Global Information Assurance Certification Paper

Linux Process Containment – A practical look at chroot and User Mode Linux

By Paul Lessard

0. Abstract

Process containment has been used for quite a long time in the computing world for the use of testing beta software and increasing the security of a process. Containing a process, which is commonly known as "jailing" a process, removes a process from the full system and stops activity inside of the container from affecting anything outside the container. There are several jailing tools available, but this paper will discuss two tools available as part of all major Linux distributions: chroot, and User-mode Linux.

This document will explore some of the general ideas of how process containment is performed with chroot and User-mode Linux, and how to help ensure that a successful attack on a jailed process does not affect the main system. The benefits of each tool is contrasted, and in conclusion is shown that neither tool is best for containing all processes for all environments individually, but rather the tools can complement each other to add even more security.

1. Introduction

The idea of jailing a process was originally conceived for beta testing applications and securing processes. Jailing a process ideally places the desired process or processes in a self-sufficient environment that cannot reach the main system outside of the jail. In general this translates to not allowing the contained process to utilized resources outside of the container such as files, memory space, devices, and such. Jailing works equally well for possibly unstable process and increasing security to a system as a whole, but the focus of this paper will be on the benefit of increased security. The idea is that if a program, or even a file system, is jailed and cannot access system resources outside of the jail, then the damage that is possible if someone does break into this container is further reduced. Two tools that are capable of jailing resources are chroot and User-mode Linux. Specifically, this document will explore the basic setup of the tools, discuss how jailing can be performed, and what documentation is available for each tool. After a brief discussion on how each tool is used, a set of methods for adding security to the tool will follow. Finally, relative comparisons of each tool's benefits with respect to the other tool will be made. The ending result will show that for some applications one tool is preferential, but for other applications the second tool is more fitted.

If a company simply must run an insecure service such as ftpd, it is opening itself attacks on that service. One method to limit the damage that can be caused is to limit the resources available to the attacker when system access is obtained.

Proper jailing can completely block off access to important system files such as shadow, passwd, the main system's log files, and properties and configuration files for other services, which may contain useful data and in some cases usernames and passwords. In addition, useful commands such as su, sudo, telnet, and even ping can be completely absent in the jail even though these fully function in the main system. By limiting these resources, even when a successful attack is made the attacker, may be unable to take control of the machine or even fully cover his or her tracks.

Perhaps the reader at this point is considering the use of containment as a way of avoiding properly setting filesystem permissions and service permissions. This should not be a consideration as there is no excuse for not setting up permissions in a least privileges fashion. Rather, the method of containment gives yet another layer of protection with respect to defense in depth. Even inside of the jail, a policy of least privileges should be used, and the contained service should not be run as root where avoidable. It must always be remembered that the container itself is a piece of software that is susceptible to attacks that can allow the hacker to break free of containment. However, if the hacker is not very determined, he or she may find the system to be too difficult to take and abandon it in search of a weaker target.

One of the tools that can be used to jail a process is the chroot (Change Root) system command. This command changes the root filesystem to a target directory and runs a command. By doing this one can essentially create a barrier to the real root filesystem and therefore limit the damage that can be done to the content of the new filesystem. This particular method of containment is commonly referred to as “chroot jailing” which is available in major Linux distributions and commercial Unix systems [1]. There is quite an abundance of information on this topic, including scripts and advice on what can, should, and should not be jailed. One of the main reasons that chroot still heavily discussed is because system administrators have complete control as to what services are available within the container down to the finest detail. Chroot environments are not indestructible, but provided proper precautions are taken the “chroot jail” can be quite a formidable advisory.

The other tool that will be discussed in this document is User-mode Linux, which will be part of future Linux distributions that run on 2.5.34 kernel or greater [2]. This operates as a sort of virtual machine on top of the already running Linux kernel. Similar to chroot, User-mode Linux can be assigned resources as they are needed, and can even run it's own networking devices. Unlike chroot however, it does not have to be started as the root user, which is a significant security bonus over chroot [3].

