Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>

Robert N. M. Watson <rwatson@FreeBSD.org>

The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually un-trusting parties are introduced, ‘‘inappropriate’’ rapidly transitions to ‘‘nightmarish’’, especially with regards to data integrity and privacy protection.

The FreeBSD ‘‘Jail’’ facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ‘‘root’’ model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality, and low total cost of ownership with the need for alternative security models that are difficult or impossible to implement with the UNIX security mechanisms.

One such consideration is the desire to delegate some (but not all) administrative functions to untrusted or less trusted parties, and simultaneously impose system-wide mandatory policies on process interaction and sharing. Attempting to create such an environment in the current-day FreeBSD security environment is both difficult and costly: in many cases, the burden of implementing these policies falls on user applications, which means an increase in the size and complexity of the code base, in turn translating to higher development and maintenance cost, as well as less overall flexibility.

This abstract risk becomes more clear when applied to a practical, real-world example: many web service providers turn to the FreeBSD operating system to host customer web sites, as it provides a high-performance, network-centric server environment. However, these providers have a number of concerns on their plate, both in terms of protecting the integrity and confidentiality of their own files and services from

This paper was presented at the 2nd International System Administration and Networking Conference "SANE 2000" May 22-25, 2000 in Maastricht, The Netherlands and is published in the proceedings.

This work was sponsored by `http://www.servetheweb.com/` and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, `http://www.safeport.com/`

their customers, as well as protecting the files and services of one customer from (accidental or intentional) access by any other customer. At the same time, a provider would like to provide substantial autonomy to customers, allowing them to install and maintain their own software, and to manage their own services, such as web servers and other content-related daemon programs.

This problem space points strongly in the direction of a partitioning solution, in which customer processes and storage are isolated from those of other customers, both in terms of accidental disclosure of data or process information, but also in terms of the ability to modify files or processes outside of a compartment. Delegation of management functions within the system must be possible, but not at the cost of system-wide requirements, including integrity and privacy protection between partitions.

However, UNIX-style access control makes it notoriously difficult to compartmentalise functionality. While mechanisms such as chroot(2) provide a modest level compartmentalisation, it is well known that these mechanisms have serious shortcomings, both in terms of the scope of their functionality, and effectiveness at what they provide [CHROOT].

In the case of the chroot(2) call, a process's visibility of the file system name-space is limited to a single subtree. However, the compartmentalisation does not extend to the process or networking spaces and therefore both observation of and interference with processes outside their compartment is possible.

To this end, we describe the new FreeBSD "Jail" facility, which provides a strong partitioning solution, leveraging existing mechanisms, such as chroot(2), to what effectively amounts to a virtual machine environment. Processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they can make use of, and neither access nor visibility of files, processes or network services outside their partition.

Unlike other fine-grained security solutions, Jail does not substantially increase the policy management requirements for the system administrator, as each Jail is a virtual FreeBSD environment permitting local policy to be independently managed, with much the same properties as the main system itself, making Jail easy to use for the administrator, and far more compatible with applications.

2. Traditional UNIX Security, or, "God, root, what difference?" [UF].

The traditional UNIX access model assigns numeric uids to each user of the system. In turn, each process "owned" by a user will be tagged with that user's uid in an unforgeable manner. The uids serve two purposes: first, they determine how discretionary access control mechanisms will be applied, and second, they are used to determine whether special privileges are accorded.

In the case of discretionary access controls, the primary object protected is a file. The uid (and related gids indicating group membership) are mapped to a set of rights for each object, courtesy the UNIX file mode, in effect acting as a limited form of access control list. Jail is, in general, not concerned with modifying the semantics of discretionary access control mechanisms, although there are important implications from a management perspective.

For the purposes of determining whether special privileges are accorded to a process, the check is simple: "is the numeric uid equal to 0 ?". If so, the process is acting with "super-user privileges", and all access checks are granted, in effect allowing the process the ability to do whatever it wants to [1].

For the purposes of human convenience, uid 0 is canonically allocated to the "root" user [ROOT]. For the purposes of jail, this behaviour is extremely relevant: many of these privileged operations can be used to manage system hardware and configuration, file system name-space, and special network operations.

