CA Technologies
CA Privileged Identity Manager
Security Research Whitepaper

Contents

Introduction

Privileged Identity Management (PIM) is an approach for reducing risk and securing superuser accounts. These accounts are required in every IT organization for performing system administrator tasks, such as installing software packages or performing database maintenance. CA Privileged Identity Manager (CA PIM), formerly CA ControMinder, is an enterprise software solution that provides privileged access and account management through a variety of controls, such as privileged password management, fine-grained access controls, user activity reporting, and cross-platform authentication bridging. CA Technologies approached Gotham Digital Science (GDS) to perform security research to substantiate whether CA PIM provides an effective set of controls against attacks that target privileged identities. This GDS Labs Security Research Whitepaper documents the research performed into the *fine-grained access controls* capability implemented within CA PIM.

Security Risks of Privileged Identities

Some of the key security concerns associated with privileged identities include the following:

- **Unrestricted Access** - Instead of delegating only the privileges required for task execution or resource access, organizations assign IT personnel unrestricted administrator access.

- **Lack of Accountability** – Use of a single administrator account is a common practice resulting in multiple employees sharing the same password.

- **Loss of Audit Trail** – Reliability of audit trails and determining who performed what action is extremely difficult when using shared privileged accounts.

Without a strategy to secure privileged identities, organizations are at increased risk to unauthorized access or disclosure (leakage) of sensitive data, fraudulent activities carried out without detection, and non-compliance. Privileged identities are highly coveted by attackers because they are soft targets that provide access to high value enterprise resources and compromise is often difficult to detect. Both the Verizon 2014 Data Breach Investigations Report[1] and the Mandiant 2014 Threat Report[2] highlight compromise of privileged identities as a common element of security breaches.

Typical Attack Patterns

To compromise a privileged identity, a threat actor must gain an initial foothold within the organization. This initial entry point can be accomplished in a number of ways including targeting organization users through phishing attacks, exploiting vulnerabilities within public facing network services and web applications, or accessing a system directly as a malicious insider / administrator. Initial compromise may only provide user-level access and so the next step in the attack plan is often elevating to administrator-level privileges. In other cases, the initial compromise results in immediate privileged access because a network service was configured insecurely to run with elevated privilege or because the victim of a phishing attack is an IT administrator whom the organization inherently trusts with elevated privileges. Privileged identities provide an attacker with everything they need to move laterally and pivot deeper in the network as well as establish persistence for APT style attacks. Look no further than the high profile hacks of Sony in 2014[3] and Target in 2013[4] for real world examples of this.

[1] http://www.verizonenterprise.com/DBIR/2014/reports/rp_Verizon-DBIR-2014_en_xg.pdf
[2] https://dl.mandiant.com/EE/library/WP_M-Trends2014_140409.pdf
[3] http://www.cnn.com/2014/12/18/politics/u-s-will-respond-to-north-korea-hack/index.html
[4] http://krebsonsecurity.com/2014/01/new-clues-in-the-target-breach/

CA Privileged Identity Manager

The security risks related to privileged identities are real; so what can organizations do to secure them? There are tactical and strategic solutions that can be implemented to reduce risk and provide protection. An effective approach is Privileged Identity Management, which is most simply defined as the monitoring and protection of superuser accounts in an organizations IT environment[5]. CA Privileged Identity Manager, a leading enterprise software vendor in the PIM space, is a unified platform of capabilities that provides privileged access and account management across the enterprise. IT organizations can leverage CA PIM to strengthen existing controls, prevent attacks and improve overall monitoring and auditing of enterprise operations.

What follows are the CA PIM security controls that were in scope for this research project that can be implemented to help organizations protect privileged identities:

- ✓ **Fine-Grained Access Controls** – Provides an additional layer of access controls on top of the native operating system. This allows for least privilege access to be configured for privileged accounts. In the event a privileged account is compromised, access to the compromised system will still be restricted.

- ✓ **Granular Audit Logging** – Provides audit logging of actions for tracking privileged accounts, which accurately tracks the user's identity even in shared account scenarios. Combined with enforced fine-grained access controls on privileged identities, this can help with early detection of an account compromise.

- ✓ **Application Jailing** – Provides the ability to enforce fine-grained access controls on applications and processes. This can provide a strong security control for hardening public facing network services. By limiting the system resources that can be accessed by an application it can restrict the resources that an application can access in the event a 0-day vulnerability is exploited.