Throughout this document, remember that both of these tools run on major distributions of Linux and are supported to some degree on most Unix based operating systems. However, for the sake of simplicity, and due to time and

space constraints, Linux RedHat 7.3 is the assumed Linux platform. If any general knowledge about differences between RedHat and other distributions is known it will be stated, but not in great detail, as the general concept should still be applicable.

2. Chroot Jailing Basics

Chroot jailing can be an incredibly tedious and painful experience. There are tools available to lessen the pain, but these generally work for a few specific services such as SSH, FTP, and Apache. In order to be able to jail any process it is still important to know the basics. Knowing the basics will give a greater flexibility when creating a jail, and give a great deal of respect for the time and effort that people have put into creating tools to make this job easier.

One of the biggest benefits of chroot jailing is that the administrator can define exactly what he/she wants to have available inside of the jail. This is also, unfortunately, one of the biggest banes to using chroot jails. The reason they are so tedious is due to fact that we tend to take for granted all of the shared libraries, configuration files, and devices that are automatically setup for us when we install Linux. When a jail is created, access to all of these resources is cut off, so these resources need to be manually included.

So how does one know what resources a service uses while it runs? The command ldd can be used to discover what libraries are need to load and launch a particular service, but what about the dynamically bound ones [4]? A simple method of dynamically bound resource discovery is to copy the desired service or command into the jail, chroot into the jail directory, and run the desired program with strace and pipe the output to a file [5]. Run the command until it crashes, completes, or until you have tested a few pieces of functionality, then check the output of strace. If you are missing a resource, the strace output will have errors indicating which files or devices were attempted to be used but failed. After identifying a missing resource, you simply add it to the jail. Sounds simple right?

Unfortunately strace resource discovery is a big reason why making chroot jails is so tedious. You may run strace once, discover a missing shared library, add it to the jail, then run it again to find that five libraries and two configuration files are missing. Then, after adding the required libraries, the new libraries require even more shared libraries and some system devices.

Very quickly you start to realize that the more complex the command or service the longer the list of needed resources will be. In addition, strace itself requires a certain set a basic libraries and devices. Luckily, there are a standard set of libraries and devices that can be added to fully support most basic commands like strace, ls, and bash. The article, “Go Directly to Jail” featured in Linux Magazine in December 2002 discusses this basic list very well. The script found

in Appendix A was based off the script found in this article. Though it is similar, it was cleaned up and separated out into commented blocks and some more generic pieces were added to it. Running it will create a jail in the current directory with the name "jail_" with a random number attached to the end. It will copy all the basic shared libraries and some simple commands such as ls, and bash.

If you do run the script, you need only run the command **chroot /path/to/my/jail/ bash** as the user root to enter the jail cell. Once inside the jail you can browse around as you would with any other filesystem. To exit the jail, simply type the command **exit**. This script was based on the authors Linux RedHat 7.3 installation so your mileage may vary when running this script on other distributions or versions.

While perusing the script you will notice that commands such as **vi**, **less**, and **more**, are not copied into the jail and are therefore not available in the jail. Depending on what service you are trying to jail, you may not need these commands. Not having editors or viewers available may be irritating to you while you configure the jail, but it would also be very irritating to a hacker as well. In order for a hacker to damage anything, he or she must have the tools available to do the damage. It maybe trivial to upload some of these tools into the jail once a hacker has broken in, but why make it easy? Remember, every second that a hacker spends floundering around trying to get the simplest commands working is extra time that you have to discover that your system has been compromised.

So a basic jail is not to hard to put together, but then a basic jail is not very useful. In order to add useful functionality to your jail you need to add more complex programs, which means that strace becomes invaluable for finding those dynamically linked resources.

Reading a strace is not always very easy, as you will often detect false positives in your strace. These false positives are often files that a service looks for in order to determine how it should be run or what options are available. For example, if you attempt to jail the daemon **sshd** after adding some of the required libraries and devices you may quickly see "No such file or directory" errors for the directory **/usr/lib/mmx**. This error could be caused by a multitude of reasons, and is most likely because your processor does not support MMX or because you do not have the MMX libraries installed. This particular error is not fatal and simply means that the service will continue without using MMX.

