Chroot for fun and profit

Chroot is a method of securing an application by effectively altering the path for its execution. With that mouthful of a definition out of the way, you may be asking what it really does or why you care about it. I'll answer the former question within the text of this article. As to the latter, by using a chroot environment you can mitigate the effects of a break-in and some other breaches of security. A chroot will not stop a successful attack but it will severely limit the damage that an attacker can do with that attack. Security is a multi-layer goal. A chroot environment provides another layer of that security goal. In this article I'll define chroot and give some examples of why you'd want to bother with it. In addition I'll show you how to create a chroot environment, sometimes referred to as a jail, with a hands-on walkthrough to chroot the BIND nameserver.

Chroot Defined

Prior to digging into chroot and how to make a jail in Linux, I think it would be helpful to better define a chroot environment. If you're itching to get into chroot'ing something, skip ahead. Chroot operates in the context of a service or an application. In other words, an application such as the Apache web server can be placed into a chroot environment but the Linux kernel cannot.

The filesystem in context of an application

When an application such as the Apache web server starts it takes advantage of shared libraries located within the Linux system. Apache reads its configuration file(s) to find out what, if any modules and additional path information it should use (among syntax checking). Apache will start at the root filesystem (/) and search various locations within the filesystem such as /lib, /usr/lib, and so on for the shared libraries that it needs. Finding the correct libraries, the web server will start normally.

The filesystem in the context of chroot

The application startup just described seems straight forward, it's roughly the same from Windows to Linux at this high level. In a chroot environment, you tell chroot to change the root filesystem, from the application's point of view, to a different location thus faking the application into thinking it's really running from the true root filesystem. As far as the application is concerned, it doesn't care that its in a chroot environment and in fact it doesn't even know that its been jailed inside a chroot. An example would be most helpful here. The root filesystem is normally /. For this example, you want to create a chroot environment for Apache in the /chroot/apache directory. Therefore, you place the appropriate directories and files into the /chroot/apache directory. In essence, you will create a small version of the main root filesystem underneath the /chroot/apache directory. In other words, there will be directories such as:

/chroot/apache/etc

/chroot/apache/bin
/chroot/apache/lib
/chroot/apache/sbin
/chroot/apache/usr
/chroot/apache/var

Within that new hierarchy you'll add the files that Apache needs to run. From here you'd move (or copy?) the files that Apache needs, and only the files that Apache needs, into the new chroot hierarchy. From that point, you'll then start Apache by calling chroot. For example, the command might look like this:

chroot /chroot/apache /usr/sbin/apachectl start

From now on, Apache only knows about the files in the /chroot/apache directory hierarchy. It has no idea or ability to access files in, say, /etc or /bin. (right?)

The benefits of chroot

All this chroot stuff certainly seems like a lot of hassle for very little benefit. I assure you that chroot provides real benefits. I will admit that configuring an application to work from within a chroot can be somewhat of a hassle but surviving that initial pain can pay off in the long run. Imagine the scenario when an attacker successfully exploits a bug in the Apache web server or within a module loaded by Apache such as PHP. Depending on the severity of the exploit, that user could gain shell access to the server itself. That shell access may be in the context of the user that Apache runs as but may also be in the context of the superuser. If the attacker gains shell access, it's normally "game over." The attacker can wreak havoc on the server, steal the password file, run programs, or launch attacks on other servers within your network or on Internet. Once discovered, you're going to have to clean up after the attacker as well. Usually this means completely cleaning off the server and starting over.