Additional core capabilities marketed by CA PIM that were outside the scope of this security research paper include:

- ✓ **Shared Account Password Management** – Provides secure storage and access to one-time use passwords for privileged user accounts. Enterprise Shared Account Password Management tools are recommended by Microsoft Trustworthy Computing as a security control of an overall approach for mitigating Pass-the-Hash (PtH) attacks[6].

- ✓ **UNIX Authentication Bridging** – Allows UNIX and Linux users to authenticate using their Active Directory credentials.

- ✓ **Virtualization Security** – Brings Privileged Identity Management, including shared password controls and access controls to virtual machines and hypervisors.

- ✓ **Reporting and Session Recording** – User activity reporting and session recording combined with the granular audit logging feature allows organizations to address corporate policy and compliance requirements.

[5] http://searchsecurity.techtarget.com/definition/privileged-identity-management-PIM
[6] http://www.microsoft.com/en-us/download/details.aspx?id=36036

The following table summarizes common threats that affect privileged identities and the relevant CA PIM countermeasures that were in scope for this research project.

Common Threat Vectors to Privileged Identities	CA Privileged Identity Manager Countermeasures
Network Service Compromise Exploits against external facing network services are a common way to gain entry to a perimeter system in order to pivot into an organization's internal network. This includes entry with a zero-day vulnerability, by running public exploits against unpatched services, or exploiting web application vulnerabilities. **Local Privilege Escalation** Once user level access to a system is obtained (compromised network service, spear phishing, etc), local privilege escalation is needed to gain root/administrator privileges. Privileged identities are useful for establishing persistence, gaining access to user and service credentials, and facilitating pivoting into internal networks. **Privileged User Compromise** Privileged accounts are frequently targeted since they provide the greatest return on investment. If an attacker is able to compromise a privileged identity, there is no need to spend time on escalating privileges. These types of attacks can occur through spear phishing attacks against administrators or compromise of a network service running with elevated privileges and/or permits remote administrator logins. **Malicious Administrator** Typically, certain individuals within an organization must be granted with privileged access to systems. Organizations that assign unrestricted administrator/root access to these individuals or engage in shared administrative account practices are at a much greater risk of the malicious administrator scenario materializing.	**Fine-Grained Access Controls** CA PIM provides a granular access control mechanism that can layer on top of native OS-level file system access controls. CA PIM administrators are able to assign user specific access control lists on various supported resource classes (files, network connections, kernel modules, etc). If the user performs actions in the context of a shared account (i.e. root, administrator), CA PIM tracks the original identity and applies the proper access control policy. This arms an organization with the ability to enforce the principles of "least privilege" and "segregation of duties" even if several accounts have the ability to run commands on the system as a privileged user. **Application Jailing** CA PIM's fine-grained access controls can be utilized in order to create jails for applications. CA PIM provides the ability to create logical users based on the account used to run an application. By applying a restricted policy on the logical user, it is possible to restrict a process to only access files that are required to function properly. This can provide strong mitigation against common web application vulnerabilities or 0-day vulnerabilities against Internet facing services. **Granular Reporting and Auditing** CA PIM's fine-grained access control mechanism enables accurate auditing of actions performed on the system. Auditing is customizable on a per resource basis and it can be configured to log all access requests to specific resources or only log failed access attempts. It is possible to track the user's true identity within the audit records, which helps reduce loss of accountability when administrative accounts are shared between users. Unlike a traditional 'syslog' log file that can be modified by a privileged user, CA PIM access control policies can be setup to restrict audit log access to only PIM auditors.

Research Overview and Summary

CA Technologies approached Gotham Digital Science to perform security research to substantiate whether CA Privileged Identity Manager provides an effective set of controls against attacks that target privileged identities.

Approach and Scope

The objective of the research performed by the GDS Labs team was determining whether CA Privileged Identity Manager (CA PIM) provides effective defenses against common threats and attacks targeting privileged identities. The CA PIM security controls in scope for the research project were the following:

- Fine-Grained Access Controls
- Granular Audit Logging
- Application Jailing

The primary threat agents considered were a malicious privileged user with existing access to a CA PIM protected host as well as an adversary external to the host attacking an exposed network service.