To discern if an error is a false positive, you generally have to use some common sense. If the error is for a resource that does not even exist outside of the jail, then it is probably nothing you need. This assumption is of course based on the program that you are trying to jail actually working outside the jail. It is generally a good idea to test the program's functionality outside of a jail before trying to jail it, or you might spend hours debugging your jail to find out that a library is

corrupt, or that you are missing a necessary resource outside of the jail.

The normal output from a strace can be lengthy to go through, but as an example, Figure 1 shows some brief snippets from running sshd with strace when all the resources were not available. The errors are fairly easy to identify, as they all follow the same format. Errors will generally return with the error ENOENT, which stands for "No such file or directory" [5]. Once you have identified a false positive you should note it, as you will be seeing it several times throughout the jailing process. All of the errors in Figure 1 are "No such file or directory" errors, but the first set towards the beginning of the strace were found to be false positives on the machine it was run on. Later on in the strace, files like /etc/host.conf show up which do exist outside the jail and therefore are most likely needed. With some additional common sense you could expect sshd to need host.conf since it is a networking application, so you just need to add this resource to your jail.

After getting the program up and stable, make sure to test as many of the application's features and options as possible to make sure nothing was missed. Just because **sshd** starts up without dying does not mean that everything is in place. Make sure you can log in using your username and password, as well as using a key. Make sure you can scp files without crashing the daemon. If you miss something, you can be assured that an attacker will find it, and try to take advantage of it.

```
towards the beginning of the strace...

open("/lib/i686/mmx/libpam.so.0", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/i686/mmx", 0xbfffed20)      = -1 ENOENT (No such file or directory)
open("/lib/i686/libpam.so.0", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/i686", 0xbfffed20)          = -1 ENOENT (No such file or directory)
open("/lib/mmx/libpam.so.0", O_RDONLY)  = -1 ENOENT (No such file or directory)
stat64("/lib/mmx", 0xbfffed20)           = -1 ENOENT (No such file or directory)
open("/lib/libpam.so.0", O_RDONLY)       = 3

several system calls later...

gettimeofday({1043819874, 821194}, NULL) = 0
getpid()                                 = 16280
open("/etc/resolv.conf", O_RDONLY)       = -1 ENOENT (No such file or directory)
uname({sys="Linux", node="webserver", ...}) = 0
open("/etc/host.conf", O_RDONLY)         = -1 ENOENT (No such file or directory)
open("/etc/hosts", O_RDONLY)             = -1 ENOENT (No such file or directory)
open("/var/nis/NIS_COLD_START", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/lib/libnss_dns.so.2", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/usr/lib/libnss_dns.so.2", O_RDONLY) = -1 ENOENT (No such file or directory)
write(2, "Could not reverse map address 19"..., 46) = 46
write(2, "debug1: PAM setting rhost to \"19"..., 46) = 46

strace continues on...
```

Figure 1 - Snippet from an strace of sshd

3. Securing a Chroot Jail

There are a few things to consider when using chroot jails. Though it can be difficult to get out of the jail, it is by no means impossible. Most known exploits

involve having root privileges, but there are ways of affecting the system outside the jail without root access if careful consideration has not been taken when constructing it.

First, when building a jail do you need all of the contents of configuration files? Just blindly adding resources to your jail as the strace shows them could be just as dangerous as not having the jailed program to begin with. These files may contain information that is not only unnecessary in the jail, but can also lead to system compromise if a hacker were to obtain the information. For instance, you may need to include the file **/etc/passwd** in your jail, which contains the list of all users on the system. Though you may not include the **/etc/shadow** file in your jail, handing a hacker the list of user id's on your system is not wise. A list of users can tell a hacker a lot about a system such as what services are running on the system, and thus possibly what exploits the system has. When including files like **/etc/passwd**, make sure you remove information that is not needed in the jail. In this case, remove all user entries that are not needed in the jail.

