INTERNET ARCHIVE
Wayback Machine
http://www.nluug.nl/events/sane2000/papers/kamp.pdf
Go
12 captures
2 Sep 2000 - 13 May 2024
AUG SEP JAN
1999 02 2000 2005

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 untrusting 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,

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/

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 maintaennce 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 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

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

Google Exhibit 1145
Google v. VirtaMove