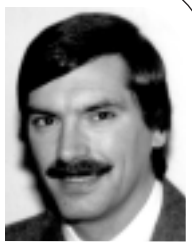
the jail facility in FreeBSD 5.2

by Kirk McKusick



Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. He has twice served on the Board and as president of USENIX.

mckusick@mckusick.com

The FreeBSD access control mechanism is designed for an environment with two types of users: those with and those without administrative privilege. It is often desirable 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. Historically, attempting to create such an environment has been both difficult and costly. The primary mechanism for partial delegation of administrative authority is to write a set-user-identifier program that carefully controls which of the administrative privileges may be used. These set-user-identifier programs are complex to write, difficult to maintain, limited in their flexibility, and prone to bugs that allow undesired administrative privilege to be gained.

[**Editor's note:** This article is a partial excerpt from Chapter 4, "Process Management," from *The Design and Implementation of the FreeBSD Operating System*, by Marshall Kirk McKusick and George Neville-Neil. Reprinted with permission from Pearson Education, Inc. (0-201-70245-2). Copyright 2005. To learn more: *http://www.awprofessional.com/title/0201702452*.]

Many operating systems attempt to address these limitations by providing fine-grained access controls for system resources [P1003.1e, 1998]. These efforts vary in degrees of success, but almost all suffer from at least three serious limitations:

1. 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. Often the increased complexity results in significant frustration for the administrator, which may result in two disastrous types of policy: running with security features disabled and running with the default configuration on the assumption that it will be secure.
2. Usefully segregating capabilities and assigning them to running code and users is difficult. Many privileged operations in FreeBSD seem independent but are interrelated. The handing out of one privilege may be transitive to many others. For example, the ability to mount file systems allows new set-user-identifier programs to be made available that in turn may yield other unintended security capabilities.
3. Introducing new security features often involves introducing new security management interfaces. When fine-grained capabilities are introduced to replace the set-user-identifier mechanism in FreeBSD, applications that previously did an appropriateness check to see if they were running with superuser privilege before executing must now be changed to know that they need not run with superuser privilege. For 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 changes 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.

This abstract risk becomes more clear when applied to a practical real-world example: Many Web service providers use FreeBSD to host customer Web sites. These providers must protect the integrity and confidentiality of their own files and services from their customers. They must also protect the files and services of one customer from (accidental or intentional) access by any other customer. A provider would like to supply

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. By putting each customer in a separate partition, customers are isolated from accidental or intentional modification of data or disclosure of process information from customers in other partitions. Delegation of management functions within the system must be possible without violating the integrity and privacy protection between partitions.

FreeBSD-style access control makes it notoriously difficult to compartmentalize functionality. While mechanisms such as chroot provide a modest level of compartmentalization, this mechanism has serious shortcomings, both in the scope of its functionality and the effectiveness of what it provides. The chroot system call was first added to provide an alternate build environment for the system. It was later adapted to isolate anonymous FTP access to the system.

The original intent of chroot was not to ensure security. Even when used to provide security for anonymous FTP, the set of operations allowed by FTP was carefully controlled to prevent those that allowed escape from the chrooted environment.

Three classes of escape from the confines of a chroot-created file system were identified over the years:

1. Recursive chroot escapes
2. Escapes using ..
3. Escapes using fchdir

All these escapes exploited the lack of enforcement of the new root directory.

Two changes to chroot were made to detect and thwart these escapes. To prevent the first two escapes, the directory of the first level of chroot experienced by a process is recorded. Any attempts to traverse backward across this directory are refused. The third escape, using fchdir, is prevented by having the chroot system call fail if the process has any file descriptors open referencing directories.

Even with stronger semantics, the chroot system call is insufficient to provide complete partitioning. Its compartmentalization does not extend to the process or networking spaces. Therefore, both observation of and interference with processes outside their compartment is possible. To provide a secure virtual machine environment, FreeBSD added a new "jail" facility built on top of chroot. Processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they may use. They are denied access to and visibility of files, processes, and network services outside their jail [Kamp & Watson, 2000].