Specific attention should be paid to the **/etc/passwd** file because it contains user to uid mappings. If a hacker gains write access to the **/etc/passwd** file within the jail, all the user mappings could be changed to map to uid zero which would grant root access to all users within the jail next time the jail is started up [6].

Just as in a normal system administration, do not run anything as root unless the program running requires root access to function. There are several sites available on the internet that contain exploits for breaking out of jail, provided you have root access, and some of them, like BPFH.net, even include source code [7]. Though these exploits maybe patched in some versions of chroot, other vulnerabilities may very well appear. Whether or not the version of chroot is vulnerable, it is still good practice not to run any process as root unless you have to. If the hacker cannot obtain root access, the damage to your jail maybe far reduced which in turn quickens most steps of incident handling.

When considering file permissions in the jail, keep in mind that the normal permissions for a directory or file are based on dozens of applications that need to run on the system, as well as allowing for users to bring up directory listings. If the jail doesn't need a wide-open tmp directory like a full system does, then restrict the permissions so that a hacker cannot take advantage of this space. Write access should be specifically prohibited for most files, but also make sure that root owns all files [6]. Since you should be running the jailed application as a user other than root (where possible), this will prevent users from modifying files or changing the permissions on them.

In most cases you need to be root in order to run chroot, so once you are inside of the jail you should “give up” root for a less privileged id. In order to do this one of the recommended methods is to run the chroot command using a “wrapper” such as the one available at the Linux Magazine site [8]. This program takes in

parameters such as the directory you will chroot into, the user to become after entering the jail, and what command to run after becoming the user. After entering the chroot jail, it "gives up" root access and changes to the specified user provided that user exists in the jail.

Do not include any hard links out of the jail. Hard links lead directly out of the jail to the main system, which may lead to a vulnerability [6]. Also, when creating your jail keep in mind that symbolic links do not work with respect to linking outside a jail. This may not seem the case as the link will work for the system admin, but it will not work for the jailed program [6].

Finally, though this seems somewhat picky, when you actually run the jail for production use make sure you change your current directory into the jail before running the chroot jail. Some implementations of chroot do not change the current directory into the jail when the chroot is initialized, which helps a hacker utilize the root user exploits [7].

4. Chroot Tools

Now that you know the grueling details of how to put a jail together and the basics of securing one, this section will briefly cover some of the tools available to make your job easier. There are two main types of tools available for automating jail setup, one of them being an actual open source project and the other being a general category of tools.

There is an open source project called, "Jail Chroot Project" which greatly reduces setup time for several basic and even more advanced jails [9]. This project is a set of programs and scripts used to setup and configure a jail. It features the ability to setup several fully functional server jails such as telnetd, ftpd, and sshd, and runs on several Unix based operating systems including all major Linux distributions.

The other tool, or rather the other set of tools, is less packaged and easy to work with. These tools are the scripts & programs written and developed for performing more specific tasks on a jail. This category includes scripts found on the Internet from various people, and individual Sourceforge.net projects. Though this seems very general and possibly not very useful, consider this: I typed "apache jail chroot linux" into the internet search engine http://www.google.com and the third result was called "Apache in a chroot jail" [10, 11]. This site contained a break down of how to put apache into a jail, and included specific list of libraries and resources needed within the jail. This very same technique could be used to find people who have made lists of resources, or maybe even scripts, to jail many common programs.

Sourceforge.net is another great resource for open source projects [12]. A great deal of the projects found at this site are focused toward Unix based systems,

and all projects are open source. By being open source, you can have a little more trust in projects you find there, as all code is under public scrutiny. However, do not take this to mean that malicious code cannot exist in the open source community; it is just harder for it to exist. To find chroot projects at Sourceforge.net, go to http://sourceforge.net and use their search engine to search for the word "chroot".

A good tip to remember is that for what you want to jail, someone will likely have already done it. Unless you are just curious as to how tedious it is to jail something, it is recommended that you first do a couple searches on the Internet for jailing the desired application.