Many limitations to this model are immediately clear: the root user is a single, concentrated source of privilege that is exposed to many pieces of software, and as such an immediate target for attacks. In the event of a compromise of the root capability set, the attacker has complete control over the system. Even without an attacker, the risks of a single administrative account are serious: delegating a narrow scope of capability to an inexperienced administrator is difficult, as the granularity of delegation is that of all system management abilities. These features make the omnipotent root account a sharp, efficient and extremely dangerous tool.

[1] ... no matter how patently stupid it may be.

The BSD family of operating systems have implemented the ‘‘securelevel’’ mechanism which allows the administrator to block certain configuration and management functions from being performed by root, until the system is restarted and brought up into single-user mode. While this does provide some amount of protection in the case of a root compromise of the machine, it does nothing to address the need for delegation of certain root abilities.

3. Other Solutions to the Root Problem

Many operating systems attempt to address these limitations by providing fine-grained access controls for system resources [BIBA]. These efforts vary in degrees of success, but almost all suffer from at least three serious limitations:

First, increasing the granularity of security controls increases the complexity of the administration process, in turn increasing both the opportunity for incorrect configuration, as well as the demand on administrator time and resources. In many cases, the increased complexity results in significant frustration for the administrator, which may result in two disastrous types of policy: ‘‘all doors open as it’s too much trouble’’, and ‘‘trust that the system is secure, when in fact it isn’t’’.

The extent of the trouble is best illustrated by the fact that an entire niche industry has emerged providing tools to manage fine grained security controls [UAS].

Second, usefully segregating capabilities and assigning them to running code and users is very difficult. Many privileged operations in UNIX seem independent, but are in fact closely related, and the handing out of one privilege may, in effect, be transitive to the many others. For example, in some trusted operating systems, a system capability may be assigned to a running process to allow it to read any file, for the purposes of backup. However, this capability is, in effect, equivalent to the ability to switch to any other account, as the ability to access any file provides access to system keying material, which in turn provides the ability to authenticate as any user. Similarly, many operating systems attempt to segregate management capabilities from auditing capabilities. In a number of these operating systems, however, ‘‘management capabilities’’ permit the administrator to assign ‘‘auditing capabilities’’ to itself, or another account, circumventing the segregation of capability.

Finally, introducing new security features often involves introducing new security management APIs. When fine-grained capabilities are introduced to replace the setuid mechanism in UNIX-like operating systems, applications that previously did an ‘‘appropriateness check’’ to see if they were running as root before executing must now be changed to know that they need not run as root. In the case of applications running with privilege and executing other programs, there is now a new set of privileges that must be voluntarily given up before executing another program. These change can introduce significant incompatibility for existing applications, and make life more difficult for application developers who may not be aware of differing security semantics on different systems [POSIX1e].

4. The Jail Partitioning Solution

Jail neatly side-steps the majority of these problems through partitioning. Rather than introduce additional fine-grained access control mechanism, we partition a FreeBSD environment (processes, file system, network resources) into a management environment, and optionally subset Jail environments. In doing so, we simultaneously maintain the existing UNIX security model, allowing multiple users and a privileged root user in each jail, while limiting the scope of root’s activities to his jail. Consequently the administrator of a FreeBSD machine can partition the machine into separate jails, and provide access to the super-user account in each of these without losing control of the over-all environment.

A process in a partition is referred to as ‘‘in jail’’. When a FreeBSD system is booted up after a fresh install, no processes will be in jail. When a process is placed in a jail, it, and any descendents of the process created after the jail creation, will be in that jail. A process may be in only one jail, and after creation, it can not leave the jail. Jails are created when a privileged process calls the jail(2) syscall, with a description of the jail as an argument to the call. Each call to jail(2) creates a new jail; the only way for a new process to enter the jail is by inheriting access to the jail from another process already in that jail. Processes may never leave the jail they created, or were created in.

Fig. 1 — Schematic diagram of machine with two configured jails

Membership in a jail involves a number of restrictions: access to the file name-space is restricted in the style of chroot(2), the ability to bind network resources is limited to a specific IP address, the ability to manipulate system resources and perform privileged operations is sharply curtailed, and the ability to interact with other processes is limited to only processes inside the same jail.