Unlike other fine-grained security solutions, a jail doesn't substantially increase the policy management requirements for the system administrator. Each jail is a virtual FreeBSD environment that permits local policy to be independently managed. The environment within a jail has the same properties as the host system. Thus, a jail environment is familiar to the administrator and compatible with applications [Hope,

The administrator of a FreeBSD machine [can] partition the host into separate jails and provide access to the superuser account in each of these jails without losing control of the host environment.

2002].

Jail Semantics

Two important goals of the jail implementation are to:

1. Retain the semantics of the existing discretionary access-control mechanisms.
2. Allow each jail to have its own superuser administrator whose activities are limited to the processes, files, and network associated with its jail.

The first goal retains compatibility with most applications. The second goal permits the administrator of a FreeBSD machine to partition the host into separate jails and provide access to the superuser account in each of these jails without losing control of the host environment.

A process in a partition is referred to as being "in jail." When FreeBSD first boots, no processes will be jailed. Jails are created when a privileged process calls the jail system call with arguments of the file system into which it should chroot and the IP address and hostname to be associated with the jail. The process that creates the jail will be the first and only process placed in the jail. Any future descendants of the jailed process will be in its jail. A process may never leave a jail that it created or in which it was created. Any given process may be in only one 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.

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. A jail has the ability to restrict the set of network services that it chooses to offer at its address. An application request to bind all IP addresses is redirected to the individual address associated with the jail in which the requesting process is running.

A jail takes advantage of the existing chroot behavior to limit access to the file system namespace 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. Thus, the integrity and confidentiality of files outside the jail file-system root are protected.

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, connecting to processes outside the jail with debuggers, or even seeing processes outside the jail with the usual system-monitoring mechanisms. Jails do not prevent, nor are they intended to prevent, the use of covert channels or communications mechanisms via accepted interfaces. For example, two processes in different jails may communicate via sockets over the network. Jails do not attempt to provide scheduling services based on the partition.

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 that jail, restricting the services that it may consume or offer.

The jail environment is 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 be signaled or debugged.

Processes running without superuser privileges will notice few differences between a jailed environment and an unjailed environment. Standard system services such as remote login and mail servers behave normally, as do most third-party applications, including the popular Apache Web server. Processes running with superuser privileges will find that many restrictions apply to the privileged calls they may make. Most of the limitations are designed to restrict activities that would affect resources outside the jail. These restrictions include prohibitions against the following:

- Modifying the running kernel by direct access or loading kernel modules.
- Mounting and unmounting file systems.
- Creating device nodes.
- Modifying kernel runtime parameters such as most sysctl settings.
- Changing security-level flags.
- Modifying any of the network configuration, interfaces, addresses, and routing-table entries.
- Accessing raw, divert, or routing sockets. These restrictions prevent access to facilities that allow spoofing of IP numbers or the generation of disruptive traffic.
- Accessing network resources not associated with the jail. Specifically, an attempt to bind a reserved port number on all available addresses will result in binding only the address associated with the jail.
- Administrative actions that would affect the host system, such as rebooting.

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

- Signaling any process within the jail is permitted.
- Deleting or changing the ownership and mode of any file within the jail is permitted, as long as the file flags permit the requested change.
- The superuser may read a file owned by any UID, as long as it is accessible through the jail file system namespace.
- Binding reserved TCP and UDP port numbers on the jail's IP address is permitted.

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

Jail Implementation

The implementation of the jail system call is straightforward. A prison data structure is allocated and populated with the arguments provided. The prison structure is linked to the process structure of the calling process. The prison structure's reference count is set to one, and the chroot system call is called to set the jail's root. The prison structure may not be modified once it is created.