Before running any scripts or projects that you find on the Internet, you should analyze the tool or tools before running them on a production system. Anything you get from the Internet might not be as benign as the author would lead you to believe. A hacker might have written the script you just downloaded in hopes that you run it on a productions system and compromise yourself. Scripts and projects that are licensed as open source may be more trust since they are open to public scrutiny, but that does not mean they do not have bugs or security holes in them. Several projects make use of md5sums which allows you to validate the software's integrity. This can help prevent outsiders from tampering with the software while in transit to your machine. If a software package utilizes md5sums you should always check the validity of the package before installing it. Even with md5sums, the most important lesson here is always run unknown code on a test system before running it on a production system.

5. User-mode Linux Basics

User-mode Linux commonly referred to as UML in the open source community is somewhat like a virtual machine that can run without special admin rights. In fact, it enables you boot a full Linux kernel inside of a user space [3]. For those who have heard of Vmware or Plex86, UML is very similar but can only serve and run on Linux kernels. The differences between UML and virtual machines such as VMware and Plex86 could probably be covered in its own paper, and even then may not be of much interest to the security community. Consequentially, these differences will not be covered in this paper. One basic and important difference will be noted though: UML has now been merged into the 2.5.34 Linux development kernel [2].

With UML now a part of the main kernel, it will be available in future distributions of Linux making it just as readily available as chroot. For now you can download it from the "User-mode Linux Kernel Home Page" [14]. To setup UML on RedHat with a kernel of 2.3 or higher, simply download the RPM and install it. If you are running a kernel less than 2.3, you should really upgrade your kernel, but there is a patch available. Also, if you are not running RedHat, the source is available and has been compiled and tested on all Linux distributions. If you do need to

compile the source, then you will have to follow the instructions as documented on the User-mode Linux Kernel site [15].

After installing UML, you only need a file system to boot in order to get started. Several pre-built filesystems are available at the "Downloading it" section of the UML Home Page, so you have a quick place to start [14]. If a pre-built system does not suite your needs there is a great deal of documentation on how to build your own filesystem under the "Creating your own filesystems" section of the UML Home Page [13].

Assuming you chose to download a pre-built filesystem, check the md5sum to validate the filesystem, decompress it, and create a symbolic link to it called **root_fs**. Then run the command **Linux** in the same directory as the symbolic link. This will boot the filesystem and bring you to a login prompt. On the pre-built filesystem that was tested for this paper, the root account's starting password is root.

An important note is that you do not have to run the virtual machine as the root user [3]. This means that if a hacker broke into a service running on your virtual machine, then later broke out of the virtual machine into the main system, the hacker would only have the privileges available to the user that started the virtual machine.

Though there is some setup needed to utilize important functionality such as networking, unlike chroot this is a very well documented and standard process. Due to the amount of tutorials and documentation available this will not be covered in this paper and will be left up to the reader to research. There are several tutorials on how to setup networking, honeypots, X-Windows, etc. on the UML Home Page under "Tutorials" on the navigation bar [16].

6. Securing a User-Mode Linux kernel

Securing a User-Mode Linux kernel is really just like securing a full production system. You should use your experience as a system administrator to guide you through securing the virtual machine, but this section will offer a few pieces of advise.

Turn off and remove anything that is not necessary. If you aren't going to run a telnet server from your virtual machine, disable it and remove it so an intruder cannot turn it back on. To find unnecessary services, run **chkconfig –list** to see what is started at different run levels for the system. Identify services that will not be used and make sure they are turned off for all run levels. If you are unsure whether the service is necessary or not, make a backup copy of the virtual machine, turn off the service, and see if anything breaks. If something you need breaks, then turn the service back on for the desired run levels. If nothing breaks then you do not need it. Additionally look for services that may be bootstrapped

to other services such as xinetd. Once you have turned off the unnecessary services you should remove them. On an RPM based system you can remove a service or software package by typing the command **rpm -e package_name.** If you do not know the name of the package you need to remove, use **rpm -qa|more** to look through the package names. To query a package to see what software it contains use **rpm -qi** to get a description of the package's contents.

