage is the most popular tool for managing jails on a single host. It's written in Python and the developers respond to bug reports. The author takes reports of missing features as a personal challenge; if this book says that iocage doesn't support a feature, check the latest version to see if that's still true.

Ezjail is one of the oldest jail management tools. Very popular in its time, ezjail has not been updated for FreeBSD's last few versions. I used ezjail, I liked it, but it's sadly out of date.

BSDPloy (https://github.com/ployground/bsdploy) is a tool for DevOps-style mass jail deployment built on top of the Ploy infrastructure. It integrates with other tools like Ansible and Fabric. If you speak DevOps, consider Ploy and BSDPloy.

CBSD (https://bsdstore.ru/) is a hosting control panel for managing FreeBSD virtualization across multiple hosts. It does jails. It does bhyve. It sings, it dances, it brings you coffee. If you don't manage servers as much as you herd them, and if you want bhyve as well as jails, look at CBSD. I don't cover CBSD because I haven't yet written a bhyve book.

And more: Fubarnetes (in Rust), jailctl, jailadmin, iocell, pot, and more.

No matter what jail management system you choose, in the medium-to-long term you must understand what actions the tool takes. You need to understand the jail's underlying mechanisms. A jail might work great at setup, but when it goes sideways only your comprehension and troubleshooting can restore service.

Additionally, you must be able to differentiate jail problems, FreeBSD issues, and management tool bugs. For that reason, this book performs all jail configurations both by hand and with iocage. To differentiate, I call jails that are managed with base system tools *standard jails*, while iocage-managed jails are *iocage jails*. Iocage jails are perfectly standard, mind you, and you can perfectly emulate all iocage functions using only the base system, but it's a useful distinction when learning the tools.

Creating and removing jails is the smallest part of the problem. You must also manage existing jails. While virtualization was created to divorce the operating system from the hardware, many organizations use it as a way to kick problems further up the stack so that maintaining all this crud becomes someone else's problem. While I fully support delegating routine maintenance to other people, host sysadmins share responsibility for the jails they provide. Perhaps managing a jailed server is the web developer's problem, but his problems can escalate to become your problems. Host administrators can manage many issues more simply than jail owners. Leverage your tools to simplify everyone's lives, and tell your jail owners that you'll do so. While spending half an hour negotiating an operating agreement with your coworkers is annoying, it beats spending days troubleshooting subtle problems created by multiple systems administrators independently attacking one issue.

What's In This Book?

You're reading Chapter 0.

Chapter 1, *The Jail Host*, covers configuring your base FreeBSD install to best support jails.

Chapter 2, *Jail Essentials*, takes you through the core concepts of jails. You'll learn to set up a simple jail using both base system tools and iocage.

Chapter 3, *Jail Management*, teaches you about tuning, viewing, and otherwise herding jails.

Chapter 4, *Jail Filesystems*, discusses jail-specific aspects of filesystem management.

Chapter 5, *Packages and Upgrades*, covers different ways to perform these standard sysadmin tasks on jails.

Chapter 6, *Space Optimization*, discusses disk-saving tricks like clones and base jails, as well as how they change software management.

Chapter 7, *Living in a Jail*, covers some of the differences of working inside a jail versus having access to the whole system.

Chapter 8, *Jail Features and Controls*, takes you through some of the special features that make jails more than a chroot, like private System V IPC space, the ability to mount filesystems, securelevels, and more.

Chapter 9, *Networking*, goes into depth on how TCP/IP works in jails. We'll discuss virtual network stacks and bridges.

Chapter 10, *Extreme Jails*, shows how you can use jails inside jails and even run Linux in a jail.

Chapter 11, *Resource Restriction and Removal*, discusses preventing a jail from consuming the host. Or, alternately, how to give a jail more access to your host than usual.

Chapter 12, *iocage Bonuses*, introduces some features iocage provides that make it a powerful choice over other base system jails.

While jails have more features than I can cover in a book, once you master the topics covered here, you'll be able to solve most of your jail challenges.

Chapter 1: The Jail Host

Virtualization is grand, but even with nested layers of virtual machines eventually you hit an operating system installed on physical hardware. Yes, many components of modern systems run their own internal operating systems, but we call those *BIOS* and *firmware* and *UEFI* and try desperately to ignore them.[3] Installing and configuring your host so that it can most easily run jails will save you a bunch of later trouble.

An intruder that cracks your host gains control of all the jails running on that host. Plan from the beginning to run zero nonessential services on the host. If it isn't absolutely needed to manage the host, jail it. The jail host needs SSH and timekeeping, so run them on a network interface not exposed to the Internet. Web, database, and application servers? Untrustworthy critters, the lot of them. Jail them all. Deny potential intruders even the smallest opportunity to ravage your host.

If you want to use virtual networking with jails, I strongly recommend using FreeBSD 12 or later. You can do virtual networking with earlier versions of FreeBSD, but it's moderately hideous and either requires compiling a custom kernel or gluing a bunch of virtual interfaces together. It can be done, but vnet in FreeBSD 12 vastly simplifies and stabilizes virtual networking.

We'll start with storage, proceed to networking, and then look at detailed configuration of the host's services and kernel.

3 Most sysadmins routinely pretend that it's *not* turtles all the way down. It's the only way we stay sane. Sane-ish.

Storage

Jails are all about storage. Yes, they need process management and networking and all that, but those can be reconfigured without ripping everything out and starting over. If possible, separate your jail data from the host's data. On a host with several hard drives, dedicate a couple of drives to your operating system and the rest to jails. If using multiple multi-terabyte drives for a small operating system install feels wasteful, get a mainboard with a couple of directly attached high endurance flash drives. Either way, mirror the operating system drives with geom_mirror(8) or ZFS. Make your jail storage redundant as well.

Your filesystem needs to support standard Unix filesystem features, such as owners and permissions and so on. Don't try to build jails on a FAT filesystem, unless you're looking to amuse yourself with new and intriguing failure scenarios. Use either UFS or ZFS. If your host can support ZFS, I recommend choosing it. Jails appeared when UFS was the only serious filesystem option in FreeBSD, but ZFS has additional features that amplify and expand jails, such as completely delegating datasets to a jail.

Two filesystems that seem like obvious choices for jails are nullfs and unionfs. We'll make heavy use of nullfs, but more than one FreeBSD developer has described unionfs as "a basket of panic." The man page warns people not to use it. Some people disregard that warning and report excellent results, while others suffer endless agony. If you use unionfs and love it, good for you—but this book doesn't suggest it and I won't recommend it.

What about networked filesystems like NFS or AFS? Any networked filesystem presents challenges, from the usual performance issues to strange edge cases that affect only your precise configuration and the environment of one lone sysadmin who posted a desperate, detailed plea for aid to Usenet in 1994, got no answer, and was never heard from again. If you're willing to cope with those problems when they manifest at the least convenient moment, you can make jails work on a networked filesystem. The nfsuserd(8) daemon needed for NFSv4 doesn't interoperate well with jails, so you should probably use NFSv3 or earlier.

Remember, networked filesystems are not the same as networked disks. The mechanism used to connect the disks to the host is irrelevant. Fiber channel and iSCSI are not networked filesystems—they're networked disks. They're fine. Nobody has escaped a jail by exploiting fiber channel. Yet.

You'll see documents declaring that you should run lsvfs(1) to see the filesystems supported by the kernel, and to only use filesystems marked with the *jail* flag. This indicates filesystems that can be safely managed from *within* the jail, not filesystems that the host can use to provide storage *to* the jail. We'll discuss managing filesystems from within a jail in Chapter 8.

This book's examples of hand-crafted standard jails all go in the */jail* filesystem. Ideally, this is a separate set of physical media than your operating system. It could use either UFS or ZFS. Any examples that run on UFS are by definition standard jails and appear in */jail*.

Iocage works only on ZFS. I call my pool dedicated to iocage *iocage*, mounted at */iocage*. Any examples under */iocage* by definition use iocage.

Python uses fdescfs(5) to improve performance. If you're using iocage, enable fdescfs on the host with an */etc/fstab* entry.

```
fdescfs  /dev/fd  fdescfs  rw  0  0
```

Each jail can have several mount points. Expect to have a whole bunch of filesystem table files like `/etc/fstab`.

Comment everything very, very well. Document it all, or regret it.

Networking

The rules of TCP/IP declare that only one process can listen on a combination of IP addresses and ports at a time. You can attach a jail to any IP address and port on a host, but if a process on the host or another jail is already attached to that port and address, the jail won't work properly. The most obvious example is with the SSH server sshd(8). SSH normally attaches to port 22 on all IP addresses on a host. If your host monopolizes port 22 on all of the jail addresses, none of the jails can use SSH on port 22. They must use a different port, which adds complexity. "Hosts use port 22, while jails use port 23" is the sort of statement that makes people plot vengeance against sysadmins.

When you're first learning about jails, use a simple host with a single network interface but multiple IP addresses. Assign one IP exclusively for the host's use. Bind all of the host's services to that IP. Once you understand the basics of jails, you can start on a production host with multiple interfaces attached to different networks, VLANs, and all that fun stuff, but keep your initial test environment simple. We'll discuss more complicated network setups in Chapter 9.

If you don't have the luxury of multiple IP addresses you must track which daemons in which jails get bound to which IP addresses and ports, possibly on multiple distributed copies of a cryptographically signed giant spreadsheet. This leads to telling users things like "add 61000 to all your port numbers to find the services for jail 61. Yes, as in 61022, 61443, and so on."[4]

4 Yes, you'll be miserable—but the good news is, your users will be miserable too! They'll be steadfast allies in your battle for more IP addresses.

While jails routinely piggyback on the host's network stack, FreeBSD's vnet(9) virtual networking stack allows you to assign each jail a virtual network stack and its own routing tables. Each jail can have a completely different routing table. We discuss this in Chapter 9.

Protect your host by reducing its network profile as greatly as possible. Identify open ports with sockstat(1) or netstat(1). Remember, any service you run on the host is also a potential attack vector for intruders. A new FreeBSD install defaults to having three daemons listen to the network: syslogd(8), ntpd(8), and sshd(8).

syslogd

FreeBSD's logger syslogd(8) proactively binds to UDP port 514 on every available IP so it can send log messages to other hosts. If you don't need remote logging, or if your remote logging solution doesn't use syslogd(8), disable syslogd's network access with `-ss`. (This will probably be the default in FreeBSD 13.)

```
# sysrc syslogd_flags="-ss"
```

Security-sensitive organizations almost certainly require remote logging. Lock syslogd to the host's management IP with the `-b` flag.

```
# sysrc syslogd_flags="-b 192.0.2.2"
```

This frees up the syslogd port on other addresses for jails.

ntpd

All jails take their time from the host. FreeBSD's timekeeper daemon, ntpd(8), listens to all available IP addresses. There's no way to restrict this with stock ntpd—you can bind ntpd to a single interface, but not a single address. This is the one case where I'll tell you to not worry about it, and to run ntpd.

Jails cannot change the kernel's time. Even if you run ntpd in a jail, it can't actually change the host time. Any program running in a jail that connects to or from UDP port 123, as ntpd does, is almost certainly trying to sneak through a packet filter. There's no reason to make that little trick easier. Have the host monopolize UDP port 123.

If you worry anyway, disable the base system's ntpd(8) and install OpenNTPd. OpenNTPd has privilege separation as a bonus.

sshd

Everyone manages Unix hosts with sshd(8). Your users will want to SSH into their jails, and forcing them to use an alternate port will annoy them. That can't be helped if all your jails use the same IP, but in our ideal every-jail-has-its-own-IP world you'll restrict the host's SSH daemon to a single IP. Use the ListenAddress keyword in */etc/ssh/sshd_config* to restrict which address OpenSSH's sshd(8) uses.

```
ListenAddress 192.0.2.2
```

Ideally, this address is reachable only from your management network.

Chapter 9 takes you deep into jail networking, but this should get you started.

Enabling Jails

FreeBSD has built-in scripts for starting and stopping jails. The scripts read the configuration file */etc/jail.conf* and feed the results to jail(8). While it's possible to configure, start, and stop jails entirely on the command line with the `jail` command, as we'll see in Chapter 3, those commands become tedious as the number of jails increases. You're better off using the configuration file and enabling jail management through service(8).

```
# service jail enable
```

Iocage uses configuration files stored with each individual jail. When iocage starts, it reads its own configuration files and fires up jail(8) to create and start all the jails you've flagged for automatic start-up. Tell FreeBSD to start iocage at boot with `service iocage enable`.

You can use both jail management systems simultaneously. Jails configured in *jail.conf* don't inherently conflict with those managed by iocage. Both systems are merely front ends to jail(8), after all. They might have administrative conflicts, such as both trying to bind to the same IP address and port, but the two systems work independently. Iocage can manage hundreds of your most common types of jail, freeing you up to painfully maintain a couple of festering hand-tweaked artisan jails that require their own exotic weirdness.

Iocage Setup

First off, iocage requires ZFS. If you don't have ZFS, don't try to use iocage. Find the current iocage package with `pkg search iocage` , or pull the bleeding-edge version from github. Once you get that far, though, iocage requires specific locale settings. Add the following to *$HOME/.login_conf*.

```
me:\
   :charset=UTF-8:\
   :lang=en_US.UTF-8:\
   :setenv=LC_COLLATE=C:
```

Log out and log back in to activate these locale settings.

The first time you run any iocage command, it automatically creates iocage datasets on your root pool. If you have a separate pool for iocage jails, tell iocage about it before it spams your root pool. Point iocage at your pool with the `iocage activate` command. Here I dedicate the pool *iocage* to iocage jails.

```
# iocage activate iocage
ZFS pool ‘iocage’ successfully activated.
```

The iocage files get mounted at `/poolname/iocage`. If you use a default install and the default root pool, as nobody does, all iocage files appear under `/zroot/iocage`. Using a pool named `iocage`, the iocage files go under `/iocage/iocage`. For brevity, and to avoid sprinkling horrible filenames like `$POOL/iocage` throughout this book, all the examples use `/iocage`.

If you mistakenly tell iocage to set itself up on the wrong pool, use `iocage clean -a` to destroy all those datasets. For older versions of iocage, you must `zfs destroy -r` whatever dataset has the iocage files.

The first time you run iocage it creates datasets for downloads, jails, templates, and all other iocage functions. If you jump ahead and create your first jail, iocage will automatically prompt you to choose a release, download it, and set up the jail for you. Or you can run `iocage list` to display all your iocage jails, of which you have none yet. We'll explain all those datasets throughout this book.

Iocage has experimental color output. I don't use color here because this book will be printed in black and white, but if you want colors in your terminal set the environment variable `IOCAGE_COLOR` to TRUE.

Reboot your host to verify that everything is secured as tightly as you thought, and that the host restarts as expected. Once you've confirmed everything, you can start adding jails.

Chapter 2: Jail Essentials

While jails can have convoluted or highly specialized configurations, a simple jail on a dedicated IP address is pretty simple. The jail itself requires a dedicated root directory, an IP address, and a hostname. You'll need an installation source for the userland. We'll start by using the FreeBSD native tools and proceed to iocage.

Jails come in "thin" and "thick" varieties. A thin jail is a ZFS clone of another jail. A thick jail is a full, independent copy of a FreeBSD system. Our standard jails are thick jails, while iocage defaults to thin jails.

To start with, we'll build two standard jails. One, **`loghost`**, runs rsyslogd to collect everything from your network, while the second, **`logdb`**, runs PostgreSQL and aggregates those log messages.[5] We'll then build two iocage jails, **`www1`** and **`www2`**. We'll use those to test apps to easily view the log data.

Each jail needs a root directory. Everything under that directory belongs to the jail. I name my root directories after the standard jail hostnames. In this chapter I'll use *`/jail/loghost`* and *`/jail/logdb`*.

Let's start by discussing core configuration and networking.

Jail Networking

A jail's network connectivity is isolated from the host and from other jails. Each jail has an IP reserved for its exclusive use. This sounds great, until you hit **`localhost`** and the loopback interface.

5 You don't have to run my example applications. Feel free to use your own, inferior, choices.

Theoretically, every computer that speaks TCP/IP has access to the loopback interface as well as the loopback addresses 127.0.0.1 (IPv4) and ::1 (IPv6). These are artificial constructs that mean "this machine right here." A host can only have one loopback address attached to it, and can't delegate it to a jail. So how do jails get loopback addresses?

The kernel lies to them.

A jail substitutes the jail's main IP address for the loopback address. When you use the address 127.0.0.1 in the jail, you're actually using the jail's public address. For many applications, this simply doesn't matter… except when it does. If you run a jailed daemon on the address 127.0.0.1 expecting it to be isolated from the public network, you're heading for disappointment. The daemon is listening on the jail's external address.

While the loopback address is not as important as it once was, many software packages still need it and many sysadmins expect it to be available. Without a loopback address you can't, say, run a DNS resolver for local use and another for public consumption. The database for your web site needs to be configured to use a local socket rather than a TCP/IP port. None of these issues are insurmountable, so long as you're aware of them.

Chapter 9 discusses giving your jail its own private network stack and a legitimate loopback interface through vnet.

Jail Sysctls

FreeBSD has a whole section of the sysctl tree dedicated to jails, *security.jail*. All the jail sysctls are run-time tunable; you don't have to reboot to adjust them. You might need to restart a jail for the new setting to affect it, however. Many of these sysctls are merely defaults, and can be overriden by jail parameters, as discussed in the next section.

Programs and users can check the *security.jail.jailed* sysctl to see if they're inside a jail. The sysctl is 0 on a host, but 1 inside a jail.

Parameters and Properties

Configure jails with *parameters* (also called *properties*) that dictate how the jail should function. The jail's root filesystem? That's a parameter. The IP address assigned to the jail? Also a parameter. Can the jail mount filesystems, and if so, which types? Yep, a parameter. Some parameters manage jail(8), while others get passed to the host's kernel.

Kernel-related parameters control functions that interact with the kernel. How does the jail present mount points and filesystems? Which devices may the jail see? Can this jail create more jails, and if so, how many? Should we allocate a chunk of memory for System V IPC? All these require twiddling the kernel. A few of these parameters, such as the jail's securelevel, can be changed while the jail is running. Some parameters, such as those for less well-known filesystems, are tied to specific kernel modules.

Non-kernel parameters are only used to initially configure the jail. These give commands like "run this command at such-and-such stage," "log the console to this file," "use this virtual network stack," and "set this hostname." They do involve the kernel, but only in the ways that every other program does.

Jail parameters are deliberately modeled after sysctls, and consist of period-separated terms like *ip4.addr*, *allow.mount.devfs*, and *allow.socket_af*. Set a jail's parameters in the jail configuration file `/etc/jail.conf`.

Iocage has a similar set of parameters. Jail and iocage parameters often share names, but they're separate entities. Iocage uses its parameters to set the corresponding flags, parameters, and options for jail(8) commands. Iocage parameters use underscores where jail parameters use periods. Those example jail parameters I gave in the last paragraph map to the iocage parameters *ip4_addr*, *allow_mount_devfs*, and *allow_socket_af*. Generally speaking, anything that applies to standard jail parameters also applies to iocage parameters.

We configure jails by assigning values to parameters. We'll use the jail(8) format for these examples, but the values work exactly the same in iocage even though the configuration method differs. Parameter values can contain any non-whitespace character. If a value contains anything other than letters, digits, dots, dashes, and underscores, put single or double quotes around the value. (You can technically escape other characters with backslashes, but that gets ugly quick.)

Some parameters have values assigned to them with equal signs. You'll use such a parameter to tell a jail its root directory, using the *path* parameter.

```
Path="/jail/mariadb";
```

This jail's root directory is */jail/mariadb*.

Some parameters have a default, shown in jail(8). The parameter's presence implies the default setting, but you can change that default. Here I set the parameter *ip4*, enabling the feature at its default value.

```
ip4;
```

Use an equal sign to set a parameter like *ip4* to a non-default value. You can disable such a feature by setting it to "false" or "no," in *jail.conf* and iocage alike.

```
ip4=false;
```

A few parameters can have multiple values. Separate those values with a comma.

```
ip4.addr="203.0.113.231,203.0.113.232";
```

Complex parameter values might not permit separating the values with commas. In this case, repeat the parameter assignment with the += syntax to add a value. Here I assign *ip4.addr* two complex values.

```
ip4.addr="jpublic|203.0.113.225";
ip4.addr+="jprivate|198.51.100.225";
```

You can safely use `+=` to initialize variables.

Other parameters alter the environment with their mere presence, much as the whole mood of a room changes when Darth Vader walks in. Here, I set the *allow.mount* parameter, allowing the jail's sysadmin to mount additional jail-safe filesystems.

```
allow.mount;
```

Many `jail.conf` parameters have a "no" version that turns off a function. It's normally used to turn off default settings, and is equivalent to setting that parameter to "no" or zero. Here I disallow mounting additional filesystems.

```
allow.nomount;
```

Iocage does not have the *no* version of parameters. You can't set *allow_noset_hostname* in iocage; you must set *allow_set_hostname* to 0 instead.

Now let's see how to use `jail.conf`.

/etc/jail.conf

The format of `/etc/jail.conf` should look familiar to any sysadmin. Each jail needs a block of configuration, set apart by curly braces. Each parameter declaration ends with a semicolon. As with almost every other Unix configuration file, a hash mark in `/etc/jail.conf` denotes a comment.

Defining Standard Jails

Here's a *jail.conf* entry for **loghost**.

```
loghost {
  host.hostname="loghost.mwl.io";
  ip4.addr="203.0.113.231";
  path="/jail/loghost";
  mount.devfs;
  exec.clean;
  exec.start="sh /etc/rc";
  exec.stop="sh /etc/rc.shutdown";
}
```

Each jail starts with a name. The *name* parameter isn't explicitly labeled, but always appears in front of the opening brace. The jail(8) command will create a jail named "loghost."

The first parameter inside the braces, *host.hostname*, gives the jail's hostname. I called the jail **loghost**, but the virtual machine identifies itself by the Internet hostname **loghost.mwl.io**.

With *ip4.addr*, I assign the jail an IP address of 203.0.113.231. For our initial testing, use an IP address already attached to the host. Chapter 9 discusses having jail(8) automatically add and remove IP addresses from the host.

The *path* parameter gives the jail's root directory.

The presence of *mount.devfs* tells jail(8) to attach a device filesystem to the jail's */dev* directory. The default device filesystem assigned to jails includes only the device nodes needed for basic jail operations. It won't get device nodes for hardware, kernel dumps, sound cards, and so on. Jails don't need any of those device nodes, and access to those nodes might allow a clever inmate to escape his jail.

Jailed processes can inherit environment variables from their parent process. This is almost always bad. Here I explicitly define the *exec.clean* parameter to strip the jail(8) command environment when starting the jail.

Finally, the jail must know what program to run at startup and shutdown. A tiny jail might only run a single command. A full-featured jail needs the full FreeBSD startup and shutdown procedures defined in *exec.start* and *exec.stop*. I discuss jail startup and shutdown in Chapter 3.

Default Settings

Many jails share common settings. All of my jails start by stripping the shell environment via *exec.clean*. Almost all of my jails run complete FreeBSD installs, starting with */etc/rc* and shutting down with */etc/rc.shutdown*. Every jail needs a device filesystem. Defining these for each and every jail would be tedious and repetitive.

Fortunately, *jail.conf* lets you set system-wide defaults for jails. Specify your defaults at the beginning of the configuration file.

```
mount.devfs;
exec.clean;
exec.start=”sh /etc/rc”;
exec.stop=”sh /etc/rc.shutdown”;

loghost {
  host.hostname=”loghost.mwl.io”;
  ip4.addr=”203.0.113.221”;
  path=”/jail/loghost”;
}
```

Every jail on this host gets the *mount.devfs*, *exec.clean*, *exec.start*, and *exec.stop* settings. Defining new jails becomes a matter of defining only the unique settings.

Overriding Defaults

When a jail needs a parameter setting other than the defaults, override the defaults. Maybe the jail **mariadb** has a unique startup command.

```
mariadb {
  host.hostname=”mariadb.mwl.io”;
  exec.start=”sh /etc/rc.mariadb”;
  ip4.addr=”203.0.113.232”;
  path=”/jail/mariadb”;
}
```

This jail inherits all the defaults, but also runs the custom startup command */etc/rc.mariadb* rather than the full FreeBSD startup.

If a default parameter activates a feature by its presence, but you must disable that feature for a jail, set the “no” version of that parameter in that jail.

Overriding defaults gets a little tricky when you use the += syntax to chain multiple statements in a default. You can override the defaults within a jail definition, but you must give all the desired statements in the jail definition. Consider this default.

```
#default settings
exec.created=”logger jail $name has started”;
exec.created+=”cpuset -c -j $name -l 2-5”;
…
```

The *exec.created* parameter runs a command when the jail is created, but before the first jailed process is started. We run two commands, logger and cpuset. (Chapter 11 covers cpuset.) I need to run a slightly different cpuset command for my jail **logdb**. I can’t use a plain += and stack a third command on top of the first two commands, because the two cpuset commands are incompatible; running the default precludes running the jail-specific command.

```
logdb {
   exec.created="logger jail logdb has started";
   exec.created+="cpuset -j logdb -cl 6-7";
   …
```

I use a plain equals sign for the first *exec.created* statement, wiping out all previous definitions. The second *exec.created* statement gets added to the list thanks to the `+=`. If you can't figure out why a jail isn't working right, double-check you didn't forget the plus sign in one of your parameters.

Those of you who carefully read examples are probably wondering about the `$name` in the default settings. Let's talk about variables.

Variables

Variable substitution simplifies jail management. You can define your own variables, or pull some from the jail's settings. Variables in double quotes and unquoted strings get expanded, but not in single-quoted strings.

One obvious place for a user-defined variable is the path to the jail filesystem. If I move my hundreds of jails from */jail* to another partition, I must update every jail's configuration entry. That seems simple, but I know me. I'll screw that up. Instead, I define a variable for that filesystem.

```
$j="/jail";
…
loghost {
   …
   path="$j/loghost";
}
```

Moving my jails to another partition now requires updating only one *jail.conf* entry.

Much more interesting, though, is the ability to use jail parameters as variables. Once you define a parameter, you can use that parameter's value in later configuration statements. This isn't terribly useful for *ip4* but works beautifully with, say, *name*.

```
$j="/jail";
path="$j/$name";
host.hostname="$name.mwl.io"

loghost {
  ip4.addr="203.0.113.231";
}
```

We can now define global defaults for a jail's hostname and root directory. The only information unique to each jail is the IP address assigned. This has the additional advantage of synchronizing hostnames with the labels jail(8) uses.

Parameters with a period in them can be used as variables, but the parameter name must be enclosed in braces. Folks who deploy multiple domain names in their jails might prefer to use the fully qualified domain name as part of the root directory path. Here's a default that supports that.

```
path="/jail/${host.hostname}";
```

Using parameters as variables, combined with a coherent directory layout, lets you configure jails by their unique parameters.

You can now perform basic standard jail configuration. Let's check out iocage.

Iocage Configuration

Perform all jail configuration with the iocage(8) command, which closely resembles the zfs(8) and zpool(8) commands. Just as with standard jails, iocage configures each jail with parameters. Unlike standard jails, iocage stores all jail configuration with the jail's files.

Iocage creates a dataset for each jail, containing an `fstab` with details on special filesystems to mount in that jail and a `config.json` to record the jail's parameters. You'll also see a `root` dataset where the jail's actual filesystem resides.

The configuration file lists the jail's parameters, in alphabetical order.

```
{
  "CONFIG_VERSION": "18",
  "allow_chflags": "0",
  "allow_mount": "0",
  "allow_mount_devfs": "0",
  "allow_mount_nullfs": "0",
…
```

It's very readable with the naked eye. Readability is a design goal of JSON. Writable? Uh… yeah, *technically*, you can hand-edit JSON. Hand-editing is a great way to introduce errors, though. Iocage reads and writes configuration file parameters for you, and validates the result as legitimate JSON.

Read a jail's parameters with `iocage get`. You need two more arguments, the parameter of interest and the jail name. Here I grab the value of the *sysvshm* parameter from the jail **`www1`**.

```
# iocage get sysvshm www1
new
```

To see all parameters for a jail, use the `-a` argument and the jail name.

```
# iocage get -a www1 | less
CONFIG_VERSION:11
allow_chflags:0
allow_mount:0
allow_mount_devfs:0
…
```

This looks an awful lot like the configuration file, doesn't it?

To retrieve a parameter from every iocage jail, add the `-r` flag. This defaults to printing a fancy table with ASCII borders highly suitable for eyeballs, but I use this most often in scripts and scripts don't like fancy borders. Add the `-h` flag to strip away the decorations and produce simple tab-delimited output. Here I want to check all of my hosts to see if anyone's adjusting a particular parameter to a forbidden setting.

```
# iocage get -rh sysvshm | grep inherit
db5     inherit
```

See Chapter 8 for why I'm going to yell at the person who set up the jail named **db5**.

Changing Parameters

Use `iocage set` to change a parameter.

```
# iocage set sysvshm=new db5
Property: sysvshm has been updated to new
```

The new setting will take effect upon restarting the jail.

The iocage configuration files contain a few parameters that don't appear in jail(8) and won't work in *jail.conf*. These support functions such as clones and base jails. We'll discuss them in Chapter 6.

Iocage Booleans

A whole bunch of parameters are functionally toggles. The feature is on or off, enabled or disabled, true or false, 0 or 1. These boolean parameters are all stored as 0 or 1.

Iocage Defaults

While iocage's defaults are decent, your organization will unquestionably need a different default, sometime, somewhere. The jail name **`default`** is reserved for iocage's default settings. To view every possible iocage parameter and its initial setting, query this jail.

```
# iocage get -a default
```

To change the default settings for all jails, change the default jail.

```
# iocage set sysvshm=disable default
```

The default settings take effect on all jails where you haven't changed the parameter. If you manually tweaked a parameter on a jail, and then manually set it back to the default, changing the default setting of that parameter won't affect that jail. If you changed *sysvshm* on a jail, changing *sysvshm* on **`default`** won't alter that jail's *sysvshm*. You touched it, you own it.

The current iocage defaults are stored in */iocage/defaults.json*. To get rid of all your customizations to iocage's defaults, remove the file. A new defaults file will appear the next time you run iocage. Changing a single parameter back to the default value means figuring out what that default value is and changing it by hand. You do have a test host, right?

Standard Jail Userland

The easiest way to get a userland is to grab the *base.txz* file from a FreeBSD release or build. Keep old userland installation sources on your jail host in a place like */jail/media*, in a subdirectory named after the release. If you rebuild, reinstall, or duplicate a jail, you'll need the installation media.

You can also use a locally-built userland from */usr/src* and */usr/obj*. Make your own userland tarball by installing in a temporary directory, though. Don't install directly to a jail; you'll need reproducible jails. FreeBSD's build process includes options to install your custom world in a non-standard directory, thanks to the DESTDIR option.

```
# cd /usr/src
# make installworld DESTDIR=/tmp/jail
# make distribution DESTDIR=/tmp/jail
```

The `make installworld` command copies FreeBSD's program files to the destination and sets the appropriate permissions. The `make distribution` command creates supporting directories and files, such as many from */etc*. The end result is a pristine userland.

Bundle it up for deployment to your jails.

```
# tar -cf /jail/media/2019-05-25-current.txz --xz -C /tmp/jail/
```

Install your actual jails from this tarball.

Remember, the userland in the jail cannot be newer than the host's kernel: June 2020's userland won't work reliably on April 2019's kernel.

Iocage Userland

Iocage is intended for large scale deployments based on official FreeBSD releases. You won't be installing and upgrading jails from */usr/src*; instead, you'll use release tarballs and freebsd-update(8). The `iocage fetch` command grabs releases for you.

```
# iocage fetch
```

You'll get a list of available releases.

```
[0] 9.3-RELEASE (EOL)
[1] 10.1-RELEASE (EOL)
[2] 10.2-RELEASE (EOL)
[3] 10.3-RELEASE (EOL)
[4] 10.4-RELEASE (EOL)
[5] 11.0-RELEASE (EOL)
[6] 11.1-RELEASE (EOL)
[7] 11.2-RELEASE
[8] 12.0-RELEASE
```

Releases marked with EOL are past their end of life. While FreeBSD continues to offer them for download, they do not receive security patches and you run them at your own risk.

```
Type the number of the desired RELEASE
Press [Enter] to fetch the default selection: (Not a RELEASE)
Type EXIT to quit: 7
```

I want the newest FreeBSD 11 release, so I hit `7`. Depending on my connection speed, I can either grab lunch from the fridge or hit La Snootiér for a twelve-course snack.

```
Fetching: 11.2-RELEASE

Downloading : MANIFEST [####################] 100%
0Mbit/s
Downloading : base.txz [######--------------] 29%
19.57Mbit/s
…
```

The downloaded files get cached in */iocage/download*. Once iocage has all the files for a release it automatically extracts them and applies the latest security updates. I wind up with 11.2-RELEASE plus all available patches.