The bad news is that chroot doesn't stop all of these attacks. The good news is that chroot can stop some of these attacks and mitigate others. The attacker still gains access to the server through the Apache web server running in a chroot. However, since that chroot jail starts at /chroot/apache instead of the true root filesystem, the furthest that the Apache web server (or anyone within that context) can go is to /chroot/apache. As far as the attacker is concerned, /chroot/apache is the / of that server. This means that the attacker cannot get to the server's main configuration files, such as the password file. The attacker cannot even get into the true /etc of the server but can only get into /chroot/apache/etc which will contain some skeleton files and no more. In effect, the attacker is imprisoned or jailed within /chroot/apache. Now you see why it's sometimes referred to as a chroot jail! In addition, since you will only load programs that Apache needs in order to run, the attacker will have a much more difficult time launching attacks towards other servers in your network or out onto the Internet. However, such attacks would still be possible. For example, many web sites use Perl as a programming language for CGI programs or other programs. Therefore, you'll need to load Perl and, more than likely, some associated

libraries and modules such as libnet or the DBI module for database access. If an attacker had access to Perl, he or she could write a program to attack other servers using that program from within the chroot. However, since most attacks are automated and scripted this particular risk is small. This specific risk can also be mitigated by using iptables.

A chroot jail provides no protection if the successful attacker with sufficient access (such as superuser) simply wishes to deface web sites or remove files. However, think of the difference in restore time if the attacker can only remove files below /chroot/apache instead of files below /. Even if all the files below /chroot/apache were removed, you could simply restore that directory from the last backup and the sites would be back online. However, if everything below the true root filesystem was removed (even though that wouldn't really work), you'd have to rebuild the entire server from scratch and then restore the web site data. I don't know about you, but I certainly prefer restoring one directory to rebuilding a server. I will note that there are other things that you can do to make your server less susceptible to a scripted attack and I will cover those throughout this article. In addition, skilled attacker can sometimes break out of a chroot jail. This can also be mitigated through the use of the GRSec kernel patch.

How to Chroot anything

Creating a chroot jail isn't as difficult as you might think. There are a few basic steps to creating the jail but you should be prepared for a lot of trial and error. Creating the chroot environment involves copying the files and directories necessary to effectively mirror the real environment under which the application normally runs. Like so many things in Linux there are programs and tools to help you accomplish this task, the trick is knowing that the helper programs are there!

Two programs help the most when it comes to chrooting applications. They are 'ldd' and 'strace'. The ldd command prints shared library dependencies for a given program. The closest analogy in Windows would be a program requiring a DLL. The ldd program will print each dependency which you can then copy into the chroot jail. Ldd works to catch most dependencies but other files are frequently accessed and required by a program. For example, many programs use the /dev/null special device. This needs to be created in most chroot jails. Finding these requirements can be a bit tricky but the job is made much easier with the strace command. Strace traces system calls for the given program and is invaluable in finding seemingly hidden accesses.

Along with normal Linux programs to assist in manually making a chroot environment, there is also a script called makejail to assist in automating the task for you. The script works natively with Debian Linux but should work with other Linux distributions as well. Makejail automates the process of creating directories, looking for library dependencies, and copying the files themselves. Makejail can be downloaded from http://www.floc.net/makejail/.

Using ldd and strace as chroot helpers

Using ldd to help create a chroot is straightforward. It simply examines the program or file that you feed as an argument and reports back what it finds. Run ldd with one argument of the program or file to examine:

ldd </program>

For example:

ldd /bin/ls

The output reveals five shared libraries required by the ls command:

```
libacl.so.1 => /lib/libacl.so.1 (0xb75d8000)
libtermcap.so.2 => /lib/libtermcap.so.2 (0xb75d4000)
libc.so.6 => /lib/tls/libc.so.6 (0xb749d000)
libattr.so.1 => /lib/libattr.so.1 (0xb749a000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb75eb000)
```

Armed with that information you can copy each library into the chroot jail along with the ls command. This is sufficient for many programs that you wish to move into a chroot jail:

. Create the directory structure in the jail.
. Copy the program(s) into the jail.
. Run ldd to determine library dependencies.
. Copy the libraries into the jail directory structure.

