Restricted Community Accounts

Securing Science Gateways at the Account Level

Kevin J. Price, NCSA

Abstract

Community accounts represent a means by which scientific research gateways can allow a dynamic user base to submit jobs on computational resources without requiring each end-user to have an individual account on the resource. However, due to their shared nature these accounts are a potential weak point in the security of both the resource and the end-user's data. The goal of this project is to explore ways to ameliorate these problems by providing a framework that a resource provider can use to restrict accounts. Specifically, the focus is on solutions that are usable on a wide variety of POSIX-compliant systems without any major modifications. Our current thrust is towards utilizing the chroot functionality included in the POSIX standard to sandbox community accounts. We have in alpha release two products geared towards this aim: (1) "chroot_jail" is a utility suite that facilitates the construction, maintenance, and regular security auditing of chroot jails; and (2) "commsh" is a shell that can verify that a jail environment is secure, invoke chroot, and check the requested command against a list of allowable commands.

1. Introduction

Scientific research portals and gateways hold the promise to greatly increase the use of computational resources by a broad group of users. Community accounts allow these gateways to maintain a dynamic base of users that can submit jobs to shared, computational resources without requiring each end-user to have individual accounts on these resources.

Due to their shared nature, these accounts are a potential weak point in the security of the resource. As a potentially large group of users can utilize these accounts, they represent a very probable vector for attack. Furthermore, as the end user might not be known to the resource administrator, such an attack could be made with a greater degree of anonymity.

The goal of NCSA Restricted Community Accounts project [1] is to explore ways to ameliorate these security problems by developing a framework for restricting community accounts in a POSIX-compliant environment. This framework, or parts of it, can be implemented by a resource provider and configured to the specific needs of that resource.

Our current thrust is towards utilizing the chroot functionality included in the POSIX standard to sandbox community accounts. Accounts restricted to the confines of such a sandbox will have limited access to the rest of the system, and so a compromise of the community account need not equate to a compromise of the entire system.

We have two products in alpha release geared towards this end: chroot_jail, a utility suite to facilitate the construction of chroot jails, and commsh, a shell that can securely chroot to a jailed environment. Ideally, the chroot_jail suite can be used to create jailed environments, and commsh can be used to restrict an account to the jail. The utilities are also designed to be usable independently of one another.

2. Jailing utility suite

Overview

The "chroot_jail" utility suite is a set of tools written in Perl that facilitate the construction, maintenance, and regular security auditing of chroot jails. A chroot jail is a directory of an existing file system that represents a sandboxed environment, containing all data files, libraries, and executables, required for a limited set of applications. A job running from within a chroot jail cannot, under normal circumstances, access parts of the file system outside of the jail. Currently, chroot_jail constructs these jails by the inclusion of files that already exist on the system. There are two major problems that the suite tries to address: dependency tracking and security.

The utilities make use of DBM file databases, which are simple key-value hash tables stored as files. These databases contain assorted metadata about files in the jail, such as MD5 checksum information and dependency information. In particular, a flag is stored that indicates whether a file was included explicitly or implicitly.

An explicitly included file is typically part of an application package. These files are likely being used directly. By contrast, implicitly included files are those required by another application or library.

There are three ways in which the utilities currently check for file dependency: (1) shared libraries, (2) interpreters, and (3) symbolic links. Shared libraries are determined by invoking 'ldd' or a similar utility (dependant on the platform). Interpreters are determined by checking if the first line of an executable utilizes the standard "#!" notation. Symbolic links are determined as expected. Dependency determination is performed recursively, as each file included might have dependencies of its own.

By storing this information in a database, implicitly included files can be automatically removed if all files that depend on them are removed, which helps prevent a dynamically changing library from accumulating unnecessary baggage. Additionally, it is possible to determine exactly why any given library has been included.

On the security front, a security checking utility is included that compares files in the jail to entries in the database, making certain that files are located where they should be and checksums and ownership information match.

A configuration file [2] can be used to specify more finely-tuned security control, such as ignoring the checksum on a file or directory or files. Additionally, this configuration file can specify default ownership and mode information for files in different locations.

Implementation limitations

The main drawback of the current implementation is that there's no easy way to modify files within the jail in a way that updates the metadata stored in the corresponding database. Additionally, there's no way to mark a group of files as being part of the same package.

Some planned features and changes that will hopefully address these limitations include: a means of directly manipulating files in the jail in a way that also updates the database (such as editing, renaming, etc.); a means of importing packages directly into the jail without first adding them to the root directory structure; and a means of tagging groups of files as being in the same group.

3. Community shell

Overview

The community shell, "commsh", is a C++ program that currently offers two distinct methods for securing a community accounts: command restriction and chroot jailing. These two methods can be used separately or together, and their behavior is fully configurable [3].

Command restriction is implemented by matching the specified command to a list of allowed commands using a variety of matching algorithms. These algorithms include sudo-style matching [4], regular expression matching, verbatim matching, and a specialized matching routine that uses the POSIX-standard fnmatch function as its basis.

Secure chroot jailing involves checking the jail for any potential security concerns, and then performing a standard chroot system call. It should be noted that in order for this functionality to work, commsh needs the setuid root bit, as it is required in order to call the chroot function. This bit is dropped after the chroot is called. The shell itself lives outside of this jailed environment, so it can have the setuid bit without compromising the security of the jail.