Practice the philosophy of least privileges and remove any unnecessary rights from all users. Reduce file permissions where possible, and make root the owner of files that users should not be touching.

Run services and programs as a non-root user where possible. A User-Mode Linux virtual machine is not indestructible, but there is more difficulty breaking out of a UML jail if an attacker cannot gain root access.

To really secure a service, run UML as a non-root user in a chroot jail. UML is easy to chroot as there is a program available from the User-mode Linux Kernel site called "jail_uml" [14]. This program runs your virtual machine in a jail so there is no reason not to chroot jail your virtual machine. The truly paranoid system administrator can run a jailed program inside of a chroot jail, which is inside a UML virtual machine, which is inside a chroot jail, and all programs run as non-root users. This may seem quite paranoid but it really depends on what you are jailing. The bottom line is that there is no reason not to combine tools together to increase security. In fact, this is what defense in depth is really about.

Finally, UML has the ability to limit the amount of RAM that a virtual machine can use [18]. Implementing a RAM limit can decrease the number of possible DoS attacks by not allowing a hacker to run a script that consumes all the memory on the machine.

7. Chroot Benefits over User-mode Linux

Chroot has a limited number of benefits over UML. The main benefit is that it is a tried, tested, and true way of securing an application, without a huge performance overhead. Chroot has been around for a long time, and it is not going away anytime soon. Some applications such as Postfix are setup to be run in a chroot jail [17].

Due to the age of chroot, system administrators can have a good deal of confidence that if they jail an application correctly, there is a very limited chance for complete system compromise. Its age has also lead to several case studies and tools available to make the job easier to do [1, 9, 10, 17]. Due to the years of public scrutiny, system administrators can have greater confidence in the integrity of this software.

Some applications have gone to lengths to be chroot jail "compatible". These programs run as a contained package that is easily separated from the main system. This by far eases the workload for jailing the process.

The performance hit induced by running User-mode Linux may not be acceptable for some systems. If an application is jail compatible, why induce the overhead of running a full Linux virtual machine? The performance hit from running a chroot jail is minimal, allowing you to run several jailed programs on one system. The overhead from UML just maybe too much for older systems or very CPU intensive applications.

8. User-mode Linux Benefits over chroot

There are several benefits that UML offers, some of which are not as obvious as others. Though there is an overhead to running a UML virtual machine, it is a full virtual machine and therefore can be relatively easy to jail programs. The documentation for setting up a UML filesystem is fairly good. UML filesystems are also generally contained within one single file on the host system. This single file offers a huge number of benefits that may not be inherently obvious. Though UML does impose an overhead, this overhead can be limited by specifying the amount of RAM the virtual machine is allowed to consume. Finally, UML itself is chroot jail compatible, so the virtual machine can be run inside a jail for even more security with very minimal additional overhead.

Chroot is a great tool for jailing small or jail compatible applications. Unfortunately, the larger and more complex the application, the higher the probability that the jailed application will be very difficult to setup and maintain. If a jail is too difficult to maintain, something might be missed which might decrease the security of the application as a whole. This problem is not an issue with User-mode Linux since it is a full Linux system onto itself. Jailing an application becomes much easier, as there is a great deal of documentation on how to get basic functionality working. After the basics are in place, adding and maintaining a complex application is just as easy as adding it to the host system. This also means you do not have to play with tools such as strace and ldd to create a working jailed process.

There is a great deal of documentation on creating your own virtual system, or if you trust the User-mode Linux developers, there are several existing virtual system available on the User-mode Linux Kernel website [13, 14]. The tutorials tell you how to get the basics such as X-Windows, networking, and even Honeypots working.

A major benefit is that virtual system can be entirely contained in one file, which makes backup and restore of the virtual system very easy. If someone attacks and even damages your system, incident containment and eradication time can be far reduced. Simply turn off the virtual system, make a copy of the file

containing the filesystem, then (if desired) restore from backup a copy of the file from before the system was compromised. The eradication step can be begin in a lab within minutes of the containment phase. Multiple copies of the same filesystem can easily be given to several incident handlers to check concurrently instead of everyone crowding around one console. In addition, all the incident handlers can analyze several copies of the compromised system on one physical machine due to UML's ability to run in the user space. Once the security hole is identified, it can be patched on the restored copy of the virtual machine and the recovery phase can begin.