To disable security updates, you can add the `-NU` or `--noupdate` option to the `iocage fetch` command. That's a bad idea. Don't do it.

If your local firewall requires proxy configuration or authentication, you might need to download the release distribution files manually and then import them into iocage. You can also import your own custom FreeBSD releases. Find the desired release on the FreeBSD download server https://download.FreeBSD.org and grab the files *MANIFEST*, *base.txz*, *doc.txz*, *kernel.txz*, and *src.txz*. Stick them all in a directory named after the release. The `-f` flag tells iocage to extract the release from local files instead of the network. Use `-d` to specify the directory and `-r` to give iocage the name of this release. Here I've fetched the files for 12.0 into the directory */home/mwl/releases/12.0*.

```
# iocage fetch -f -d /home/mwl/releases/ -r 12.0
```

You might want to have iocage provide files from your host's software build in */usr/src*, without all the trouble of making a full release. That requires leaving iocage, but it's doable. First, create a dataset for the release, and a separate *root* dataset beneath it. My host is running FreeBSD 13-current. If you're going to create new releases from -current every so often, you might find it useful to name the iocage release after the SVN version you're building from.

```
# zfs create -p iocage/iocage/releases/r343907/root
```

Now install a jail userland to that directory.

```
# cd /usr/src
# make installworld DESTDIR=/iocage/releases/r343907/root
# make distribution DESTDIR=/iocage/releases/r343907/root
```

If you plan to regularly update your jails to the latest -current, I recommend using thick jails or perhaps even a base jail with a thick origin. ZFS clones will quickly diverge from their origin. If a jail sticks around long enough that divergence is a problem, export and import the jail as discussed in Chapter 12.

Iocage only applies security updates if freebsd-update(8) supports that release. I must patch my -current jails by hand. I wouldn't do this for standard production jails, but I'm contributing to FreeBSD by performing pre-release testing in my environment.

You can run `iocage fetch` multiple times for a release. It won't re-download the files, but it re-extracts previously downloaded files and applies all current security updates. If you went into the release directory and mucked with the files, this cleans up your mess.

If I know what release I want to grab, I can specify it on the command line with the `-r` option.

To see which releases you've already downloaded, run `iocage list -r`.

```
# iocage list -r
+---------------+
| Bases fetched |
+===============+
|  9.3-RELEASE  |
+---------------+
| 11.2-RELEASE  |
+---------------+
| 12.0-RELEASE  |
+---------------+
```

I can install a jail with any release on this list.

To get rid of a downloaded release, use the `iocage destroy -d` command. Specify the release with `-r`. FreeBSD 9.3 is long gone, and I don't want to install any more jails with it.

```
# iocage destroy -d -r 9.3-RELEASE

This will destroy RELEASE: 9.3-RELEASE

Are you sure? [y/N]: y

9.3-RELEASE has dependent jails (who may also have
dependents), use --recursive to destroy:
  www1
```

Uh oh. I still have a 9.3-RELEASE jail running. I must eliminate that jail before iocage will let me destroy the downloaded release. Iocage installs are ZFS clones, so even upgrading the jails to a newer release won't let me eliminate the underlying release. If you need independence, consider thick jails or base jails (Chapter 6).

To change a cloned, or "thin" jail so that it no longer depends on a release or template, export and re-import the jail (Chapter 12). All exported jails become thick jails.

Your First Jail

Now that you have software to run in your jail, you can set up a first jail. You'll use different methods for raw jail(8) and iocage(8).

Creating and Destroying Jails with jail(8)

Building a jail with jail(8) means populating a directory tree with a FreeBSD userland and then configuring it in *jail.conf*. Before you stick a userland in a directory, think about how you want to arrange your jail files. A jail might need additional support files that don't go inside the jail, but are necessary to start or configure the jail. A key example would be *fstab* files (Chapter 8). Do you want to keep these files with the jail, or separately?

I put all jail userlands in a directory directly beneath */jail*, like */jail/loghost*, and then add-on files in topic-specific directories like */jail/fstab*. Other people prefer having the jail's userland one directory further down, like */jail/loghost/root/*, letting them put add-on files under */jail/loghost*. Do whatever makes sense to you.

Here I want to set up a jail called **loghost**, running FreeBSD 11.2. The original distribution file is in */jail/media/11.2/base.txz*, and I want the jail in */jail/loghost*. Extract the release with tar(1). If you have a disk image handy, remember that tar can pull files straight from an unmounted ISO.

```
# tar -xpf /jail/media/11.2/base.txz -C /jail/loghost
```

If you need additional distribution sets, such as the system sources or documentation, extract them to the directory tree as well.

Now create a *jail.conf* entry. We created a perfectly suitable configuration when discussing jail configuration, including variables for additional jails.

```
$j="/jail";
path="$j/$name";
host.hostname="$name.mwl.io"

loghost {
  ip4.addr="203.0.113.231";
}
```

The jail is now ready to start.

To remove a jail, delete its directory, all its contents, and the *jail.conf* entry.

Creating, Viewing, Renaming, and Destroying Iocage Jails

Much like installing a standard jail, to create an iocage jail you must know the jail's IP address, FreeBSD release, and the jail's name.

Use the `-n` option to name the jail. If you don't assign a name, iocage assigns a Universally Unique Identifier (UUID), a thirty-two-character hexadecimal string. That's fine for your load-driven deployment system, but terrible for human beings.

The `-r` option lets you specify the release.

You can also specify any additional parameters on the command line. Here I specify the IP address with the `ip4_addr` parameter.

```
# iocage create -n www1 ip4_addr="203.0.113.234" \
-r 11.2-RELEASE
www1 successfully created!
```

The jail creation process is very quick because it leverages ZFS clones. Clones retain an attachment to their parent dataset, though. You might expect a jail to quickly diverge from the parent dataset, though, and want to prepare for that beforehand. A *thick* jail uses a copy of a dataset, not a clone, as its base. It's slower to create, but will save you pain in the long run—particularly if you expect this jail to *have* a long run. Use the `-T` flag to specify a thick jail.

```
# iocage create -T -n dns4 -r 11.2-RELEASE \
ip4_addr="203.0.113.243"
```

Creating a thick jail is much slower than cloning a jail, but it's a one-time cost.

If you want to create a highly customized jail—say, if you want to do an old-school install from */usr/src* and */usr/obj*—create an empty jail with `-e`.

View existing iocage jails with `iocage list`.

```
# iocage list
+-----+------+-------+--------------+---------------+
| JID | NAME | STATE |   RELEASE    |      IP4      |
+=====+======+=======+==============+===============+
| -   | www1 | down  | 11.2-RELEASE | 203.0.113.234 |
+-----+------+-------+--------------+---------------+
```

Iocage stores the jail's files in */iocage/jails/www1*. You'll find iocage's configuration file for the jail, *config.json*, and an *fstab* file iocage uses to mount extra filesystems before starting the jail. The *root* subdirectory is the jail's actual root directory.

To see more detail on each iocage jail, add the `-l` flag to `iocage list` to produce the "long list." You'll need a wide terminal to use the long list. The `-q` flag strips the output down to the jail name and IP address. Adding `-s` lets you sort by any of the headers. Here I list all of my jails, sort them by the release they were installed with, and strip out the fancy headers.

```
# iocage list -s RELEASE -h
13      wdb1    up      11.2-RELEASE    203.0.113.236
12      www1    up      11.2-RELEASE    203.0.113.234
15      wdb2    up      12.0-ALPHA2     203.0.113.237
14      www2    up      12.0-ALPHA2     203.0.113.235
```

Use `iocage destroy` to eliminate a jail and its dataset.

```
# iocage destroy www1
This will destroy jail www1

Are you sure? [y/N]: y
Stopping www1
Destroying www1
ioc-www1: removed
#
```

To tell iocage to skip confirmation, add the `-f` flag. Be careful, though; accidentally running `iocage destroy -f` on something you needed can cause iorage.

Destroy multiple jails simultaneously by naming all the jails on the command line.

```
# iocage destroy www3 www2 www1
```

If you screw up and mis-name your jail, don't destroy and reinstall it. Use `iocage rename` to change the jail's name. Here I fix one of my most common typos.

```
# iocage rename wwww1 www1
Jail: wwww1 renamed to www1
```

Now let's test our jails.

Jail Directory Testing

Test your jail directory tree by running a single command within the jail.

For standard jails, use jail(8) to run a command within a jail. The `jail` command takes four arguments: the jail's path, name, and primary IP address, and the command to run inside the jail. This starts the jail and runs the command. When you exit the command, the jail shuts down as well.

```
# jail path name address command
```

I want to run a shell inside my jail **loghost**. It's in the directory */jail/loghost*, and has the IP 203.0.113.231.

```
# jail /jail/loghost/ loghost 203.0.113.231 /bin/sh
#
```

I get a command prompt back… but it's not the same command prompt.

```
# uname -a
FreeBSD loghost 13.0-CURRENT FreeBSD 13.0-CURRENT #5
r336638: Mon Jul 22 13:32:15 EDT 2019     root@storm:
/usr/obj/usr/src/amd64.amd64/sys/GENERIC  amd64
```

I'm jailed! Specifically, I'm locked in */jail/loghost*, and now see it as my root directory. This jail is running in something that closely resembles single user mode. Look at the processes running in this jail.

```
# ps -ax
ps: /dev/null: No such file or directory
```

There's no */dev*? We haven't mounted one yet, so: no. No processes other than the shell are running, but you can run some basic commands like ls(1) and mount(8).

To test-start an iocage jail, run `iocage console`.

```
# iocage console www1
```

The jail **`www1`** is now running. Exactly as with a standard jail, the host is stripped bare. There are no user accounts or installed packages. Exiting the console won't shut down the jail, however.

Configure each jail before truly starting it.

Final Jail Configuration

Whether you install with raw FreeBSD tools or iocage, your new jail is even more bare than that created by the FreeBSD installer. There are no users. There is no root password. The network has an IP, but there's no resolver. "Time zone? What is this 'time zone' of which you speak?" Spend a minute decorating the jail before locking users into it.[6]

Many of the setup commands you must run aren't jail-aware. The passwd(1) and adduser(8) programs don't know anything about jails, and can only affect files in */etc*. You don't need to start them under jail(8) to set up your jail, though: a simple chroot(8) suffices.

```
# chroot /jail/loghost adduser
# chroot /jail/loghost passwd root
```

If you're using commands that affect processes, such as running a database client, you really must execute the program from within the jail's process context. Start the whole jail.

DNS

Each jail performs its own DNS resolution. You must provide a resolver configuration for standard jails. In many environments you can copy it straight from the host. Perhaps you have a recursive nameserver jail you want hosts to use. The local caching server you get by setting *local_unbound* to YES in */etc/rc.conf* has trouble because it can't

6 Every prisoner appreciates a considerate gaoler.

really attach to the loopback address. Either set up a place for your jails to get their DNS queries answered, or point all the jails at your provider.

You can use local_unbound on the host to provide DNS to the jails if you like; set up an ACL to only permit access from the jail network and **`localhost`**, and use the *interface-automatic* unbound option to help the server correctly return responses.

Iocage automatically copies the host's */etc/resolv.conf* into the jail when the jail starts. The *resolver* parameter lets you specify a different file on the host to be copied to the jail's *resolv.conf*.

Time Zone

Jails probably share the host's time zone. You can copy the host's */etc/localtime* into the jail. If a jail needs a unique time zone setting, though, run tzsetup(8). Conveniently, tzsetup is jail-aware. Use the `-C` flag to specify a chroot directory. If you know the desired time zone, give it as a final argument.

```
# tzsetup -sC /jail/loghost UTC
```

The iocage program automatically copies the host's time zone settings into the jail. Disable this by setting the *host_time* parameter to no.

I encourage using the same time zone in all your jails and hosts. Others disagree.

/etc/fstab

All core filesystem mounting should happen in the host, so jails work without a file system table. Many programs that can run within a jail, however, expect to find */etc/fstab* and assume that everything has gone horribly wrong if the file doesn't exist. Use touch(1) to create an empty filesystem table.

```
# touch /jail/loghost/etc/fstab
```

The jail's sysadmin can add their own custom filesystem entries to this file. Those mounts will only work if you've delegated them that permission, however. See Chapter 4.

/etc/rc.conf

If a jail runs a complete FreeBSD install, as discussed in this chapter, you'll probably use the FreeBSD startup script */etc/rc*. Jail creation works without */etc/rc.conf*, but you'll probably want the jail to start other programs at boot. If nothing else, users will manage most jails via SSH. Thankfully, sysrc(8) has a `-j` flag to modify a jail's *rc.conf* from the host.

```
# sysrc -j loghost sshd_enable="YES"
```

Add any other *rc.conf* settings the jail will need.

Root Password and Accounts

You must be within the jail filesystem to add user accounts or change passwords. User management commands don't need to run in the jail's process namespace, though; they merely must be rooted in the jail's root filesystem. You can use chroot(8).

You could also run a shell as discussed in "Jail Testing" earlier this chapter, and set a root password with passwd(1).

If you're going to manage this jail with SSH, run adduser(8) to add at least one user.

Other Configuration

If your network has additional requirements that can be met by editing configuration files, this is the time to set them up. Do you use LDAP for authentication? Copy your configuration files into the jail now. You won't want to change *nsswitch.conf* until we install packages in Chapter 5, but having your *ldap.conf* in place won't hurt.

While iocage installs all the security patches available when it downloads the FreeBSD release, it won't install any patches released between when you downloaded the release and when you install the jail. Similarly, with a standard jail you must run freebsd-update(8) (Chapter 4) to install all current patches.

Your jail is now ready for use.

Chapter 3: Jail Management

Starting and stopping jails is the least of your problems. You must be able to run commands inside jails, control the jail's environment and the environment of commands run inside jails, configure startup sequences, and install software.

When it comes to system management, iocage is a front end to jail(8). No matter how you manage your jails, you must understand this core jail functionality. We'll cover controlling jails with built-in system tools, and then discuss the same tasks with iocage.

Startup and Shutdown

A host treats each jail as an independent service, just like a web server or a database. Each jail runs its own processes, but from the host's perspective a jail is a single self-managing entity that contains a whole bunch of processes and memory. The host is only responsible for starting and stopping the jail, and disavows responsibility for managing the jail's processes.

Each jail must start and stop its own processes.

Startup and jail(8)

Start, stop, and restart jails with service(8). I've configured two jails using *jail.conf*, **loghost** and **logdb**.

```
# service jail restart
Stopping jails: loghost logdb ioc-www2.
Starting jails: loghost logdb.
```

I've configured two jails in `jail.conf`; where did **`ioc-www2`** come from? That's an iocage jail. I had an iocage jail already running, so service(8) stopped it. My command didn't restart **`ioc-www2`**, however, as it's not configured in `jail.conf`.

If you run both standard jails and iocage jails, you'll need to stop, start, and restart standard jails by name.

```
# service jail restart loghost
Stopping jails: loghost.
Starting jails: loghost.
```

When you set `jail_enable` to YES in `/etc/rc.conf`, FreeBSD runs `service jail start` at boot and starts everything configured in `jail.conf`.

I'll often configure jails for testing and debugging that I don't want to automatically start at boot. I don't want the mere presence of a `jail.conf` entry to make service(8) automatically start or stop these jails. The *jail_list* option in `rc.conf` restricts the jails that service(8) manages.

```
jail_list="loghost logdb"
```

This restricts service(8) to only running the jails **`loghost`** and **`logdb`**. It exchanges the problem of service(8) starting everything at boot for the problem of maintaining *jail_list*.

To shut down all configured jails, run `service jail stop`. To stop only a single jail, add the jail name.

Startup and Iocage

Iocage integrates with the FreeBSD service(8) architecture. If you set *iocage_enable* to YES in `/etc/rc.conf`, FreeBSD runs `service iocage start --rc` at boot. The iocage *boot* parameter dictates which jails autostart, and defaults to off.

```
# iocage get -r boot
+------+-------------+
| NAME | PROP - boot |
+======+=============+
| www2 | off         |
+------+-------------+
| www1 | off         |
+------+-------------+
```

I want iocage to automatically start **www1** at boot, so I change its *boot* parameter.

```
# iocage set boot=on www1
Property: boot has been updated to on
```

Running `iocage get` verifies this.

The `iocage start` command starts jails. Use the `--rc` flag to affect only jails set to start at boot, just like `service iocage start`. You can also select a jail by name.

```
# iocage start www1
* Starting www1
  + Started OK
  + Starting services OK
```

Use the ALL flag to start every jail, whether it's set to start at boot or not.

```
# iocage start ALL
```

If some iocage jails are running, using ALL starts only jails that aren't already running. It doesn't trigger reboots.

To have iocage reboot running jails, use the `restart` command. Specify a jail, or use ALL to bounce every jail.

```
# iocage restart www2
```

The good news is that jails boot very quickly, so those service interruptions won't last long.

Stop jails with the `iocage stop` command, giving either a jail name or, if you want to stop every jail, ALL. If iocage is enabled in *rc.conf*, you can also run `service iocage stop`.

```
# iocage stop ALL
```

Iocage reports the status of every jail as it shuts it down.

When iocage is enabled in *rc.conf*, FreeBSD runs `service iocage stop` at system shutdown. This turns off all machines that have the boot parameter set to on. It ignores any jails without that parameter, so you'll need to shut down those jails yourself—or let them unceremoniously die with the host.

Tuning Startup and Shutdown

When you start a jail, FreeBSD creates it in something that closely resembles single user mode. To enter multiuser mode the jail runs */bin/sh /etc/rc*. On an otherwise unconfigured jail, */etc/rc* starts cron and syslogd and awaits your commands.

Maybe you don't want to run a full startup, though, and for some reason you don't want to disable cron and syslogd in */etc/rc.conf*. Perhaps you want to not even mount */dev*, although lack of critical devices like standard in and standard out will cause most programs to fail quickly.[7] You might get the bright idea to write your own jail startup script. This impulse is almost always a mistake. If you insist on making this mistake, set the parameter *exec.start* to your preferred startup script.

```
exec.start="/bin/sh /etc/rc.custom"
```

7 Remove */dev/null* from a host and a surprising number of programs crash and burn. Experienced sysadmins understand that most software requires an uninterruptible supply of nothing.

When you shut down the jail, the jail runs `/etc/rc.shutdown`. Custom startup script users probably won't want the standard shutdown script. Use the *exec.stop* parameter to set your custom shutdown script. If you have no live data, including syslogd's log files, you might use true(1).

```
exec.stop="/bin/true"
```

The start and stop scripts run as the user starting and stopping the jail, normally **`root`**. Don't do this if your jail writes data to disk. Databases in particular get really cranky about being brutally terminated.

You might want to run a command on the host when starting and stopping a jail. The *exec.prestart*, *exec.poststart*, *exec.prestop*, and *exec.poststop* parameters support this. Before starting the jail, the host runs any command in *exec.prestart*. Once the jail's `/etc/rc` completes, the host runs the command in *exec.poststart*. Similarly, *exec.prestop* gets run before the jail's `/etc/rc.shutdown`, and *exec.poststop* runs after the jail has been fully killed. Here I set defaults for all of these in `/etc/jail.conf`.

```
exec.prestart="logger trying to start jail $name...";
exec.poststart="logger jail $name has started";
exec.prestop="logger shutting down jail $name";
exec.poststop="logger jail $name has shut down";
exec.timeout=90;
```

Now I get a `/var/log/messages` entry every time you start or stop a jail.

If you want to run multiple commands, stack them with the += syntax. Here I want to log that a jail has started, but I also want to add a CPU limiting command discussed in Chapter 11.

```
exec.poststart="logger jail $name has started";
exec.poststart+="cpuset -c -j $name -l 2-5";
```

Iocage's *exec_prestart*, *exec_poststart*, *exec_prestop*, and *exec_poststop* parameters work exactly the same way.

Where did this *exec.timeout* come from, though? Scripts, including `rc.shutdown` and your custom shutdown script, can fail. A simple crash isn't so worrisome, but what if the script hangs forever? Jails use *exec.timeout* to set a limit on how long to wait for *exec.prestart*, *exec.poststart*, *exec.prestop*, and *exec.poststop*. If these commands are still running after the timeout, the jail is not started or stopped. You must fix the command to start or stop the jail. If you keep this unset, jail(8) will wait forever for the jail to finish its commands.

Iocage sets a default *exec_timeout* to 60 seconds. Standard jails have no default timeout; you must set one with *exec.timeout*. I set *exec.timeout* to 90 in this example to match the 90 second timeout `/etc/rc.shutdown` uses before it forcibly terminates processes. If the jail's `rc.shutdown` can't end a process, the jail deletion will end it with prejudice.

Speaking of timeouts, we've all watched a host console as the system is shutting down and seen FreeBSD announce that it's forcibly massacring some program that flat-out refuses to exit. Such programs can monopolize jails as well. When a big program like a database is told to shut down, it's not unreasonable for it to take a few seconds to finish tidying itself up. Standard jails offer no grace time; they terminate all such processes immediately when the jail's `rc.shutdown` completes. The parameter *stop.timeout* gives a number of seconds the jail will wait for such programs to exit. It defaults to 0 in standard jails, but iocage defaults to a *stop_timeout* of 30.

Always set a timeout. *Always*. Maybe you're not using any exec. pre- or post- commands, and you don't think any of your jails will be running processes that need extra time at shutdown, but every real sysadmin knows that "things happen" and sometimes those things are terrible. Setting *exec.timeout* and *stop.timeout* to 30 at the top of `jail.conf` when you set up your host can prevent a whole bunch of pain later.

Tuning Before Startup

FreeBSD creates the jail, and then runs the first command in the jail. It's possible that you might need to run a command on the host once the jail exists, but before the jail actually starts. That's where *exec.created* comes in.

Most of the *exec.created* use cases are fairly advanced, such as jailing ZFS datasets and imposing resource limits. I'll demonstrate *exec.created* in those sections. It's one of those tools that's almost never needed, but on those rare occasions it's invaluable.

Dying and Life After Death

Jails have parameters that dictate how they behave during shutdown, and indeed have special parameters that allow them to exist even when nothing's running inside them.

When a jail is in the process of shutting down, the read-only parameter *dying* is set to 1. It's really hard to catch this in action, as dying is only set once the jail's shutdown script finishes running and the host kernel is in the process of ripping the jail out. In the unlikely event that you want to make changes to a jail as it's being killed, though, setting the parameter *allow.dying* to 1 permits such changes.

Normally, when a jail no longer has any processes running FreeBSD deallocates all resources, erases all the private namespaces, and forgets the jail ever existed. You might need a jail that doesn't have any processes in it to stick around, though. The `jail.conf` *persist* parameter tells FreeBSD to keep the jail around even when it has no processes. Iocage does not have a *persist* parameter. Most often, *persist* is used during the creation of complex jails.

Users and Jail Commands

Only **`root`** can start jails, but many add-on tasks such as pre- and post- scripts don't need to be run with privilege. Running such commands as **`root`** is probably unwise. While the *exec.jail_user*, *exec.system_jail_user*, and *exec.system_user* parameters let you set special users for certain tasks, they're not as useful as they might seem.

Remember, a jail is created by running a jail(8) command. That command needs permission to mount a device filesystem for the jail. Most users can't do that. You can use the *exec.system_user* to specify a user to run commands to create the jail, but you'll need to do a bunch of troubleshooting to make it work correctly.

Similarly, the *exec.jail_user* parameter defines a user in the jail's `/etc/passwd` to run in-jail commands as. Unprivileged users can't run `/etc/rc`, so it's largely useless unless you've written your own startup script—and even then, you'll wind up performing a huge amount of troubleshooting.

Finally, the *exec.system_jail_user* parameter has jail(8) use an account in the host's `/etc/passwd` to run commands as inside the jailed environment.

Jail Debugging

The `service` commands provide handy front ends to jail management, but what do you do when things go wrong? At that point, you must fall back onto raw jail(8) commands.

Debugging Standard Jails

That's not as bad as it might sound. The `-c` flag creates a jail, and `-v` adds verbosity. Here I start the jail **`loghost`** through jail(8).

```
# jail -vc loghost
loghost: run command: /sbin/ifconfig jailether inet
  203.0.113.231 netmask 255.255.255.255 alias
loghost: run command: /sbin/mount -t devfs -oruleset=5 .
  /jail/loghost/dev
loghost: run command: logger trying to start jail
  loghost...
loghost: jail_set(JAIL_CREATE) persist name=loghost
path=/jail/loghost host.hostname=loghost.mwl.io ip4.
addr=198.51.100.225 enforce_statfs=1 allow.mount=true
allow.mount.tmpfs=true allow.mount.zfs=true devfs_rule-
set=5 allow.raw_sockets
loghost: created
loghost: run command in jail: sh /etc/rc
loghost: run command: logger jail loghost has started
loghost: jail_set(JAIL_UPDATE) jid=4 nopersist
```

When a jail fails to start you can see exactly why. And by the time you finish this book, you'll understand all this gobbledygook.

To remove a running jail, run `jail -r` and give the JID or jail name. The jail will run its shutdown script and exit.

```
# jail -vr loghost
```

Shutdown problems are rarer, but still happen. To unconditionally, immediately destroy a jail without running any shutdown scripts, use the `-R` flag.

```
# jail -vR loghost
```

In addition to providing jail information via jls, jail(8) stores a jail IDs in a file in */var/run*. Each file starts with *jail_*, adds the jail name, and ends in *.id*. The JID for **logdb** would be stored as */var/run/jail_logdb.id*.

Debugging iocage Jails

Unfortunately, iocage does not provide a way to watch it create problem jails. It does provide error messages indicating where the problem lies, however.

The iocage program creates a configuration file in */var/run* for each jail it starts, named *jail.ioc-*, the jail name, and *.conf*. For example, the configuration file for iocage jail **www1** is */var/run/jail.ioc-www1.conf*. If you want to know why an existing jail is behaving a certain way, double-check its configuration file.

If an iocage jail refuses to shut down, you can blow it away with `jail -vR` and the jail name. Raw jail commands trump iocage, service(8), and all other add-on tools.

Jail Return Codes

You might think you want to use a jail's return code to see what happened inside a jail, and take action. You'd be wrong. The return code from jail(8) represents what `jail` experiences, not the specific error code from programs running or shutting down inside a jail. If the jail starts or stops, `jail` returns the usual 0. If the jail didn't start or stop correctly because a command inside the jail erred out, `jail` returns a 1.

In short, you can't rely on a jail's return code to represent events inside the jail. All it can tell you is "stuff worked," or "stuff didn't work."

Jail Boot Order

The order your jails start in might or might not be vital. If you have a jail with a web server backed by the database in another jail, the web site won't function until both jails are fully running. If the database comes up first, users will get no response until the web server starts. If the web server starts first, though, users who hit the site before the database is accessible will see some variant of the "Oh crud, my database is dead" page. As a sysadmin, I understand what's happening and don't really care about such transient errors. My managers, however, uniformly want to eliminate user-visible errors.[8] Both `jail.conf` and iocage let you manage jail startup and shutdown order.

Standard Jail Start Order

You can manage standard jail boot order with either *jail_list* or the *depend* parameter.

The *depend* parameter is probably the easiest way to control boot order. Rather than explicitly listing an order the jails should start in, depend lets you say that this jail requires another jail. You don't care what order your flock of jails starts in, so long as certain jails start before certain other jails. Suppose I need **`logdb`** to start before **`loghost`**. I can specify that in loghost's `jail.conf` entry.

```
loghost {
  ip4.addr="203.0.113.232";
  depend="logdb";
}
```

If you have no *depend* parameters set, but use *jail_list* in `rc.conf`, FreeBSD starts and stops jails in the order listed in *jail_list*. If the shutdown order should reverse the startup order, set the `rc.conf` option *jail_reverse_stop* to YES.

The *depend* parameter overrides a jail's placement in *jail_list*, allowing required jails to jump ahead in line.

8 My perennial suggestion to eliminate user-visible errors by eliminating users often makes it surprisingly far up the org chart.

Iocage Start Order

Iocage uses the *depends* parameter rather than jail(8)'s *depend*, but it works exactly the same way. If you want a jail to require that another jail to be running before it starts, list the required jail in *depends*. Here I need database server **`wdb1`** to be running before **`www1`**.

```
# iocage set depends=wdb1 www1
Property: depends has been updated to wdb1
```

If I need multiple jails to be running before starting this jail, I'd enter them in quotes.

```
# iocage set depends="wdb1 wdb2" www1
Property: depends has been updated to wdb1 wdb2
```

The *depends* parameter overrides *boot*. If a jail set to run at boot requires another jail to start, that other jail is started. Look at the boot property of my iocage jails.

```
# iocage get -rh boot
www2    off
wdb2    off
www1    on
wdb1    off
```

Only **`www1`** explicitly starts at boot.

Let's shut everything down and see what starts at boot.

```
# iocage stop ALL
…
# iocage start --rc
# iocage get --rh state
www2    down
wdb2    up
www1    up
wdb1    up
```

The jail **`www1`** needs jails **`wdb1`** and **`wdb2`** running before it can start, so iocage started them. That seems handy, but this implicit dependency breaks during system shutdown. While your web server might require its database servers to function, the web server doesn't care if the database servers are running when it's off.

```
# iocage stop -rc
# iocage get -rh state
www2    down
wdb2    up
www1    down
wdb1    up
```

The database servers are still running. If this had been a real host shutdown, your databases would be ungracefully terminated. That's widely regarded as a bad idea.[9] Set the boot parameter for every machine you want FreeBSD to start for you.

Iocage computes startup dependencies before starting any jails, letting you daisy-chain the *depends* parameter. If **www1** needs **wdb1**, and **wdb1** needs **iscsi1**, iocage starts and stops the jails in proper order. The downside of this is that, predictably, circular dependencies make iocage unhappy. If **www1** depends on **wdb1**, but **wdb1** requires **www1**, you've created a loop. Iocage won't start anything.

Dependencies can quickly grow frustratingly convoluted. Much of the time we want to follow guidelines like "start the iscsi jails first, then the database jails, then the web servers." Establish this sort of system with the *priority* parameter. Priority is an arbitrary integer, defaulting to 99. A *priority* of 1 gets started first, while everything else gets started in order. Group your jails by role and assign their priority accordingly, then patch up the exceptions with *depends*.

```
# iocage set priority=50 wdb1
# iocage set priority=10 iscsi1
```

Iocage shuts down jails in reverse priority order.

9 Databases in general are a bad idea, but Big Data keeps suppressing the truth.

Viewing Jails

Just because you configure a jail to start at boot doesn't mean that the jail will keep going. View currently running jails with jls(8). You'll see the jail's IP address, hostname, and the path to the root directory. In this case, the root directory also makes it obvious which jails are standard and which are iocage.

```
# jls
   JID  IP Address      Hostname        Path
     1  203.0.113.234   www1            /iocage/jails/www1/root
     2  203.0.113.236   wdb1            /iocage/jails/wdb1/root
     3  203.0.113.235   www2            /iocage/jails/www2/root
     4  203.0.113.237   wdb2            /iocage/jails/wdb2/root
     6  203.0.113.232   logdb.mwl.io    /jail/logdb
     7  203.0.113.231   loghost.mwl.io  /jail/loghost
```

Each jail also has a unique *jail ID*, or JID. Every jail is assigned a unique jail ID at startup, and retains that JID as long as it runs. When you stop, restart, or reboot the jail, the jail gets a new JID. The JID gets attached to jail-related processes and memory, as we'll see throughout this book.

While you can hard-code a jail ID in *jail.conf* with the *jid* parameter, this feature exists mostly to support obsolete management scripts. Jail IDs get tied to system resources, and a jail cannot be restarted until those resources are freed from the shutdown. This takes longer than you would expect, and leads to frustration as well as language that would shame your mother. Let JIDs be disposable. Iocage does not include this option.

If you want to list a particular jail, add the `-j` flag and the jail's name.

```
# jls -j loghost
   JID  IP Address      Hostname        Path
     9  203.0.113.231   loghost.mwl.io  /jail/loghost
```

Listing jails by name becomes very important when you use some of jls(8)'s more advanced views.

Viewing Parameters

Jails are created by jail(8) commands. All of the jail configuration methods, from `jail.conf` to iocage and beyond, are only tools that simplify creating the jail(8) command to start the jail. They're abstraction layers. If you want to verify that the abstraction layer hasn't gone sideways, and that all the options you set are actually applied, you'll need to view the parameters the jail's running with.

A few parameters, such as *securelevel* (Chapter 8), can be adjusted after jail creation. You'll need to be able to view the jail's current settings, as opposed to how what you set the parameter to when you created the jail. We'll specifically mention these parameters when we get to them.

The jls(8) command can query the host's kernel and display a jail's current parameter settings. Parameters that were not passed to the kernel are not shown; they're run-time ephemera, and could have been changed since the jail started anyway. To find pseudo-parameters like *depend* and *exec.prestart*, you must read `jail.conf` or query iocage.

