tinyPM: Lightweight Policy Machine for Linux

1 Motivation

Enterprise computing applications aim to provide data services (DSs) to its users. Examples of such services are email, workflow management, and calendar management. NIST Policy Machine (PM) [4] was proposed so that a single access control framework can control and manage the individual capabilities of the different DSs. Each DS operates in its own environment which has its unique rules for specifying and analyzing access control. The PM tries to provide an enterprise operating environment in which policies can be specified and enforced in a uniform manner. PM was initially developed for enterprise applications and is targeted at Microsoft Windows Server environment where the Active Directory is responsible for user authentication and authorization. The PM follows the attribute-based access control model and can expressive a wide range of policies that arise in enterprise applications and also provides the mechanism for enforcing such policies.

We wanted to develop a lightweight version of the PM, which we term *tinyPM*, that is suitable for the Linux environment where applications need access to system resources. Traditionally, applications running on the Linux were given super user or root privileges. However, this is not acceptable for reasons of security. Applications containing malicious codes or trojan horses may compromise the system if they have super user privileges. Towards this end, the later versions of Linux kernel supported the notion of capabilities [1] which advocate breaking up the super user's privileges into sets of meaningful privileges. Applications can be given capabilities on a need-to-know basis, making the overall system less vulnerable.

Our tinyPM is a policy management utility that can be executed by the super user or by a special account created by the super user who is permitted to change the capabilities of other application binaries. tinyPM controls access to resources required by an application in a transparent manner without the application being aware of its existence. Thus, a rogue application who wants control of the operating system resources cannot gain root privileges and compromise the system. The application will only get the privileges bestowed upon it by the tinyPM utility.

Our aim is to deploy tinyPM in Linux servers or for new emerging Linux applications that makes use of the Docker Engine [3]. In the Docker model, the application, its libraries and binaries are packaged in a container that can execute in any Linux environment. Initially, such containers are given limited set of capabilities [6] and more capabilities are conferred on a need to know basis. Our tinyPM is ideally suited for such

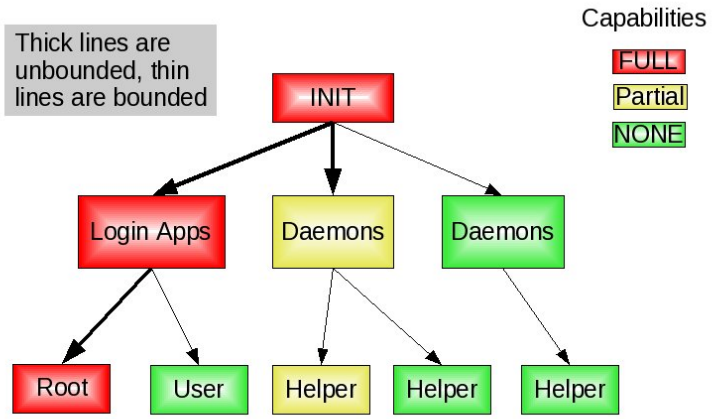

Figure 1: Linux Capabilities Model

environments. We have implemented an initial prototype to demonstrate the feasibility of our approach.

2 Background

In the traditional UNIX or Linux environment, we have a super user who has complete access to the system resources. This administrative identity does not have to abide by the access control rules of any application and has the freedom to read or update user's data. Moreover, this entity can perform many privileged actions, such as, binding a process to a port, loading kernel modules, mounting and unmounting file systems, and a variety of other system activities. Traditionally, applications developed for Linux often executed with the super user privilege and had more access than they really needed. This makes the system inherently vulnerable as trojan horses embedded in the applications may exploit the super user privileges and cause security breaches.

In version 2.1 of the Linux kernel, the notion of capabilities [2] were added to eliminate the dependence on the root account for certain actions. The new model [1, 5] introduced a separation of root privileges into capabilities. These capabilities break the super-user's privilege into a set of meaningfully separable privileges. In Linux, for instance, the ability to switch UIDs is enabled by CAP_SETUID while the ability to change the ownership of an object is enabled by CAP_CHOWN. Figure 1 describes how the capabilities can be given to the various applications. As of the 2.6 and 3.0+ kernels, this functionality has been successfully incorporated into the Linux kernel distributions.

The new emerging concept of application containers such as Docker service [3] heavily leverages the Linux capabilities model [1, 5]. Docker allows an application and its dependencies to be packaged in a virtual container that can be executed on any Linux

server. In contrast, the virtual machine (VM) model requires more resources as it needs the application, its dependencies, and the operating system to be packaged. In Docker, the application runs as an isolated process in the userspace of the host operating system sharing the kernel with other containers. Thus, the application enjoys the resource isolation and allocation of VMs but is much more portable and efficient.