Jail takes advantage of the existing chroot(2) behaviour to limit access to the file system name-space for jailed processes. When a jail is created, it is bound to a particular file system root. Processes are unable to manipulate files that they cannot address, and as such the integrity and confidentiality of files outside of the jail file system root are protected. Traditional mechanisms for breaking out of chroot(2) have been blocked. In the expected and documented configuration, each jail is provided with its exclusive file system root, and standard FreeBSD directory layout, but this is not mandated by the implementation.

Each jail is bound to a single IP address: processes within the jail may not make use of any other IP address for outgoing or incoming connections; this includes the ability to restrict what network services a particular jail may offer. As FreeBSD distinguishes attempts to bind all IP addresses from attempts to bind a particular address, bind requests for all IP addresses are redirected to the individual Jail address. Some network functionality associated with privileged calls are wholesale disabled due to the nature of the functionality offered, in particular facilities which would allow ‘‘spoofing’’ of IP numbers or disruptive traffic to be generated have been disabled.

Processes running without root privileges will notice few, if any differences between a jailed environment or un-jailed environment. Processes running with root privileges will find that many restrictions apply to the privileged calls they may make. Some calls will now return an access error — for example, an attempt to create a device node will now fail. Others will have a more limited scope than normal — attempts to bind a reserved port number on all available addresses will result in binding only the address associated with the jail. Other calls will succeed as normal: root may read a file owned by any uid, as long as it is accessible through the jail file system name-space.

Processes within the jail will find that they are unable to interact or even verify the existence of processes outside the jail — processes within the jail are prevented from delivering signals to processes outside the jail, as well as connecting to those processes with debuggers, or even see them in the sysctl or process file system monitoring mechanisms. Jail does not prevent, nor is it intended to prevent, the use of covert channels or communications mechanisms via accepted interfaces — for example, two processes may communicate via sockets over the IP network interface. Nor does it attempt to provide scheduling services based on the partition; however, it does prevent calls that interfere with normal process operation.

As a result of these attempts to retain the standard FreeBSD API and framework, almost all applications will run unaffected. Standard system services such as Telnet, FTP, and SSH all behave normally, as do most third party applications, including the popular Apache web server.

5. Jail Implementation

Processes running with root privileges in the jail find that there are serious restrictions on what it is capable of doing — in particular, activities that would extend outside of the jail:

- Modifying the running kernel by direct access and loading kernel modules is prohibited.
- Modifying any of the network configuration, interfaces, addresses, and routing table is prohibited.
- Mounting and unmounting file systems is prohibited.
- Creating device nodes is prohibited.
- Accessing raw, divert, or routing sockets is prohibited.
- Modifying kernel runtime parameters, such as most sysctl settings, is prohibited.
- Changing securelevel-related file flags is prohibited.
- Accessing network resources not associated with the jail is prohibited.

Other privileged activities are permitted as long as they are limited to the scope of the jail:

• Signalling any process within the jail is permitted.

• Changing the ownership and mode of any file within the jail is permitted, as long as the file flags permit this.

• Deleting any file within the jail is permitted, as long as the file flags permit this.

• Binding reserved TCP and UDP port numbers on the jails IP address is permitted. (Attempts to bind TCP and UDP ports using INADDR_ANY will be redirected to the jails IP address.)

• Functions which operate on the uid/gid space are all permitted since they act as labels for filesystem objects of proceses which are partitioned off by other mechanisms.

These restrictions on root access limit the scope of root processes, enabling most applications to run un-hindered, but preventing calls that might allow an application to reach beyond the jail and influence other processes or system-wide configuration.

6. Implementation jail in the FreeBSD kernel.

6.1. The jail(2) system call, allocation, refcounting and deallocation of `struct prison`.

The jail(2) system call is implemented as a non-optional system call in FreeBSD. Other system calls are controlled by compile time options in the kernel configuration file, but due to the minute footprint of the jail implementation, it was decided to make it a standard facility in FreeBSD.

The implementation of the system call is straightforward: a data structure is allocated and populated with the arguments provided. The data structure is attached to the current process' `struct proc`, its reference count set to one and a call to the chroot(2) syscall implementation completes the task.

Hooks in the code implementing process creation and destruction maintains the reference count on the data structure and free it when the last reference is lost. Any new process created by a process in a jail will inherit a reference to the jail, which effectively puts the new process in the same jail.

There is no way to modify the contents of the data structure describing the jail after its creation, and no way to attach a process to an existing jail if it was not created from the inside that jail.