For system setup and initial research, the single file filesystem also means you can experiment with a copy of the virtual system without worrying that you will damage the system and have to re-install. Wonder if the system will be able to function properly without a device or library but worried you will not be able to boot up if you remove it? Simply make the desired change on a copy and see what happens. If things blow up in your face, delete the copy and go back to your pre-change copy.

The single file filesystem is also very useful for applying patches as well. Most company policies require that you test a security patch on a test system before applying it to production. The problem with testing patches is sometimes they can be complex. They are not always as easy as going to a site, downloading a binary and running a patch command. When using a virtual system, you can repeatedly apply the patch to a pre-patched system. If your production servers are running UML virtual machines, you can obtain a copy of the production filesystem and test the patch in a lab. Having done these repeated patches and patching production, you will be more confident that you will apply the patch correctly to the production systems.

Once you have achieved a secure virtual machine, you can easily deploy it to multiple physical machines. If you have a standard way of setting up a web server within your infrastructure, you can setup an initial server with your default settings, then deploy and tweak as necessary. This method gives you a strong base to start on, and avoids missing those small but important steps of setting up security.

UML's ability to limit RAM usage is not just a security bonus. It is useful for limiting a process from consuming all of your systems resources in the case of memory leaks [18].

A final noteworthy benefit of UML is that it is chroot jail compatible. As previously discussed the utilities tarball available at the download site for UML comes with a program called "jail_uml" which runs your virtual machine within a chroot jail. There does not have to be a choice between chroot and UML, as they run well together. If you can afford the overhead of running UML, you really should run it in a chroot jail.

9. Conclusion

The use of jailing processes for security has been used for quite some time in the industry, but has in the past been quite a lot of work. With new tools such as the Jail Chroot Project and User-Mode Linux the job has become much easier for the system administrator. By use of jailing jails, a new level of defense in depth can be achieved with little setup by the system administrator. No one tool will ever be the catch all for security, but jailing an application may drastically slow or irritate an attacker to the point of leaving your server in search of an easier target.

There is no rule that works 100% of the time for choosing chroot over UML, but there are some basic suggestions. If a program is simple, or is chroot jail compatible, then the application should be run in a chroot jail. If it is complex and the overhead is not an issue, then a UML virtual machine is probably the better candidate. If the application is to be run in UML, you should run UML within a chroot jail for optimal security. Jailing all processes on a machine would be very difficult to maintain and would most likely cause too much overhead (whether system overhead, or system administration overhead) so jail only the most security sensitive applications that you feel have the highest probability of being attacked.

10. References

1. Friedl, Steve. "Go Directly to Jail". Dec 2002. URL: http://www.linux-mag.com/2002-12/chroot_01.html (accessed on March 23, 2003)
2. Nimrod. "Linux: UML Merged Into 2.5". 13 Sept 2002. URL: "http://kerneltrap.org/node.php?id=409 (accessed on March 23, 2003)
3. LeBlanc, Dee-Ann. "User-mode Linux: Coming to a Kernel Near You, Part 1". URL: http://www.linuxplanet.com/linuxplanet/tutorials/4712/1/ (accessed on March 23, 2003)
4. Friedl, Steve. "Go Directly to Jail". Dec 2002. URL: http://www.linux-mag.com/2002-12/chroot_02.html (accessed on March 23, 2003)
5. Friedl, Steve. "Go Directly to Jail". Dec 2002. URL: http://www.linux-mag.com/2002-12/chroot_03.html (accessed on March 23, 2003)
6. Friedl, Steve. "Best Practices for UNIX chroot() Operations". 18 Jan 2001. URL: http://www.unixwiz.net/techtips/chroot-practices.html (accessed on March 23, 2003)
7. Simon. "How to break out of a chroot() jail". 12 May 2002. URL: http://www.bpfh.net/simes/computing/chroot-break.html (accessed on March 23, 2003)
8. Friedl, Steve. "runchroot.c". Dec 2002. URL:

http://www.linuxmagazine.com/downloads/2002-12/jail/runchroot.c
(accessed on March 23, 2003)
9. Casillas, Juan. "Jail Chroot Project". 29 Oct 2001. URL: http://www.gsyc.inf.uc3m.es/~assman/jail/index.html
(accessed on March 23, 2003)
10. Mourani, Gerhard. "Chapter 29. Software – Network Server, web/Apache". URL: http://en.tldp.org/LDP/solrhe/Securing-Optimizing-Linux-RH-Edition-v1.3/chap29sec254.html
(accessed on March 23, 2003)
11. "Google", URL: www.google.com
(accessed on March 23, 2003)
12. "Sourceforge.net". URL: http://sourceforge.net
(accessed on March 23, 2003)
13. Dike, Jeff. "Creating your own filesystems". URL: http://user-mode-linux.sourceforge.net/fs_making.html
(accessed on March 23, 2003)
14. Dike, Jeff. "Downloads". URL: http://user-mode-linux.sourceforge.net/dl-sf.html
(accessed on March 23, 2003)
15. Dike, Jeff. "Compiling the kernel and modules". URL: http://user-mode-linux.sourceforge.net/compile.html
(accessed on March 23, 2003)
16. Dike, Jeff. "Site Home Page". URL: http://user-mode-linux.sourceforge.net
(accessed on March 23, 2003)
17. Borland, Matt. "Locking Down Your Daemons: An Overview of 'chroot jailing' services in Linux". 20 May 2001. URL: http://www.sans.org/rr/linux/daemons.php
(accessed on March 23, 2003)
18. LeBlanc, Dee-Ann. "User-mode Linux: Coming to a Kernel Near You, Part 1". URL: http://www.linuxplanet.com/linuxplanet/tutorials/4712/2/
(accessed on March 23, 2003)

Appendix A

A Sample make jail script. It was successfully tested on a Linux RedHat 7.3 system, so you mileage may vary depending on what flavor of Linux you test this on. Also, though the principle of this script is applicable to most Unix based systems, I can almost guarantee it will not function without major tweaking in a non-Linux system.

```
# Make the Jail directory
JAIL_NAME=jail_$RANDOM
mkdir $JAIL_NAME
if [ ! -e $JAIL_NAME ]
then
echo "Could not create the jail directory."
exit 1
fi
cd $JAIL_NAME

# Create directory structure
mkdir bin dev etc lib usr tmp
mkdir usr/bin usr/sbin
mknod dev/null c 1 3
mknod dev/zero c 1 5

# Copy Basic libraries
cp /lib/ld-linux.so.2                   lib
cp /lib/libc.so.6    lib
cp /lib/libnsl.so.1  lib
cp /lib/libnss_nisplus.so.2             lib
cp /lib/libtermcap.so.2                 lib
cp /lib/libcrypto.so.2                  lib
cp /lib/libdl.so.2   lib
cp /lib/libnss_files.so.2               lib
cp /lib/libpam.so.0                     lib
cp /lib/libutil.so.1 lib

# Copy some basic commands
cp /bin/bash        bin
cp /bin/ls          bin
cp /usr/bin/strace usr/bin

# Create etc entries
echo "root:x:0:0:root:/:/bin/bash"> etc/passwd
echo "root:x:0:root"> etc/group
echo "passwd:     files nisplus
shadow:     files nisplus
group:      files nisplus
hosts:      files nisplus dns
networks:   files
protocols:  files nisplus
services:   files nisplus
publickey:  nisplus"> etc/nsswitch.conf
cp /etc/localtime etc

# Change permissions
#First bolt down the permissions
chmod -R 000 *
chown -R root *
chgrp -R root *
#Now gradually open the permissions
chmod -R 550 bin
chmod -R 550 usr/bin
chmod -R 550 usr/sbin
chmod -R 770 tmp
chmod -R 440 etc
chmod -R 550 lib
```