Docker starts containers with a restricted set of capabilities. Thus, instead of giving an application root or non-root privileges, capabilities provide more fine-grained access control. For example, a web server normally runs at port 80. To start listening on one of the lower ports (below 1024), the web server would need root permissions. Now this web server daemon needs to be able to listen to port 80, however it does not need access to kernel modules. Instead of giving this daemon all root permissions, we can set a capability on the web server binary, like CAP_NET_BIND_SERVICE. With this specific capability it can open up port 80 (and 443 for HTTPS) and listen to it, like intended, without getting access to the kernel modules.

3 tinyPM Architecture

tinyPM is the lightweight management subsystem (written in Java) developed with the goal of making the management and tracking of application-level Linux capabilities easier. It is best suited for server-based applications and applications deployed in containers where each container is dedicated to servicing a single type of application in a security constrained environment. Most common examples include server-based containers to serve various system and user oriented services such as HTTP web/application servers, DNS server, NTP server, and NFS server. tinyPM allows one to keep track and update the privileges for the desired applications in an easy manner.

tinyPM as a management utility must be run by the privileged user account. It is owned and invoked by the super user account or a privileged user account created by the superuser that has the permission to change the CAP_SETFCAP capability in the tinyPM package binary. This capability allows the executable to change the capabilities of other applications binaries. So no other account except the privileged user is allowed to execute tinyPM to manage the capabilities.

Figure 2 illustrates how tinyPM can be deployed. tinyPM consists of the following components:

- Shell – provides the command-line-interface to input the policy commands
- Parsing engine – parses and processes the models given at the command-line-interface
- Enforcer – libcap wrapper module that is invoked by the parser to call the libcap system functions to manage Linux capabilities on a particular application
- Database – stores the capabilities privilege models on disk and allows one to perform CRUD (Create/Read/Update/Delete) operations on them.

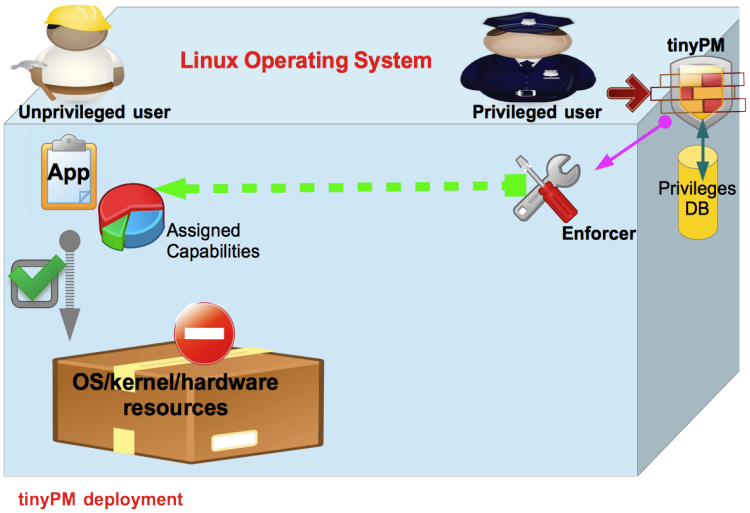

Figure 2: tinyPM deployment

The following commands are currently implemented to manage and enforce the Linux capabilities model from tinyPM command-line shell interface:

- EXIT
- HELP
- SHOW_APP_POLICIES
- ADD_APP_POLICY
- DELETE_APP_POLICY
- COUNT_APP_POLICIES

4 Future Directions

One of our future goals is to have a distributed tinyPM where application capabilities could be deployed, managed, and tracked on a cloud like installation with Linux servers located on the network. In the distributed tinyPM architecture, we need to address issues related to scalability, authentication, and access control in the case of a large cloud-based (server-farm) deployments. We also need to design and implement a distributed coordination protocol that will allow the orchestration and coordination of a multitude of tinyPM installations across a server farm. We must provide a representation that allows one to express, analyze, and enforce non-local policies in a distributed manner. In the future deployments, we may also want to group a set of applications and manage the groups' capabilities, and also enforce capabilites with temporal restrictions.

References

[1] Capabilities. Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/capabilities.7.html`, 2015. accessed 18-March-2015.

[2] Capabilities. POSIX Capabilities & File POSIX Capabilities. `http://www.friedhoff.org/posixfilecaps.html`, 2015. accessed 18-March-2015.

[3] Docker. What is Docker? `https://www.docker.com/whatisdocker/`, 2015. accessed 18-March-2015.

[4] D. F. Ferraiolo, V. Atluri, and S. I. Gavrila. The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement. *Journal of Systems Architecture – Embedded Systems Design*, 57(4):412–424, 2011.

[5] S. E. Hallyn and A. G. Morgan. Linux Capabilities: Making them Work. In *Proceedings of the Linux Symposium*, pages 163–172, Ottawa, Canada, July 2008.

[6] Docker Security. Linux Kernel Capabilities. `https://docs.docker.com/articles/security/#linux-kernel-capabilities`, 2015. accessed 18-March-2015.