6.2. Fortification of the chroot(2) facility for filesystem name scoping.

A number of ways to escape the confines of a chroot(2)-created subscope of the filesystem view have been identified over the years. chroot(2) was never intended to be security mechanism as such, but even then the ftp daemon largely depended on the security provided by chroot(2) to provide the ‘‘anonymous ftp’’ access method.

Three classes of escape routes existed: recursive chroot(2) escapes, ‘‘..’’ based escapes and fchdir(2) based escapes. All of these exploited the fact that chroot(2) didn't try sufficiently hard to enforce the new root directory.

New code were added to detect and thwart these escapes, amongst other things by tracking the directory of the first level of chroot(2) experienced by a process and refusing backwards traversal across this directory, as well as additional code to refuse chroot(2) if file-descriptors were open referencing directories.

6.3. Restriction of process visibility and interaction.

A macro was already in available in the kernel to determine if one process could affect another process. This macro did the rather complex checking of uid and gid values. It was felt that the complexity of the macro were approaching the lower edge of IOCCC entrance criteria, and it was therefore converted to a proper function named `p_trespass(p1, p2)` which does all the previous checks and additionally checks the jail aspect of the access. The check is implemented such that access fails if the origin process is jailed but the target process is not in the same jail.

Process visibility is provided through two mechanisms in FreeBSD, the `procfs` file system and a sub-tree of the `sysctl` tree. Both of these were modified to report only the processes in the same jail to a jailed process.

6.4. Restriction to one IP number.

Restricting TCP and UDP access to just one IP number was done almost entirely in the code which manages "protocol control blocks". When a jailed process binds to a socket, the IP number provided by the process will not be used, instead the pre-configured IP number of the jail is used.

BSD based TCP/IP network stacks sport a special interface, the loop-back interface, which has the "magic" IP number 127.0.0.1. This is often used by processes to contact servers on the local machine, and consequently special handling for jails were needed. To handle this case it was necessary to also intercept and modify the behaviour of connection establishment, and when the 127.0.0.1 address were seen from a jailed process, substitute the jails configured IP number.

Finally the APIs through which the network configuration and connection state may be queried were modified to report only information relevant to the configured IP number of a jailed process.

6.5. Adding jail awareness to selected device drivers.

A couple of device drivers needed to be taught about jails, the "pty" driver is one of them. The pty driver provides "virtual terminals" to services like telnet, ssh, rlogin and X11 terminal window programs. Therefore jails need access to the pty driver, and code had to be added to enforce that a particular virtual terminal were not accessed from more than one jail at the same time.

6.6. General restriction of super-users powers for jailed super-users.

This item proved to be the simplest but most tedious to implement. Tedious because a manual review of all places where the kernel allowed the super user special powers were called for, simple because very few places were required to let a jailed root through. Of the approximately 260 checks in the FreeBSD 4.0 kernel, only about 35 will let a jailed root through.

Since the default is for jailed roots to not receive privilege, new code or drivers in the FreeBSD kernel are automatically jail-aware: they will refuse jailed roots privilege. The other part of this protection comes from the fact that a jailed root cannot create new device nodes with the mknod(2) systemcall, so unless the machine administrator creates device nodes for a particular device inside the jails filesystem tree, the driver in effect does not exist in the jail.

As a side-effect of this work the suser(9) API were cleaned up and extended to cater for not only the jail facility, but also to make room for future partitioning facilities.

6.7. Implementation statistics

The change of the suser(9) API modified approx 350 source lines distributed over approx. 100 source files. The vast majority of these changes were generated automatically with a script.

The implementation of the jail facility added approx 200 lines of code in total, distributed over approx. 50 files. and about 200 lines in two new kernel files.

7. Managing Jails and the Jail File System Environment

7.1. Creating a Jail Environment

While the jail(2) call could be used in a number of ways, the expected configuration creates a complete FreeBSD installation for each jail. This includes copies of all relevant system binaries, data files, and its own `/etc` directory. Such a configuration maximises the independence of various jails, and reduces the chances of interference between jails being possible, especially when it is desirable to provide root access within a jail to a less trusted user.