One useful setting is to view the command line used to start the jail. Parameters that can be changed on a live jail (such as securelevel) are shown at their current value rather than the startup value, but it's good enough for most debugging. Use the `-s` flag to see the parameters arranged as jail(8) arguments. The output is quite substantial, so I normally combine this with `-j` to view only the jail I'm interested in.

```
# jls -sj loghost
devfs_ruleset=0 enforce_statfs=2 host=new
ip4=disable ip6=disable jid=9 linux=new
name=loghost osreldate=1300007 osrelease=13.0-CURRENT
path=/jail/loghost nopersist securelevel=-1 sysvmsg=new
sysvsem=new sysvshm=new vnet=inherit allow.nochflags
allow.nomlock allow.nomount allow.mount.nodevfs
allow.mount.nofdescfs allow.mount.nolinprocfs
allow.mount.nonullfs allow.mount.noprocfs
allow.mount.notmpfs allow.mount.nozfs
allow.noquotas allow.noraw_sockets allow.noread_msgbuf
allow.reserved_ports allow.set_hostname
allow.nosocket_af allow.nosysvipc
allow.unprivileged_proc_debug children.max=0
host.domainname=/””}”” host.hostid=0
host.hostname=loghost.mwl.io
host.hostuuid=00000000-0000-0000-0000-000000000000
linux.osname=Linux linux.osrelease=2.6.32
linux.oss_version=198144
```

Now you see why jail(8) has abstraction layers.

A handful of parameters are internal to jail(8). To pull these as well, use the `-n` flag.

```
# jls -nj loghost
devfs_ruleset=0 nodying enforce_statfs=2 host=new ip4=-
disable ip6=disable jid=9 linux=new name=loghost osrel-
date=1300007 osrelease=13.0-CURRENT
…
```

For a nice table with a header listing all parameters, and each jail’s value for that parameter listed beneath it, add the `-h` flag. This is useful when you want to demonstrate to your boss that you need a bigger monitor because your current one can only handle a 600-character-wide terminal. You can add parameter names to view only the parameters of interest, though. Here I check the values of *sysvshm*, *sysvsem*, and *sysvmsg*. I also grab the *name* parameter, so I know which jail the results apply to. To tidy the results, I run them through `column -t`.

```
# jls -h name sysvshm sysvsem sysvmsg | column -t
name        sysvshm  sysvsem  sysvmsg
ioc-wdb2    0        0        0
ioc-squid1  0        0        0
loghost     1        1        1
ioc-www1    2        2        2
```

Adding `-j` lets you check a parameter for a single jail.

Other Output Formats

To get a nice multi-line listing for each jail and select parameters, use the `-v` flag.

```
# jls -vj loghost
   JID  Hostname                       Path
        Name                           State
        CPUSetID
        IP Address(es)
     9  loghost.mwl.io                 /jail/loghost
        loghost                        ACTIVE
        4
        203.0.113.231
```

This is also the only simple way to see which processor the jail is assigned to—a fact that's most often irrelevant, but it's nice to know a jail can't fork-bomb your host.

Jails support libXO, letting you print jail information in HTML, XML, JSON, or even plain text. This is ridiculously helpful if you're looking to run complicated or extensive reports and have tools to parse these formats. Add the `--libxo` argument and the format you want.

```
# jls -n --libxo xml
```

Us old-school sysadmins will continue using `grep` and `cut` in shell scripts, painfully parsing the results, and adding special edge cases to our audit scripts forever, all in the name of technical superiority.

Viewing Iocage Jails

If you're only interested in iocage jails, the `iocage list` command displays which jails are running and which aren't. We've done this a few times. To get more information on each jail, print a long list with `iocage list -l`. While the width of the output depends on the information displayed, you'll need a terminal at least a hundred characters wide to show it properly.

The `iocage list` command has a whole bunch of flags and arguments to view iocage-related information. It's a general front end to examining iocage jails. We'll examine those in the relevant sections.

Process Management

Use standard systems administration tools to manage jailed processes. Neither iocage(8) nor jail(8) include any special process-management tools.

Jails do not have unique ranges or separate pools of process IDs; they all share the same range of process IDs as the host. If you're **`root`** on the host and list all running processes, you'll see the processes running in all of the jails. Users in a jail can see only processes tied to that jail. Processes running inside a jail are flagged with a `J` in ps(1).

```
# ps -ax
…
 1951  -  SsJ     0:00.10 /usr/sbin/syslogd -c -ss
 2005  -  IsJ     0:00.29 /usr/sbin/cron -J 15 -s
 2090  -  SsJ     0:00.10 /usr/sbin/syslogd -c -ss
 2144  -  IsJ     0:00.30 /usr/sbin/cron -J 15 -s
 2229  -  SsJ     0:00.11 /usr/sbin/syslogd -c -ss
 2283  -  IsJ     0:00.32 /usr/sbin/cron -J 15 -s
…
```

All of these processes have the J flag, showing they're tied to a jail.

The `-J` flag to ps(1) lets you view the processes that belong to a specific jail. Give `-J` one argument, the jail name or JID. Here I see what's running inside **`logdb`**.

```
# ps -ax -J logdb
 PID TT  STAT    TIME COMMAND
3652  -  SsJ  0:00.10 /usr/sbin/syslogd -s
3705  -  SsJ  0:00.45 sendmail: accepting connections
  (sendmail)
3708  -  IsJ  0:00.02 sendmail: Queue runner@00:30:00
  for /var/spool/clientmqueue (sen
3712  -  IsJ  0:00.32 /usr/sbin/cron -s
```

That's a pretty minimal system. Don't worry, I'll be adding to it soon.

To exclude all jailed processes and see only processes belonging to the host, tell `-J` to use jail ID 0. This lets you more easily debug the host itself.

```
# ps -ax -J 0
  PID TT  STAT       TIME COMMAND
    0  -  DLs     2:38.25 [kernel]
    1  -  ILs     0:00.08 /sbin/init --
    2  -  DL      0:00.00 [crypto]
…
```

Most process-related commands, such as pkill(1), pgrep(1), and killall(1) accept a `-j` argument to let you specify a jail. Each jail runs its own cron process. Here I find out which belongs to the jail **logdb**.

```
# pgrep -j logdb cron
3712
```

Most of us don't troubleshoot this way, though. We'll log into a troubled host, see that one of the MySQL processes is running amok and eating all the CPU, and need to backtrack which jail it belongs to. The `-o` option to ps(1) reveals many handy facts about processes through keyword arguments. (If you're trying to extract information about a process, definitely look at the list of eighty-plus keywords in ps(1).) Adding `-o jail` prints the name of the jail.

```
# ps -ax -O jail | grep 8219
8219 wdb1 - IsJ 0:00.00 /usr/local/libexec/mysqld
```

Process 8219 is running in the jail **`wdb1`**.

For a more dynamic view of a particular jail's processes, top(1) also accepts the `-J` flag. Add the jail name as an argument, and you'll see the standard top view of what's running within the jail.

```
$ top -J ioc-www1
```

Running top with the `-j` flag gives you the host's top listing, including a JID column. The `j` is also an interactive top command, letting you toggle the JID display on and off.

You could use jexec(8) or `iocage exec` to manage processes inside jails, yes. There's no practical difference. Choose what works for you. (My fingers are not yet accustomed to these newfangled "pgrep" and "pkill" commands, but you youngsters are welcome to them.)

Jailing Commands

While you can log into a jail to run commands, you can run commands inside an active jail from the host using jexec(8). You might want to SSH into the jail for complicated work, such as server configuration or troubleshooting, but for one-off commands—especially on multiple jails—jexec is almost always easiest. You'll need three arguments: `-l`, the name of jail to run the command in, and the command to run. (We'll see why you need `-l` in the next section.) If you don't enter a command, it runs a shell. You must be **`root`** to run jexec.

```
# jexec -l loghost uname -a
FreeBSD loghost.mwl.io 13.0-CURRENT FreeBSD
13.0-CURRENT #10 r338496: Fri Sep  6 12:29:00 EDT 2019
root@storm:/usr/obj/usr/src/amd64.amd64/sys/GENERIC
amd64
```

I've run this command from my host, but it's run inside the jail **`loghost`**.

Suppose I'm first setting up the jail and I want to set a root password. I give the name of the jail, the command I want to run, and any arguments to that command.

```
# jexec -l loghost passwd root
Changing local password for root
New Password:
```

The jexec(8) command only works on running jails. To run a command inside a jail that isn't working, you'll need to jail an individual command as discussed in Chapter 2. Here I start a shell in a non-active jail.

```
# jail /jail/loghost/ loghost 203.0.113.231 /bin/sh
```

Technically speaking this starts the jail, but only in a limited way. The only process running in the jail is your shell, plus anything you start.

You can use jexec on iocage jails, but you'll need to add the *ioc-* in front of the jail name—remember, iocage jails get that tag in front of their name. Rather than having to remember that, use `iocage exec`. Like jexec it takes two arguments, the host and a command. If you don't provide a command, it runs a shell. Here I add a user to my iocage jail **`loghost`**.

```
# iocage exec www1 adduser
```

If you want a shell inside your jail, you're better off using the `iocage console` command. This simulates a full root login of your jail. Give it the jail name as an argument.

```
# iocage console www1
```

Why use `iocage console` and not just jexec a shell? Because of the jail environment.

Jailed Process Environment

Processes inherit their environment from their parent. Processes started by the jail's startup process are children of the jail's */etc/rc* (or whatever command you used to start the jail). That environment's pretty well sanitized. While your `jexec` and `iocage exec` commands are flagged as belonging to a jail, they're children of your existing shell. This seems like a minor detail, but ask a jailed command about its environment. We'll skip the `-l` this time.

```
# jexec loghost env
HOME=/home/mwlucas
TERM=xterm
LC_COLLATE=C
…
SSH_CLIENT=203.0.113.70 59076 22
SSH_CONNECTION=203.0.113.70 59076 203.0.113.50 22
SSH_TTY=/dev/pts/2
SSH_AUTH_SOCK=/tmp/ssh-ZfvZOatcsu/agent.60492
…
```

Hang on here—a jailed process should *not* have my terminal session's SSH environment variables! It has them because this process, while created in the jail's process context, inherits the parent's environment. Never mind that the directory where my SSH agent's authentication socket lurks doesn't even exist in the jail, they're part of the shell environment. Similarly, it inherits a bunch of other crud from my login files.

Adding `-l` tells jexec to eradicate every setting except $HOME, $SHELL, $TERM, and $USER. It also preserves anything from user's login class.

Similarly, the initial process that boots the jail is a child of another process—possibly */etc/rc* from the host's initial boot, or maybe your shell if you start the jail by hand. The *exec.clean* parameter tells jail(8) to purge the environment before starting the jail. It defaults to 1, cleaning the environment. If you want to have a jail inherit the environment of the process that starts it, set *exec.noclean* to learn exactly how you're wrong.

Iocage's `exec` command defaults to running everything in a clean environment. The iocage parameter *exec_clean* controls if commands should be run in a clean environment. The default, 1, purges the environment. When set to 0, `iocage exec` inherits the user's environment.

Console Access

You could run `jexec -l /bin/sh` or `iocage exec /bin/sh` to get a shell, but that won't really simulate the effect of being logged in as **`root`** on the jail. A real login experience should source dotfiles and process login classes.

Get a true login on a jail from the host through jexec(8) by running login(1). Give login the `-f` argument and the account you want to log into. Here I get a **`root`** shell on the jail **`loghost`**.

```
# jexec -l loghost login -f root
FreeBSD 13.0-CURRENT (GENERIC) #10 r338496: Thu Sep  6
12:29:00 EDT 2019

Welcome to FreeBSD!

Release Notes, Errata: https://www.FreeBSD.org/releases/
…
root@loghost:~ #
```

Yep, you get the full */etc/motd* and everything.

To get a **`root`** login using iocage, use the `console` command.

```
# iocage console www1
…
root@www1:~ #
```

Iocage does not have a feature to log into a jail as a non-root user but you can fall back on `jexec -l` ***`jail`*** `login -f` ***`user`***, login as **`root`** and use su(1), or use an add-on package like `jailme`.

Console Logging

We've all had FreeBSD fail to start, fail to stop, or inexplicably hang for an excessive time somewhere in the `reboot` process. Usually you need to check the system console to see what misconfiguration is causing the delay.[10] Jails can log their console output to a file.

On a standard jail, set the parameter *exec.consolelog* to a file. Here I set a default console log at the top of *jail.conf*.

```
exec.consolelog="/var/tmp/$name";
```

This creates files like */var/tmp/loghost* and */var/tmp/logdb*.

You might think that console logs should be carefully protected, and there's certainly an argument to be made for that. Your jail host should be carefully protected as a whole, however, and regular users shouldn't have any access to it. Illicit access to read jail console logs is the least of your problems. If you're concerned, though, put those logs in a directory only readable by **root**.

Iocage automatically logs all consoles to the *log* directory on iocage's dataset, so my console logs go in */iocage/log/*. Each console log is named after the jail, with *-console.log* appended. The log for iocage jail **www1** is */iocage/log/www1-console.log*.

Now that you can create, start, stop, reboot, fold, and spindle jails, let's perform some common sysadmin tasks with them.

10 I want to say "it's always DNS," but that's obviously not true. It's *overwhelmingly* DNS. When it's not, it's some balky service. `service -R` is your friend.

Chapter 4: Jail Filesystems

If you haven't noticed yet, filesystem management is a huge part of jails. As we discuss jails we'll dive into all sorts of filesystem tricks and details. You need to understand a few dusty corners about filesystems before we go too far, though. Many of these are not jail-specific, but they're not commonly used.

This chapter is mostly about managing filesystems from the host. Chapter 8 provides insight into managing filesystems within a jail.

mount(8) And umount(8)

You've used mount(8) and umount(8). You've mounted disks, and disk images. You've rebooted systems after accidentally mounting something incorrectly and destroying the system's stability. If you've been a sysadmin long enough, you've accidentally unmounted the */bin* disk pack as payroll was running. What more is there to learn about mount(8)? A bunch. Sometime when you have a few minutes to spare, reread mount(8). Play with some of the options. Revisiting commands you think you know is a great way to improve your skills. We'll cover the options I find useful for jail management here.

Running mount(8) on its own shows all mounted filesystems, of all types. But sometimes you want to view only a certain type of filesystem. To show only one filesystem, use the `-t` flag and the filesystem name. Here I show every device filesystem mounted on this host.

```
# mount -t devfs
devfs on /dev (devfs, local, multilabel)
devfs on /iocage/jails/wdb2/root/dev (devfs, local,
  multilabel)
devfs on /iocage/jails/squid1/root/dev (devfs, local,
multilabel)
devfs on /jail/loghost/dev (devfs, local, multilabel)
devfs on /iocage/jails/www1/root/dev (devfs, local,
  multilabel)
```

If you're searching for strangeness, you might find piping this to `column -t` useful.

While `mount` can read */etc/fstab*, it can also write files in that format. Add the `-p` flag to display the results in this style.

```
# mount -pt devfs
devfs  /dev                           devfs  rw  0  0
devfs  /iocage/jails/wdb2/root/dev    devfs  rw  0  0
devfs  /iocage/jails/squid1/root/dev  devfs  rw  0  0
devfs  /jail/loghost/dev              devfs  rw  0  0
devfs  /iocage/jails/www1/root/dev    devfs  rw  0  0
```

I find this useful when I've manually tweaked filesystems and want to produce *fstab* entries. You might need to tweak the last two fields, for dump(8) level and mounting order, but it's at least a really good start.

Finally, `mount` and `umount` can read *fstab* files other than */etc/fstab* with the `-F` flag. You can create a per-jail *fstab* and only mount that when starting the jail. The jail system hides this behind the *mount.fstab* parameter, but you need to know how that magic happens. If you accidentally blow up a jail and its custom filesystems are still mounted, unmount everything in its *fstab*.

```
# umount -aF /jail/fstab/loghost.fstab
```

This will save you huge amounts of annoyance once you start destructive testing on base jails (discussed in Chapter 6).

Filesystem Visibility

Log into one of your jails and run mount(8). What you get depends on the underlying filesystem. A jailed user can see where the files for their jail are mounted on the host, but can't see other mount points on the jail. Jails running on ZFS can view the dataset their jail is rooted in. Here, the jail **`www1`** is locked into the directory *`/iocage/jails/www1/root`*.

```
# mount
iocage/iocage/jails/www1/root on / (zfs, local, nfsv4acls)
```

Jails on UFS see the device their root filesystem is mounted on. The jail **`loghost`** is locked into *`/jail/loghost`*, but that's a directory on the UFS filesystem *`/jail`*.

```
# mount
/dev/gpt/ufs1 on / (ufs, local)
```

The *`/jail`* filesystem is on device *`/dev/gpt/ufs1`*, so that's what the jail sees.

Jails on both UFS and ZFS have more filesystems than a single root directory, though. Every jail has *`/dev`*, unless you deliberately turn it off. Iocage jails mount fdescfs(5) by default. Each should show up in a list of mounted filesystems, no?

No. Jails deliberately restrict what filesystems jailed processes can see. By default, jailed processes can only see their root directory and the device that it's mounted on. Use the parameter *enforce_statfs* to adjust this behavior. The default, 2, is the highest setting, and most tightly restricts filesystem visibility. The other mount points are still present, and jailed processes can access them as permissions permit. The jail can't see them as separate mount points, however.

Loosen the restrictions by setting *enforce_statfs* to 1, and a jail can perceive filesystems mounted within the jail.

```
# mount
iocage/iocage/jails/www1/root on / (zfs, local, nfsv4acls)
devfs on /dev (devfs, local, multilabel)
fdescfs on /dev/fd (fdescfs)
```

If you want jailed processes to be able to mount new filesystems within the jail, the process must be able to perceive the mount points. Any sort of allowed mount requires setting *enforce_statfs* to 1 or lower. The jail must have additional privileges defining which types of filesystems can be mounted, as discussed elsewhere this chapter, but the kernel doesn't unilaterally block mount-related syscalls.

Setting *enforce_statfs* to 0 removes all limits on what the jail can see. The jail can see all of the mount points and filesystems on the host, and if granted permission can mount them. If the host has a separate */var* partition, the jail can't read the contents but knows that the partition exists. If you think a jail needs this, you're probably wrong.

Most filesystem requirements can be solved by having the host mount the appropriate filesystem when starting the jail.

Mounting Filesystems at Startup

If a jail needs access to a device or filesystem beyond its root directory, you can mount that filesystem automatically when creating the jail. This can be space shared between jails, a read-only data store, special filesystems such as */proc* or fdescfs(5) or anything else.

Remember that sharing read-write disk space doesn't control how different jails write to that space. If multiple jails have MariaDB[11] installs and you configure them to share the same data directory, bad things will happen to your data—exactly as if different database servers accessed that data directory via NFS or SMB or any other file sharing protocol. Different jails cannot see others' processes. Be very careful in what you choose to share.

11 MariaDB is MySQL without the Oracle taint.

Having said that, sharing directories between multiple jails can make sense. How many copies of `/usr/ports` or `/usr/src` do you need? Sharing such infrastructure directories between hosts might make perfect sense. For maximum safety, mount such directories read-only.

The jails system has a few ways to mount additional filesystems in a jail at startup.

Mounting Special Filesystems

Some special-purpose filesystems do not need configuring. They're either present, or not. Examples include procfs(5), fdescfs(5), and devfs(5). Jails have a parameter for each.

Almost every jail needs `/dev`. The *mount.devfs* parameter mounts a standard device filesystem for the jail. You can customize which device nodes a jail can access, as discussed in "Customizing /dev" later this chapter.

The file descriptor filesystem is only needed if you use exotic software like the Bash shell. Enable it with the *mount.fdescfs* parameter (*mount_fdescfs* in iocage).

FreeBSD developers loathe the process filesystem, `/proc`. If your software demands `/proc`, better you enable *mount.procfs* (*mount_procfs* in iocage) and run it in a jail than expose your host to it. Also, get better software.

Mounting other filesystems inside a jail is slightly more complicated.

Standard Jails and Extra Mounts

If a jail needs only one special mount, we might use the *mount* parameter. It takes one argument, a formatted `fstab` line. The jail system mounts this filesystem at jail creation, and unmounts it when shutting down the jail.

Suppose my standard jail **`loghost`** needs really speedy scratch space, and I decide to assign it a tmpfs(5) memory filesystem for *`/tmp`*. The jail's root filesystem is *`/jail/loghost`*, so I need this tmpfs mounted at *`/jail/loghost/tmp`*. I need jail(8) to mount this filesystem before starting the jail. I can accomplish this with the following *`jail.conf`* snippet.

```
loghost {
  ip4.addr="203.0.113.231";
  mount="tmpfs  /jail/loghost/tmp  tmpfs rw,size=1g,-
  mode=1777 0 0";
}
```

That's almost straight from fstab(5); the only thing I've changed is the directory to mount the memory filesystem on.

The problem with one-off solutions is that they're never one-offs. Once you realize you can mount extra filesystems in a jail, you'll find more and more uses for them. Plus, these *`fstab`* entries can really clutter up your nice tidy *`jail.conf`*. The jail system can mount an additional *`fstab`* when creating the jail, and automatically unmount these filesystems when shutting down the jail. Give the full path to this *`fstab`* file in the *mount.fstab* parameter.

```
mount.fstab="/jail/fstab/loghost.fstab";
```

I normally create a directory for *`fstab`* files, and name the individual files after the jail. This lets you do daft things like make *mount.fstab* a default, useful when you have many standard base jails (Chapter 6).

With each jail's additional filesystems contained in a separate file, you can easily mount and unmount that jail's filesystems without activating the jail itself.

```
# mount -aF loghost.fstab
# umount -aF loghost.fstab
```

While mounting and remounting additional filesystems are integral to base jails, they add much-needed flexibility to more routine jails.

Iocage and Extra Mounts

Iocage provides several ways to edit *fstab* files, all through the `iocage fstab` command. Newly created iocage jails (that aren't clones or base jails) have no *fstab* entries, so we'll start by adding some.

Use the `-a` option to add a line to the *fstab*. The command has the format:

```
# iocage fstab -a jail /src/path /jail/directory \
  fstype mount-options 0 0
```

This command differs from creating an *fstab* for a standard jail. You don't give the full path to the destination mount point. Iocage moves jails between datasets as needed, so you don't specify the full path to the current mount location: instead, you give the mount point inside the jail where you want the new mount to wind up. Here I give the jail **www4** a tmpfs */tmp*.

```
# iocage fstab -a www4 tmpfs /tmp tmpfs rw,mode=1777 0 0
```

If you want the source directory nullfs-mounted in the same location in the jail, such as mounting the host's */usr/src* on the jail's */usr/src*, you can shorten this command to:

```
# iocage fstab -a jailname /src/path
```

In my environment I put all custom scripts in */usr/local/scripts*. I want to make this directory on the host available inside all the jails. I won't need all those scripts on all the hosts, so I can use nullfs(5) to make this directory available on multiple mount points.

```
# iocage fstab -a www1 /usr/local/scripts
Destination: /iocage/jails/www1/root/usr/local/scripts
  does not exist!
```

Iocage refuses to create the directory because the jail **www1** doesn't have a */usr/local/scripts* directory to mount on. The command-line tool provides some sanity checking that you don't get from editing an *fstab* by hand. Once I create the directory, the command works and iocage immediately performs the mount.

After a while, you'll forget what you've added to each jail's *fstab*. To view the current *fstab*, use `iocage fstab -l`. It prints out a formatted table by default, but you can use `-h` to strip all that out and get the information.

```
# iocage fstab -lh www1
0  /usr/local/scripts /iocage/jails/www1/root/usr/local/
scripts nullfs ro 0 0
1  /usr/src /iocage/jails/www1/root/usr/src nullfs ro 0 0
```

I guess I added */usr/src* to this jail sometime. Each entry has an index number, shown at the far left. My */usr/local/scripts* mount is index 0, while */usr/src* is index 1.

To remove an *fstab* entry, get the entry's index number and use the `iocage fstab -r` command. Give the jail name and the *fstab* table entry's index number as arguments. This jail no longer needs */usr/src*, so I'm going to remove it. The */usr/src* mount is index 1, so I run:

```
# iocage fstab -r www1 1
Successfully removed mount from www1's fstab
```

Note that removing an *fstab* entry changes index numbers. Everything below the deleted entry moves up one. Don't willy-nilly remove indexes 0, 1, and 2 one after the other, because after each removal they'll point to different entries.

You can replace entries with `iocage fstab -R`. Eventually I realized that I didn't want to make all of my host's scripts available on jails. The script to purge templates and perform other jail management tasks shouldn't be on the jails themselves; they leak information about my sloppy sysadmin habits to potential intruders. I needed to create an expurgated scripts directory on the host, */usr/local/jail-scripts*, and mount that on the jail. The */usr/local/scripts* mount is at index 0, so I ran:

```
# iocage fstab -R 0 www1 www1 /usr/local/jail-scripts \
    /usr/local/scripts nullfs ro 0 0
Index 0 replaced.
```

If all this programmatic stuff feels like too much work, and you'd rather edit a jail's *fstab* directly, use `iocage fstab -e` and the jail name. This brings up the jail's *fstab* in your default editor.

```
# iocage fstab -e www1
```

When you directly edit a jail's *fstab*, iocage performs no sanity checking and mounts no filesystems for you. If you create a bogus *fstab* entry, iocage won't tell you until you try to restart the jail. If you're worried about uptime, let iocage edit the filesystem table for you.

Device Filesystems and Jails

Every jail gets a stripped-down device filesystem. Typical jails need access to the devices that let users log in, transfer data to and from programs, generate numbers, and discard data. They don't need disk devices, the physical console, sound cards, and so on.

Except when they do.

Eventually, you'll need to grant a jail access to a different device. Perhaps you're running your backup software in a jail, or ripping CDs from the backup vault, or using a webcam, or who knows what? That means exposing a device in a jail, which means configuring a special device filesystem for that jail.

Configuring devfs(5)

In-depth configuration of a device filesystem is really beyond the scope of this book, but I'll give you a very brief example for a simple case because I'm ~~such a nice guy~~ not a complete monster. Configuring a device filesystem means creating rulesets that hide and reveal device nodes. Configure a devfs rulesets in */etc/devfs.rules*. Perhaps the most common mistake in working with devfs(5) is putting rules in */etc/devfs.conf*. Don't do that.

Each rule is assigned a number. FreeBSD includes a few rulesets in */etc/defaults/devfs.rules*. These are intended as building blocks for further rulesets, mainly jails. Each ruleset is identified with a unique name and number. Ruleset 4 is specifically for jails.

```
[devfsrules_jail=4]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path fuse unhide
add path zfs unhide
```

The initial brackets set off the ruleset name (devfsrules_jail) and the ruleset number (4).

The following lines include other rulesets, by name. The standard rulesets have names that reflect their purpose: devfsrules_hide_all hides all the devices, devfsrules_unhide_basic unhides essential devices like */dev/random* and */dev/stdin*, and devfsrules_unhide_login unhides the virtual terminal devices needed for logging in.

Building on this basic model, a ruleset for jails needs two more special device nodes. The fourth rule reveals */dev/fuse*, and the fifth */dev/zfs*. This permits use of mount_fusefs(8) and zfs(8) from within a jail. Exposing these generically within jails means that you can control access to them through jail parameters without having to assign a new set of devfs rules for the jail. If you're running jails on UFS, and you don't allow use of FUSE, you could strip out these rules.

Creating a new ruleset is as simple as picking a rule number, including a ruleset to build on, and adding the new rules you need. Suppose you want to let a particular jail access the CD reader, */dev/cd0*. (You cannot read the CD filesystem from within a jail, but presumably you have a ZFS pool or some other filesystem on it.) I'm going to copy the jail ruleset devfsrules_jail and add a new unhide statement.

```
# allow CD in jail
[devfsrules_jail_cd=1000]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path cd0 unhide
```

This ruleset has the name of devfsrules_jail_cd, and is ruleset number 1000. This gives me lots of room for FreeBSD to add more rulesets. It includes most of the devfsrules_jail ruleset and adds a rule to unhide the cd0 device.

Run `service devfs restart` to make devd reread its rules, and verify that the new ruleset shows up in `devfs rule showsets`. Once it's available in devd, assign the ruleset to a jail.

Standard Jails and Devfs

The parameter *devfs_ruleset* tells jail(8) which ruleset to assign to a jail. The default is ruleset 4, also called devfsrules_jail. Here I want to assign a specific jail my custom devfsrules_jail_cd ruleset, also known as ruleset 1000.

```
loghost {
  ip4.addr="203.0.113.231";
  devfs_ruleset=1000;
}
```

Restart the jail, and your device will be available.

View all devfs rulesets assigned to jails using jls(8).

```
# jls -h name devfs_ruleset |column -t
name      devfs_ruleset
ioc-www2  9
loghost   1000
ioc-www1  10
logdb     0
```

Or, you could log into the jail and verify that */dev/cd0* exists while */dev/acpi* doesn't… but where's the fun in that?

Iocage and Devfs

Control an iocage jail's device filesystem rules with the *devfs_ruleset* parameter. Sort of.

Iocage has a hard-coded initial devfs ruleset. As I write this, the default iocage ruleset is the same as the standard devfs.rules ruleset 4 that the base system uses for jails. If you enable DHCP on a jail, it adds */dev/bpf*.[12] When you start the jail, iocage builds a brand-new set of devfs rules and assigns it to the jail. If *devfs_ruleset* is set to 4, the common default for jails, iocage creates this new ruleset every time it starts the jail. When you stop the jail, it removes that jail's ruleset.

Why does iocage do this? One little known feature of devfs is that you can edit its rules on the command line, with the devfs(8) command. I don't know anyone who actually does this, but I'm assured that the people who do have perfectly good reasons. Each iocage jail gets assigned a unique ruleset at startup, so you can manipulate that jail without affecting any other jails. Whenever you start a jail, iocage tells you the number of the devfs ruleset it created for that jail. Query jls(8) for which devfs rulesets are applied to which jails.

12 "Keeping servers running is already hard enough, and you want to add a dependency on DHCP?" (throws up hands) "Fine. It's your life."

When you assign a specific devd ruleset to a jail, iocage does not copy this ruleset to a new ruleset. It believes that if you assigned ruleset 1000 to a jail, you dang well *meant* it. If you assign a specific ruleset to several jails, and then manipulate that ruleset live, the changes apply to all jails that use that ruleset.

Changes to *devfs_ruleset* do not take effect until you restart the jail.

So to give the jail **wdb2** access to the CD drive via ruleset 1000, which I added to */etc/devfs.rules*, I would run:

```
# iocage set devfs_ruleset=1000 wdb2
Property: devfs_ruleset has been updated to 1000
# iocage restart wdb2
```

This jail can now see */dev/cd0*.

Iocage Disk Usage

Iocage creates at least two datasets for each jail. As we get into more complicated iocage uses and leverage ZFS features, it'll quickly become obvious that figuring out how much disk space a jail uses isn't as easy as running df(1) inside your jail. As your ZFS clones diverge, you'll see pool utilization creep upward and need to figure out where all your disk space is going.

Iocage provides the `df` command to sort out exactly how much disk space each jail uses.

```
# iocage df
+-------------+-------+------+------+-------+------+
|    NAME     |  CRT  | RES  | QTA  |  USE  | AVA  |
+=============+=======+======+======+=======+======+
| dns3        | 1.03x | none | none | 604K  | 888G |
+-------------+-------+------+------+-------+------+
| dns4        | 1.03x | none | none | 25.2M | 888G |
+-------------+-------+------+------+-------+------+
| dnstemplate | 1.55x | none | none | 567M  | 888G |
+-------------+-------+------+------+-------+------+
| squid1      | 2.01x | none | none | 209M  | 888G |
+-------------+-------+------+------+-------+------+
…
```

The first column gives the name or UUID of the jail.

The CRT column shows the jail's ZFS compression ratio.

The RES column shows any ZFS reservation for the jail, while QTA displays the ZFS quota.

USE shows how much space the jail actually uses.

Use the `-s` flag to sort by a particular column, from smallest to largest. Here I sort by how well the jails compress.

```
# iocage df -s CRT
```

If you want to find, say, the five largest jails, sort by USE and pipe the output to tail(1). The `-h` flag strips out all the fancy table formatting, leaving you the raw data.

```
# iocage df -hs USE |tail -5
```

Or, you can trawl through zfs(8) commands and figure out disk space usage. Whatever floats your fish.

Chapter 5: Packages and Upgrades

Empty jails have uses—just not many uses. You almost certainly want to install FreeBSD packages on your jail. Similarly, unpatched FreeBSD installs have uses, but those uses are all for intruders. You must install security updates on your jails! FreeBSD's packaging and upgrade tools support jails, and iocage has specific support for packages.

Packages

You'll see there's a bunch of ways to manage packages in jails. Pick a method and stick to it. Don't bounce between iocage packages and host-based pkg(8) and jail-based pkg(8). The packaging tools might be subtly different, especially if the jail runs an older FreeBSD release than the host.