The security checks performed can include checks for: setuid root utilities, setuid/setgid utilities, block or character device files, UNIX sockets, FIFO pipes, and mount points. Additionally, a list of allowable block and character device files can be specified. If a security problem is detected in the jail, an error is logged via syslog and the shell exits, disconnecting the remote user.

Implementation limitations

There are several security problems that still exist even when both command restriction and chroot jailing are used. Primarily, there is no sandboxing performed within a community account. This means that jobs submitted by one user of the community could potentially access another user's data. As a related concern, a method must exist by which users can securely access their own data and results.

Auditing and accounting of jobs based on the end user (instead of the community account) is also not considered. It is impossible for the resource administrator to determine exactly which end user is running any given job.

A possible solution to these problems involves the generation of dynamic accounts with a UID selected from a configurable range. If these dynamic accounts are properly associated with an end user, they can be used to solve both the data access problem and end-user auditing problem.

It should be noted that possible solutions to the accounting and auditing problem exist outside the scope of this project. Notably, Martin and Foster have proposed an extension to GRAM that would enable better auditing in the community account scenario. [5]

4. Limitations of chroot jails

Even with a perfect implementation, chroot environments are not perfect sandboxes. We will briefly discuss some of the limitations present when using chroot jails.

It is possible with a poorly constructed jail for a user to "break out" of the jailed environment and access the rest of the system. These compromises are possible if a user within the system can run arbitrary code as the root user or directly access the file system or memory. Both chroot_jail and commsh try to prevent either of these from being possible.

Additionally, chroot jailing only restricts file system access. It is still possible for a job running with in a chroot jail to open network connections to other systems or to over-use memory or CPU (if allowed by the operating system). These can all present potential system compromises that must be addressed separately.

Chroot jailing is also not supported by most job managers. For example, GRAM, the Globus standard for remote job launching, is not equipped to properly handle chroot environments. Specifically, jobs launched through GRAM are launched directly, not through the user's shell. This bypasses "commsh", thereby bypassing the secure chroot into a jailed environment.

Access to data from within a chroot jail is also a potential problem. For the most part, UNIX systems can mount a file system (or networked file system) in multiple places, and if a file system is mounted within a chroot jail then it is accessible by an application restricted to that jail. However, GPFS [6] has security restrictions that make it difficult or impossible to mount them in multiple locations.

5. Related work

Several other products exist that provide similar functionality or that can be used towards similar goals. Among these are Jailkit, chrsh, Samhain, virtual servers, and kernel modules.

Jailkit [7] is, like chroot_jail, a set of utilities geared towards the construction of jail environments. However, it does not include a database component and so dependency information is not tracked, and as such the clean removal of components is tedious.

Chrsh [8] is a shell wrapper that provides chroot functionality similar to that of commsh. Unlike commsh, chrsh does not perform any security auditing on the jailed environment.

Samhain [9] is a file integrity checker that uses methods similar to chroot_jail to determine if files on a host have been modified. It should be noted that chroot_jail is better able to report inconsistencies in a jail as all official changes to applications, libraries and other system files within the jail will be made by the chroot_jail and therefore will update in the database appropriately. Using Samhain or another external product will generate warnings every time a change is made to a jail, even an approved change.

Virtual servers, such as VMware [10] and Virtual Server [11], provide an alternative way to create secured sandboxes that could be used to house community accounts. Virtual servers represent a trade-off in sandboxing. They create more secured sandboxes than chroot jailing, but this is at the expense of both significant resources and trust relationships. As each virtual server is an independent system, they use RAM and disk space even when no applications are running, and any servers that trust the host server will not, a priori, trust the virtual server.

Kernel modules can allow for very finely-tuned security control on a machine. They can base their control on either the user level or the application level. One example of such a kernel module is AppArmor [12], which comes fully integrated into modern versions of SUSE [13]. However, any such modules are very platform-specific, and customized kernel modifications of this sort have the potential to make future upgrades more difficult.

6. Future work

In addition to the proposed features detailed above, there are several other areas where future work is warranted. One such area is working with the Globus GRAM team to address chroot jailing problems is one such area; another is making use of the Community Authorization Service (CAS) functionality bundled with the Globus Toolkit; and a third is working towards a reasonable accounting and auditing framework for community accounts.

7. Availability

Source code for this product is available for download at the project website [1].

8. Acknowledgements

This work was funded by the NCSA NSF CORE award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

9. References

1. Restricted Community Accounts project website http://security.ncsa.uiuc.edu/research/commaccts
2. Sample chroot_jail configuration http://security.ncsa.uiuc.edu/research/commaccts/chroot_jail/sample.conf
3. Sample commsh configuration http://security.ncsa.uiuc.edu/research/commaccts/commsh/sample.conf
4. Sudoers Manual - Wildcards http://www.courtesan.com/sudo/man/sudoers.html#wildcards
5. Globus Toolkit Extensions for Auditing and Accounting. Stuart Martin and Ian Foster. Work in progress.
6. IBM General Parallel File System (GPFS) http://www.ibm.com/servers/eserver/clusters/software/gpfs.html
7. Jailkit: Jail construction utilities http://olivier.sessink.nl/jailkit/
8. chrsh: Aaron Gifford's chroot jail wrapper http://www.adg.us/computers/chrsh.html
9. Samhain file integrity/intrusion detection system http://la-samhna.de/samhain/index.html
10. VMware Virtual Desktop Infrastructure http://www.vmware.com/
11. Microsoft Virtual Server http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx
12. Novell AppArmor http://forge.novell.com/modules/xfmod/project/?apparmor
13. SUSE Linux http://en.opensuse.org/