On a box making use of the jail facility, we refer to two types of environment: the host environment, and the jail environment. The host environment is the real operating system environment, which is used to configure interfaces, and start up the jails. There are then one or more jail environments, effectively virtual FreeBSD machines. When configuring Jail for use, it is necessary to configure both the host and jail environments to prevent overlap.

As jailed virtual machines are generally bound to an IP address configured using the normal IP alias mechanism, those jail IP addresses are also accessible to host environment applications to use. If the accessibility of some host applications in the jail environment is not desirable, it is necessary to configure those applications to only listen on appropriate addresses.

In most of the production environments where jail is currently in use, one IP address is allocated to the host environment, and then a number are allocated to jail boxes, with each jail box receiving a unique IP. In this situation, it is sufficient to configure the networking applications on the host to listen only on the host IP. Generally, this consists of specifying the appropriate IP address to be used by inetd and SSH, and disabling applications that are not capable of limiting their address scope, such as sendmail, the port mapper, and syslogd. Other third party applications that have been installed on the host must also be configured in this manner, or users connecting to the jailbox will discover the host environment service, unless the jailbox has specifically bound a service to that port. In some situations, this can actually be the desirable behaviour.

The jail environments must also be custom-configured. This consists of building and installing a miniature version of the FreeBSD file system tree off of a subdirectory in the host environment, usually `/usr/jail`, or `/data/jail`, with a subdirectory per jail. Appropriate instructions for generating this tree are included in the jail(8) man page, but generally this process may be automated using the FreeBSD build environment.

One notable difference from the default FreeBSD install is that only a limited set of device nodes should be created. MAKEDEV(8) has been modified to accept a “jail” argument that creates the correct set of nodes.

To improve storage efficiency, a fair number of the binaries in the system tree may be deleted, as they are not relevant in a jail environment. This includes the kernel, boot loader, and related files, as well as hardware and network configuration tools.

After the creation of the jail tree, the easiest way to configure it is to start up the jail in single-user mode. The sysinstall admin tool may be used to help with the task, although it is not installed by default as part of the system tree. These tools should be run in the jail environment, or they will affect the host environment’s configuration.

```
# mkdir /data/jail/192.168.11.100/stand
# cp /stand/sysinstall /data/jail/192.168.11.100/stand
# jail /data/jail/192.168.11.100 testhostname 192.168.11.100 \
     /bin/sh
```

After running the jail command, the shell is now within the jail environment, and all further commands will be limited to the scope of the jail until the shell exits. If the network alias has not yet been configured, then the jail will be unable to access the network.

The startup configuration of the jail environment may be configured so as to quell warnings from services that cannot run in the jail. Also, any per-system configuration required for a normal FreeBSD system is also required for each jailbox. Typically, this includes:

- Create empty /etc/fstab
- Disable portmapper
- Run newaliases
- Disabling interface configuration
- Configure the resolver
- Set root password
- Set timezone
- Add any local accounts
- Install any packets

7.2. Starting Jails

Jails are typically started by executing their /etc/rc script in much the same manner a shell was started in the previous section. Before starting the jail, any relevant networking configuration should also be performed. Typically, this involves adding an additional IP address to the appropriate network interface, setting network properties for the IP address using IP filtering, forwarding, and bandwidth shaping, and mounting a process file system for the jail, if the ability to debug processes from within the jail is desired.

```
# ifconfig ed0 inet add 192.168.11.100 netmask 255.255.255.255
# mount -t procfs proc /data/jail/192.168.11.100/proc
# jail /data/jail/192.168.11.100 testhostname 192.168.11.100 \
     /bin/sh /etc/rc
```

A few warnings are generated for sysctl's that are not permitted to be set within the jail, but the end result is a set of processes in an isolated process environment, bound to a single IP address. Normal procedures for accessing a FreeBSD machine apply: telneting in through the network reveals a telnet prompt, login, and shell.

```
% ps ax
  PID  TT  STAT      TIME COMMAND
  228  ??  SsJ    0:18.73 syslogd
  247  ??  IsJ    0:00.05 inetd -wW
  249  ??  IsJ    0:28.43 cron
  252  ??  SsJ    0:30.46 sendmail: accepting connections on port 25
  291  ??  IsJ    0:38.53 /usr/local/sbin/sshd
93694  ??  SJ     0:01.01 sshd: rwatson@ttyp0 (sshd)
93695  p0  SsJ    0:00.06 -csh (csh)
93700  p0  R+J    0:00.00 ps ax
```

It is immediately obvious that the environment is within a jailbox: there is no init process, no kernel daemons, and a J flag is present beside all processes indicating the presence of a jail.