Other programs like Apache and related modules require a bit more work. Frequently this work involves using the strace command to find out why the program won't start in the chroot. Strace has many more options than ldd and there are some that are helpful for the purposes of creating a chroot. The basic syntax for strace is:

strace </program>

Running strace /usr/sbin/httpd reveals an incredible amount of information scrolling past the screen, only some of which isn't useful:

```
2471  open("/lib/libkrb5.so.3", O_RDONLY) = -1 ENOENT (No such file or
directory)
2471  open("/usr/lib/libkrb5.so.3", O_RDONLY) = 3
2471  read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\340\360"...,
512) = 512
2471  fstat64(3, {st_mode=S_IFREG|0755, st_size=385220, ...}) = 0
2471  old_mmap(NULL, 384604, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xb7453000
2471  old_mmap(0xb74af000, 8192, PROT_READ|PROT_WRITE,
```

```
MAP_PRIVATE|MAP_FIXED, 3, 0x5c000) = 0xb74af000
2471  close(3)                          = 0
2471  open("/lib/tls/libcom_err.so.3", O_RDONLY) = -1 ENOENT (No such file
or directory)
2471  open("/lib/libcom_err.so.3", O_RDONLY) = -1 ENOENT (No such file or
directory)
2471  open("/usr/lib/libcom_err.so.3", O_RDONLY) = 3
2471  read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\t\0"...,
512)  = 512
```

Strace prints its output to STDERR which means that normal shell commands like less and grep won't work. Luckily you can send the output to a file rather than the terminal using the -o <filename> option:

```
strace -o <filename> </program>
```

Continuing on with the Apache httpd example:

```
strace -o straceoutput.txt /usr/sbin/httpd
```

The output now ends with:

```
chdir("/")                              = 0
clone(child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb7131808) = 2615
exit_group(0)                           = ?
```

However, this output may not be exactly useful. Since Apache creates child processes the output from strace may not reveal the things you need. Therefore, adding the '-f' option onto strace will cause it to attempt to follow those forks.

```
strace -f -o <filename> </program>
```

The '-f' option is helpful but sometimes not quite enough. Therefore, another option is to attempt to get the program itself to stay in the foreground. Many programs offer the option to stay in the foreground as a command-line switch when starting. Looking closer at some snippets of strace output from a failed attempt at chrooting Apache might be helpful. The output was generated with the command:

```
chroot /chroot/apache /usr/bin/strace -o outputfile /usr/sbin/httpd
```

Here's the snippet:

```
execve("/usr/sbin/httpd", ["/usr/sbin/httpd"], [/* 23 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...}) = 0
```

```
brk(0)                          = 0x809a748
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/mmx/libssl.so.4", O_RDONLY) = -1 ENOENT (No such file or
directory)
open("/lib/mmx/libssl.so.4", O_RDONLY)  = -1 ENOENT (No such file or
directory)
open("/lib/libssl.so.4", O_RDONLY)      = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320\226"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=216004, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb75ea000
old_mmap(NULL, 216708, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xb75b5000
old_mmap(0xb75e7000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3,
0x32000) = 0xb75e7000
close(3)                        = 0
```

From the previous output snippets you can see that Apache attempts to open /etc/ld.so.preload and /etc/ld.so.cache but it can't find either (as evidenced by the "-1 ENOENT (No such file or directory)" error. These two particular files are not necessary in an Apache chroot. However, the next two lines show that Apache is looking for libssl.so.4 in /lib/tls/i686/mmx/ and /lib/mmx/ and cannot find it. Notice that Apache was able to find libssl.so.4 in /lib as evidenced by the fact that the status was a positive integer (3) and there is no ""-1 ENOENT (No such file or directory)" on that line. Therefore, it's safe to assume that Apache was able to find this library, even though it took it three tries to find it.

```
open("/lib/tls/libcrypto.so.4", O_RDONLY) = -1 ENOENT (No such file or
directory)
open("/lib/libcrypto.so.4", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\260\250"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=971612, ...}) = 0
```