Managing jail packages from the host with pkg(8) is not the same as running `jexec -l jailname pkg`. The former uses the host's packaging tools, while the latter runs the jail's packaging tools.

A jail owner might want to install his own packages using the standard pkg(8) tools inside the jail. That's perfectly reasonable—a jail has a complete FreeBSD userland, after all. Carefully consider if you want a particular user installing packages or not. Remember, if the user screws up you'll get a phone call from the user. If the user doesn't apply security updates to his packages and an intruder penetrates the jail, you'll get a call from law enforcement. For users who shouldn't even have shell access let alone their own jail, I replace the jail's */usr/sbin/pkg* with a shell script that prints instructions to ask the helpdesk to install all packages and then emails me a warning that my special friend is meddling with things he doesn't understand. If the user has a modicum of competence, though, have them manage their own packages with pkg(8) like any other FreeBSD install.

pkg(8)

Jails are a first-class citizen of the pkg(8) ecosystem. The host's package tools can identify running jails by name and manage their installed packages. I generally recommend managing jail packages from the host rather than inside the jail. While host-based package management uses the host's packaging tools, it stores the jail's package database inside the jail and uses the jail's *pkg.conf*, certificates, and so on. This lets jails use different repositories.

Managing a jail's packages with the host's pkg(8) requires the `-j` flag and the jail name. This flag goes right after `pkg`, not the sub-commands. Here I'm telling `pkg` to run on jail **`logdb`**, and install the package *sudo*.

```
# pkg -j logdb install sudo
```

The first time you run pkg(8) on a host it upgrades itself, installs a database of packages available in the repository, and does whatever you asked for. Running the host's pkg(8) on a jail avoids installing a current pkg(8) in the jail, but still installs the current package database in the jail. The pkg(8) program will detect version differences between the host and the jail, and will issue warnings. You can safely ignore these warnings.

Other than `-j` and the jail name, managing a jail's packages from the host is *exactly* like managing packages on any other host.

Iocage Packages

Iocage integrates package management into its command line. Use the `iocage pkg` command. Give it the name of the jail and the pkg(8) command you want to run. Here I install sudo on the jail **`www1`**.

```
# iocage pkg www1 install sudo
```

The `iocage pkg` command checks the repository's package database, installs an in-jail package database if needed, and installs sudo, exactly as if I ran raw pkg(8) commands on the host. Only the syntax differs slightly.

Why would you use `iocage pkg` over `pkg -j`? Your preference. Iocage aims to be a complete jail management solution, and provides `iocage pkg` as a convenience for iocage users, that's all.

Patching Jails

FreeBSD's binary upgrade tool, freebsd-update(8), supports jails. A jail is merely a FreeBSD install mounted at a particular directory, after all. While you'll need to use freebsd-update for standard jails, iocage provides a simpler front-end and leverages ZFS snapshots.

Patching with freebsd-update

You cannot reliably use freebsd-update(8) within a jail. The key word here is *reliably*. I've gotten away with in-jail upgrades, but it's not supported and if you make it a regular practice it'll eventually sink its fangs into your arm. Perform all jail upgrades on the host.

To update or upgrade a jail, freebsd-update(8) must know the version of FreeBSD running in the jail and the jail's base directory. Always back up your jails before upgrading. If the upgrade fails, a backup tarball or ZFS snapshot will save you a world of pain.

Patching or upgrading a jail has different requirements than a host. You'll need a different update configuration for your jails.

```
# grep -v ^# /etc/freebsd-update.conf | \
    grep -v ^$ > /etc/jail-update.conf
```

Open the file and start trimming. You need a server and the KeyPrint for that server, so leave those alone. Start with the Components line. Jails have no kernel, so don't try to update it. Most (not all) jails don't include the system source code. The only component you must always update is the world.

```
Components world
```

You can use the host's intrusion detection functions inside a jail if you want.

Some commands should never be run inside my jails, like pkg(8) and freebsd-update(8). I replace them with shell scripts reminding me to not use them. I also don't care about updates to files like */etc/motd*. Block freebsd-update from replacing such files with the IgnorePaths statement.

```
IgnorePaths /usr/sbin/pkg /usr/sbin/freebsd-update /etc/motd
```

You'll need MergeChanges in */etc*, but */boot/device.hints* is irrelevant in jails. You might modify UpdateIfUnmodified to suit your configuration.

The freebsd-update(8) command checks the kernel to identify which FreeBSD release it's running on, much as `uname -a` does. Jails run on the host kernel, but the jail's userland is often entirely different. FreeBSD 12 jails running on a FreeBSD 13 system would default to applying FreeBSD 13 patches. That would be bad. Use freebsd-version(1) instead to get the jail's current patchlevel.

Here, my standard jail **ldap1** currently runs FreeBSD 11.1-RELEASE. I need to install the current security patches on it. I'm running this from the host.

```
# freebsd-update -f /etc/jail-update.conf \
  -b /jail/ldap1 --currently-running \
  `jexec -l ldap1 freebsd-version` fetch install
```

The jail now has updated binaries, as freebsd-version(1) will confirm.

```
# jexec -l ldap1 freebsd-version
11.1-RELEASE-p14
```

Any currently running programs are still the old versions, though. It's best to restart the whole jail.

```
# service jail restart ldap1
```

Your jail is now upgraded.

It's an annoyingly long command line, but ridiculously amenable to scripting if your jails are in a directory named after the jail. Here's my script `jailpatch.sh`. It takes one argument, the name of the jail to be patched.

```
#!/bin/sh

freebsd-update -f /etc/jail-update.conf -b /jail/$1 \
      --currently-running \
      `jexec -l $1 freebsd-version` fetch install
service jail restart $1
```

Run this through a loop to patch and restart all your jails.

The brave can add a `pkg -j $1 upgrade -y` in the middle of the script.

Backups and Patching with Iocage

Iocage's snapshot features simplify pre-upgrade backups. The `iocage snapshot` command takes a snapshot. Give it the jail name, and then use `-n` to provide a snapshot name. Here I take a snapshot of jail **www1** and call it "install." If you don't provide a snapshot name, iocage creates one based on the current date and time.

```
# iocage snapshot -n install www1
```

View all snapshots with `iocage snaplist`. Add the jail name. As with most iocage commands you can use the `-h` option to get rid of the table formatting and/or sort on a column with `-s`.

```
# iocage snaplist www1
+--------------+-----------------------+-------+------+
|     NAME     |        CREATED        | RSIZE | USED |
+==============+=======================+=======+======+
| install      | Thu Sep 20 17:34 2018 | 92K   | 0    |
+--------------+-----------------------+-------+------+
| install/root | Thu Sep 20 17:36 2018 | 680M  | 0    |
+--------------+-----------------------+-------+------+
```

Always take a snapshot before updating a jail.

Iocage simplifies jail patching with the `iocage update` command. Give one argument, the name of the jail to patch.

```
# iocage update www1

* Updating www1 to the latest patch level...
```

You'll see the usual freebsd-update messages.

If the upgrade goes terribly wrong, you can revert an upgrade with the `iocage rollback` command. Give the snapshot name with `-n`, and then the jail name.

```
# iocage rollback -n install www1

This will destroy ALL data created including ALL snap-
shots taken after the snapshot install
Are you sure? [y/N]: y
Rolled back to: iocage/iocage/jails/www1
```

You can now try the patch again when you have a higher pain tolerance.

Delete snapshots with `iocage snapremove`. Give the snapshot name with `-n`, then the jail name. After a few successful upgrades the install-time snapshot won't ever be used, so you can get rid of it.

```
# iocage snapremove -n install www1
```

This particular jail runs FreeBSD 9.3. I don't really want to patch it. I want to upgrade it to a supported version of FreeBSD. Let's discuss upgrades.

Upgrading Jails

Upgrading a host across major (12.0 to 13.0) or minor (12.0 to 12.1) FreeBSD versions is a whole bunch easier than it used to be, thanks to freebsd-update(8). Even so, if you have an automation system such as Ansible or Puppet that make deployment trivial, I recommend installing new jails at the new FreeBSD version rather than upgrading. Automation makes server installation faster and safer than upgrading. This is especially true with iocage, as default iocage jails are ZFS clones and upgrading from 12.2 to 13.0 vastly increases disk space usage.

No matter how you upgrade your jails, be sure to take a backup before starting.

Upgrades with freebsd-update

Everything about patching a jail applies to upgrading a jail. You need that same jail-specific *freebsd-update.conf* to upgrade a jail. You must feed freebsd-update the jail's current version of FreeBSD and the root directory. Once you're set up to update a jail, though, an upgrade isn't much different.

The jail **ldap1** runs FreeBSD 11.1 patchlevel something-or-other. I want to upgrade it to a fully patched 11.2. Specify the target release with `-r` and give the `upgrade` sub-command.

```
# freebsd-update -f /etc/jail-update.conf \
   -b /jail/ldap1 --currently-running \
   `jexec -l ldap1 freebsd-version` \
   -r 11.2-RELEASE upgrade
```

You'll be asked to confirm which components are installed and which aren't. After inspecting the system and downloading patches, you'll see lists of files to be updated, added, and removed. Once all that finishes, you'll need to install the updates as a second command.

```
# freebsd-update -f /etc/jail-update.conf \
   -b /jail/ldap1/ --currently-running \
   `jexec -l ldap1 freebsd-version` \
   -r 11.2-RELEASE install
Kernel updates have been installed.  Please reboot and
run "/usr/sbin/freebsd-update install" again to finish
installing updates.
```

It's nice that freebsd-update patched the jail's copy of the kernel file, but it's irrelevant. Jail hosts always run a FreeBSD equal to or newer than any jail. Hit the up arrow to repeat the command and install the new userland.

It's a new version of FreeBSD, so you really should reinstall all of your packages. Use the `-f` flag to force reinstalls even if the software doesn't seem to have changed.

```
# pkg -j ldap1 upgrade -fy
```

Now restart the jail.

```
# service jail restart ldap1
```

Much as with patching, these upgrades are highly amenable to scripting. This script takes one argument, the jail to be upgraded, and upgrades it to FreeBSD 11.2. When I upgrade a bunch of jails, I edit the script to change the version number. You could make the FreeBSD version an argument to the script, but that would mean longer command lines and I can't be bothered.

```
#!/bin/sh
freebsd-update -f /etc/jail-update.conf -b /jail/$1 \
  --currently-running `jexec -l $1 freebsd-version` \
  -r 11.2-RELEASE upgrade
freebsd-update -f /etc/jail-update.conf -b /jail/$1 \
  --currently-running `jexec -l $1 freebsd-version` \
  -r 11.2-RELEASE install
pkg -j $1 upgrade -fy
service jail restart $1
```

If you're upgrading across major releases, you'll also be prompted for merging `/etc/`.

Upgrades with Iocage

You could use freebsd-update(8) with iocage jails, but iocage assimilated a whole bunch of the command line into the `iocage upgrade` command. Give the name of the jail as one argument, and the desired upgrade target with `-r`. My jail **www1** is running FreeBSD 9.3, and I want to upgrade it to the newest FreeBSD 10.

```
# iocage upgrade www1 -r 10.4-RELEASE
```

You'll see the familiar "inspecting system" and "preparing to download files" messages. This particular upgrade required 35,266 patches and 50,864 files, so it's a good time to check your Mastodon feed.

Once all of this finishes, you'll have the joy of merging `/etc` changes into your jail. Iocage automatically updates all the files you haven't changed. Iocage automatically handles the repeated runs of freebsd-update needed to complete the upgrade.

Upgrading iocage jails across major versions is unwise. Iocage creates jails as ZFS clones unless told otherwise. ZFS clones only use an amount of disk space equal to the differences between the snapshot they were cloned from and the current files on disk. Even minor version upgrades can use half a gig of disk, while major revisions suck up the space of a complete FreeBSD install.

```
# zfs list -rt snap iocage/iocage/jails/www1
NAME                                  USED AVAIL REFER MOUNTPOINT
iocage/iocage/jails/www1@install       56K     -   92K  -
iocage/iocage/jails/www1/root@install    0     -  680M  -
```

Upgrading this jail from 9.3 to 10.4 used an additional 680 MB of disk space. That's not a killer, but it's certainly not a space savings over full 10.4 install.

Fortunately, you can configure jails in ways that conserve disk during upgrades. We'll see some of those in the next chapter.

Chapter 6: Space Optimization

Standard jails plop a copy of FreeBSD down onto the disk and use it like any other install. When you have a bunch of jails running the same release, and perhaps even running the same packages, storing terabytes of duplicated files feels inelegant.

Iocage jails optimize disk usage by using ZFS clones. This is great, so long as you don't keep your jails around long enough to need to upgrade across major versions. Once you replace all of the files in a ZFS clone it becomes another copy of the operating system, but with a dependency on the underlying snapshot. Again: inelegant.

No jail system has an elegant solution to this problem. Instead, I offer three different ways to work around it.

Iocage offers a *clone* command that lets you duplicate an existing jail. You can do the same thing with standard jails, of course, but iocage provides a handy script to copy for you.

The next step up from a clone is a *template*, a jail that you configure perfectly and then copy on demand.

Some environments have many jails that must be completely synchronized. The environment demands that all servers run exactly the same FreeBSD version, or that all packages be provably, programmatically identical. Or, perhaps you're desperately short on disk space but need several jails. Use *base jails* to share one filesystem between all these jails.

Which should you use? It depends entirely on your needs. Don't skip ahead to what you think is your preferred solution, though. Each section in this chapter refers to concepts discussed in earlier parts, so read this entire chapter before deploying.

This is not a complete list of optimizations. People reimplement base jails with hard links and even symlinks. Folks painstakingly optimize jail contents, build custom crunched binaries, and set up any number of clever systems. If you find something that works better for you than what I discuss, use it. Unix offers an endless supply of rope, you can hang yourself any way you please.

Clones

If you want an exact copy of an existing jail, clone it. A clone is a fork of a jail; it starts off identical to the original, but diverges when anything changes on either jail. Updates to the original jail do not propagate to its clones: consider base jails if you need such propagation.

Clones are most often used for one-off duplication, such as duplicating an application server so you can test an upgrade. They're generally short-lived.

Standard Jail Clones

With standard jails, clones are pretty simple. On UFS, use tar(1) or even `cp -rp` to duplicate the source directory tree in another directory, then configure the copy as a new jail in *jail.conf*. If you're using ZFS, you can use a ZFS clone.

Really. That's it.

Creating a template from a standard jail is more useful, however, as we'll see in the next section.

Iocage Clones

An iocage clone is a ZFS clone of the original jail's dataset, plus a fresh copy of the iocage support files for that jail. You cannot clone a running jail. Use `iocage clone` to clone a jail. The first argument must be the name or UUID of the jail to be cloned. Assign a new name to the jail with the `-n` argument, and add any parameters you want to change as command-line arguments after the name. Here I clone the jail **www1** to **www2**, assigning it a new IP address with the *ip4_addr* parameter.

```
# iocage clone www1 -n www2 ip4_addr="203.0.113.234"
www2 successfully cloned!
```

When you assign the clone a name, iocage changes the *hostname* value in the clone's */etc/rc.conf* to match. If you don't assign the clone a name, iocage generates a random UUID for the new jail. Again, while this is perfectly fine for automation, it's not so great for human beings.

Maybe you need a whole herd of clones. The `-c` argument to iocage clone lets you specify how many clones to create at once.

```
# iocage clone www1 -c5
6ed60d8f-0274-47bf-86d4-b4483cbaa1d2 successfully
  cloned!
bc586521-6313-4ef3-9a39-e6f569e3c0c9 successfully
  cloned!
…
```

Each clone is assigned a UUID, printed on the command line. If you want to name the jails as you create them, create them individually and use `-n` to assign names.

All of these jails are exactly identical. That brings up a new problem…

Cleaning Clones

Clones start off as exact copies of the original jail. Do you want your jails to all have exactly the same files? Almost certainly not. If nothing else, each jail needs unique SSH host keys. User accounts might have ephemeral information you don't want to carry to the clones.

Fixing sshd(8) host keys is easy. If host key files exist, sshd(8) uses them. Removing the key files forces sshd to create new ones.

You'll probably have other elements that you don't want to replicate across your jails. In my environment, I particularly don't want the original host's */root/.ssh/known_hosts* distributed across new jails; host keys change too frequently, and a bogus entry will pain me months from now. I use a ridiculously simple script, *newjailclean.sh*, that takes one argument, the name of the new jail.

```
#!/bin/sh
rm /iocage/jails/$1/root/etc/ssh/*key*
rm /iocage/jails/$1/root/root/.ssh/known_hosts
```

Clones inherit all of the original jail's settings and configuration, including those in */etc/passwd*. This is guaranteed to cause contention among any group of sysadmins: should the template have user accounts or a root password? Is it better to have an initial default root password that you need to change on each new host? Or should you risk having no root password? In my mind, the failure mode of "this host has an old root password" is less horrific than "this host has no root password," but you must assess the risks in your environment. Whatever you decide, put the result in your jail-cleaning script.

Your script will become especially important when using templates.

Templates

A *template* is a perfect jail specifically designed to be duplicated for other jails. It's basically a variant of a clone, except that the template jail is set to read-only to avoid unintended changes. A template is almost never run as an actual jail, outside of testing. When you deploy a jail from the template, you add on other programs and configurations depending on the jail's role. Templates are only worthwhile if you intend to deploy several jails from the template.

Configure your template to the standard baseline for your network, including all files and services critical to your environment. All of my systems, regardless of operating system or virtualization method, share a *sudoers* sudo configuration file. My servers authenticate via LDAP, so they need *ldap.conf* and *nsswitch.conf*. As a responsible sysadmin I disable password-based authentication on all production, staging, and test servers, so my jails need a custom *sshd_config*. Some environments have custom PAM configurations. I add management users like **ansible** or **nagios** to the template, as well as fallback accounts to handle those inevitable LDAP failures.

Just as with clones, any mistakes in your template get replicated to all jails based on that template. It's a little more high-risk than a clone, however. While a clone is generally used to duplicate an existing jail, a template is intended for deploying new jails. Configure template jails perfectly. Ideally, your organization has a checklist for how to set up a new system. While a jail can't encrypt its own disk or have a serial console, steps like "configure LDAP" and "install tmux" certainly apply. Use everything you can steal from the checklist, and then test everything. Does the SSH server start with your new configuration? Does LDAP authentication work? How about logging; does each jail run its own syslogd(8), or do they forward all log messages to a syslog jail?

After you test the template, have someone else test it again. You can either fix the template once, before deployment, or fix all of your jails later.

It's entirely possible to create templates from templates, but I advise against it. The thought of an organizational template, which you snapshot and clone to create templates for different types of servers, which you snapshot and clone to create individual servers, might feel tempting, but such trees are fragile and make changes difficult. Resist this temptation, especially if you're using ZFS.

Standard jails on ZFS can use a ZFS snapshot as a template. On UFS, build a standard jail template by tarring up the template jail's directory and extracting it when needed.

Iocage's template support is more complicated, and more featureful.

Creating Iocage Templates

Initially, your iocage template is a boring old jail exactly like every other jail. Give it a temporary IP that will allow it access to all necessary network resources. I'm going to test LDAP and SSH functionality, so the jail needs an address on my network.

```
# iocage create -n dnstemplate \
   ip4_addr="203.0.113.245" -r 11.2-RELEASE
```

Exactly like other jails, **`dnstemplate`** is a ZFS clone of iocage's directory of a FreeBSD release. In this case it's a clone of 11.2. You'll modify the clone **`dnstemplate`**, and then clone **`dnstemplate`** to create more jails.

All of my servers need the tmux, openldap-client, emacs-nox, and sudo packages. Additionally, my DNS servers run BIND 9.13. I install all of these in the template jail.

```
# iocage pkg dnstemplate install openldap-client sudo \
   bind913 emacs-nox tmux
```

Now I copy my network's vital configuration files into the template jail and create standard users.

My template jail is intended as a base template for DNS servers, and while I've installed the organization's preferred DNS server, I won't configure DNS yet. Authoritative and recursive DNS servers have very different configurations.

When iocage creates a jail based on a template, it duplicates all of the template's parameters as well. If jails based on this template need specific non-default iocage parameters, adjust those too.

Once you're certain that the template jail works correctly, use the clone-cleaning script discussed under "Clones" to remove any ephemera that must not be replicated on jails based on this template.

Reclaim any temporary settings, such as temporarily loaned IPs, before finalizing the template. Iocage lets you assign one IP to multiple jails, but you'll need to either configure those jails to avoid TCP/IP port overlaps or make sure you don't run multiple jails with that IP simultaneously. Future You will appreciate you preventing those overlaps while they're in front of you.

Once your template jail is set up exactly as desired, convert it to a template by setting the *template* parameter. This makes the jail read-only at the ZFS level and moves the dataset from */iocage/jails* to */iocage/templates*.

```
# iocage set template=yes dnstemplate
dnstemplate converted to a template.
```

Template jails do not start at system boot, even if the *boot* property is set to on, and cannot be manually started. Running the jail would involve writing to the dataset, and we can't be having that.

View all templates with `iocage list -t`.

```
# iocage list -t
+-----+------------+-------+-------------+---------------+
| JID |    NAME    | STATE |   RELEASE   |      IP4      |
+=====+============+=======+=============+===============+
| -   | dnstemplate | down  | 11.2-RELEASE | -             |
+-----+------------+-------+-------------+---------------+
```

For more details on your templates, add the `-l` flag.

```
# iocage list -tl
```

We can now deploy jails based on this template.

Using Iocage Templates

To create a jail based on a template, use `iocage create` with the `-t` flag. Give the template name as an argument. Here I create a jail, **`dns3`**, based on the template **`dnstemplate`**.

```
# iocage create -t dnstemplate -n dns3
```

Take a look at the results.

```
# iocage list
+-----+--------+-------+-------------+---------------+
| JID |  NAME  | STATE |   RELEASE   |      IP4      |
+=====+========+=======+=============+===============+
| -   | dns3   | down  | 11.2-RELEASE | -             |
…
```

Iocage does not copy uniquely per-jail items, such as IPv4 and IPv6 addresses, to the new jail. I must set IP addresses if I want them. I also must set booting at system startup. Here I do everything in a single command.

```
# iocage set boot=on ip4_addr="203.0.113.243" dns3
Property: boot has been updated to on
Property: ip4_addr has been updated to 203.0.113.243
```

I can now start the jail and configure its specific function, confident that all its core services work.

Templates and ZFS

Let's look at how iocage templates work at the filesystem level. The default output of `zfs get` is pretty wide, so I use the `-o` option to trim unneeded columns from the output. I also ordered the output to more easily explain what's going on and to fit the page.

```
# zfs get -o name,value -r origin iocage | grep dnstemplate
iocage/iocage/releases/11.2-RELEASE/root@dnstemplate  -
iocage/iocage/templates/dnstemplate                   -
iocage/iocage/templates/dnstemplate/root
               iocage/iocage/releases/11.2-RELEASE/root@dnstemplate
iocage/iocage/templates/dnstemplate/root@dns3         -
iocage/iocage/jails/dns3/root
                      iocage/iocage/templates/dnstemplate/root@dns3
```

The first entry, *`iocage/iocage/releases/11.2-RELEASE/root@dnstemplate`*, is a snapshot of iocage's FreeBSD 11.2 installation directory. All jails installed running 11.2 start as a clone of this dataset.

The second entry, *`iocage/iocage/templates/dnstemplate`*, is a dataset to contain the **`dnstemplate`** jail and its iocage configuration file.

Third is the dataset for the files belonging to jail **`dnstemplate`**, *`iocage/iocage/templates/dnstemplate/root`*. It's a clone of the 11.2 release in the first snapshot.

Fourth, we have the snapshot *`iocage/iocage/templates/dnstemplate/root@dns3`*. This is the state of the template **`dnstemplate`** when we created the jail **`dns3`** from it.

Last we have *`iocage/iocage/jails/dns3/root`*, the dataset for the files inside jail **`dns3`**. It's a clone of the dataset in the fourth entry.

Taken as a whole, we've cloned the 11.2 installation directory to create the template **`dnstemplate`**, configured **`dnstemplate`**, and then cloned **`dnstemplate`** to create the jail **`dns3`**. All jails based on dnstemplate have a dependency on the 11.2 release, even if we eventually upgrade them to FreeBSD 13 or 15 or 30. This tree will get more complex if you use a template as a basis for another template, which you then create jails from.

From this it's easy to see how you'd create a ZFS-based jail template without iocage.

Standard Jail Templates

If you're using standard jails on ZFS, you can use ZFS snapshots to build your own templates. If you're using UFS, build templates with tar(1). Either way, build your pristine jail first. Test it exactly as you would when using iocage.

To use templates via ZFS clones, each jail must be its own dataset. Here I create a ZFS dataset for my DNS server template.

```
# zfs create jail/dnstemplate
```

If you're running UFS, create the directory */jail/dnstemplate*.

No matter what filesystem you're using, extract your chosen *base.txz* into that directory and make an */etc/jail.conf* entry for **dnstemplate**. Copy */etc/resolv.conf* into the jail. Start the jail and install the needed packages.

```
# pkg -j dnstemplate install openldap-client sudo \
   bind913 emacs-nox tmux
```

Now add other critical common files to the template jail, such as *sudoers* and *ldap.conf*. Assign a default root password and create any local users. Remember, jails based on this template inherit all of these users and passwords. Test your template jail to verify that the core functions like LDAP and sudo behave as expected.

As a very last step, eliminate files that should not be replicated: */root/known_hosts*, SSH host keys, and any other ephemera. Ideally, create a list of files that should be removed and write a script to destroy them. You'll need it again, when you update the template if nothing else.

Shut the template jail down. It's now ready to duplicate.

For ZFS, snapshot the jail's dataset and create a clone from the snapshot. I name my snapshots by date. Here I create the filesystem for jail **dns1** from the template.

```
# zfs snapshot jail/dnstemplate@2018-11-08
# zfs clone jail/dnstemplate@2018-11-08 jail/dns1
```

While this would be a lot of work to create one server, creating additional servers off this template is a single command.

```
# zfs clone jail/dnstemplate@2018-11-08 jail/dns2
# zfs clone jail/dnstemplate@2018-11-08 jail/dns3
# zfs clone jail/dnstemplate@2018-11-08 jail/dns4
```

This illustrates how templates are only worthwhile in bulk.

Templates on UFS use more space, but many people find throwing lumps of files around conceptually simpler than ZFS snapshots. Back up your template jail to a tarball. I strongly recommend including the date in the tarball name, so you can easily update your template.

```
# cd /jail/dnstemplate
# tar -czvf /jail/media/dnstemplate-2018-11-08.tgz .
```

When you want to deploy the jail, extract the tarball.

```
# cd /jail/dns1
# tar -xpf /jail/media/dnstemplate-2018-11-08.tgz
```

Whether you're on UFS or ZFS, configure **dns1** in */etc/jail.conf* and you're ready to go.

Updating Templates

Experienced sysadmins all understand that crafting and deploying a perfectly polished server template is the only excuse the universe needs to obsolete that template. Security advisories and patches arrive daily. Every time you deploy a new jail, via any method, immediately run `freebsd-update` and `pkg upgrade` (Chapter 5) to update it. Templates complicate patching, however.

A template and jails created from the template are different entities. Updates to a template affect only new jails created from that template, not existing jails. Patching iocage jails built from templates, or standard jails built from ZFS clones, causes those jails to grow larger. You'll want to update your template so that newly deployed jails are less obsolete and require less patching.

Should you update your template every time you deploy a new jail? Not necessarily. Most security updates are comparatively tiny, and constantly updating the template means extra testing. *Never* updating templates isn't a good idea either, though. I generally update my templates every time the underlying FreeBSD has a new point release. If I install a template with 12.0, I update the template at 12.1, 12.2, and so on. This minimizes size increases on new ZFS clones.

When you declare an iocage jail to be a template, iocage makes the ZFS dataset read-only. You'll need to declare the template jail to no longer be a template, start and update it, and declare it to be a template again.

```
# iocage set template=no dnstemplate
dnstemplate converted to a jail.
Property: template has been updated to no
```

While your template should have an existing snapshot for every jail based on the template, I'd recommend creating a specific pre-upgrade snapshot right now. I name snapshots after the date. If you don't specify a snapshot name as the final argument, iocage assigns one based on the current date and time.

```
# iocage snapshot dnstemplate
Snapshot: iocage/iocage/templates/
   dnstemplate@2019-02-25_20:17:46 created.
```

Now upgrade the jail, restart the jail, and upgrade the packages.

```
# iocage update dnstemplate
# iocage restart dnstemplate
# iocage pkg dnstemplate upgrade -y
```

Your system and packages are now updated, so something's subtly busted. That's the rule. Fix the problems, verify that everything works as expected, then shut down the jail.

Remove any files that shouldn't be propagated to new jails, such as the template's SSH host keys. Remember in the last section, when I suggested you write a script to remove these files? This is why. Run your convenient template-cleaning script.

```
# templateclean.sh dnstemplate
```

Now switch the jail back to a template.

```
# iocage set template=yes dnstemplate
```

Your jail is now updated. New jails cloned from this template will get the most secure versions of all installed software… if you create them before the next security advisory.[13]

Base Jails

Perhaps the biggest pain point in jail management is applying security patches. You must apply patches immediately if not sooner, but disks have limited throughput. When you have hundreds of jails, patching is a not-inconsiderable undertaking and imposes a noticeable performance hit. Similarly, some environments need their jails to all run the exact same FreeBSD release, often down to the packages. If you manage dozens of jails or require meticulous version synchronization, consider *base jails*.

13 "Sysadmin" is just another way to say "professional patcher." You're a roofer, but indoors and with different tar.

The term "base jail" originated in the days when people would remount the base operating system as a filesystem for all the jails. Once upon a time, disks were so small that an operating system took up a substantial portion of a system's storage. Reusing the base install for a jail was a substantial savings. That feels quaint in this age of multiterabyte disks.

Nowadays the term describes any jail setup derived from that type of configuration. Most often there's a FreeBSD install somewhere on the disk, exactly as for a regular jail, but which gets remounted for use by other jails. The term "base jail" could mean the type of jails being used, the jail upon which others are based, or a jail that's based on another jail. In other words, a base jail is a base jail based on the base jail. In the interest of clarity, I'll refer to a jail underlying other jails as an *origin* jail. Jails built on that base (dang it!) I'll call *derived* jails. Once you understand how everything works and can infer the exact meaning of "base" from context, feel free to call everything a base jail and perplex the next generation.

FreeBSD's nullfs(5) allows connecting directories to multiple points of the directory tree. A modern base jail takes advantage of nullfs to mount key directories from a single jail's filesystem over many other jails. The derived jail mounts those directories read-only, so a derived jail can't muck with its peer derived jails. Updating the origin jail magically updates the binaries and other critical files in all the jails. It won't update `/etc` for you, but handles all the binaries and libraries.

Base Jail Packages

Base jails complicate software management. Think about templates for a moment. The main reason to deploy jails from a template rather than from a release is because all your jails need a common set of packages and configuration files. Including packages in an origin jail gets complicated quickly, though. It's not that including packages in the origin jail is technically complicated, but the administration of those packages can spiral into a tangled mess.

The obvious approach is to mount the origin jail's */usr/local* in the derived jail. This works great, if all of your derived jails need exactly the same packages. But do they? Do they *really*? How many of your web servers have that one special package required by no other server? With */usr/local* mounted read-only, jail administrators cannot install additional packages. Any package you install in the origin jail is available to all jails. You must separate jails by role. Fine. Let's suppose you need a whole bunch of nginx server jails, so you create an nginx origin jail. Some of your applications need PHP 5.6, though, while others need PHP 7. FreeBSD's packaging system doesn't support installing multiple versions of PHP. Do you install both by hand, one as */usr/local/bin/php56* and the other as */usr/local/bin/php7*, and manually install all the modules? Or do you carefully divide your jails by very precise software requirements? What do you do when one jail needs a specific php-xmlreader package, but another jail explicitly needs that package to *not* be installed?[14]

You choose your pain, that's what you do.

Another option would be to use poudriere to build packages that use a different package root directory by setting $PREFIX. You could put your origin jail's packages in */usr/pkg*, like NetBSD, and reserve

14 It's a short step from here into complete containerization. Or insanity. But I repeat myself.

/usr/local for truly local packages. A whole bunch of software doesn't recognize software installed outside the root directory or */usr/local*, however. NetBSD's valiant pkgsrc developers do an inordinate amount of work to make software in */usr/pkg* work correctly, but many upstream projects don't accept those patches because what sort of loony needs software installed outside */* or */usr/local*? Default FreeBSD packages might not recognize software in */usr/pkg*, resulting in duplicated or conflicting software installs.

It can be done. It's a great deal of work. Do you want to do that work? If you elect to perform that work, submit patches.