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

Some changes were needed to restrict process visibility and interaction. The kernel interfaces that report running processes were modified to report only the processes in the same jail as the process requesting the process information. Determining whether

Making the jail environment appear to be a fully functional FreeBSD system allows maximum application support and the ability to offer a wide range of services within a jail environment.

one process may send a signal to another is based on UID and GID values of the sending and receiving processes. With jails, the kernel adds the requirement that if the sending process is jailed, then the receiving process must be in the same jail.

Several changes were added to the networking implementation:

- Restricting TCP and UDP access to just one IP number was done almost entirely in the code that 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 preconfigured IP number of the jail is used.
- The loop-back interface, which has the magic IP number 127.0.0.1, is used by processes to contact servers on the local machine. When a process running in a jail connects to the 127.0.0.1 address, the kernel must intercept and redirect the connection request to the IP address associated with the jail.
- The interfaces 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.

Device drivers for shared devices such as the pseudo-terminal driver needed to be changed to enforce the restriction that a particular virtual terminal cannot be accessed from more than one jail at the same time.

The simplest but most tedious change was to audit the entire kernel for places that allowed the superuser extra privilege. Only about 35 of the 300 checks in FreeBSD 5.0 were opened to jailed processes running with superuser privileges. Since the default is that jailed superusers do not receive privilege, new code or drivers are automatically jail-aware: They will refuse jailed superusers privilege.

Jail Limitations

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. Making the jail environment appear to be a fully functional FreeBSD system allows maximum application support and the ability to offer a wide range of services within a jail environment. However, there are limitations in the current implementation. Removing these limitations will enhance the ability to offer services in a jail environment. Three areas that deserve greater attention are the set of available network resources, management of scheduling resources, and support for orderly jail shutdown.

Currently, only a single IP version 4 address may be allocated to each jail, and all communication from the jail is limited to that IP address. It would be desirable to support multiple addresses or possibly different address families for each jail. 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, thus allowing the use of ping and other network debugging and evaluation tools.

Another area of great interest to the current users of the jail code is the ability to limit the effect of one jail on the CPU resources available for other jails. Specifically, they require that the system have ways to allocate scheduling resources among the groups of processes in each of the jails. Work in the area of lottery scheduling might be leveraged to allow some degree of partitioning between jail environments [Petrou & Milford, 1997].

Management of jail environments is currently somewhat ad hoc. Creating and starting jails is a well-documented procedure, but jail shutdown requires the identification and killing of all the processes running within the jail. One approach to cleaning up this interface would be to assign a unique jail-identifier at jail creation time. A new jailkill system call would permit the direction of signals to specific jail-identifiers, allowing for the effective termination of all processes in the jail. FreeBSD makes use of an init process to bring the system up during the boot process and to assist in shutdown. A similarly operating process, jailinit, running in each jail would present a central location for delivering management requests to its jail from the host environment or from within the jail. The jailinit process would coordinate the clean shutdown of the jail before resorting to terminating processes, in the same style as the host environment shutting down before killing all processes and halting the kernel.

References

Hope, P. 2002. "Using Jails in FreeBSD for Fun and Profit," *;login:*, vol. 27, no. 3, pp. 48–55, *http://www.usenix.org/publications/login/2002-06/pdfs/hope.pdf*, USENIX Association, Berkeley, CA (June 2002).

Kamp, P. & R. Watson. 2000. "Jails: Confining the Omnipotent Root," *Proceedings of the Second International System Administration and Networking Conference (SANE), http://docs.freebsd.org/44doc/papers/jail/* (May 2000).

P1003.1e. 1998. Unpublished Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface—Amendment: Protection, Audit and Control Interfaces [C Language] IEEE Standard 1003.1e Draft 17, ed. Casey Schaufler, Institute of Electrical and Electronic Engineers, Piscataway, NJ (1998).

Petrou, D. & J. Milford. 1997. Proportional-Share Scheduling: Implementation and Evaluation in a Widely Deployed Operating System, *http://www.cs.cmu.edu/~dpetrou/papers/freebsd_lottery_writeup98.ps* and *http://www.cs.cmu.edu/~dpetrou/code/freebsd_lottery_code.tar.gz* (1997).