The same situation happens with libcrypto.so.4. Apache looks first in /lib/tls/ before finding it on its second try in /lib/. These types of entries are typical but you should be on the lookout for any instances of "-1 ENOENT (No such file or directory)" where the program is not able to find the file anywhere. In these instances chances are that you should copy the file into the chroot jail. I use the phrase "chances are" because

sometimes the file isn't absolutely required for the program to run successfully. The end of this output was:

```
close(4)                        = 0
chdir("/")                      = 0
clone(child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb713d808) = 2045
exit_group(0)                   = ?
```

This output looks strangly similar to output from a successful run of Apache. However, Apache still won't start in the chroot. The problem is that Apache is forking into the background and the error is occuring after the fork. Strace's '-f' option to the rescue. Re-running the earlier command and adding -f:

```
chroot /chroot/apache /usr/bin/strace -f -o outputfile /usr/sbin/httpd
```

Looking through the strace output towards the end reveals the problem:

```
2671  exit_group(0)                 = ?
2672  --- SIGSTOP (Stopped (signal)) @ 0 (0) ---
2672  --- SIGSTOP (Stopped (signal)) @ 0 (0) ---
2672  setsid()                      = 2672
2672  close(0)                      = 0
2672  open("/dev/null", O_RDONLY)       = -1 ENOENT (No such file or
directory)
```

Looks like someone (namely me) forgot to create dev/null into the chroot. Creating /chroot/apache/dev/null using mknod (mknod is shown in detail later) causes Apache to start like a charm. Actually, I created /chroot/apache/dev/random and /chroot/apache/dev/urandom while I was there too, otherwise Apache probably still wouldn't have started. Using ldd and strace along with basic Linux commands such as grep and less and a healthy dose of solid troubleshooting methodology makes creating chroots easy. With those commands in mind, I'll now give a recipe for creating a couple chroot environments. The first chroot for ISC BIND will help to get your feet wet. It shouldn't require too much heavy lifting.

Your first chroot environment: ISC BIND

Enough background, let's look at how to create a chroot environment. We'll start with the BIND nameserver from ISC. BIND is a popular DNS server which probably still commands the lion's share of DNS on the Internet. For a long time, BIND was the sole DNS server software running on all of the DNS root servers on the Internet. This was changed because of the security considerations in having all of these highly valuable servers running the same software.

BIND presents a great opportunity to learn how to create a chroot environment. It's not incredibly difficult to make BIND work within a chroot but it shows the steps necessary to successfully create a chrooted application.

I'll assume that you have BIND working successfully in a non-chroot environment. If this is not the case then you shouldn't try to create your first chroot until you're familiar with how BIND works in the normal Linux environment. BIND normally runs as a non-root user for added security. The name of this user varies among Linux distributions but is usually either named 'bind' or 'named'. Of course, as with other things Linux, you can create your own special user for working with BIND as well.

Basic Infrastructure

The first step is to create the directories within which the jail will reside. For example, one method is to create a directory called /chroot and then create subdirectories for each chrooted application underneath /chroot, such as /chroot/bind, /chroot/apache, and so on. You need to create directories to mirror the real root filesystem as it will be seen by the application, in this case, by BIND.

```
mkdir -p /chroot/bind
mkdir /chroot/bind/bin
mkdir /chroot/bind/etc
mkdir -p /chroot/bind/usr/sbin
mkdir /chroot/bind/lib
mkdir -p /chroot/bind/var/run
mkdir /chroot/bind/dev
```

In SuSE create this directory as well:

```
mkdir /chroot/bind/var/run/named
```

Some distributions such as Debian store BIND configuration and some data files such as named.conf within /etc/bind. Therefore, this directory will need to be created within the chroot if you are using Debian. Distributions such as Red Hat stored the configuration files in /etc/named. Still other distributions store the BIND configuration file (usually named.conf) within the /etc directory itself. If your distribution calls for the BIND configuration file in a directory below /etc, create it within the chroot jail. For example, to mirror Debian's /etc/bind behavior it must be created within the chroot as well:

```
mkdir /chroot/bind/etc/bind
```

BIND can also accept an argument specifying the configuration file location so the step of creating the directory for the configuration file is not absolutely necessary. But at this point there's no need to add any confusion. Your best bet is to create mirror copies of the directory structure that the real daemon uses.