Or, you could use poudriere to build packages that install directly under the root directory. Rather than installing sudo(1) as */usr/local/bin/sudo*, put it in */bin/sudo*. I abhor this solution, even though it works. Mostly. One of FreeBSD's key advantages is the clear separation between the base system and packages. A file from a package might overwrite a base system file. Again, basing packages in */* can be done, but requires careful testing.

Even if your software is all identical, chances are your server configuration will differ. All those configuration files that normally appear in */usr/local/etc* but the jail admin needs to edit must go someplace local to the derived jail. For example, while FreeBSD's Apache port uses a default configuration file of */usr/local/etc/apache24/httpd.conf*, that directory would be mounted from the derived jail. The jail admin will be rightfully upset if she can't edit it. You'll need a configuration directory like */etc/apache24*. Perhaps you'll need an */etc/rc.conf* entry to point Apache at a configuration file in that directory. Maybe you'll keep the standard configuration file location, but edit it to suck in additional configuration files from the local configuration directory. Configuration files used globally, like *ldap.conf* and *sudoers*, can remain in */usr/local/etc*.

Or, you can meticulously group jails by role and software requirements and develop a comparatively painless process to migrate derived jails between origin jails. Fortunately, that's not terribly difficult.

Iocage Base Jails

Iocage defaults to using a release as the origin jail. This maximizes flexibility for each derived jail, but minimizes shared local customization. Patching the release updates the FreeBSD install for each jail, but doesn't affect the packages or configuration.

Create a derived jail by adding the `-b` flag when creating the jail.

```
# iocage create -r 11.2-RELEASE -b -n dns3
dns3 successfully created!
# iocage start dns3
```

Before customizing the new jail, let's examine the filesystems and mounts iocage uses for this base jail. Widen your terminal to 120 characters or so and follow along.

```
# mount | grep dns3
iocage/iocage/jails/dns3 on /iocage/jails/dns3
   (zfs, local, nfsv4acls)
iocage/iocage/jails/dns3/root on /iocage/jails/dns3/root
   (zfs, local, nfsv4acls)
```

These datasets are for the jail's iocage directory and the jail's root directory, exactly like any other jail.

```
/iocage/releases/11.2-RELEASE/root/bin on
  /iocage/jails/dns3/root/bin (nullfs, local, read-only)
```

Here's the first piece of base jail magic. The */bin* directory from the FreeBSD release is mounted read-only on the jail's */bin* directory. This same remounting happens for */boot*, */lib*, */libexec*, */rescue*, */sbin*, */usr/bin*, */usr/include*, */usr/lib*, */usr/libexec*, */usr/sbin*, */usr/share*, */usr/libdata*, and */usr/lib32*. After all that, you'll find the standard devfs(5) and fdescfs(5) mounts that appear in all iocage jails.

With all of these remounts and overlays from the origin jail, what's left to handle inside the derived jail? For one, everything in */etc*. Local configuration is entirely delegated to the derived jail. As user accounts aren't centrally managed by the jail, */usr/home* needs to remain with the derived jail. Each jail manages its own logs and packages, so the derived jail includes */usr/local* and */var*. If your jail includes the FreeBSD source code it's presumably so you can hack on it, so the jail gets its own copy. Directories like */var* and */home* all go with the derived jail.

When creating the derived jail, iocage populates only the directories managed by the derived jail. As */bin* and */sbin* get remounted from the origin jail, iocage empties the derived jail's */bin* and */sbin*. If you're in doubt about which files iocage copies into a derived jail and which get remounted from the origin jail, stop the jail and examine the derived jail's directory tree. Test this yourself. Here, base jail **dns3** is running.

```
# cd /iocage/jails/dns3/root
# ls bin
[                df               kill             ps
…
```

Stop the jail and look again.

```
# iocage stop dns3
# ls bin
#
```

With the origin jail's directories unmounted, the derived jail's */bin* is empty.

Jailed users perceive their filesystem differently. The jail appears as a single contiguous filesystem rather than a morass of nullfs mounts. A jail has no knowledge of those mythical filesystems beyond its root directory, so those mounts are invisible.

```
# iocage exec dns3 mount
iocage/iocage/jails/dns3/root on / (zfs, local,
   nfsv4acls)
```

A curious user with root access to a jail can trivially identify read-only directories with touch(1). Once she starts looking, she'll quickly realize that she's in a base jail. Don't count on the invisibility of base jail mounts for any sort of security.

To see all base jails on the system, run `iocage list -B`.

```
# iocage list -B
+-----+------+-------+--------------+---------------+
| JID | NAME | STATE |   RELEASE    |      IP4      |
+=====+======+=======+==============+===============+
| -   | dns3 | down  | 11.2-RELEASE | 203.0.113.243 |
+-----+------+-------+--------------+---------------+
```

For the long jail list that includes only base jails, add the `-l` flag.

```
# iocage list -Bl
```

You can use other iocage list flags like `-q` and `-s` to modify the output.

Custom Iocage Base Jails

Derived jails that use a FreeBSD release as the origin are handy when you want to update a base operating system install once and have those updates automatically replicate to all the derived jails. But what if you want a customized origin jail, including additional files? Use a template as the base jail.

Create the derived jail as you would any other template-based jail by using the `-t` flag, but add the `-b` flag to create it as a base jail. Here I create a derived jail named **`dns4`** that uses the template **`dnstemplate`** as the origin jail.

```
# iocage create -t dnstemplate -b -n dns4
```

Any updates you apply to **`dnstemplate`** automatically propagate to **`dns4`**.

Iocage Base Jail Rebasing

No matter how carefully you assign your base jails to their roles, eventually you'll need to move a derived jail to another origin jail. Future releases of iocage will include an option to do this automatically, but for the moment you'll need to manually rebase your jails by backing up the old derived jail and restoring it in a new derived jail.

On the host, use tar(1) to back up the derived jail. The `--one-file-system` flag tells tar to not cross filesystem boundaries. It won't cross the nullfs mounts holding the files remounted from the origin jail, thus including only what's in the derived jail's filesystem.

```
# tar -czf /home/mwlucas/dns4derived.tgz \
   --one-file-system /iocage/jails/dns4/root/
```

Create a new derived jail and restore the critical files. You'll need to manually compare files like */etc/rc.conf* and build new configurations. If the origin jail provides */usr/src*, you can use mergemaster(1). Which exact files you can restore, which you mustn't, and which you must evaluate and rebuild depends on exactly how you're using base jails.

You could leave the derived jail in place and manually change the jail's *fstab* to point at the new origin, but this goes outside iocage management and something might break.

Standard Base Jails

Using a base jail with FreeBSD's standard jail management tools is only slightly more complicated than using iocage. You must prepare the origin jail, create an *fstab*, prepare the derived jail, and configure *jail.conf*. We'll create an origin jail (**`ldapbase`**) and a derived jail (**`ldap3`**), and also create a template to ease deployment of further jails using the same origin jail.

Create your origin jail exactly like any other jail. If you'll be including packages with your base jail, install those packages in the origin jail. If you wish, create the origin jail from a template. Each of your derived jails will have an independent `/etc`, so the origin jail's cryptographic keys can't leak into the derived jails. Configure the origin jail in the host's `/etc/jail.conf` and verify that it functions as expected. The jail should be able to reach the Internet, provide a shell, cleanly start and stop, and all the other functions you'd expect. My origin jail is intended for LDAP servers and is called **`ldapbase`**. As this origin jail shouldn't start automatically at system boot, I exclude it from the host's `rc.conf` settings.

Decide which directories your base jail provides from the origin jail and which must be in the derived jail. Base jails default to "all directories not provided by the origin jail belong in the derived jail," but it's worth considering if this specific application must write to specific directories normally provided by the origin jail before trying to deploy. At a minimum, an origin standard jail must provide the directories an iocage origin jail provides. You'll use this list to build the filesystem table for your jail.

Remember back in Chapter 4 when we discussed how mount(8) can use `fstab` files other than `/etc/fstab`? We'll use the *mount.fstab* parameter to give each jail its own filesystem table, in the exact same syntax as `/etc/fstab`. The jail system mounts these filesystems before creating the jail and unmounts them after shutting down the jail.

Store each jail's `fstab` outside the jail. You might find it useful to create a directory for each jail to contain jail metadata, much as iocage does, or perhaps create directories for different types of per-jail files.

I create a `/jails/fstab/` directory for `fstab` files, each named after the jail. My first derived jail will be called **`ldap3`**, so it uses `ldap3.fstab`. Here's a typical `fstab`, providing the same filesystems iocage-based derived jails inherit.

```
/jail/ldapbase/bin          /jail/ldap3/bin         nullfs ro 0 0
/jail/ldapbase/boot         /jail/ldap3/boot        nullfs ro 0 0
/jail/ldapbase/lib          /jail/ldap3/lib         nullfs ro 0 0
/jail/ldapbase/libexec      /jail/ldap3/libexec     nullfs ro 0 0
/jail/ldapbase/rescue       /jail/ldap3/rescue      nullfs ro 0 0
/jail/ldapbase/sbin         /jail/ldap3/sbin        nullfs ro 0 0
/jail/ldapbase/usr/bin      /jail/ldap3/usr/bin     nullfs ro 0 0
/jail/ldapbase/usr/include  /jail/ldap3/usr/include nullfs ro 0 0
/jail/ldapbase/usr/lib      /jail/ldap3/usr/lib     nullfs ro 0 0
/jail/ldapbase/usr/libexec  /jail/ldap3/usr/libexec nullfs ro 0 0
/jail/ldapbase/usr/sbin     /jail/ldap3/usr/sbin    nullfs ro 0 0
/jail/ldapbase/usr/share    /jail/ldap3/usr/share   nullfs ro 0 0
/jail/ldapbase/usr/libdata  /jail/ldap3/usr/libdata nullfs ro 0 0
/jail/ldapbase/usr/lib32    /jail/ldap3/usr/lib32   nullfs ro 0 0
```

Now create a template for derived jails.

A derived jail includes everything that isn't in the origin jail. The easiest way to create the derived jail template for standard jails is to take the template used to create the origin jail and remove everything the base jail provides. The *fstab* file provides a convenient list of directories to be emptied. Don't remove the directories—you can't mount anything on nonexistent directories! Only remove the contents.

```
# tar -xpf template.tgz -C /jail/ldap3
# rm -rf /jail/ldap3/boot/*
# rm -rf /jail/ldap3/bin/*
…
```

You might save this as a new template for derived jails. As the origin jail contains all of the stuff that gets patched during an upgrade, a derived jail template remains valid longer than most templates.

You might find you can't delete some files.

```
# rm -rf /jail/ldap3/lib/*
rm: /jail/ldap3/lib/libc.so.7: Operation not permitted
rm: /jail/ldap3/lib/libcrypt.so.5: Operation not permitted
rm: /jail/ldap3/lib/libthr.so.3: Operation not permitted
```

These files are installed with the immutable flag set. Clear that flag and try again.

```
# chflags -R noschg /jail/ldap3/lib/
```

If your standard jail template included packages, those packages got copied into the origin and derived jails. The *fstab* above doesn't remount the origin jail's */usr/local* onto the derived jail, so the jail has its own independent copy of those packages. The template also includes */var/db/pkg*, the package database, so the templated jail can manage those packages. You might want to include the origin jail's packages instead… or not. Again, package management with base jails is a matter of agony selection.

Last, make a *jail.conf* entry for your derived jail. "Base jail" and related words doesn't appear anywhere in *jail.conf*. All you need to create a derived jail is the *mount.fstab* parameter that specifies an *fstab* to mount before starting the jail.

```
ldap3 {
   ip4.addr="203.0.113.239";
   mount.fstab="/jail/fstab/ldap3.fstab";
}
```

Default Base Jail fstab Files

I'm always tempted to make *mount.fstab* a default setting. If it's a default, however, you can't start or stop any configured jail without an *fstab*. Creating an empty *fstab* file with touch(1) satisfies the jail system, though, or you could set *mount.nofstab* in the jail definition. If you set *mount.fstab* as a default, you must either create an *fstab* for every jail or set a property for *fstab*-free jails.

```
mount.fstab="/jail/fstab/$name.fstab";
```

Which is right for your environment? All I know is, whatever choice you make will annoy you.

Standard Base Jail Crashes

I treat my jails badly. When they threaten to stay up, I put a stick between their legs so they trip. At times, I unceremoniously crash them. For normal jails this isn't a big deal, other than recovering any dirty databases, but a crashed base jail leaves wreckage behind. The jail shutdown process unmounts any filesystems mounted during the startup. If the jail crashes, you need to manually unmount those filesystems.

```
# umount -a -F /jail/fstab/ldap3.fstab
```

Derived jails won't restart until you unmount leftover filesystems. That might seem unnecessary, but jail(8) thinks that if it's responsible for those filesystems it's gonna dang well be sure the right ones are mounted.

Packages and Standard Base Jails

A standard set of packages that's centrally managed in a single origin jail and yet meets every need for every jail is the Platonic ideal, long-sought and forever unattainable. All those problems that apply to distributing packages in iocage base jails apply exactly as well to standard base jails. You're going to wind up with several different origin jails that each have their own package sets, and shifting derived jails between those origin jails as needed. Here we run into the deadliest part of jail management, and the system administrator's nemesis: *organization*.

Sysadmins love creating organizational schemes. When we start a new assignment or job, the first thing we do is gripe about our predecessor's lack of organizational expertise, and loudly declare that our new and improved scheme will handle all problems for every conceivable use case, forever, while simultaneously optimizing resource use. Inside six months we're looking for a new way to organize everything, as this environment is "unexpectedly complicated." We struggle with the system for a while longer, eventually giving the whole thing up as a bad job and searching for a new gig.[15]

One factor many people cite in deciding to use base jails is optimizing disk space usage. There's nothing wrong with this fully natural tendency for efficiency, but combined with base jails it leads directly to an organizational nightmare. I've seen several setups where people install a single origin jail, derive jails from that for different package loads, and then carefully cross-mount and remount specific */usr/local* and */var/db/pkg* directories to other derived jails depending on each jail's needs. It's unquestionably space-efficient, yes, but exchanges a few gig of disk for a vastly increased chance of outages incurred from administrative overhead.

Once you find yourself contemplating deriving jails from derived jails, step back and slap yourself until you regain your senses. Base jails are *complicated*. Don't increase their complexity. Instead, do this complex thing in the simplest possible manner.

I recommend creating an origin jail for each type of package load. Suppose I have a bunch of web server jails. They all run the same web server software but have different PHP versions. Create an origin jail for each set of packages you want available, and give the origin jail a name that represents that package set. If I have a group of jails running FreeBSD 12 with (now-EOL'd) PHP 5.6, and another running FreeBSD 12 with PHP 7.1, I might call them **12php56** and **12php71**. I don't include FreeBSD minor versions in the names, because I'll be upgrading these jails throughout the FreeBSD 12 lifespan.

15 I know, I know. You're the exception, and your systems and records are pristine. I'm talking about those *other* sysadmins.

Preparing an Origin Jail With Packages

Let's start with creating the 12php56 origin jail. Extract `base.txz` from FreeBSD 12 into `/jail/12php56`.

```
# tar -xpf /jail/media/12.0/base.txz -C /jail/12php56/
```

Now do the basic setup.

```
# cp /etc/resolv.conf /jail/12php56/etc/
# cp /etc/localtime /jail/12php56/etc/
# chroot /jail/12php56/ passwd root
# touch /jail/12php56/etc/fstab
```

As I've set *mount.fstab* as a default in `jail.conf`, each origin jail needs a blank jail(8) `fstab` file.

```
# touch /jail/fstab/12php56.fstab
```

Now create a `jail.conf` entry for this origin jail.

```
12php56 {
  ip4.addr="203.0.113.221";
}
```

If everything is correct, the jail should start.

```
# service jail start 12php56
Starting jails: 12php56.
```

The jls(8) command shows the jail is running. I did it right? Huh. Cool.

Get the latest security patches.

```
# freebsd-update -f /etc/jail-update.conf \
   -b /jail/12php56/ --currently-running \
   `jexec -l 12php56 freebsd-version` fetch install
```

Now install the desired packages in the jail. Make sure you put the correct versions in the correct jail—installing PHP 7.1 in the PHP 5.6 jail wrecks the whole endeavor.

```
# pkg -j 12php56 install apache24 php56 mariadb103-server
```

Technically, you have a complete origin jail for web servers using PHP 5.6. A little extra effort here can save you work later, though. Remember, the default configuration files all live in */usr/local*, which will be read-only mounted from the origin jail. Your derived jail will need its own configuration files for many services. Providing suitable flags and examples in */etc/rc.conf* right now will save you much scrambling later. For example, Apache lets you set a custom configuration file with the `-f` flag. Adding a placeholder entry to *rc.conf* will help Future You appreciate Present You.

```
apache24_flags="-f /etc/httpd.conf"
```

Even if you need to change the configuration file location, a good example helps.

You don't need to provide alternate configurations for all programs. Some configurations, like the *sudoers* used by sudo(1), probably should be managed entirely from the origin jail.

Now let's derive a jail from this origin.

Preparing a Derived Jail with Packages

Let's set up **www8**, a base jail derived from **12php56**. A derived jail is identical to an origin jail, minus the directories provided by the origin jail. Figure out what comes from the origin jail, and create an *fstab* for that. Here's an *fstab* for **www8**, *www8.fstab*. It looks much like the *fstab* for any other derived jail, but includes entries for */usr/local* and */var/db/pkg*.

```
/jail/12php56/bin           /jail/www8/bin           nullfs ro 0 0
/jail/12php56/boot          /jail/www8/boot          nullfs ro 0 0
/jail/12php56/lib           /jail/www8/lib           nullfs ro 0 0
/jail/12php56/libexec       /jail/www8/libexec       nullfs ro 0 0
/jail/12php56/rescue        /jail/www8/rescue        nullfs ro 0 0
/jail/12php56/sbin          /jail/www8/sbin          nullfs ro 0 0
/jail/12php56/usr/bin       /jail/www8/usr/bin       nullfs ro 0 0
/jail/12php56/usr/include   /jail/www8/usr/include   nullfs ro 0 0
/jail/12php56/usr/lib       /jail/www8/usr/lib       nullfs ro 0 0
/jail/12php56/usr/libexec   /jail/www8/usr/libexec   nullfs ro 0 0
/jail/12php56/usr/sbin      /jail/www8/usr/sbin      nullfs ro 0 0
/jail/12php56/usr/share     /jail/www8/usr/share     nullfs ro 0 0
/jail/12php56/usr/libdata   /jail/www8/usr/libdata   nullfs ro 0 0
/jail/12php56/usr/lib32     /jail/www8/usr/lib32     nullfs ro 0 0
/jail/12php56/usr/local     /jail/www8/usr/local     nullfs ro 0 0
/jail/12php56/var/db/pkg    /jail/www8/var/db/pkg    nullfs ro 0 0
```

The whole point of this derived jail is that it gets all packages from the origin jail, so why drag along */var/db/pkg*? Users can use pkg(8) to query the package database, saving me the trouble of answering trouble tickets with "yes, php56-xmlwriter is installed, the problem is your busted code."

Our derived jail runs FreeBSD 12.0, like the origin jail. If you have a tarball containing a 12.0 derived jail, you can use it. If not, prepare one from our new origin jail. Here I copy our origin jail, */jail/12php56*, in its entirety into */jail/www8*.

```
# tar cfC - /jail/12php56/ . | tar xpfC - /jail/www8/
```

Now empty all directories the origin jail will provide. Remember, don't destroy the directories, only the content of the directories. You can't mount filesystems on nonexistent directories. Use the *www8.fstab* file as a checklist.

```
# rm  /jail/www8/bin/*
…
# rm -rf /jail/www8/usr/local/*
# rm -rf /jail/www8/var/db/pkg/*
```

I recommend using full file paths, to avoid accidentally emptying the host's */bin*.[16]

16 Again.

We never started sshd(8) or used an SSH client, so there's no need to blow away the host keys or any *known_hosts* files.

You now have a directory tree for a FreeBSD 12.0 jail. Save it so you can skip purging these directories next time you want a jail.

```
# tar cfC /jail/media/12.0-pkgs.tgz /jail/www8/ .
```

Now configure the derived jail in */etc/jail.conf*. If you've defined *mount.fstab* as a default property, all you need to set is an IP address.

Now that you've set up a single jail and saved a tarball of the derived jail, creating other derived jails is a matter of extracting that tarball in another directory, creating an *fstab*, and making a *jail.conf* entry. As that tarball doesn't include any packages, you can use it for any origin jail of the same release.

Rebasing Standard Jails

All software becomes obsolete. PHP 5.6 is past its end-of-life, and eventually you'll get the time to stop using it. You'll need to migrate all your applications to a newer version of PHP. Rather than installing the newer software in the origin jail and attempting to migrate all of your derived jails simultaneously, rebase each derived jail one-by-one to an origin jail running a different version of that software. Changing a derived jail to rely on a different origin jail running the same FreeBSD release is trivial. Merely edit the derived jail's *fstab* to pull in the other origin jail's directories. You can safely edit a jail's *fstab* while the jail is running; jail(8) can unmount nullfs mounts even with an incorrect mount device in *fstab*.

Yes, changing a standard base jail's origin jail is as simple as a global search-and-replace and a restart.

The tricky part of rebasing a jail comes in integrating changes between the different FreeBSD versions. Jails make building test servers fairly straightforward, especially if you're using ZFS and use clones. Yes, you must configure your applications to run on a different IP address or perhaps use host files, but that's wholly unrelated to jails. While you can minimize some application changes, you're still stuck with upgrading the base jail's */etc*. The simplest way is to share */usr/src* on your origin jail and use mergemaster(8). Once you've performed a few of these upgrades, you'll have a really good idea what files can be kept, which need wholesale replacement, and which need careful handling.

Chapter 7: Living In A Jail

A lightweight virtual machine isn't the same as a host or a fully virtualized system. For one, the host imposes restrictions on the jail. The jail owner can't change those restrictions from within the jail; he must request those privileges from the host sysadmin. These restrictions are a feature—if there's no reason for your database server to be pinging the outside world, why permit that activity?

Being a jail administrator means knowing your limits, understanding them, and working around them. The most obvious limits involve rebooting, processes, and network access.

Restarting Your Jail

Commands that shut down or restart the system, like halt(8), reboot(8), and so on, don't work in jails. These commands work by signaling the kernel and the master process init(8) that it's time to carry out their final tasks: signal all the processes to shut down gracefully, kill all the processes, flush the disk cache, and either halt or restart the hardware. Jails can query but not change the kernel, and they have no init process. The result?

```
# reboot
reboot: SIGTSTP init: No such process
```

Strictly speaking, a jail sysadmin should never need to reboot their jail. A jail is only a bunch of processes trapped in a collection of transformed namespaces. Seasoned sysadmins understand that it's perfectly possible to restart all those processes and achieve the same effects as a reboot. Seasoned sysadmins also know that sometimes the easiest and least risky way to achieve these restarts is to reboot the dang jail.

The only way to restart a jail is from the host. You must ask your host sysadmin to perform the reboot. If you hunt around, you can find add-on scripts that will let jail owners reboot their jails. None of them have been nicely packaged for installation, though.

Processes and Jails

A host's processes and process IDs are each unique. A jail does transform a processes' namespace, but only by restricting which processes can be viewed within the jail. If PID 2029 is in the jail **loghost**, no other PID 2029 appears anywhere else on the host, jailed or not. PID 2029 can only see processes in its own jail, however. Jails cannot see PID 1, or */sbin/init*.

If you're managing your jails with iocage, remember that iocage puts the prefix **ioc-** before every jail name. Perhaps the host doesn't think the jail **www1** exists, but the jail **ioc-www1** does.

If you run ps(1), jailed processes show up with the J flag. I'm inside a jail here.

```
$ ps
  PID TT  STAT    TIME COMMAND
23615  0  IJ   0:00.01 login [pam] (login)
23616  0  SJ   0:00.01 -csh (csh)
24808  0  R+J  0:00.00 ps
```

Look under STAT. All of the processes have the J flag. The jail can't see anything that's not in its own jail. If I checked processes on the host, PIDs 23615, 23616, and 24808 would show up exactly the same.

Network Visibility

Networking commands like netstat(1) are tightly tied to the kernel's data structures. If the version of FreeBSD running inside the jail is older than the version running on the host, they probably won't work. You must use netstat from the host.

The sockstat(1) command has a `-j` flag, to let you specify a jail to observe. You must run it from the host.

```
# sockstat -j www1 -6
```

Always check a command's man page to see if it has a jail-specific flag that will help in troubleshooting. Now let's head towards kernel-level customization of individual jails.

Chapter 8: Jail Features And Controls

A jail's **`root`** account has complete control of the jail's userland. She can add accounts, install packages, configure login classes and servers, and more. Yes, the host administrator can block out a bunch of these functions through tricks like mounting filesystems read-only, but for the most part the jail owner controls her jail.

The host administrator can also permit jail owners greater kernel-level access to their jails. This includes functions like allowing the jail owner to change their jail's hostname, allocate kernel memory for System V IPC, mounting filesystems, and enforcing chflags(1). All this requires that the jail owner interacts with the kernel more authoritatively, and the host administrator must deliberately enable all of these features.

Many advanced jail features have two levels of access control. First, the feature must be enabled on the host. If the host administrator wants the jail owner to be able to mount filesystems, he must toggle "jails can mount filesystems" on the host. This doesn't actually grant any individual jail the power to mount filesystems, however. The host administrator needs to set that permission on individual jails. While not all features have this dual-layer access control, many do. When you set a permission on a jail and it doesn't work, check to see if there's a host-level control for that feature.

In this chapter we'll look at memory management, securelevels, and filesystems as examples of per-jail features, but the underlying access control concepts will be used throughout the rest of this book.

System V IPC

System V IPC, or *sysvipc*, is a way a program or set of programs can set aside a chunk of memory and use it as a shared resource. Programs can send each other messages, leave records, manage queues of objects, scribble “Hey Bill, look at this!” and so on. Think of sysvipc as a shared whiteboard where different processes can communicate with one another.

While sysvipc was designed with access controls, those access controls considerably predate Unix virtualization, let alone jails. For many years, all the jails on a host shared one common sysvipc namespace. This meant you couldn’t securely isolate programs like databases within jails. All supported versions of FreeBSD offer sysvipc namespace translations, though, so you can now safely use sysvipc in jails.

Sysvipc is broken up into three components: message primitives (sysvmsg), semaphore primitives (sysvsem), and shared memory primitives (sysvshm). Each of these primitives can be set to new, inherit, or disable.

New, or 1, means that the host provisions a private sysvipc namespace for that feature, totally isolated from the other jails. *Inherit*, or 2, tells the host to permit access to the host’s sysvipc memory space. Using inherit is generally inadvisable, unless you’re deliberately using sysvipc for communication between jails. Finally, *disable*, or 0, tells the host to disable that type of sysvipc in the jail.

I recommend treating all of these primitives as a unit and setting them all to the same value. Yes, it’s entirely possible that a program might need only one or two primitives. Even if this particular release of DoofusSQL only needs shared memory and runs fine without the other primitives, a point release might well add another primitive and cause weird failures.

Jails use the *sysvmsg*, *sysvsem*, and *sysvshm* parameters for sysvipc. There is no host-level access control for sysvipc.

security.jail.sysvipc

There's a sysctl *security.jail.sysvipc*, and you'll see references to an *allow.sysvipc* parameter. These are obsolete, and are used to control per-jail access to the host's untransformed sysvipc namespace. Enabling it is equivalent to setting all three sysvipc parameters to "inherit."

If it's obsolete, why mention it? There's almost twenty years of documentation out there referring to it, for one thing. If a document says that you must enable it, stop and consider what you want to achieve. For many years, sysadmins had two choices for jail sysvipc: disable sysvipc entirely, or allow access to the host's sysvipc namespace. Many programs, notably databases, won't run without sysvipc, so the host administrator had to delicately judge whether to jail the service or not.

In most cases, if a document says "enable sysvipc," on modern FreeBSD you can safely set the specific jail's sysvipc parameters to "new." It's extremely rare that you would truly want "inherit," as the documents might imply.

Using sysvipc on older FreeBSD meant enabling the *security.jail.sysvipc* sysctl, and then enabling *sysvipc* on individual jails. If you find this still in use on your hosts, migrate them to the newer system.

Sysvipc and Standard Jails

The default sysvipc setting for standard jails is "disable," or 0. This was FreeBSD's default for many years, and most programs work fine with it. Set the new parameter values in `jail.conf`, using either the word or numerical values. Here we enable a new sysvipc namespace for the jail **`logdb`**, so I can run a database and aggregate my log entries.

```
logdb {
  ip4.addr="203.0.113.232";
  sysvmsg=new;
  sysvsem=new;
  sysvshm=new;
}
```

Restart the jail and you'll see sysvipc enabled therein.

```
# jls -j logdb -h name sysvmsg sysvsem sysvshm |column -t
name   sysvmsg  sysvsem  sysvshm
logdb  1        1        1
```

We can now run a database in this jail.

Sysvipc and Iocage

Iocage has the same *sysvmsg*, *sysvsem*, and *sysvshm* parameters. The default is "new," creating a new sysvipc memory space for every jail. It uses a scrap more memory than disabling sysvipc, but programs Just Work. To change these parameters, use the word values rather than the numbers. Here, for a variety of daft reasons including but not limited to managerial fiat, I'm letting this jail access the host's sysvipc namespace.

```
# iocage set sysvmsg=inherit sysvsem=inherit \
   sysvshm=inherit wdb2
```

The jail **wdb2** is now ready to leak information into any other jail set to inherit sysvipc namespace.[17]

Securelevels and chflags(1)

While you can change a jail's securelevel, many securelevel protections and access controls simply don't apply to a jail. Jails don't get access to disk devices by default, so disallowing the ability to edit those devices is irrelevant. In the unlikely disaster of a jail panicking the kernel, the

17 But I'm sure nothing will go wrong. Ignore those resumes on the printer.

host handles all the debugging functions. While we'll discuss jails using their own packet filter in Chapter 9, many sysadmins perform all packet filtering at the host level. Jails can't change the system clock.

The only securelevel control that has practical impact in a jail is file flag enforcement, so we'll focus on chflags(1). This means that the only securelevels that really matter are -1 and 1. (Securelevel 2 matters if you're running a jailed firewall.)

Standard jails default to securelevel -1, while iocage jails default to 2.

Setting Jail Securelevel

Jails can run at any securelevel equal to or higher than the host's. A jail cannot have a securelevel lower than the host's. The securelevel can be increased when creating the jail, or from within the jail.

The parameter *securelevel* controls the jail's securelevel at creation time. Standard jails default to the host's securelevel, -1 by default, while iocage jails default to securelevel 2. Adjust this in *`jail.conf`* or with `iocage set`, exactly as you would any other parameter.

A jail owner can increase the jail's securelevel through the *kern.securelevel* sysctl, or by setting a new securelevel in the jail's *`/etc/rc.conf`*.

This complicates checking a jail's securelevel. An `iocage get securelevel` or checking *`jail.conf`* will give you the securelevel the jail was created with, but the jail owner might have changed the jail's securelevel. Use jls to get a jail's actual securelevel.

```
# jls -h name securelevel
```

Securelevel is a parameter where I always query the current state of the jail, not the way the jail was created.

File Flags and Securelevel

While most file flags don't interact with securelevel, securelevel directly controls schg and sappnd. At securelevel 1 or higher, these flags prevent removing or altering files they're applied to.

If a host is running at securelevel 0 or -1, but the jail is running at securelevel 1 or greater, the jail owner can't remove these flags or muck with files marked with these flags—but the host administrator can. If you have an overly enthusiastic but inexperienced jail administrator, this can result in trouble tickets like "I made my home directory immutable, send help!" You can also use it as part of your security enforcement, though. If you want to keep jail owners from replacing system binaries in the jail, flag them all as immutable and crank up the jail's securelevel. You must run `chflags -R noschg` on those critical directories before upgrading the jail, but for certain environments (and certain users) it's worth the trouble.

Jail owners cannot use chflags(1) by default, as that leads to those aforementioned support tickets. If you want a jail owner to be able to set file flags, you must set the *allow.chflags* parameter on the jail.

Mounting Filesystems

Filesystems are dangerous. Corrupt filesystems can panic the host. Jails can't mount filesystems by default. If you grant a jail owner permission to mount a type of filesystem, they can mount any filesystem of that type. It's truly best for the host administrator to mount any needed filesystems for the jail, preferably through an *`fstab`* file (Chapter 4) or the automounter.

But sometimes you can't help it. Perhaps the jail exists specifically to examine and process filesystems, or maybe it needs a filesystem mounted at irregular intervals but specifically not mounted at other times. Reality is often inconvenient that way. You can grant a jail permission to mount specific types of filesystem.

Remember, not all filesystems can be managed from within a jail. Identify jail-safe filesystems by running lsvfs(8) and looking for the "jail" flag. Some common filesystems like FAT, cd9660, NFS, and even UFS cannot be safely managed from within a jail. They are tightly integrated into the virtual memory system in such a way that they cannot be easily contained. ZFS was written for virtualizing operating systems, so making it jail-safe was fairly straightforward. You can manage synthetic filesystems like devfs, fdescfs, and tmpfs, as well as FUSE and (sadly) procfs from within a jail. You can also manage Linux synthetic filesystems like linprocfs and linsysfs, but not EXT-based filesystems.