As with any FreeBSD system, accounts may be created and deleted, mail is delivered, logs are generated, packages may be added, and the system may be hacked into if configured incorrectly, or running a buggy version of a piece of software. However, all of this happens strictly within the scope of the jail.

7.3. Jail Management

Jail management is an interesting prospect, as there are two perspectives from which a jail environment may be administered: from within the jail, and from the host environment. From within the jail, as described above, the process is remarkably similar to any regular FreeBSD install, although certain actions are prohibited, such as mounting file systems, modifying system kernel properties, etc. The only area that really differs are that of shutting the system down: the processes within the jail may deliver signals between them, allowing all processes to be killed, but bringing the system back up requires intervention from outside of the jailbox.

From outside of the jail, there are a range of capabilities, as well as limitations. The jail environment is, in effect, a subset of the host environment: the jail file system appears as part of the host file system, and may be directly modified by processes in the host environment. Processes within the jail appear in the process listing of the host, and may likewise be signalled or debugged. The host process file system makes the hostname of the jail environment accessible in /proc/procnum/status, allowing utilities in the host environment to manage processes based on jailname. However, the default configuration allows privileged processes within jails to set the hostname of the jail, which makes the status file less useful from a management perspective if the contents of the jail are malicious. To prevent a jail from changing its hostname, the "jail.set_hostname_allowed" sysctl may be set to 0 prior to starting any jails.

One aspect immediately observable in an environment with multiple jails is that uids and gids are local to each jail environment: the uid associated with a process in one jail may be for a different user than in another jail. This collision of identifiers is only visible in the host environment, as normally processes from one jail are never visible in an environment with another scope for user/uid and group/gid mapping. Managers in the host environment should understand these scoping issues, or confusion and unintended consequences may result.

Jailed processes are subject to the normal restrictions present for any processes, including resource limits, and limits placed by the network code, including firewall rules. By specifying firewall rules for the IP address bound to a jail, it is possible to place connectivity and bandwidth limitations on individual jails, restricting services that may be consumed or offered.

Management of jails is an area that will see further improvement in future versions of FreeBSD. Some of these potential improvements are discussed later in this paper.

8. Future Directions

The jail facility has already been deployed in numerous capacities and a few opportunities for improvement have manifested themselves.

8.1. Improved Virtualisation

As it stands, the jail code provides a strict subset of system resources to the jail environment, based on access to processes, files, network resources, and privileged services. Virtualisation, or making the jail environments appear to be fully functional FreeBSD systems, allows maximum application support and the ability to offer a wide range of services within a jail environment. However, there are a number of limitations on the degree of virtualisation in the current code, and removing these limitations will enhance the ability to offer services in a jail environment. Two areas that deserve greater attention are the virtualisation of network resources, and management of scheduling resources.

Currently, a single IP address may be allocated to each jail, and all communication from the jail is limited to that IP address. In particular, these addresses are IPv4 addresses. There has been substantial interest in improving interface virtualisation, allowing one or more addresses to be assigned to an interface, and removing the requirement that the address be an IPv4 address, allowing the use of IPv6. Also, access to raw sockets is currently prohibited, as the current implementation of raw sockets allows access to raw IP packets associated with all interfaces. Limiting the scope of the raw socket would allow its safe use within a jail, re-enabling support for ping, and other network debugging and evaluation tools.

Another area of great interest to the current consumers of the jail code is the ability to limit the impact of one jail on the CPU resources available for other jails. Specifically, this would require that the jail of a process play a rule in its scheduling parameters. Prior work in the area of lottery scheduling, currently available as patches on FreeBSD 2.2.x, might be leveraged to allow some degree of partitioning between jail environments [LOTTERY1] [LOTTERY2]. However, as the current scheduling mechanism is targeted at time sharing, and FreeBSD does not currently support real time preemption of processes in kernel, complete partitioning is not possible within the current framework.