BIND data files for zones can be stored within a directory specified within the BIND configuration file therefore creation of a specific directory for this data is not necessary. For this example, I will be storing BIND data in var/cache/bind so I'll need to create it within the chroot jail: mkdir -p /chroot/bind/var/cache/bind BIND requires some special devices within the chroot and these need to be created manually. Specifically, BIND needs dev/null and dev/random. These devices are created with the mknod command:

```
mknod -m 0666 /chroot/bind/dev/null c 1 3
mknod -m 0644 /chroot/bind/dev/random c 1 8
```

Copy the existing BIND configuration files into the appropriate place in the chroot. For example, in Debian the file would be copied like this:

```
cp -p /etc/bind/* /chroot/bind/etc/bind
```

In Red Hat the files would be copied like this:

```
cp -p /etc/named/* /chroot/bind/etc/named
```

If you have existing data or zone files, make sure those get copied into the appropriate directory within the chroot. As I stated earlier, I'll be using var/cache/bind to store data and zone files which is where they are stored normally in Debian, so here goes:

```
cp -Rp /var/cache/bind/* /chroot/bind/var/cache/bind
```

It is important to note that whatever location you choose that you should specify it within the BIND named.conf configuration file:

```
options {
      directory "/var/cache/bind";
};
```

You'll want many more options than that for a normal BIND server, but the directory option is used to configure the location of the zone files. Don't forget that the path is relative to the chroot. You'll need to give the BIND user (either named or bind depending on your flavor of Linux) permission to write into the var/run directory within the chroot. Some distributions don't have a bind or named user by default. In these instances, add the user manually:

```
groupadd named
useradd -g named named
```

Then change ownership on a couple directories within the jail for that user:

```
chown -R named.named /chroot/bind
```

BIND also needs access to a time interface which is provided through /etc/localtime in most Linux flavors. Therefore, copy this file into the chroot jail:

cp /etc/localtime /chroot/bind/etc

Now copy the BIND binary into the chroot jail:

cp /usr/sbin/named /chroot/bind/usr/sbin

Logging for the new BIND

Logging within the chroot can be accomplished through two means. First, you can get a special program that enables logging from inside to outside a jail. Meaning that the logs of what's going on in the jail are stored outside the jail? However, on nearly every version of Linux you can use the second method which involves a few configuration changes to use the existing logging infrastructure. It is this second method that will be covered in this article. The syslog daemon (syslogd) is the default mechanism for logging in Linux. Normally syslogd uses the /dev/log socket for logging. However, since BIND will be in a chroot jail it won't be able to access /dev/log. To correct this you can have syslogd listen on an additional socket, in this case one inside the chroot. Syslogd must be started with an additional option, namely -a <socket>. For example, to configure syslogd to also listen on the dev/log within the chroot for BIND you would add the following to the configuration file for syslog's startup:

-a /chroot/bind/dev/log

In Debian you add this into the file /etc/init.d/sysklogd on SYSLOGD line:

```
# Options for start/restart the daemons
#   For remote UDP logging use SYSLOGD="-r"
#
SYSLOGD="-a /chroot/bind/dev/log"
```

In SuSE or Red Hat, edit /etc/sysconfig/syslog and change the SYSLOGD_PARAMS or SYSLOGD_OPTIONS line:

SYSLOGD_PARAMS="-a /chroot/bind/dev/log"

Once you've edited the syslogd startup file restart syslogd using the init.d script.