While an unsafe filesystem cannot be managed from within a jail, it can provide data storage to a jail. Mount and manage such filesystems from the host.

If you still want to manage filesystems from within a jail, read on.

Mounting Prerequisites

The *enforce_statfs* parameter (Chapter 4) controls mount point visibility. The default setting, 2, means that mount points other than the jail's root directory are invisible as mount points. The jail owner can't even see mount points that they know exist (like */dev*) as mount points, but only as directories. This blocks any chance of mounting anything anywhere other than */*. To allow a jail owner to view other mount points, and thus to mount anything that can be perceived from within the jail, set *enforce_statfs* to 1. (You could also set it to 0, but that removes all mount point visibility restrictions and is almost certainly a terrible idea.)

Invisible mount points can cause problems. The jail owner can think that their jail has a great deal of disk space, but maybe the host administrator has mounted a separate partition as the jail's */home* and that partition is almost full. The jail owner can't see that. Moving open files between partitions won't happen. I'm not saying don't configure your jail with multiple invisible partitions, but I am saying you should let the jail owner know about it.

Now that the jail can see mount points, let's talk permissions. Jails have a generic mount parameter, *allow.mount*. It defaults to zero, disallowing all mount(8) commands from within the jail. Setting it to 1 opens the possibility of mounting and unmounting filesystems.

You still need to grant permissions by filesystem type, however. Jails provide separate parameters for permissions to new mounts and unmounts of each jail-safe filesystem, in the form *allow.mount.* and the filesystem name, such as *allow.mount.zfs* and *allow.mount.devfs*. If the jail has a parameter granting permission to mount that filesystem type, the jail's **`root`** account can use the standard mount(8) commands to mount and unmount those filesystems.

Permission to mount and unmount a filesystem does not include the power to mount filesystems mounted by jail(8) when the jail was created. Consider */dev* and devfs. Certain software configurations, such as chroots, might well require an additional devfs in, say, the jail's */var/named*. The jail's **`root`** can mount and unmount that devfs instance. The jail's proper */dev*, however, needed to exist before the jail was started. Its mount is controlled by the host, not the jail. A jail owner can't unmount the jail's */dev*, or the */dev/fd* created by *mount.fdescfs*, and so on.

Let's discuss each of the jail-manageable filesystems in turn.

fdescfs

The *mount.fdescfs* parameter controls whether a jail should mount fdescfs at startup. Iocage mounts fdescfs by default. I normally set *mount.fdescfs* as a default in `jail.conf` for standard jails as well, because it costs almost nothing and its presence reduces the number of trouble calls I get.

Why would a jail owner need to manage additional fdescfs instances? Linux emulation requires an additional fdescfs within the Linux chroot. Set *allow.mount.fdescfs* in the jail configuration to let the jail owner manage their own fdescfs.

devfs

The jail's main device filesystem, `/dev`, exists outside the jail's process namespace and cannot be managed from within. The host must make all changes to a jail's `/dev`. The *mount.devfs* parameter controls whether the host creates `/dev` when starting the jail, and enabling it is the universal default. I am not aware of any jailable applications that don't require `/dev`.

If a jail needs additional devfs instances, set *allow.mount.devfs* in the jail configuration.

Jails cannot run devd(8), so the jail does not have access to dynamic devices or devfs rules. New devfs instances are an exact copy of the jail's `/dev`. If you need a different set of devfs rules in a jail, configure it from the host with *mount.fstab*.

nullfs

Loopback-style mounts with nullfs(5) work perfectly fine inside a jail. The jail can only null-mount directories it can see. Enable the jail owner to perform them with *allow.mount.nullfs*. There is no *mount.nullfs* parameter; configure null mounts at jail creation with an `fstab` file.

procfs, linprocfs, and linsysfs

All three of these synthetic filesystems work with jails.

The *mount.procfs* parameter tells jail(8) to mount */proc* when starting the jail. This mount remains in place for the life of the jail. The jail owner only needs *allow.mount.procfs* if she intends to mount additional procfs instances, or if she needs to mount and unmount `/proc`. Use of the process filesystem is heavily discouraged, and I encourage unmounting it when not in use.

FreeBSD's Linux mode requires linprocfs and linsysfs. The jail owner must have the *allow.mount.linprocfs* and *allow.mount.linsysfs* parameters set to use Linux software inside a jail. You can use Linux compatibility within a jail, and the hard-core can run Linux *as* a jail (Chapter 10).

tmpfs

Set the *allow.mount.tmpfs* parameter to allow a jail owner to create and destroy memory filesystems. The older style of memory disks, md(4), cannot be managed from within a jail. Even if you could get the `/dev/md0` device, the UFS filesystem traditionally used on them can't be managed within a jail. Use tmpfs(5).

If you grant this permission, a jail can exhaust the host memory. You'll need to restrict the amount of memory the jail can access, as discussed in Chapter 11.

ZFS In A Jail

ZFS is more jail-aware than any other filesystem. You can delegate administration of a ZFS dataset and all its children to a jail. You can allow a jail owner to create new datasets in their jail, adjust those datasets, and assign new datasets to a jail. The *jailed* property indicates that a dataset belongs to a jail.

A jail owner with the ability to make arbitrary changes to a ZFS dataset can make changes that make the dataset incompatible with the host. If they create a new dataset with a mountpoint of */var/log*, she obviously wants that dataset to be used as */var/log* within the jail. If the host mounted that dataset, it would mount over the host's */var/log*. The amount of mayhem this would inflict is not worth the amusement. ZFS datasets with the *jailed* property set cannot be mounted by the host. You can certainly unset this, but check the dataset's mount point first.

Don't delegate the jail's primary dataset to the jail. Managing a jail from the host means that the host must access the jail's base system and its packages. You could temporarily free a dataset from the jail, reconfigure all the mount points, perform maintenance, and restore the jail owner's configuration, but that's a tricky process that's likely to go wrong. Instead, provide the jail owner a private dataset that they can use and abuse to their heart's content. If the dataset needs to go in a critical system location like */home*, perform the initial setup before handing control of the jail over to the user.

Configuring a Dataset

On the host, create the dataset that you want to delegate to the jail. Copy any necessary files to that dataset. Remember, once the dataset is delegated to a jail you won't be able to access the contents from the host without removing the dataset from the jail and carefully checking the dataset's compatibility with your host.

That's really it. Everything else happens in the jail.

Configuring a ZFS-Delegated Standard Jail

To manage ZFS, the jail must have access to `/dev/zfs`. Don't allow all your jails access to `/dev/zfs`; while they won't have permission to manage datasets, you don't need jail owners nosing about. Create a special devfs ruleset specifically for ZFS management and assign it only to jails that need it.

For a jail to manage ZFS, the jail must have the *allow.mount.zfs* parameter. It also needs the generic *allow.mount* parameter and must have *enforce_statfs* set to 1 or lower, exactly as when managing any other filesystem. Additionally, use *exec.start* to run `zfs mount -a`, telling the jail to mount all the ZFS datasets it can see.

Here I'm jailing the dataset `cdr/cdr`, a top-level dataset in a pool dedicated to retaining call detail records.[18] The jail **`cdrpit`** relies on global defaults for boring details like path and IP addresses.

```
cdrpit {
  allow.mount=true;
  allow.mount.zfs=true;
  enforce_statfs=1;
  devfs_ruleset=5;
  exec.created+="zfs set jailed=on cdr";
  exec.created+="zfs jail cdrpit cdr";
  exec.start="zfs mount -a";
  exec.start+="/bin/sh /etc/rc";
}
```

The magic pixie dust of ZFS delegation is entirely in the two *exec.created* parameters. The first, `zfs set jailed=on cdr`, sets the *jailed* property on the target ZFS dataset. Theoretically you need only do this once, but in case the host administrator mucked with the dataset we'll verify it as the jail starts. The second command,

18 This dataset and jail are named in honor of anyone sentenced to maintaining decades of telco call detail records that nobody ever looks at, until they're suddenly the most vital data in the company. I am delighted beyond words to no longer be you.

`zfs jail cdrpit cdr`, tells ZFS to assign the dataset *`cdr`* to the jail **`cdrpit`**. This is another command that should need to be done only once, but double-checking might prevent trouble.

The jail's startup script might require data in the jailed dataset. Before we can start the jail, therefore, we must mount all ZFS datasets. This means redefining the default *exec.start*, so that the jails mounts all its ZFS datasets before running its startup script.

Inside the Jail

Log into the jail and examine the available datasets.

```
# zfs list
NAME   USED  AVAIL  REFER  MOUNTPOINT
cdr    376K   899G    88K  /cdr
```

The jail's ZFS knows about our delegated dataset, and expects to manage it. If you run mount, you'll see the jail's other filesystems. If you've delegated a ZFS dataset to the jail, chances are that the jail's root directory is also ZFS. The jail recognizes the root filesystem as a filesystem, but not as ZFS. The jail does not have access to the root filesystem's ZFS characteristics, as that dataset hasn't been delegated to the jail.

As the jail owner, the delegated dataset is yours. You cannot adjust quotas on delegated datasets but it's otherwise yours.

Configuring a ZFS-Delegated iocage Jail

The jail **`cdrpit2`** handles a different set of call detail records, on the dataset *`cdr2/cdr`*.

Set the *jail_zfs* parameter to "on" to enable ZFS management inside a jail, then set the *jail_zfs_dataset* parameter to list a dataset to be delegated to the jail. Give the dataset name without the pool name. Here I create the jail **`cdrpit2`**, and assign it the top-level *`cdr`* dataset.

```
# iocage create -n cdrpit2 ip4_addr="198.51.100.228" \
   -r 12.0-RELEASE jail_zfs=on jail_zfs_dataset=cdr
```

You can't set the delegated dataset's mount point from within iocage, though. You must use ZFS commands. Log into the jail and look at your datasets.

```
# zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
iocage        14.7G   884G  4.04M  /iocage
iocage/cdr      88K   884G    88K  none
```

The delegate dataset shows up under the *iocage* pool, even though it's really a separate pool. Even this amount of detail is concealed from the jail owner. Change its mountpoint as desired.

```
# zfs set mountpoint=/cdr iocage/cdr
```

That's it. The jail now owns that dataset.

Minor Features

Jails have a bunch of small features that don't really fit into any other section, from setting the hostname to special memory and buffer access.

Change Hostname

A jail owner might want to change the jail's hostname using the standard in-jail methods like hostname(1) or */etc/rc.conf*. The *allow.set_hostname* parameter (*allow_set_hostname* in iocage) permits the jail owner to change the jail's hostname, and it defaults to on.

Changed jail hostnames can confuse the host administrator. The jls command defaults to showing jail hostnames, rather than the jail name. Remember that the hostname is not the same as the jail name. You set the jail name at the beginning of the jail definition, and it goes in the *name* parameter. The jail's hostname is the *host.hostname* parameter.

If hostname consistency is important to you or your organization, set the *allow.noset_hostname* parameter in `jail.conf`; in iocage, set *allow_set_hostname* to 0.

Dmesg

Jails don't get to view the host's dmesg(8) buffer or the contents of the *kern.msgbuf* sysctl. The kernel's messages are hidden from jails. If you want a jail to be able to view the host's message buffer, set *allow.read_msgbuf* (*allow_read_msgbuf* in iocage).

Once you enable message buffer access, any user in the jail can access it. To restrict message buffer visibility to only **`root`** on both the host and allowed jails, set the sysctl *security.bsd.unprivileged_read_msgbuf* to 0.

No combination of settings permits dmesg(8) access to all users on the host but only **`root`** on a jail.

Lock Physical Memory

When a program requests memory, the kernel hands it a chunk of memory addresses. The program has no way to know if this memory lies on the memory chips or if it's swapped out to disk; it's memory, it stores stuff for the program, enough said.

Data like encryption keys, however, should never be swapped. Programs that handle such confidential data need to request memory specifically on memory chips, or to *lock physical memory*. Jails default to not allowing programs to lock physical memory, but you can permit locking physical memory with the *allow.mlock* parameter (*allow_mlock* in iocage).

If the sysctl *security.bsd.unprivileged_mlock* is set to 0, only **`root`** can lock physical memory on both jails and the host.

Now that you can fine-tune your jail exactly as you like, let's interconnect them in strange and terrible ways.

Chapter 9: Networking

Up until now, we've delegated all the networking to the host. You can attach networking configuration to a jail rather than the host, granting the jail more independence and making it possible to more easily move jails from host to host. You can also use virtual networking features to attach jails to different parts of the network, have one jail firewall another, and even build a cluster of jails that can network with each other but not the outside world.

This chapter assumes you understand networking. If anything in here confuses you, go read a basic networking primer and come back. My *Networking for System Administrators* (Tilted Windmill Press, 2015) would suffice, although there's a fair number of competitors.

Jails can get much more complicated than what I show here. I discuss bridging with vnet; you could choose to route instead. You could use setfib(1), although vnet has largely obsoleted it in a jail context. You could build a jailed load balancer and distribute traffic between jails. I won't show you everything. Once you understand these examples, though, you'll be able to implement as many different structures as your networking knowledge permits.

Before diving into that, though, let's consider the host.

Interface Names and Jails

Rather than configuring a jail to use an IP address already attached to a network interface, you can have a jail attach its IP to a selected interface before starting. This keeps you from having to constantly edit the host's */etc/rc.conf* as you add and remove addresses, but it adds minor snags.

FreeBSD names network interfaces after the network card driver. Normally, the only time you need the interface name is when you're troubleshooting or performing basic system configuration—that is, infrequently. The interface name mostly gets used in */etc/rc.conf*, so who cares what it's called?

Once you start using an interface name in userland configurations, though, the interface name really quickly becomes important. Changing the network card from gigabit to ten gigabit doesn't mean only a search-and-replace in *rc.conf*; you must update all those userland entries as well. Jails with multiple network interfaces amplify the problem.

Fortunately, FreeBSD has the ability to rename network interfaces. Before you start assigning network interfaces in jail configurations, assign the network interface a name like "jail." When you change the network card, you can assign the new interface the same name and leave all your jail configurations alone.

I'm dedicating this host's `em0` network interface to jails, so in this *rc.conf* snippet I rename the interface and bring it up.

```
ifconfig_em0_name="jail"
ifconfig_jail="203.0.113.50/24"
```

All of my jail configuration can now use the interface name *jail* and it will all be correct.

In cases where I want to assign multiple interfaces to jails, I'll name interfaces after their role. Suppose I have one for the jails' public Internet, one for the jails to access my private network, and one for host management. I'll assign the public interface a name like *jpublic*, the private one *jprivate*, and—as long as I'm here—the host management interface *mgmt*.

I'll use these interface names for all of the networking examples from here on out.

TCP/IP and Jails

Jails get a limited view of the network, depending on the addresses assigned to it. A jail can open standard TCP/IP sockets, allowing you to run typical servers. If your jail has access to `/dev/bpf` and you fire up a packet sniffer, you only receive broadcast traffic and traffic addressed to the jail's IP address. You can control exactly how these network restrictions work, though.

Managing Network Access

Jails default to only accessing the network addresses made available to them, but that's not the only option. The *ip4* and *ip6* parameters control access to the kernel's TCP/IP stack. Both work identically.

The default setting, *new*, tells the kernel to permit the jail access to only the assigned IP addresses. If a jail requests a specific IP address, the host attempts to add that address as a new alias. All traffic sourced from the jail uses the first IP assigned as the source IP.

Set these to *inherit* and the jail can access all of the host's IP addresses. This is usually undesirable, but unusual is not uncommon as we'll see in Chapter 11.

Finally, to prohibit a jail from using an IP family, set this value to *disable*. If I want to block a jail from using IPv4 but allow access to the host's entire collection of IPv6 addresses, I would use this.

```
ip4=disable;
ip6=inherit;
```

Even a simple `ping localhost` will now get a “Protocol not supported” error. I might use this to see if an application *really* works on pure IPv6.

Note that if you disable an IP protocol, but also assign an address in that protocol family, the protocol is re-enabled. The jail assumes that if you assigned an address, you must want the address to function.

Even if I permit a jail control of an IP address, it still can’t control the whole network stack unless I let it.

Multiple Interfaces and Addresses

Large production hosts often have multiple interfaces, or at least multiple VLANs, each dedicated to a particular purpose. If you want jail(8) to add the jail’s IP addresses to a specific interface at jail startup, and remove that address at jail shutdown, list the interface before the IP address in the IP address parameter like so.

```
loghost {
   ip4.addr="jpublic|203.0.113.231";
   ip6.addr="jpublic|2001:db8::1234";
…
```

The advantage to storing jail interface information with the IP address is that the jail configuration can easily be moved to an entirely different host.

Iocage works exactly the same way, except it uses the *ip4_addr* and *ip6_addr* parameters.

```
# iocage set ip4_addr="jpublic|203.0.113.234" www1
Property: ip4_addr has been updated to
   jpublic|203.0.113.234
# iocage set ip6_addr="jpublic|2001:db8::2" www1
ip6_addr: none -> jpublic|2001:db8::2
```

These jails will now attach themselves on startup to whatever network interface happens to be called `jpublic` at the moment.

Jail addresses are IP aliases on the network interface. This means they are assigned a /32 or /128 netmask, but inherit a practical netmask from the interface's primary address. That primary address is probably attached to the host, which is fine for most applications. If the interface doesn't already have an address, specify the netmask with the address assignment.

```
ip4.addr=”jpublic|203.0.113.225/24”;
```

A jail can have multiple IP addresses. If a requested address isn't already on the host, the host will automatically add each as an alias on the main network interface. List the addresses in *ip4.addr* or *ip6.addr*, separated by commas.

```
ip6.addr=” jpublic|2001:db8:0::225/64, jpublic|2001:db8:0::226/64”;
```

Once you involve multiple interfaces, though, the *jail.conf* syntax changes. You must use multiple *ip4.addr* or *ip6.addr* statements and the += syntax. Here I assign a jail two addresses on the `jpublic` interface and another on the `jprivate` interface.

```
loghost {
  ip6.addr=”jpublic|2001:db8:0::225/64”;
  ip6.addr+=”jprivate|2001:db8::226/64”;
  ip6.addr+=”jprivate|2001:db8:1::225/64”;
  …
```

You cannot combine multiple IP addresses and interface assignments in *jail.conf*. Once you start including interface information with addresses you must list each address on its own line, even when multiple addresses get assigned to one interface.

You can also give iocage jails addresses on multiple interfaces. With iocage, you can combine these assignments in a single line.

```
# iocage set ip4_addr="jpublic|203.0.113.234,\
    jprivate|198.51.100.234" www1
```

You can list however many interfaces and addresses you like in a jail, but remember that the jail will attach any IP not marked with an interface to what it thinks is the host's main interface. I encourage you to explicitly label interfaces to prevent iocage from changing its mind and choosing a different interface.

Multiple addresses and interfaces force the jail to make decisions about which source address to use for outbound connections. When a system can initiate traffic from multiple addresses or multiple interfaces, it must choose which source address to put on the outgoing connection. If a jail has fifteen IP addresses, and you SSH out of the jail, where does the jail say the connection is coming from?

The rule of thumb is that an outgoing connection uses the first address on the interface that the traffic leaves the host from as its source address. If the interface `jprivate` has a primary IP address of 198.51.100.234, and the system routes traffic out that interface, the connection comes from the IP 198.51.100.234.

Jails slightly change source address selection. The first address assigned to a jail is the jail's primary address. If the jail can't find a more appropriate address to use for an outgoing connection, it uses that. It's a meaningless distinction most of the time, but has unusual effects on communication with the host and other jails on the host.

Multiple jails can share a single IP, if and only if that's the only IP assigned to both jails. You cannot have one jail with multiple addresses, and a second jail that shares one of those addresses. You can't have multiple shared addresses on multiple jails. Jails can share one and only one address, and they'll bicker like children over who gets to bind to which TCP/IP ports.

You can run jails on a host with only one network-facing IP address. If you explicitly assign a jail the host's only IP address, both the jail and the host will lose network connectivity.[19] You can solve this by allowing the jails to inherit the host's IP stack, or by binding jails to the loopback address.

If you set a jail's *ip4* and *ip6* parameters to "inherit," the jail can access every IP address on the host. You'll need to run the jail's external-facing services like SSH on alternate ports, or manage the jails locally through jexec. We discuss this model in Chapter 11.

You could attach private IP addresses to the loopback interface `lo0` and bind the jails to those addresses. You'll need a NAT on the host's packet filter to give those jails network access and to make desired services accessible to the outside world.

Ping, Traceroute, and Raw Sockets

A jail cannot access raw TCP/IP sockets. These are more flexible than the standard "cooked" sockets, and allow a host to hand-craft packets. A knowledgeable user could use raw sockets to twiddle various network stack internals. The problem is, network tools like ping and traceroute require raw socket access.

While ping and traceroute are the go-to tools for troubleshooting network connectivity, there are alternatives for testing network functionality. My usual command is host(1), especially when the jail's nameserver is not on the local network. If you can run a command like `host mwl.io` and get an answer, chances are your jail has network connectivity. If your nameserver is on the local LAN, try using another network-aware command like fetch(1). Running

19 Everything will continue to run, mind you. Only without any network connectivity.

`fetch https://mwl.io` will generate an OpenSSL error on a new FreeBSD install, as FreeBSD doesn't ship with a trusted certificate authority bundle—but the mere appearance of that error means that your jail interacted with a remote host. The network works. (You could run `fetch -o - --no-verify-peer https://mwl.io` to ignore the TLS error, but that's poor practice so don't.)

If you trust the jail owner and the applications running in that jail, you can permit raw IP socket access with the *allow.raw_sockets* parameter (*allow_raw_sockets* for iocage). Setting this parameter and restarting the jail will allow ping and traceroute to work.

```
# iocage set allow_raw_sockets=1 www1
```

Generally speaking, a jail's network restrictions only impact IPv4, IPv6, local sockets, and routing. You might want to jail applications that include TCP/IP components beyond those, or even have non-TCP/IP components. Such applications need privileges to create general IP sockets. The *allow.socket_af* parameter (*allow_socket_af* in iocage) lets a jail create network sockets beyond the jailed protocols. By enabling this, you're allowing the jail to twiddle arbitrary network characteristics. If you need to enable this for an untrusted application or jail owner, you might consider switching this jail to full virtualization.

Jails and the Loopback Interface

All TCP/IP communication between jails and the host, and between jails on the same host, happens on the loopback interface lo0. (Private jails that are attached to an interface like `lo1` communicate with each other on that interface, but any communication with the host still traverses `lo0`.) If you're on a jail and you SSH to the host, the connection never hits the external network. This is standard behavior for most operating systems, but it has consequences for jails.

An IP address can attach to more than one jail if and only if that's the only IP all the jails have. Most jails need an external address so they can access the network, so they can't also attach to the loopback addresses 127.0.0.1 and ::1. Many software packages expect **`localhost`** to be available, including a bunch of stuff in a standard FreeBSD install. Jails don't have a loopback address by default, so they use their default address: the first IP attached to the jail. Try it yourself; run `tcpdump -i lo0 ip` on the host, then go to a jail and run `ping 127.0.0.1`.

This satisfies the software's requirements. It also means that when a jailed daemon attaches to a port on the "loopback address," it's actually attaching to the jail's external IP address. If you're counting on a daemon binding to only the loopback address as a security measure, your assumption is invalid. That daemon listening on "localhost" is actually listening on the network-facing address.

The obvious thing to do is to create a loopback interface just for jails, or perhaps even give each jail its own loopback, and assign each jail a unique private IP. While it isolates those addresses from the network it doesn't isolate the jails from one another. Loopback addresses can reach each other. The underlying problem is that every jail uses the host's network stack. The host's network stack knows dang well how to reach all of these addresses. TCP/IP on the local host is designed to communicate, not isolate.

The only way to truly separate jails from one another is by assigning each jail a virtual network stack, or vnet.

Virtual Networking Stacks: VNET

A Unix system traditionally has a single network stack. It might support multiple routing tables, letting different applications send traffic to different networks, but for the most part all of the networking stack is a monolithic whole. Every program can see every interface on the host.

FreeBSD can create virtual network stacks, or *vnets*. A vnet completely controls any interfaces assigned to that vnet. You can create virtual interfaces for jails, plug them together as needed, and create your own network entirely inside your machine.

You'll read complaints about how vnets are buggy. They first appeared in FreeBSD 8-CURRENT and had problems, like any other big new complicated software. Vnets have undergone extensive testing for over a decade and were only enabled by default in FreeBSD 12. If you see a message complaining about vnets, check the date. It's not that vnets are perfect—bugs can happen anywhere. But they are expected to work, are in wide use, and bug reports are taken as seriously as any other FreeBSD bug.

Placing each jail inside its own vnet gives the jail a completely isolated network stack. Each jail gets its own loopback interface, permitting local networking without leaking information to other jails. You can delegate packet filter management to the jail, allowing your jail owners to lock themselves out of their jails rather than doing it at the host level. Vnet isolation even protects against debuggers like DTrace and packet sniffers.

Vnets are not perfect. Jails running a version of FreeBSD older than the host's might find that programs like netstat(1) and sockstat(1) don't work properly due to differing kernel structures. Some network-aware programs require access to `/dev/mem` or `/dev/kmem`, which should *never* be exposed to a jail. (The jail should get a "permission denied" message when trying to use them, but still; don't expose them.) Overall, though, vnets are a vast improvement to inheriting a narrow slice of the host's network stack.

A jail has complete control over its vnet. The jail gets raw socket access. Parameters like *allow.raw_sockets* and *allow.socket_af* are effectively on and cannot be turned off.

Throughout this section we'll assume that your jails run a FreeBSD version fairly similar to the host. The greater the difference between the jail and the host the more difficulty you'll have, as discussed in Chapter 10.

Virtual Network Design

Don't slap the *vnet* parameter on all your jails and restart them. You'll totally disrupt connectivity. Start by designing your virtual network with interfaces and bridges.

Interfaces are conduits that connect a host to something else. FreeBSD has a couple of different sorts of virtual interfaces usable for jails. We're going to focus on epair(4), which is the virtual equivalent of two Ethernet interfaces connected by a network cable. Epair interfaces have an A and a B end. For consistency, always attach the A end to the bridge and the B end to the jail. (Or decide that "B is for bridge" and do it the other way. Whatever you choose, be consistent!) Each epair interface has a name starting with e, an interface number, and which end this interface is, so either `e0a` or `e0b`. It then gets a trailing label, describing the epair's function. For jails, use the jail name. This means the interface `e0a_loghost` is the first epair interface on the jail **`loghost`**, and is attached to the bridge.

Bridges connect interfaces together. A bridge performs the same function as the standard Ethernet switch. Networking purists should note that FreeBSD's bridge(4) virtual bridge filters traffic exactly like a switch. All the interfaces connected to a bridge can communicate with each other, but can't snoop each others' traffic.

The trick to this virtual network is that not all the interfaces need be virtual. I can plug all my jails into a virtual bridge so they can see each other, and then add in my host's physical external interface. Suddenly, all my virtual interfaces plugged into the virtual bridge can communicate with the non-virtual network beyond my host.

You might want your jails to communicate only with each other. Don't connect a real interface to the bridge, and the jails will remain isolated from the world.

Maybe your jails need to connect to multiple networks. Give each jail multiple interfaces and connect to multiple bridges. Add one of the host's VLANs to the bridge and the bridge can see that VLAN.

If your jail network is even slightly more complicated than "all my jails bridge to the outside world via the host's jail interface," stop. Get out a piece of paper. Draw a map. Label all the network segments. Draw arrows on which components lead to which segments. Use colors.[20] This might seem childish, but if you can't draw a picture you can't assemble the virtual network. If you are attempting to replicate an existing environment—say, if you're building a testbed that should mirror your organization's production environment—get the diagram of that environment and use it as a base.

20 I always enjoy asking the company supply officer to order me new crayons.

Our First Virtual Network

For our initial tests we're using a very simple network. We have a single physical interface, `jailether`. All our jails will connect to a bridge that contains that interface, and through that to the local network. We'll consider this with both standard jails and iocage.

Either way, pick a physical interface to dedicate to the jail. Remove all IP networking configuration from the interface; only bring it up and give it a name.

```
ifconfig_em1_name="jailether"
ifconfig_jailether="up"
```

You can now configure vnet jails against that interface.

Standard VNET Jails

It's entirely possible to manually create an epair interface, attach one end to a jail and the other to a bridge, attach the bridge to an interface, and bring up the jail. Nobody has time for all that, though, especially when FreeBSD includes scripts to do that for you.

The directory */usr/share/examples/jails/* includes a whole bunch of scripts and documents for jail administrators. Some of it is only applicable to older FreeBSD releases, while others aren't relevant for what we're doing. Copy the */usr/share/examples/jail/jib* script and put it wherever you put your system scripts. I use */usr/local/scripts/jib* in these examples. "Jib" stands for Jail-Interface-Bridge, and automates epair interface creation and destruction on any virtual network with four or fewer bridges.

Now configure our first vnet jail. Here's the beginning of a vnet-friendly *jail.conf*.

```
$j="/jail";
path="$j/$name";
host.hostname="$name.mwl.io";
exec.clean;
exec.start="sh /etc/rc";
exec.stop="sh /etc/rc.shutdown";
mount.devfs;
exec.prestart="logger trying to start jail $name...";
exec.poststart="logger jail $name has started";
exec.prestop="logger shutting down jail $name";
exec.poststop="logger jail $name has shut down";
exec.consolelog="/var/tmp/$name";
```

These entries start with the basic setup needed for any jail. We define a path to place the jail's files, set a hostname, define the startup and shutdown commands, and put a devfs in the jail. The *exec* parameters that log when jail(8) starts and stops jails aren't mandatory, but I'll use them to illustrate some features.

Now let's set up a vnet jail.

```
loghost {
  vnet;
  vnet.interface="e0b_loghost";
  exec.prestart+="/usr/local/scripts/jib addm loghost jailether";
  exec.poststop+="/usr/local/scripts/jib destroy loghost";
}
```

The *vnet* parameter tells jail(8) to assign this jail its own virtual network stack.

The *vnet.interface* parameter tells jail(8) which interface to plug into this jail. We're using epair interfaces, so I call the interface `e0b_loghost`.

The jail's personal *exec.prestart* parameter calls `jib` to add the interface. Note that I set this parameter with the `+=` syntax. I already defined a global default *exec.prestart* at the top of `jail.conf`. This new entry gets added to the default.

The `jib addm` command creates the jail's epair interface and adds one end to a bridge. If the bridge doesn't exist, `jib` creates it. I've given two arguments here, `loghost jailether`. This translates to: create an epair interface labeled `loghost`, and attach one end to the bridge containing the interface `jailether`.

The jail's *exec.poststop* calls `jib destroy`, which removes unnecessary epair interfaces. It needs one argument, the epair interface to destroy.

Now start the jail.

The Jail and VNET

Log into the jail and run `ifconfig`. You'll see two interfaces, `lo0` and `e0b_loghost`. You have a network!

```
root@loghost:~ # ping mwl.io
ping: cannot resolve mwl.io: Host name lookup failure
```

A closer look will show that the network has no IP configuration. A non-vnet jail piggybacks off the host's network, but a vnet jail has no access to the host's network. Go into the jail's */etc/rc.conf* and configure networking exactly as you would on any other FreeBSD host.

```
ifconfig_e0b_loghost="203.0.113.231/24"
defaultrouter="203.0.113.1"
```

Reboot the jail, and you'll be on the network!

The private vnet permits greater flexibility in jail configuration without sacrificing security. If you permit access to */dev/bpf*, you can both sniff packets and use DHCP to assign jail addresses. If the host has a firewall kernel module loaded and you expose the firewall devices to the jail, you can run PF inside the jail.

The Host and VNET

Before you log into your new vnet jail, take a look at the host. You'll notice two new interfaces, `jailetherbridge` and `e0a_loghost`. Let's look at the bridge first. A bunch of what's in the bridge is only of interest to network folks, but some bits are illuminating.

```
jailetherbridge: flags=8843<UP,BROADCAST,RUNNING,
    SIMPLEX,MULTICAST> metric 0 mtu 1500
  ether 02:07:c5:a6:aa:00
…
  member: e0a_loghost flags=143<LEARNING,DISCOVER,
   AUTOEDGE,AUTOPTP>
     ifmaxaddr 0 port 5 priority 128 path cost 2000
  member: jailether flags=143<LEARNING,DISCOVER,
   AUTOEDGE,AUTOPTP>
     ifmaxaddr 0 port 2 priority 128 path cost 20000
…
```

This bridge is named after the first physical interface added to it. The interface was named `jailether`, so this is `jailetherbridge`.