8.2. Improved Management

Management of jail environments is currently somewhat ad hoc--creating and starting jails is a well-documented procedure, but day-to-day management of jails, as well as special case procedures such as shutdown, are not well analysed and documented. The current kernel process management infrastructure does not have the ability to manage pools of processes in a jail-centric way. For example, it is possible to, within a jail, deliver a signal to all processes in a jail, but it is not possibly to atomically target all processes within a jail from outside of the jail. If the jail code is to effectively limit the behaviour of a jail, the ability to shut it down cleanly is paramount. Similarly, shutting down a jail cleanly from within is also not well defined, the traditional shutdown utilities having been written with a host environment in mind. This suggests a number of improvements, both in the kernel and in the user-land utility set.

First, the ability to address kernel-centric management mechanisms at jails is important. One way in which this might be done is to assign a unique jail id, not unlike a process id or process group id, at jail creation time. A new jailkill() syscall would permit the direction of signals to specific jailids, allowing for the effective termination of all processes in the jail. A unique jailid could also supplant the hostname as the unique identifier for a jail, allowing the hostname to be changed by the processes in the jail without interfering with jail management.

More carefully defining the user-land semantics of a jail during startup and shutdown is also important. The traditional FreeBSD environment makes use of an init process to bring the system up during the boot process, and to assist in shutdown. A similar technique might be used for jail, in effect a jailinit, formulated to handle the clean startup and shutdown, including calling out to jail-local /etc/rc.shutdown, and other useful shutdown functions. A jailinit would also present a central location for delivering management requests to within a jail from the host environment, allowing the host environment to request the shutdown of the jail cleanly, before resorting to terminating processes, in the same style as the host environment shutting down before killing all processes and halting the kernel.

Improvements in the host environment would also assist in improving jail management, possibly including automated runtime jail management tools, tools to more easily construct the per-jail file system area, and include jail shutdown as part of normal system shutdown.

These improvements in the jail framework would improve both raw functionality and usability from a management perspective. The jail code has raised significant interest in the FreeBSD community, and it is hoped that this type of improved functionality will be available in upcoming releases of FreeBSD.

9. Conclusion

The jail facility provides FreeBSD with a conceptually simple security partitioning mechanism, allowing the delegation of administrative rights within virtual machine partitions.

The implementation relies on restricting access within the jail environment to a well-defined subset of the overall host environment. This includes limiting interaction between processes, and to files, network resources, and privileged operations. Administrative overhead is reduced through avoiding fine-grained access control mechanisms, and maintaining a consistent administrative interface across partitions and the host environment.

The jail facility has already seen widespread deployment in particular as a vehicle for delivering "virtual private server" services.

The jail code is included in the base system as part of FreeBSD 4.0-RELEASE, and fully documented in the jail(2) and jail(8) man-pages.

Notes & References

[BIBA] K. J. Biba, Integrity Considerations for Secure Computer Systems, USAF Electronic Systems Division, 1977

[CHROOT]
Dr. Marshall Kirk Mckusick, private communication: "According to the SCCS logs, the chroot call was added by Bill Joy on March 18, 1982 approximately 1.5 years before 4.2BSD was released. That was well before we had ftp servers of any sort (ftp did not show up in the source tree until January 1983). My best guess as to its purpose was to allow Bill to chroot into the /4.2BSD build directory and build a system using only the files, include files, etc contained in that tree. That was the only use of chroot that I remember from the early days."

[LOTTERY1]
David Petrou and John Milford. Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System, December 1997.
`http://www.cs.cmu.edu/~dpetrou/papers/freebsd_lottery_writeup98.ps`
`http://www.cs.cmu.edu/~dpetrou/code/freebsd_lottery_code.tar.gz`

[LOTTERY2]
Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management, Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94), pages 1-11, Monterey, California, November 1994.
`http://www.research.digital.com/SRC/personal/caw/papers.html`

[POSIX1e]
Draft Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment: Protection, Audit and Control Interfaces [C Language] IEEE Std 1003.1e Draft 17 Editor Casey Schaufler

[ROOT] Historically other names have been used at times, Zilog for instance called the super-user account "zeus".

[UAS] One such niche product is the "UAS" system to maintain and audit RACF configurations on MVS systems.
`http://www.entactinfo.com/products/uas/`

[UF] Quote from the User-Friendly cartoon by Illiad.
`http://www.userfriendly.org/cartoons/archives/98nov/19981111.html`