Debian:

/etc/init.d/sysklogd restart

SuSE:

/etc/init.d/syslog restart

Red Hat:

/etc/rc.d/init.d/syslog restart

Starting BIND in the chroot

It's time to start BIND within the chroot. Actually, prior to working to start BIND you should stop or kill any instances of BIND that are running. Once that's done then you can start the process of getting BIND to start from the jail.

The best way to get BIND working within the jail is to edit the init.d script for the daemon by setting the default options through sysconfig or /etc/default. If /etc/sysconfig or /etc/default are not options then you can edit the init.d script for BIND directly.

The init.d script for BIND in Debian is setup to read from /etc/default/bind. You'll most likely need to create the file /etc/default/bind since it isn't created by default. The file should contain one line:

OPTIONS="-u named -t /chroot/bind"

In Debian you'll also need to edit the init.d script directly to change the location of the PID file for BIND. Edit /etc/init.d/bind and look for instances of:

--pidfile /var/run/named.pid

Change those instances to read:

--pidfile /chroot/bind/var/run/named.pid

In Red Hat edit the file /etc/sysconfig/named and add the following line:

OPTIONS="-t /chroot/bind"

In SuSE make the following two changes to /etc/init.d/named. First, change the line:

NAMED_PID=/var/run/named/named.pid

To:

NAMED_PID=/chroot/bind/var/run/named/named.pid

Second, change the line that reads:

startproc -p $NAMED_PID $NAMED_BIN -u named

To:

startproc -p $NAMED_PID $NAMED_BIN -u named -t /chroot/bind

You can now start BIND within the chroot using the init.d script. Watch the log files for errors, usually /var/log/daemon.log or /var/log/syslog.

Special Configuration for Slave Servers

If you are running BIND version 8 (run /usr/sbin/named -v to check) and your BIND server is acting as a slave server for any zones you'll need to do a little extra work to get BIND running. This section looks at those extra settings. BIND 8 uses a program called named-xfer to transfer DNS zones from a master server. Therefore, the first concern is getting that program into the chroot jail:

cp /usr/sbin/named-xfer /chroot/bind/usr/sbin

Next you must get the libraries for that program into the jail as well. The libraries are found by using hte ldd command:

ldd /usr/sbin/named-xfer

The output will be similar to:

```
libc.so.6 => /lib/libc.so.6 (0x40018000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

For each library returned, copy it into the jail in the corresponding jail directory. For example:

cp /lib/libc.so.6 /chroot/bind/lib
cp /lib/ld-linux.so.2 /chroot/bind/lib

Things that can chroot themselves

It's somewhat of a misnomer to say that there are applications that can chroot themselves. The chroot environment created for BIND in this article uses an entirely new directory hierarchy for the entire application and related libraries thus creating a full jail environment that's virtually inescapable (with the addition of the GrSec kernel patch). However, there are some applications that can utilize a chroot for some of their functions, thus making their operation somewhat safer.

Postfix

Postfix is a popular mail program that started as a replacement for Sendmail. Many of Postfix's functions are run as separate programs. Many of these programs can be chrooted by configuring them as such in Postfix's master.cf configuration file.

MySQL

MySQL also accepts a command-line option called --chroot=path to enter the server into a chroot jail upon startup. More information on this option, which was included since version 3.23 in some form, can be found at http://www.mysql.com.

Final Chroot Thoughts

Some final thoughts on creating chroot environments:

. Think in terms of the chroot. In other words, envision the directory hierarchy as a jailed application would see it.
. Get used to prefixing things with /chroot/apache.
. Read errors carefully, they almost always give you very, very relevant clues to solve the problem. Usually they tell you what file is missing.

Conclusion

Chroot environments provide an important layer in ensuring the security of a Linux host. Using a chroot environment you can provide an extra level of safety to prevent the entire server from being compromised after a successful attack through a chroot application. You should consider chrooting as many network-facing services as possible. It is important to note that even with a chroot environment it's possible for a skilled attacker to break out of the jail and get access to the main server filesystem. Adding a kernel patch such as GrSecurity can help to prevent breakout attempts from the jail.