Each bridge member appears on the following list. The first member is the A end of the jail's epair interface, `e0a_loghost`. The second member is the physical interface, `jailether`. This bridge will gain a member with every vnet jail added to the bridge. If you see an interface for a jail that doesn't belong on this bridge, you botched the *jail.conf* entry.

One thing you won't see is the `e0b_loghost` interface, the end that's attached to the jail. The `ifconfig` command only shows interfaces attached to the current network stack. You can't see the interface in the jail any more than you can run `ifconfig` on your desktop and see the server's interfaces. Go log into the jail.

iocage VNET Jails

The iocage program can manage bridges and interfaces for you. It defaults to using a very simple network, a single bridge that accesses the network through the host's main interface. ("Main interface" being defined as "whatever interface can hit the default route." We'll discuss changing that, but for now go with it.)

Create a vnet jail by setting *vnet* to on. You must also set an IP address, including the netmask. If you want the jail to reach networks beyond the local LAN, also define the *defaultrouter* parameter. Then you'll need a name and a release, just like any other iocage jail.

```
# iocage create -n www1 ip4_addr="198.51.100.234/24" \
   defaultrouter=198.51.100.1 vnet=on -r 13.0-CURRENT
```

When the jail is created, take a look at your interfaces. You'll discover a newly added `bridge0`, containing the host's main interface and half of the jail's epair. The iocage program names the new epair interface `vnet` with a number. The interface description gives the jail name.

Shutting down all the jails does not destroy the bridge. People might configure services like DHCP or packet filters or NetFlow[21] on that bridge, so it needs to remain in place. If it offends you, remove it manually with `ifconfig bridge0 destroy`—but it'll reappear as soon as you start another vnet jail.

That's really it. Unless we want to get complicated. Which we do.

Alternate Interfaces

I don't want my jails using the host's management interface. That's why I have this `jailether` interface, so jails can spew their garbage on the network and not interfere with my pristine host. The parameter *vnet_default_interface* specifies a system interface to add to the bridge instead of the host's main interface. Not only do I want to use the interface `jailether`, I want all iocage jails to use that interface. This means I need to change iocage's default setting for this parameter.

```
# iocage set vnet_default_interface=jailether default
```

Shut down all of your iocage vnet jails. Remove the existing `bridge0` interface. Restart your iocage vnet jails, and iocage will create a new bridge with the proper interface.

21 No, I'm honestly not shilling my book *Network Flow Analysis* (No Starch Press, 2010) here. That would require subtlety, which I lack.

Complicated Networks

What if you want a whole bunch of bridges, with jails plugged into each willy-nilly? Again, make a network diagram. Name your bridges. Show where you want every jail to plug in. You can then use iocage's interface mapping parameter, *interfaces*, to assign jail interfaces to bridges. The *interfaces* parameter always takes the format *vnet:bridge*. The iocage program creates the vnet interfaces, but it assumes that if you're mapping interfaces to specific bridges you're taking responsibility for those bridges. Here I map the vnet interface on jail **www1** to connect to the bridge `jailetherbridge`.

```
# iocage set interfaces="vnet0:jailetherbridge" www1
```

If you have multiple interfaces, separate them with commas.

```
# iocage set \
   interfaces="vnet0:jailetherbridge,vnet1:jailprivbridge" \
   www2
```

Assign IP addresses to each interface as well.

```
# iocage set ip4_addr=\
   "vnet0|198.51.100.235/24,vnet1|192.0.2.235/24" \
   www2
```

Why would a jail have multiple interfaces? I'm glad you asked!

Multiple Bridges

Having a single bridge with a single interface attached to the outside world works fine for most applications, but what if you're trying to simulate a more complicated setup? Perhaps you want to test an application behind a firewall or one of those multi-tier web server/database setups with layers of load balancers and packet filters and who knows what? Maybe you can't include that proprietary multimillion-dollar[22] firewall in a jail, but you can emulate the basic architecture and work

22 but basically worthless

out the obvious bogusness before deploying your application to the staging environment.

We're going to add a second layer to our virtual network. The existing bridge, `jailetherbridge`, will remain in place. We're adding another bridge, `jailprivbridge`, that uses the IP range 192.0.2.0/24. The database servers will only connect to the private bridge, while the application servers will connect to both the Ethernet-facing and private bridges.

You know how to attach the database hosts to a bridge; merely change the `jailetherbridge` entries to `jailprivbridge`. The only trick of this configuration is understanding how to create the bridges, and then how to attach multiple interfaces to a jail.

Standard Jails with Multiple Interfaces

The `jib` command requires an existing interface to build a bridge on, so we'll create a loopback interface on the host. I'll call this interface `jailpriv`. (I would spell out `jailprivate`, but the resulting bridge name is too long.) Create this interface at boot-time with these *rc.conf* entries.

```
cloned_interfaces="lo1"
ifconfig_lo1_name="jailpriv"
```

Upon reboot, you can attach the private bridge to this interface.

Now configure a jail with multiple interfaces. The jail **loghost** will connect to both the public and private networks.

```
loghost {
   vnet;
   vnet.interface=e0b_loghost, e1b_loghost;
   exec.prestart+="/usr/local/scripts/jib addm loghost \
     jailether jailpriv";
   exec.poststop+="/usr/local/scripts/jib destroy loghost";
   allow.raw_sockets;
}
```

The *vnet.interface* parameter has two interface names, separated by a comma. Note that the interfaces are not quoted. Quoting causes them to be treated as a single unit, which is fine when you only have one interface but is counter-productive when you want multiple entries.

The tricky part is the *exec.prestart* `jib` command. The `addm loghost` part basically means "add the loghost interfaces to these bridges." It then lists the bridges to add each of the interfaces to. Interfaces are added to bridges in order—that is, the first interface is attached to the first bridge, the second interface to the second bridge, and so on.

The first interface shown in *vnet.interface*, `e0b_loghost`, gets glued to the first bridge shown at the end of the *exec.prestart* `jib addm loghost` command, `jailether`. The second interface shown in *vnet.interface*, `e1b_loghost`, is connected to the second bridge shown, `jailpriv`. The `jib` script lets you list up to four interfaces and bridges.

If you make a mistake configuring the jail and wind up with freshly created interfaces but no running jail, running `jib destroy loghost` will remove all of that jail's interfaces.

Boot up the jail and configure IP addresses on all the interfaces. You should be able to ping across the private and Ethernet-facing bridge. Hosts on the private bridge have no connectivity to the outside world—unless you configure one of the dual-interface hosts as a packet filtering firewall, that is.

iocage With Multiple Interfaces

You must create the `jailprivbridge` and `jailetherbridge` interfaces in *rc.conf* before trying to use them. Here I create the interfaces bridge0, bridge1, and lo1, give them the desired names, and add interfaces to the bridges.

```
ifconfig_em1_name="jailether"
ifconfig_jailether="up"
cloned_interfaces="bridge0 bridge1 lo1"
ifconfig_bridge0_name="jailetherbridge"
ifconfig_bridge1_name="jailprivbridge"
ifconfig_lo1_name="jailpriv"
ifconfig_jailetherbridge="addm jailether up"
ifconfig_jailprivbridge="addm jailpriv up"
```

We can now use these bridges.

Use iocage's *interfaces* property to assign virtual interfaces to bridges, in comma-separated pairs. Here I attach the virtual interfaces `vnet0` to the bridge `jailetherbridge`, and `vnet1` to `jailprivbridge`.

```
# iocage set interfaces=\
    "vnet0:jailetherbridge,vnet1:jailprivbridge" www1
```

Configuring multiple IP addresses is rather similar, except the IP address parameters use a pipe rather than a colon to pair up interfaces and addresses.

```
# iocage set ip4_addr=\
    "vnet0|203.0.113.234/24,vnet1|192.0.2.234/24" www1
```

You can now start **www1** and ping across both the private and public bridges. (Remember, vnet jails control their own sockets and need no special permissions for ping.)

You'll find a minor gotcha when moving an iocage jail to a fully private bridge. That bridge has no external access, but iocage won't start a jail without a legitimate-looking default route. The key here is legitimate-*looking*. Set the *defaultrouter* parameter to an unused IP on the jail's subnet and iocage will start the jail right up.

Manual Interface Plumbing

Scripts and commands that hide all tedious command lines are nice, but sometimes you have no choice but to dig in and set up the network manually. Perhaps you need a virtual crossover cable between two jails, or you're attempting to simulate that rat's nest of switches and cables your network administrator jokingly calls a production network. The `ifconfig` command lets you create logical interfaces and shift interfaces between vnets. As an example, I'm creating a single `epair` interface and attaching one end to the host **`firewall`** and another to the bridge `failover`. This is exactly what iocage and jib do when they create interfaces.

Creating a virtual interface, like a bridge or an epair, merely requires declaring it exists. The `ifconfig` command returns the device name and number.

```
# ifconfig epair create
epair0a
```

While ifconfig only tells you about the one end of the epair, it creates both ends. If you want a specific device number, declare it when creating the interface. If I want to specifically use device 9:

```
# ifconfig epair9 create
```

Assign your epair interfaces meaningful names.

```
# ifconfig epair0a name e0a_firewall
e0a_firewall
# ifconfig epair0b name e0b_firewall
e0b_firewall
```

Now we must decide where to put each of these interfaces. For all of our previous examples, one end of the epair was attached to a bridge. The other was attached to a jail. As a general rule, I put the interface's `a` end as close to the host as possible.[23] Here I attach `e0a_firewall` to the bridge `failover`.

```
# ifconfig failover addm e0a_firewall
```

The e0b_firewall interface at the other end of the epair needs to be moved into the vnet jail **`firewall`**. Each vnet is named after its jail. (Before you ask: yes, the host's networking stack also runs as a vnet, called vnet 0.) Assign the interface's vnet with `ifconfig`.

```
# ifconfig e0b_firewall vnet firewall
```

The interface will disappear from the host's network stack and reappear inside the jail.

You can move any interface between vnets. Suppose you're running your web server inside a jail, but it has high enough traffic that it needs a dedicated interface.

```
# ifconfig em0 vnet www1
```

Your jail now has a private physical interface.

Other Networking Features

Jails have a variety of other networking features. You can get a jail's IP address from DNS, leverage PF to host multiple jails on one IP, and have iocage configure a jail's resolver.

23 This echoes an interface IP addressing rule from my ISP career: "we are low, and we are odd." Both technically and socially accurate, and handy for the kids that don't yet understand subnets.

Get Jail Address from DNS

You already have an authoritative list of hostnames and addresses in your DNS server, so why would you possibly want to maintain a second list in `jail.conf`? In my case, it's because I make up all my jail names and don't bother putting them into DNS. Taking jail IP addresses from DNS makes perfect sense for many environments, though.

The *ip_hostname* parameter tells jail(8) to look up the IPv4 and IPv6 addresses of the value of the *host.hostname* parameter, and assign the first addresses returned as the jail's IPv4 and IPv6 addresses.

A jail's name parameter is never a default, but if you set *ip_hostname* as a default you can get your jail definitions down to something like this.

```
www4 {}
```

This feature is available in iocage as the *ip_hostname* parameter.

The *ip_hostname* parameter adds DNS as a startup dependency. If you jail your DNS servers, verify that your DNS servers can bootstrap themselves even in a completely cold startup. How will your network restart after a power outage that outlasts your UPS and diesel generators? I recommend avoiding *ip_hostname* on jails that provide DNS service. Chicken-and-egg bootstrap problems amuse everyone who doesn't have to solve them.

iocage and the Resolver

The iocage program defaults to copying the host's `/etc/resolv.conf` to each jail. This isn't a bad default, but you might want a different configuration. If your host has a whole bunch of jails, you might add one more to serve as a recursive DNS server, or perhaps you want to direct a specific jail to a certain DNS server.

The *resolver* parameter lets you configure the jail's *resolv.conf* on the command line. Semicolons become newlines. Suppose I want the jail **www1** to use this *resolv.conf*.

```
search mwl.io
nameserver 198.51.100.3
nameserver 2001:db8::bad:code:cafe
```

I'd set the resolver property like so.

```
# iocage set resolver="search mwl.io;nameserver \
   198.51.100.3;nameserver 2001:db8::bad:code:cafe;" \
   www1
```

To revert to the default, set *resolver* to `/etc/resolv.conf` or `none`.

Resolver configuration is a prime candidate for setting as a global default, using the **default** jail.

At this point you should be able to configure the network any way you need. Let's see about using jails to handle the weird stuff.

Chapter 10: Extreme Jails

One reason I like jails so much is that they let you cope with weird problems. We've all encountered those ancient, vital systems that nobody dares touch, let alone upgrade, and yet the hardware is so feeble that you know they're going to implode any time. We have customers and applications that need to run their own jails, but don't need an entire host. Or you might even want to run a Linux install inside a jail.

We'll start with jails that can run jails.

Hierarchical Jails

Jails can run their own jails, with their host administrator's permission. These *hierarchical jails* let you establish further layers of access control. Hierarchical jails let you delegate jail management down from the host. A jail that runs a jail is called a *parent jail*, while jails run inside a jail are *child jails*. It's possible for children to run their own child jails, making them both parents and children.[24] A parent jail can restart its children, which empowers users. Both standard jails and iocage can create a top-level parent jail, but you can't yet use iocage within a jail.

24 When using hierarchical jails some folks call the parent jail a "host" and the FreeBSD install running on hardware the "base host," but I absolutely refuse to torment the word "base" any further.

The *children.max* parameter (*children_max* in iocage) sets the maximum number of jails that a jail can run. This defaults to zero, blocking jails from spawning jails. Parent jails can delegate a number of their allowed children to a child jail, allowing children to have their own children.

If a jail is a collection of altered namespaces, hierarchical jails offer child jails subsets of those altered namespaces. For example, a child jail's process namespace is a subset of the parent jail's process namespace. A parent jail can see all of the processes in its namespace and all of its child's processes, where the child can only see its own processes. The parent jail's process namespace is a subset of the host's process namespace, so the host can see all the processes. A parent jail is restricted to a portion of the filesystem, so child jails are limited to a subset of that portion. If a parent jail only gets partial access to kernel functions, the child jail gets a subset of that.

This means that a parent jail cannot grant a child jail access rights the parent doesn't have. If a parent jail is not allowed to mount filesystems, the child jail cannot either. A jail can't assign its children access to device nodes that it doesn't have.

Hierarchical Jail Networking

Before configuring a parent jail, think about the network configuration. Some of the constraints that seem minor when managing jails from a host get really inconvenient when a parent jail tries to corral its children.

A common desire is to assign each parent several IP addresses, so that it can allocate those addresses to its children. Each IP can be assigned to one and only one jail, though. The parent jail is a jail. You *can* have a parent jail delegate addresses to its children, if the parent jail has its own vnet. Remember, IP addresses belong to the vnet. If the parent jail controls the vnet, this restriction doesn't apply.

A parent jail without vnet cannot control its IP address stack. It can tell its children to inherit its IP settings, which works fine but limits its jail utility. You can configure the host firewall to redirect incoming connections to particular addresses and ports bound to the loopback interface. This is all ugly and clunky, though.

I recommend always configuring parent jails in their own vnet.

Creating a Parent Jail

Creating a parent jail requires setting *children.max* to the maximum number of child jails this parent should have. A parent jail must be able to mount filesystems for its children—if nothing else, each child needs a device filesystem. We'll use the parent jails **dba1** and **dba2** in our examples, each in its own vnet. The jail **dba1** is used for MariaDB testing, while **dba2** is for Postgres.

Here's **dba1** in *jail.conf*. It can support five child jails, and gets a virtual interface attached to a bridge as discussed in Chapter 9. I'm inheriting parameters such as *path*, *host.hostname*, and so on from the defaults at the top of *jail.conf*.

```
dba1 {
   vnet;
   vnet.interface=e0b_dba1;
   exec.prestart+="/usr/local/scripts/jib addm dba1 jailether";
   exec.poststop+="/usr/local/scripts/jib destroy dba1";
   allow.mount;
   allow.mount.devfs;
   enforce_statfs=1;
   children.max=5;
}
```

Creating a parent jail with iocage is very similar to creating any other jail. It will have a much longer address entry than most, and you must set the *children_max* parameter. Here I create the jail **dba2**, with the IP addresses 198.51.100.230, running FreeBSD 12.0, and let it start up to five children.

```
# iocage create -n dba2 -r 12.0-RELEASE vnet=on \
   interfaces="vnet0:jailetherbridge" \
   ip4_addr="vnet0|198.51.100.230/24" \
   defaultrouter=198.51.100.1 children_max=5
```

While you can set all the necessary parameters when creating the jail, the command line is already too long for my comfort. Here I give **dba2** permission to mount a */dev* in its child jails.

```
# iocage set allow_mount=1 dba2
# iocage set allow_mount_devfs=1 dba2
# iocage set enforce_statfs=1 dba2
```

You must use standard jails to manage children.

Creating Child Jails

I'll set up the child jails much like I do jails on the host, with all configuration in */etc/jail.conf* and all child jail data files in the jail's */jail*. Here's how I configure */etc/jail.conf* on **dba1** for a whole bunch of MariaDB jails on **dba1**.

```
$j="/jail";
path="$j/$name";
host.hostname="$name.mwl.io";
exec.clean;
exec.start="sh /etc/rc";
exec.stop="sh /etc/rc.shutdown";
```

All of the above comes straight from the host's *jail.conf*.

```
mount.devfs;
devfs_ruleset="0";
```

The jails must have a devfs. Jails don't have access to the host's devd, though, and have no ability to create their own device filesystems. Their */dev* is a copy of the parent jail's. Attempting to apply a ruleset is an error, though, so by using *devfs_ruleset* to specify ruleset 0 we tell jail(8) to not bother attempting to apply a ruleset.

```
ip_hostname;
mariadb1 {}
mariadb2 {}
mariadb3 {}
mariadb4 {}
mariadb5 {}
```

By pulling the child jail IP addresses out of DNS, our jail definitions become a set of names that all use the default settings.

The `jail.conf` on **dba2** looks exactly like this, except with a global search-and-replace of "mariadb" with "pg."

The parent jails can now run `service jail start` and spawn their children.

Managing Children

A jail can view all of its own children, and all of its children's children. A host can view and manage every jail on the system. If I run jls on **dba2**, I see its child jails. The default jls view isn't terribly enlightening here, so I'm going to look specifically at jail names, jail IDs, and parents.

```
# jls -h name jid parent |column -t
name  jid  parent
pg1   28   0
pg2   29   0
pg3   30   0
pg4   31   0
…
```

Each jail is named as this parent jail configured it, much as you'd expect. If you've just started your jails you might expect your jail IDs to start with 1, but these higher numbers are fine. The jail's parent should show the JID of the parent jail. A parent of 0 means "this system."

The exact same command on the host shows something else.

```
# jls -h name jid parent | column -t
name           jid  parent
ioc-dba2       21   0
dba1           22   0
dba1.mariadb1  23   22
dba1.mariadb2  24   22
…
ioc-dba2.pg1   28   21
ioc-dba2.pg2   29   21
ioc-dba2.pg3   30   21
…
```

The first two jails are **ioc-dba2**, or "jail **dba2** run via iocage," and standard jail **dba1**. Their parent is 0, so they're running on the host.

Child jails get names starting with their parent, a period, and the child jail name. Jail **dba1.mariadb1** is the jail **mariadb1** running on jail **dba1**. Jail **ioc-dba2.pg3** is jail pg3 running on jail **ioc-dba2**. If the child jails have their own children, these names can get quite long.

The "parent" column shows the JID of each child jail's parent.

The host has ultimate control over all its jails. As root on the host I can run `top -j ioc-dba2.pg4` and see what's going on in this jail. I can kill arbitrary processes. If a child jail runs amok I can shut it down hard. One way is to use jexec on the parent jail and shut the jail down, but if it's really a problem I'll break out raw jail(8) commands and blow the child away. The `-r` flag unceremoniously removes jails without bothering to run any of the shutdown commands.

```
# jail -r dba1.mariadb3
dba1.mariadb3: removed
```

Problem child removed. And with hierarchical jails, problem users can now own any problems with their jails.

Old FreeBSD Versions

One advantage to jails is that you can avoid upgrading old systems. Every enterprise has hosts that are old enough that the hardware's going to die any day, but the operating system and applications are so ancient that no current sysadmin dares touch it. You can pick up almost any FreeBSD 4.0 or later system and jail it on a current FreeBSD host. Some people have had success jailing FreeBSD 3 and earlier, but that requires a custom kernel with a.out support.[25]

We're going to go through jailing an ancient FreeBSD server. My first and foremost recommendation is to not do it. It's almost always better to reinstall a modern operating system release and current applications. The packages collection includes compatibility libraries back to FreeBSD 4. Try to make your application work with those before converting the old host to a jail. Sometimes installing a new OS release, the compatibility libraries, and copying the old server's `/usr/local` to the new host suffices to get those old applications running.

Sometimes it doesn't, though. The last ancient server I jailed ran FreeBSD 4, with PHP 5 and MySQL 4. Unfortunately, it ran a mission-critical application that had been written in-house and was continually modified for years and years afterwards. Nobody still employed there knew anything about the application's innards, nobody was willing to even try to rewrite it, commercial replacements ran hundreds of thousands of dollars, and the underlying hardware had started life as a discarded desktop. Being responsible for the system, I decided I would rather jail it than try to resurrect the hardware after its inevitable smoky death.

25 And desperation. *Piles* of desperation.

The goal of such a project is not to perfectly replicate the old FreeBSD install in a jail. The goal is to restore the vital application by any means necessary.

Identifying Your Jail

Before you try converting an old server to a jail, investigate the existing system. You at least want your jail to lie consistently.

Many people use some variant of uname(1) to get the operating system version. That's incorrect practice, but most of the time it's okay. The first thing I do on an unfamiliar system is run `uname -a` to see what I'm really dealing with. The problem is, uname queries the kernel. The host kernel must be equal to or newer than the jail's operating system version. If uname declares you're running FreeBSD 15, the only thing you can be sure of is that you're not running FreeBSD 16 or higher. On a hypothetical FreeBSD 4 jail, this is wildly misleading.

The kernel provides two sysctls that uname uses to provide this information. The kern.osreldate sysctl provides the kernel release date for `uname -K`, while kern.osrelease gives the FreeBSD release returned by `uname -r`. The *osreldate* parameter lets you set a jail's kern.osreldate, while *osrelease* lets you set kern.osrelease. The problem with these is that you need to know those values for the FreeBSD version you're jailing. If the host runs you can check the sysctls. If not, spend some time with search engines and dig up reasonable values.

As of FreeBSD 10, use freebsd-version(1) to print the release you're running.

Pillaging the Old Server

Take the server destined for jailing and drop it to single user mode. If it's old enough to have */proc*, unmount it. Mount some sort of removable storage such as a USB drive or even an NFS share. Tar up every file on the server, including temporary directories. If directories like

/usr/obj, */usr/src*, or */usr/ports* exist, grab them. They won't take up much space on a modern hard drive, and having the exact source used to build the host's binaries might be vital later.

Keep the original server on hand until you're absolutely, positively certain you don't need it any more.

Replicating the Old Server

Extract the old server's files in their new home on the jail host. Be sure you use the `-p` flag, to preserve permissions.

Study the jail's */etc/fstab*. A whole bunch of it is now irrelevant—you won't need separate mounts for */var* and */tmp*, but this jail might expect a special NFS mount or some such. Is that mount a convenience, or does the application truly need that? You might have to provide such mounts from the host rather than the jail. If the application doesn't need */proc*, get rid of it.

The jail's */etc/rc.conf* is where stuff gets really interesting. The host handles all networking functions, so all those settings can go away. Eliminate any daemons that provide services the host now handles. Your jailed host needs only the functions that directly support the application. For my jail, that meant MySQL and Apache.

Resist the temptation to improve the application. If the web server compiled PHP scripts on every hit rather than using PHP-FPM, *leave it alone*. Premature optimization will doom this project.

Does the original server predate devfs? If so, remove the contents of */dev*. You can't use old device nodes on a modern system.[26] Have the jail mount a devfs.

26 Technically you *can* use twenty-year-old device nodes on a modern system. It'll unpredictably damage the system, but you can do it.

Use the host to protect the new jail. Ancient packet filters and SSH daemons are probably vulnerable to intruders, so don't use them. Use the host's SSH daemon to provide user access, and then jexec into the jail. Use a packet filter to isolate the jail from absolutely everyone who doesn't specifically need access.

In theory, the jail should now run. You'll have to perform your usual sysadmin debugging to find all the edge cases that drove you to attempt this migration.

Old Release Compatibility

FreeBSD's binary compatibility with older releases is generally excellent, but programs that directly access the kernel have problems. Programs like netstat(1) and route(1) expect to read kernel memory structures, and choke when those structures differ from what it expects. I have trouble running a FreeBSD 12 sockstat(1) on a FreeBSD 13 host. And there's no way a FreeBSD 4 */bin/ps* can read process information from a modern FreeBSD kernel.

One possible solution is to copy static binaries for those programs into the jail, overwriting the original programs. The */rescue* directory includes a whole bunch of programs hard-linked to a single crunched binary, */rescue/rescue*. It's intended for repairing badly damaged systems. There's a whole bunch of useful stuff in there, from gzip to route. You can copy the host's */rescue/ps* over the jail's */bin/ps* and restore that function, so long as your custom application didn't involve parsing ps(1) output. If you need to replace multiple programs, use hard links instead of copying the crunched binary again.

The Jetpack container toolkit (https://gitlab.com/jetpack-containers/) contains jexec-static, a program to run static binaries on the host inside a jail. You might find this useful in kludging together a solution.

If FreeBSD doesn't ship with a static version of a program that you can copy into the jail, you might have to build that binary yourself. Figuring out how to do that is a heck of a lot easier than waiting for a twenty-year-old hard drive to shatter and bring down your whole company. It's not like you're doing something truly unspeakable, like running Linux in a jail.

Running Linux in a Jail

While you can run Linux in a jail I must say up front: I don't recommend it. Here be feral wombats, refurbished Tiger tanks, and Sauron. Running Linux in a jail is more complicated than running twenty-year-old FreeBSD in a jail, and requires moral compromise and tedious debugging. But people successfully run Linux software in Linux jails. People have developed working device drivers in Linux jails. Sometimes, running Linux in a jail is the least hideous way to solve a problem.[27]

You must understand how to use FreeBSD's Linux mode before you can run a Linux in a jail. If you've never used Linux mode, trying to run a Linux jail is likely to drive you to madness. I'm going to touch on a few basics here to remind those of you that haven't used it for a while.

FreeBSD's Linux compatibility system isn't really a compatibility system. The FreeBSD kernel provides a Linux-compatible application binary interface (ABI). The ABI is how programs request services from the kernel. As far as software is concerned, the ABI *is* the kernel. By providing the Linux ABI, FreeBSD can answer system calls from Linux programs. You still have to provide the userland libraries and programs, but those you can grab from any number of download

27 Sysadmin rule #14 – The only thing worse than going with your least appalling idea is to do nothing.

sites. FreeBSD's standard Linux emulator uses a Linux userland in `/compat/linux` and runs most of the programs there, but we want to take this a step further and run a pure Linux userland inside a jail.

Always remember that Linux mode is imperfect. The ABI provides most of what a Linux kernel does. Some system calls are missing, and will remain missing until someone gets sufficiently annoyed to add them. The sysctl *compat.linux.osrelease* provides a Linux kernel version that we claim to support, but really that's just a lie FreeBSD tells because certain Linux programs get upset when that string isn't available.

Not all programs work; you won't be formatting filesystems using Linux tools. You'll have to piece together the software and configurations needed to solve your specific problem. Definitely review the basics of Linux compatibility before starting on such a project; you must understand how to install packages and how to debug with tools like truss or DTrace. Modern FreeBSD should handle binary branding for you; if you need brandelf(1), it's a bug.

Before starting, enable Linux mode on the host. The initial installation process also requires the Linux synthetic filesystem linsysfs and linprocfs, as well as fdescfs and tmpfs. Load everything into the kernel before starting.

Which Linux?

A quick Internet search shows that people have successfully installed different distributions into a jail. Which should you pick? I recommend choosing the simplest and most BSD-ish distribution that will meet your needs. Avoid systemd-based distributions if possible—again, it *can* be done, but systemd is intrusive and that dependency further complicates the whole effort.

For this example I'm using Devuan (https://www.devuan.org), a systemd-free Debian variant. The userland tools are familiar to a majority of Linux administrators, and it has a reasonably numerous user base.

If you need another specific Linux distribution, or if you're attempting to jail an old Linux in the same way you'd jail an old FreeBSD, look around and you'll probably find a post by someone who's already done it. Learn from their mistakes.

Creating the Devuan Jail

Start with an empty jail. Here I create the jail **devuan** with iocage.

```
# iocage create -e -n devuan ip4_addr="203.0.113.229"
```

Linux uses startup and shutdown scripts other than FreeBSD's `/etc/rc`. Figure out what scripts your chosen Linux uses and set them in the jail's *exec.start* and *exec.stop*. Here I tell iocage to use the Devuan scripts. Set these parameters in `jail.conf` if you're using standard jails.

```
# iocage set exec_start="/etc/init.d/rc 3" devuan
# iocage set exec_stop="/etc/init.d/rc 0" devuan
```

The jail will also need a linprocfs, linsysfs, and tmpfs.

```
# iocage fstab -a devuan tmpfs /tmp tmpfs rw,mode=1777 0 0
# iocage fstab -a devuan linprocfs /proc linprocfs rw 0 0
# iocage fstab -a devuan linsysfs /sys linsysfs rw 0 0
```

Now to put some files in the jail. Use `debootstrap` to download Devuan into the jail. The debootstrap program is a Debian management tool for installing Debian to a directory, but it's available as a FreeBSD package. We can only use part of debootstrap's functionality, but it's enough to get packages on the disk.

Our debootstrap command has this format.

```
# debootstrap --foreign --arch=architecture \
   devuan-version /directory site
```

The `--foreign` flag tell debootstrap that it's running on something other than the destination platform, and that it shouldn't try to run Debian-specific commands because they aren't present. It will download and unpack programs, but stop before making device nodes and other Linuxy stuff.

The `arch` flag is the hardware architecture you're running on. This is probably `amd64`, but if you're on 32-bit hardware use `i386`.

Devuan, like Debian, comes in several versions simultaneously. You can't use Devuan release names, but you can use the `stable`, `unstable`, and `testing` labels. Stick with `stable` for your first attempt.

The directory is the destination for these files—in this case, */iocage/jails/devuan/root/*.

Finally, the site is the Internet location to download the files from. We'll use the Devuan main site, *http://deb.devuan.org/merged/*.

This gives us a final command of:

```
# debootstrap --foreign --arch=amd64 stable \
   /iocage/jails/devuan/root/ \
   http://deb.devuan.org/merged/
```

You'll get a warning that debootstrap cannot validate the package signatures because your FreeBSD machine doesn't have the Devuan keys installed, and then you'll see a bunch of packages being downloaded.

Once the packages are downloaded, go into the jail's root directory and temporarily mount the synthetic filesystems.

```
# mount -t linprocfs linprocfs proc
# mount -t linsysfs linsysfs sys
# mount -t tmpfs tmpfs tmp
```

In theory, you can run a simple chroot now and be inside the Linux jail. If you get errors involving binary types, either you haven't loaded all the Linux kernel modules or you need to fix the branding on all the binaries in the jail's */bin* and */sbin*.

```
# chroot /iocage/jails/devuan/root/ /bin/bash
I have no name!@storm:/#
```

The prompt includes the standard Debian unconfigured host complaint, and has dragged my jail server's hostname in as well. I don't know a better sign that this jail is a total mess—but it's progress.

We only did the initial Debian bootstrapping. The package database is in an inconsistent state, and while packages are extracted they're not really configured. Inside the chroot, tell dpkg to configure everything and set up the package database.

```
# dpkg --force-depends -i /var/cache/apt/archives/*.deb
```

It'll generate warnings because you're not really running on a Linux host, but you'll wind up with configured packages. You'll be prompted for a couple of choices, depending on your Devuan version. I take the defaults, because I'm going to be butchering this system and there's no reason to inflict additional damage. Some of those errors left packages in a "to be configured" state, though, so let's configure those.

```
# dpkg --configure --pending
```

At this point everything should be installed. You can check the package database for problems, though. Any package flagged with "ii" is having trouble.

```
# dpkg -l |grep -v ^ii
```

If a package is having trouble, try reinstalling it. Find the original download for that package in */var/cache/apt/archives/* and run `dpkg --force-all -i` on the file.

Once the userland looks as good as you can get it, start the jail.

```
# iocage start devuan
iocage console devuan
Linux devuan 2.6.32 x86_64 GNU/Linux

The programs included with the Devuan GNU/Linux system
are free software; the exact distribution terms for each
program are described in the individual files in
/usr/share/doc/*/copyright.

Devuan GNU/Linux comes with ABSOLUTELY NO WARRANTY, to
the extent permitted by applicable law.
root@devuan:~#
```

Congratulations, the monster is alive and lurching towards the village. Toast your successful abomination and find a way to cleanse your soul.

At this point, you're on your own. Will your Linux-only program run in a jail? The only way to find out is to install it and resolve the inevitable errors.

Helping Linux Jails

While you're on your own at this point, here's some tricks I use to help make Linux jails manageable.

You can't use Linux `ifconfig` or `route` in a Linux jail. If you want the jail in its own vnet, that's a problem. You need to use FreeBSD tools to configure FreeBSD networking. The */rescue* directory includes quite a few standard commands, including networking commands. Don't just copy the whole directory, though; there's only one binary in that directory, */rescue/rescue*. It's a crunched binary, and all the other commands are links to that binary. If you copy the directory, you'll have umpteen copies of the same crunched binary. Tar up the host's */rescue* and extract it in the jail's */rescue* to preserve those links. Perhaps replacing the Linux ifconfig and route commands with the FreeBSD versions would simplify your life, or perhaps it would ruin everything.

I've found strange differences between standard jails and iocage. Sometimes a Linux jail works fine under one or the other. I've run Linux jails with their own vnet, which works fine so long as I use an external script to create and connect the interfaces rather than relying on the build-in jail configuration.

Running a Linux jail is an exercise in patience and persistence. Good luck.

Other Jail Tricks

If you can imagine it happening in a jail, someone's tried it. People run X servers inside jails, using Xnest. People develop device drivers in jails. People ranch cattle in jails—no, wait. Not cattle. But jails can subsume almost anything, and people have successfully jailed software I wouldn't have imagined could be constrained that way.

Do some research. See what people have successfully jailed, and where they've failed. Pay attention to dates; the jail toolchain constantly improves, and what was impossible several years ago might be very doable today.

Now let's make sure your experiments don't monopolize your host.

Chapter 11: Resource Restriction and Removal

We've all seen amok programs entirely consume a host. Jailing a program doesn't solve this problem. You can do various `login.conf` tweaks to help restrict particular programs, but that doesn't constrain a jail as a whole. FreeBSD lets you restrict the resources a program, user, login class, or jail can use with rctl(8) and cpuset(8). We're going to focus on their application to jails, but you can use them on any host for any reason.

Resource limiting is not a miracle. One of a sysadmin's major responsibilities is rearranging bottlenecks to coax optimal performance out of limited resources. Resource limiting is the reverse of that: imposing bottlenecks to impose a specific distribution of limited resources. How many times have you removed a bottleneck, only to discover another bottleneck right behind it? I once fixed a database server's slow disk reads, and discovered they were the only thing keeping the database from exhausting the host's memory. Impose a limit on a jail's memory and it might page so fiercely that your disk becomes unusable. Resource limits are a source of infinite frustration or boundless sardonic amusement, depending on your personality.

Some resources shortages can be worked around easily, though.

Periodic(8) and Jails

FreeBSD runs periodic(8) once a day, once a week, and once a month, as scheduled by */etc/crontab*. This is how FreeBSD performs routine maintenance and generates those daily, weekly, and monthly status reports. Jails need that maintenance as well, but these tasks eat storage I/O. Once a host has more than a couple jails, it drags to a crawl when the scheduled jobs kick off.

The easiest thing to do is to reschedule the periodic tasks in your jails. I allow my host an hour for periodic tasks, and then stagger the jails five minutes apart. You might schedule jails attached to different storage controllers to run simultaneously.

You might find it easier to schedule jail periodic tasks from the host, rather than the jails. Disable the periodic entries in each jail. Use the host's **root** crontab and launch periodic with jexec.

You can also audit which tasks the jails perform. I don't want to see network interface output from my jails as part of the daily maintenance: jails don't have enough access to give me accurate results. The only interface reports I care about come from the host. I can disable that output in my daily status mails. Does a host handle email? If not, stop checking the sendmail queue.[28] Adjust all of these tasks in the jail's */etc/periodic.conf*.

When you experience recurring performance issues, always check to see if your jails are simultaneously running scheduled tasks. It doesn't matter if you have the biggest, baddest storage arrays and controllers known to man; enough processes beating on it will make it stagger. My forty database jails can run their dump-to-file backups

28 If a jail has a problem emailing you, the email telling you there's a problem sending email won't arrive, so why watch it? Monitor all services outside the service.

ten minutes apart and nobody will notice anything, but simultaneous database dumps interrupt service. As with periodic tasks, I often find scheduling software maintenance simplest from the host's crontab.

Some problems aren't so easy to solve.

Resource Limiting

Software cannot be trusted. Even a highly skilled developer with the best of intentions and meticulous coding practices writes programs that bursts into flames when poked wrong. They can't help it; the software they're using to write the software also cannot be trusted. One of the risks of jails is that one jail might demand an unfair share of host resources, starving other jails. FreeBSD uses RCTL and rctl(8) to limit resource consumption. While older versions of RCTL impacted system performance, it now has negligible effects on system capacity and non-limited processes.

RCTL is a kernel feature specifically for restricting resource usage. You must enable it at boot by setting *kern.racct.enable* to 1 in `/boot/loader.conf`. If RCTL is enabled, rctl should run without errors.

```
# rctl
```

It won't produce any output until you have rules.

RCTL Rules

RCTL manages resources through rules. Use rctl(8) to add, remove, and view rules. Rules all have four components, separated by colons.

```
subject type : subject : resource : action
```

The *subject type* is what sort of entity this rule applies to. RCTL supports four types of rules: *process*, *user*, *loginclass*, and *jail*. The subject type differentiates the user **phk** from the jail named **phk**.

The *subject* is where you specify the entity to be limited. Give a username, a process ID, a login class, or a jail name here.

The *resource* is the formal name of the system resource we're restricting access to.

The *action* sets the limit and declares what's to be done when the limit is hit.

Taken as a whole, this lets you declare rules such as "limit user **phk** to 50% of one CPU."

```
user:phk:pcpu:deny=50
```

Here I restrict the jail **loghost** to 10 disk read operations per second.

```
jail:loghost:readiops:throttle=10
```

Restricting the amount of memory a big application can use makes sense. In this rule I limit the jail **dba1** to 2 GB of memory.

```
jail:dba1:vmemoryuse:deny=2g
```

Rules let you impose nearly arbitrary limits. In this rule, if process 1 runs for longer than thirty seconds I send it SIGKILL. (You can only assign process rules to processes that exist, and the rules self-destruct when the process exits.)

```
process:1:wallclock:sigkill=30
```

We'll mostly focus on jail rules from here on out. Writing good rules is about understanding resources and actions.

RCTL Resources

RCTL supports over two dozen resources, but some are friendlier than others. Perhaps you know how to calculate the appropriate amount of CPU time a jail should use before it gets unceremoniously terminated, but I don't. Setting a maximum limit on how long a jail can run is similarly an edge case. Or you might be one of the folks unlucky enough to need to restrict the amount of sysvipc memory a jail can use, or limit the amount dedicated to one of the sysvipc data types. RCTL can do that, but it's certainly uncommon. We'll focus on the resources I find most useful: processor, memory, and disk throughput. Once you understand these rules, you'll be able to limit the other resources.

The *pcpu* resource lets you restrict the amount of CPU power a jail can access, measured in percentages of a processor. A limit of 100 means "no more than one processor," while 50 would mean "no more than half a processor." We'll look at another approach to CPU limits in "CPU Sets" later this chapter.

The *vmemoryuse* resource restricts the total number of bytes of physical memory (RAM) and swap the jail can use. It is possible to separately restrict the amount of physical memory and swap the jail may access through the *memoryuse* and *swapuse* resources, but it's also possible to thrash your disks that way. You're better off setting a maximum amount of memory the jail can access and letting the kernel sort out questions of allocation.

RCTL offers four resources for disk I/O: *readbps*, *writebps*, *readiops*, and *writeiops*. Each is about restricting reads versus writes, and measuring in bytes per second or I/O operations per second. Use whatever measurement annoys you least.

RCTL Actions

The five actions RCTL supports are deny, throttle, signal, log, and devctl. Not all actions make sense for all resources.

The *deny* action tells the kernel to refuse the resource request. You can have only one deny action for a rule: you can't deny a jail memory at 2 GB and again at 3 GB. The kernel cannot deny certain resource types, like the various time measurements. The clock keeps moving even if the server locks up. The kernel can't refuse requests for disk I/O—well, it theoretically *could*, but software isn't written to gracefully handle rejection, so RCTL doesn't support it.

While you can't deny disk I/O requests, you can sure slow them down. The *throttle* action lets you slow down how quickly the kernel allocates a resource, giving the program soft feedback on its limits.[29] The throttle action doesn't make sense for resources like processor time or memory—either it's there, or it's not—but throttle is perfect for disk activity.

The *signal* action sends a signal to a process, letting you SIGTERM or SIGKILL errant processes. This action is mostly useful for process rules. You can set multiple rules for a process, so long as you use different signals.

The *log* action sends a warning to syslog. Creating a rule to log when a jail is approaching an upper limit is a great way to get advance warning of impending resource shortages, provided you read the system logs. Create log rules to your heart's content.

Finally, the *devctl* action triggers a devd rule, letting you take arbitrary actions when the limit is exceeded. See rctl(8) for the devd event details.

Define the limit in the action, by setting it with an equal sign.

29 Not that most software processes such soft feedback, but it's nice to dream.

Managing Rules

Now that you've looked at the various actions, let's assemble some rules.

My jail **logdb** keeps eating all the memory it can steal, and I need to restrict it to 2 GB or less. The first two entries in my rule will be `jail:logdb:`. The resource to control the virtual memory a jail can access is *vmemoryuse*, so that's my third entry. The action is the tricky part. I don't want to log violations or send devd events: I want to flat-out refuse to allocate more memory to this jail, and let the software running inside it deal with the consequences. That's a *deny* action.

Add a RCTL rule with `rctl -a`.

```
# rctl -a jail:logdb:vmemoryuse:deny=2g
```

View all the rules by running rctl without any arguments.

```
# rctl
jail:logdb:vmemoryuse:deny=2147483648
```

Note that the limit has changed. While rctl accepts `2g` on the command line, it automatically converts that to bytes. Use the `-h` flag if you want human-readable output.

```
# rctl -h
jail:logdb:vmemoryuse:deny=2048M
```

Remove a rule with the `-r` flag and the rule to be removed. You don't need to specify the action in the rule, but if you do it must exactly match rule's action.

```
# rctl -r jail:logdb:vmemoryuse:deny=3g
rctl: failed to remove rule ‘jail:logdb:vmemoryuse:
   deny=3g’: No such process
```

We have no rule defining a 3 GB memory limit on this jail's virtual memory size. It's a 2 GB limit. Cut the action off your command line and try again.

```
# rctl -r jail:logdb:vmemoryuse
```

Store RCTL rules in */etc/rctl.rules*. Enable the rctl service to read them at boot.

```
# service rctl enable
```

You can also set them through *exec.created* parameters with each jail.

Filters

On a complicated host, RCTL rules can get very long. While a rule must have all four components, you can have rctl show and remove rules based on partial rules. These *filters* tell rctl to only show or remove the provided rule components. A string like `jail:`, `::vmemoryuse:`, or `:logdb::` is a filter. Here I tell rctl to show me only the rules for jails.

```
# rctl jail:
```

You must use enough colons so that rctl can figure out which part of the rule you've specified. Here I want to see all the rules applied to the resource *vmemoryuse*. I give the first two colons because rctl needs to know I'm not talking about the jail **vmemoryuse**, but I don't need the last colon.

```
# rctl ::vmemoryuse
```

I can match more than one field at a time. Suppose I want all of the readiops rules applied to jails.

```
# rctl jail::readiops
```

You can remove rules based on filters. I've had enough of resource limits, and have decided to let the jails battle it out for dominance.

```
# rctl -r jail:
```

All the jails now have no resource restrictions, but that sample rule limiting PID 1 to a thirty second lifespan remains in place.

RCTL and iocage

Iocage provides a parameter for each RCTL resource. It adds RCTL rules when starting the jail, and removes them when shutting down the jail. You don't have to worry about the RCTL subject type—it's always "jail." Similarly, the RCTL subject is always the jail you're setting the parameter for. You only need concern yourself with the RCTL resource and action.

Here I want to set the vmemoryuse resource to two gigabytes for the jail **`www1`**.

```
# iocage set vmemoryuse="deny=2g" www1
```

While generic RCTL rules for jails remain in place even when the jail doesn't exist, iocage adds its RCTL rules when starting the jail and removes them when the jail is shut down.

Processor Restrictions

Modern servers can have dozens of processors and far too many processor cores. FreeBSD's scheduler does a good job of assigning processes to cores, and you should let it do its job. While RCTL lets you restrict how much processor time a jail can monopolize, it might be necessary to lock down a set of processor cores for a particular jail, limit jails to running on a subset of available processors, or reserve a few cores for the host's exclusive use. FreeBSD lets you meddle with these details via cpuset(1).

The effects and impact of cpuset are intimately tied to Non-Uniform Memory Access, or NUMA. NUMA is basically about taking into account what parts of a process' data is stored where in the system. If a process last ran on CPU 19, that process' data is probably still in CPU 19's cache. Moving the process to CPU 204 means dragging all that data across the system to the new processor. FreeBSD's NUMA functionality is under active development, and while you'll see references to several NUMA policies you should leave them alone. Satisfy yourself with tying a jail to a set of processors, and let FreeBSD handle the nitty-gritty details of scheduling processes.

These CPU restrictions make sense only when the jail host is running on real hardware. Hypervisors deliberately conceal the details of their underlying hardware from their guests. A virtual server configured with 192 cores might be running on a server with only one core.

Before attaching jails to processors, you should understand the processors you have.

Host CPU Topology

During boot, FreeBSD explores the available CPUs and discovers how they're interconnected. The scheduler uses this information to efficiently assign processes to processors. When running on real, non-virtual hardware, you can get this same information and use it to sensibly assign jails to processors. The sysctl *kern.sched.topology_spec* shows what FreeBSD thinks your host has in the way of processors and cores. Here's what you might find on a small host.

```
# sysctl kern.sched.topology_spec
kern.sched.topology_spec: <groups>
  <group level="1" cache-level="0">
    <cpu count="1" mask="1">0</cpu>
  </group>
</groups>
```

Processors are divided into groups. This shows a single group, group 1. Within that group, processor 1 is core number 0. This host has one processor, with one core.

A host that might be more a more realistic jail server looks more like this.

```
kern.sched.topology_spec: <groups>
  <group level="1" cache-level="3">
    <cpu count="8" mask="ff,0,0,0">0, 1, 2, 3, 4, 5, 6,
      7</cpu>
```

This host has eight processors. They're numbered zero through seven. Go down to view the first group.

```
<group level="2" cache-level="2">
  <cpu count="2" mask="3,0,0,0">0, 1</cpu>
  <flags><flag name="THREAD">THREAD group</flag>
  <flag name="SMT">SMT group</flag></flags>
</group>
```

This group includes two processor cores, number zero and one. They're flagged SMT, or Simultaneous Multi-Threading. They're two logical cores on the same CPU. Going down the whole list of processors, you'll see that all eight processors exist as pairs. This host has four processor chips, with two processor cores per chip. When you assign a jail to run on only a single core, this information isn't terribly vital. But if you grant a jail multiple cores, it generally makes sense to put them on a single processor to take advantage of any shared cache space.

CPU Sets

A CPU set, or *cpuset*, is a list of processors that a process or set of processes may be assigned to. Every process belongs to a cpuset. Processes normally belong to cpuset 1, the root cpuset, which grants access to all processors. View the root dataset with `cpuset -g`.

```
# cpuset -g
pid -1 mask: 0, 1, 2, 3, 4, 5, 6, 7
pid -1 domain policy: first-touch mask: 0
```

The first line lists the processors that processes in this cpuset may utilize. The second line gives the NUMA policy for allocating processes.

Every jail automatically gets its own cpuset at creation, but it's a copy of the root cpuset. View the cpuset assigned to a jail by using the `-j` flag to cpuset.

```
# cpuset -g -j logdb
jail 20 mask: 0, 1, 2, 3, 4, 5, 6, 7
jail 20 domain policy: first-touch mask: 0
```

This jail can access all processors. Let's change that. Use the `-c` flag to change an existing cpuset. The `-l` flag lets you specify which processors the cpuset may access. Here I let the cpuset for jail **logdb** to permit access to only processors 4 and 5.

```
# cpuset -j logdb -cl 4-5
```

Verify it worked with `cpuset -g`.

```
# cpuset -gj logdb
jail 25 mask: 4, 5
jail 25 domain policy: first-touch mask: 0
```

This jail is restricted to processors 4 and 5.

If you have multicore processors you should try to restrict a jail to cores on the same processor, but reality has a way of making that difficult. If you must allocate processors that aren't in a tidy range, separate them with commas. Here I restrict the jail **logdb** to processors 1, 3, 6, and 7.

```
# cpuset -j logdb -cl 1,3,6-7
```

You can reduce the number of processors that a jail can run on, but you cannot add in previously forbidden processors. In this example, I could run cpuset again to trim the allowed processors down to 6 and 7, but I can't add processor 0 to the permitted list. Increasing the number of permitted processors requires restarting the jail.

Note that the kernel is not a process, and does not belong to a cpuset. If the host is being hammered by a network attack or computationally expensive storage encryption, the kernel spreads its pain out amongst the host's processors.

Configuring a cpuset looks simple at the command line, but how do we hook it into jails?

Cpuset and Standard Jails

While `jail.conf` has no parameters to define a cpuset, it's easy to trigger via *exec.created*. You don't want to use *exec.poststart*, because the processes will have already started and might have overloaded other processors. Here I want the jail **logdb** to use only processors 6 and 7.

```
logdb {
  exec.created="cpuset -j logdb -cl 6-7";
   ...
```

You can globally restrict your jails to a subset of processors by making the cpuset command a default.

```
exec.created="cpuset -j $name -cl 2-5";
```

This reserves processors 0 and 1 exclusively for the host. There's no guarantee that the host will only use processors 0 and 1—much like the kernel, the host does what it must to continue functioning. That's how every virtualization system works. But the kernel is highly likely to schedule host processes on these otherwise idle processors.

The read-only parameter *cpuset.id* gives the number of the current cpuset. If you need to access the cpuset, though, it's much easier to use `-j` and the jail name.

Cpuset and iocage

The *cpuset* property in iocage lets you directly set the processors a jail may access.

```
# iocage set cpuset=4-7 www1
```

The most common use case is to restrict all jails to a specific set of processors, leaving other processors for the host. Adjust the default settings to achieve this. If you have a jail that should get access to other processors, set that separately.

```
# iocage set cpuset=2-5 default
# iocage set cpuset=6-7 dba2
```

Presumably, processors 0 and 1 are left for the host.

Incomplete Jails

A jail is a set of files and programs existing in a set of altered namespaces. Sometimes you might not want all of those namespaces altered, or you might want to change the files. A jail might not need a complete FreeBSD userland. You might want to deliberately share the network stack, or even the root filesystem. It might sound weird, but in systems administration today's weird is tomorrow's normal.

Trimming FreeBSD

It's a perennial discussion: if you don't run an SSH server in a jail, why install one? Should the jail have a compiler? If not, remove it. All those shared libraries—discard everything that programs don't need! This harks back to the golden age of hand-crafted chrooted daemons, where an attacker would break into a service only to discover himself locked in a barren cave with only `/dev/null` for company.

It's not that this is hard. Once you've played with ldconfig a few times and have learned about `/bin/sh -x`, you can successfully flense a userland down to the barest minimum. The problem is, flensing is excruciatingly tedious and must be retested every time the FreeBSD Project releases a new security update.

You might also try building a custom userland with the various WITHOUT options. Such builds tend to be fragile, and need rebuilding every time FreeBSD has a security update, but they're certainly an option.

The FreeBSD Project has an impending solution, though: *pkgbase*, a packaged base system. It breaks the base system up into about 800 packages. The packages all have dependency information so you can install the barest possible system and be sure of getting only what you require. Originally scheduled to be integrated with FreeBSD 12, it's currently been pushed back to FreeBSD 13. It's highly usable and is used in production by quite a few people, but the FreeBSD developers are sticklers about quality. (For reference, vnet appeared in FreeBSD 8. Despite heavy use, it wasn't made a default until FreeBSD 12.)

If you want to install a minimal system, do an Internet search for "FreeBSD pkgbase" and play along. I'm not giving an example because pkgbase is not finalized; you'll have to wait for my packaging book for that. If you happen to find a bug, getting it fixed will be much easier than spending hours debugging your personally fileted userland.

Network Inheritance

You can allow a jail access to the host's network namespace by setting the *ip4* and *ip6* parameters to "inherit." This permits the jail access to every IP address on the host. This violates one of the original purposes of jails, but makes sense on hosts with limited network resources.

Most virtual server companies will be delighted to offer you an additional IPv4 address, for a merely usurious monthly fee. I don't want to pay that fee for each jail. I could bind the jail to a private address on the loopback interface and use the packet filter to NAT traffic to and from it. I could assign each jail its own VNET, bridge them all together with a cloned loopback interface, and NAT that. Your security model might require that.

If your security model doesn't demand such isolation in your jails, though, consider letting the jails inherit the host's network stack. Consider this standard jail (although you can do the same thing with iocage). I've set all of the vital parameters like the path and hostname as defaults, leaving us with.

```
www1 {
  ip4=inherit;
}
```

This gives the jail access to the two IP addresses on the jail, the public address and 127.0.0.1. The jail cannot ping, as it doesn't have access to raw sockets. It cannot sniff packets, as there is no */dev/bpf*. All its files are jailed. If anyone breaks into the jail, they'll be blocked from reaching the host.

This has downsides, of course. A misconfigured jail might try to bind to a TCP port the host expects to use. Your host should start all of its services before starting its jail, though. You must track which daemons use which ports, exactly as you would with a firewall. If you have a complicated network environment, you probably don't want the jail to inherit the whole network stack—but you probably also have additional addresses to assign to a jail, which makes the whole problem moot.

I'm not saying that you should use an inherited network stack instead of NAT-ing a jail bound to the loopback address. I'm saying you should look at your specific application and consider if setting up a loopback-bound jail is an improvement over a fully inherited network stack in this particular instance. I find an inherited stack most sensible when I'm using a stripped-down userland in the jail.

If you have a single IP and the inherited stack isn't sufficient, you really should go all the way and give each jail its own vnet. Create a loopback address and attach to a bridge. Give the bridge an IP address, then bridge all the vnets to it. Set up your firewall and proceed.

Jails as Control Groups

Sometimes the only part of the jail you really care about is the shared process namespace. You're not trying to protect files on the host, or restrict network addresses. You just want to treat a group of processes as a single entity. Maybe they need to all run together, or share resource restrictions. A jail lets you create an entity that closely resembles a Solaris control group, or *cgroup*, and manage it discretely.

Consider this jail definition for a host running a very complicated Java Application Server that includes a whole bunch of different daemons and countless processes. Again, I'm taking a bunch of configuration from the global defaults, including the network courtesy of *ip_hostname*.

```
jas1 {
  path=/;
  mount.nodevfs;
  exec.start="/bin/sh /etc/rc.java-apps.sh start";
  exec.stop="/bin/sh /etc/rc.java-apps.sh stop";
}
```

The jail can access the entire filesystem. Any operating system or software updates are immediately available to the jail. Processes in the jail can only access other processes in its namespace, but they can access local sockets to communicate with other software.

If this jail mounted a device filesystem, it would cover the host's `/dev`. The host would lose its device nodes. This is one of the rare cases where a jail doesn't need devfs.

However complicated the tangle of services is, they're a single entity. If I need to restart the application server, I shut down the jail. The kernel definitively terminates every process in the jail, gracefully or not. When I restart the jail, each program starts deterministically and in the proper order. I can limit the jail via CPU sets and RCTL.

I've given a few different examples here, but the real point is that you can leverage jails any number of ways. Namespace transformation doesn't merely create lightweight virtual machines; it creates a variety of highly-tunable lightweight virtual machines.

Jails are a many-faceted multitool. Used with care, they'll make your life easier. Used poorly, you'll lop off a finger and have to explain the mess before anyone will give you a bandage.

Chapter 12: iocage Bonuses

So far, we've used iocage as a command-line alternative to editing */etc/jail.conf*. It has a few other features that make it compelling, however. While you can emulate iocage import/export with standard jails, iocage handles them in a highly convenient way. And there's no real equivalent to iocage plugins elsewhere in the FreeBSD world, although other lightweight virtualization systems have had them for years.

Migrating Jails

The iocage program lets you trivially bundle up a jail on one host and extract it on another, retaining all its settings. While creating a tarball of a jail and copying the settings from *jail.conf* isn't onerous, we all know that someone will eventually mess it up and cause more downtime than necessary. Migrating jails from one host to another with iocage is automatable, deterministic, and reliable.

Assuming all your hosts are configured similarly, that is.

Migrations and Defaults

A jail's configuration contains only the differences between iocage's defaults and the jail's settings. If all of a host's jails need a specific setting, you can set that setting in the special jail **`default`** and it will apply to all jails.

The problems I experience migrating jails between hosts come from different defaults on each host. If one host defaults to running vnet jails, and the other uses shared networking, a migrated jail's configuration will require tweaks to work. You're better off keeping your default iocage settings synchronized on all hosts that you migrate jails between. You can always compare the default configuration file, `/iocage/defaults.json`, to identify differences.

Physical differences between hosts can also interfere with jail migration. Suppose one host uses interface `em0` for jails, while the new one uses `ixgbe0`. All those *ip4_addr* and *ip6_addr* parameters that have interface names coded into them must be touched before the migrated jail can start. That's why I recommend giving interfaces dedicated to jails a single standard name across all of your hosts. It doesn't matter what device driver underlies the interface `jailether`; a migrated jail can find it by name and use it without your clumsy fingers having to change anything.

Assuming your jails have sufficiently similar defaults, you can migrate jails transparently between hosts.

Exporting a Jail

Exporting a jail duplicates the jail in a bundle for convenient transport and installation on another iocage host.

You must shut the jail down before exporting it with the `iocage export` command. Here the database team needs more resources for the host **`dba2`** than its current host can provide, so I export the jail.

```
# iocage stop dba2
# iocage export dba2
Exporting dataset: iocage/iocage/jails/dba2
Exporting dataset: iocage/iocage/jails/dba2/root

Preparing zip file: /iocage/images/dba2_2019-02-08.zip.

Exported: /iocage/images/dba2_2019-02-08.zip
```

This takes a while, depending on the size of the jail and the speed of your storage. All exported jails end up in */iocage/images*. Exported jails are thick jails; they no longer depend on any templates or releases.

The export creates two files: one named after the jail and the current date, and the other a cryptographic hash of the first file. Send both to the target host by the best available means.[30]

Importing a Jail

On the exported jail's new host, copy the jail file and the validate that the export file was not damaged in transit. Here I generate a cryptographic hash for the file and compare it to the hash generated on the origin host.

```
# sha256 -r dba2_2019-02-08.zip
# cat dba2_2019-02-08.sha256
```

Positioned one above the other, the hashes are easy to compare. Note that this check does not protect against willful malfeasance, only accidental damage. If an intruder can replace the jail's zip file in transit, they could also replace the included sha256 hash file. If you're truly paranoid, compare the hash of the jail that arrived with the hash of the file still on the original host. If the export file survives your rigorous validation, the correct jail arrived intact and you can now import it. Give the jail's name or short UUID, not the whole file name.

30 Depending on your organization, "best available means" might mean a pack mule loaded with tapes.

```
# iocage import dba2
Importing dataset: dba2
Importing dataset: dba2/root

Imported: dba2
```

The jail will now show up in `iocage list`, and can be started if the host's iocage defaults aren't too different.

Migration Cleanup

Despite what you might assume, an exported jail remains on the host after export. Exporting a jail doesn't free up any disk space on the original host until you `iocage destroy` it. Destroying the jail doesn't destroy the export file. If an exported jail is set to automatically start at system boot, it'll restart itself with the host. Be sure you remove jails you've deployed elsewhere.

Exported jails remain available for sysadmin convenience. Jails can be quite large, and moving them around the network can take a long time. You might export a database jail, restart it, send the export across the country, install it, and transmit the database afterwards before switching over. Or perhaps the export is a cold backup, and you're going to keep the original jail running. Destroying a stopped jail after migration is much easier than restoring the jail from backup after a failed migration.

And a bundled-up jail is easier to shift around the network than a standard jail tarball and a snippet of *jail.conf*.

Iocage Plugins

We install jails to run and isolate collections of software. You might have a jail that runs a backup server, a web server, a database server, Nethack, whatever. But some software goes into a jail so frequently that its setup has been fully automated. An iocage *plugin* is an automated install of a jail, complete with packages and configurations for running a service. While plugins are still new, the number of plugins is quickly expanding and their quality is improving. Plugins are not the same as Docker, but there are similarities.

Get the current plugin information with `iocage fetch`. The `-P` flag indicates that you're working on plugins, while `-R` triggers checking the public remote plugin repository.

```
# iocage list -PR
```

This displays the current plugins, and installs a copy in */iocage/.plugin_index*. Go take a look. While the *INDEX* describes the available plugins, I find it easiest to just look at the file names. Each file represents one plugin.

There's a file *plexmediaserver.json*. I've been meaning to install a Plex server to see what all the fuss is about. Let's give that a try. Run `iocage fetch -P` to grab a plugin. Give the plugin name with `-n`, and assign a valid IP for the jail.

```
# iocage fetch -P -n "plexmediaserver" \
   ip4_addr="203.0.113.222"
Plugin: Plex
  Official Plugin: True
  Using RELEASE: 11.2-RELEASE
  Using Branch: 13.0-RELEASE
  Post-install Artifact: https://github.com/freenas/io-
cage-plugin-plexmediaserver.git
  These pkgs will be installed:
    - multimedia/plexmediaserver
    - multimedia/ffmpeg
…
```

The plugin will install the jail and all necessary packages, then run a post-install script to configure the jail. At the end you'll see:

```
Starting plexmediaserver.

Admin Portal:
http://203.0.113.222:32400/web
#
```

Plex is now installed in the jail. I still have to configure it for my environment, though. No services other than that needed for the jail are enabled, so if you want SSH or sudo or somesuch you'll need to set it up.

You can use plugins outside the official repository by downloading the plugin definition file and using the plugin name defined in the file.

```
# iocage fetch -P -n /tmp/bitcoinminer.json \
   ip4_addr="192.0.2.2"
```

While plugin jails show up in `iocage list`, display only the plugin jails with `iocage list -P`.

Plugin jails are overwhelmingly maintained by the community, and as such their quality and completeness varies. They've achieved critical mass, though, and the number of available plugins is constantly increasing.

Plugins let you quickly deploy many purpose-built jails.

Having read this book, though, you can now deploy jails any way you want, for any purpose. Good luck.

Afterword

Jails are way cool. They're sufficiently cool that in 2013 I set out to write a book about them. That's when everything went horribly wrong.

Writing about jails means understanding filesystems—perhaps not to the depth of a kernel hacker, but pretty thoroughly for a sysadmin. It means understanding processes and packaging and upgrades and a whole bunch of other stuff to a degree I simply wasn't at.

I had to write the four *FreeBSD Mastery* storage books, as well as the third edition of *Absolute FreeBSD*, to prepare to write this book. I'm not saying that you have to *read* all of those to use this book, but I had to *write* them to gather the knowledge to write this book. Now that it's finished, I find myself pleased to have escaped without also being compelled to also write books on networking, packaging, and probably cat(1).

The strength of jails isn't merely administrative. It's how they provide a whole new set of tools that can help you solve real world problems. A host can support several jails for each fully virtualized host it could run, chopping your energy and space needs.

All that power is now in your brain.

Use it wisely.

Or… unwisely, I suppose. Up to you.

Sponsors

I offer people the opportunity to sponsor my books. They send me money as I'm writing the book, and I put their name in the book. Sponsors help provide financial stability as I work on a book, and have more than once bailed me out of a tight spot. Fortunately, I didn't have a water heater split open while writing this book, so I was able to spend the money on useless stuff like food and water.

These fine people made this book a whole bunch easier. Thank you all.

Print Sponsors

Theodore Durst
Giorgio Arcamone
Brian Downs
Philip Jocks
Niall Navin
Benedict Reuschling
Phi Network Systems
tanamar corporation
Allan Jude
Mischa Peters
John W. O'Brien
Sebastian Oswald
Mason Egger
Stefan Johnson
Brad Sliger
Anthony Carpenter
Shaun Addison

Russell Folk
Trix Farrar
Jake Champlin
Rogier Krieger
Trond Endrestøl
Chris Dunbar
Phillip Vuchetich
Joachim Ernst
Melissa R. Muth
Adrian Jaskuła

Patronizers

Many people back me on Patreon (https://www.patreon.com/mwlucas), and I appreciate every one of them. But only Kate Ebneter patronizes me so much that she gets her name in the *print* version of every book. Thank you, ma'am!