The GDS research team evaluated the CA PIM solution in a dedicated lab environment over the course of four weeks in Q1 of 2015. CA Technologies provided version 12.8 of the CA PIM software and GDS installed the agent software on a CentOS 6 Linux distribution. What follows is a brief description of the research approach employed:

- **Learning PIM and Initial Setup**: The GDS Labs research team started with zero knowledge of the platform and learned about its features and capabilities through setting up common deployment scenarios, reviewing administrator guides, and receiving instruction from CA PIM product support. Additionally, profiling of the relevant agent processes was performed to identify system resources accessed, network communications, etc.

- **Threat and Countermeasure Enumeration**: Research activities were aimed at investigating how CA PIM can be deployed to mitigate common security threat vectors and attacks that target privileged users. The threats and attacks were narrowed to those relevant to the in-scope CA PIM components. Various CA PIM access control policies and configuration settings were identified as potential countermeasures.

- **Solution Mitigation Verification:** Selected validation testing was performed to determine the resiliency of configured CA PIM policies against common bypass techniques and exploits relevant to fine-grained access controls. Additionally, CA PIM's intercepting kernel agent architecture as well as sudo, shell wrapper, and proxy control architectures were compared and evaluated to determine their resiliency to the threats and attacks.

Evaluating enterprise PIM products that compete with CA PIM was outside the scope of this project. Additionally, penetration testing CA PIM was not a primary objective of the security research. Recommendations for furthering the security posture were provided to CA where appropriate.

Summary Results and Highlights

The following summarizes the results and highlights from GDS's research of the CA Privileged Identity Manager platform:

- ✓ CA Privileged Identity Manager's fine-grained access controls provide organizations a simple and flexible security capability when designing a defense-in-depth strategy for protecting privileged identities. When deployed correctly, CA PIM increases the difficulty required to successfully exploit privilege escalation vulnerabilities.

- ✓ CA Privileged Identity Manager's fine-grained access controls are implemented as a kernel interceptor architecture. GDS' research and experience in the field suggests that access controls enforced at the OS kernel-level are less susceptible to bypass when compared to sudo, shell wrapper, and proxy approaches.

- ✓ CA Privileged Identity Manager's "Fine-Grained Access Controls" were capable of preventing the following:
  - User level privilege escalations against vulnerable SUID/SGID applications
  - Common techniques used in popular kernel privilege escalations, such as using address locations from the /proc file system to assist in exploits

- ✓ GDS' testing supported that CA Privileged Identity Manager's "Application Jailing" feature provides stronger jailing capabilities than chroot based defenses. The deployment of this capability can provide an effective mechanism to mitigate zero-day vulnerabilities in core infrastructure services.

- ✓ CA Privileged Identity Manager's "Application Jailing" functionality was capable of preventing the following critical application based attacks given appropriately configured ACLs:
  - Execution of common shell code such as reverse or bind shells delivered through memory corruption or command injection vulnerabilities
  - Local File Include exploits to read any files not explicitly required by the application to function
  - Arbitrary file write exploits in order to gain remote code execution through a web shell

Common Approaches to Fine-Grain Access Controls

Privileged Identity Management solutions offer different approaches to fine-grain access controls to address the security problem of protecting privileged identities. This section provides an overview of the typical approaches, highlighting their strengths and weaknesses. An underlying assumption for these approaches to provide any level of assurance is first for the host to have native OS hardening in-line with best practices.

Sudo

The 'sudo' application allows a permitted user to execute a command as the super user or another user, as specified in the '/etc/sudoers' file on Unix/Linux systems. One of the major limitations with 'sudo' based solutions is they do not support the ability to apply file-based access controls. Therefore, any PIM solution that leverages 'sudo' would need to implement support for handling file based authorization controls.

An additional pitfall with 'sudo' as a fine-grained access control solution is the potential for attacks that result in breaking out from the allowed application in order to gain access to unauthorized applications. The most common form of 'sudo' based breakouts is utilizing functionality within the restricted program that allows the execution of an arbitrary command. The following simple example walks through how this could be performed on the popular Vim file editor.

Sudo Breakout Example

Shown below are 'sudo' privileges set to the 'lowuser' user account.

```
lowuser ALL=(ALL) /usr/bin/vim
```

The configuration is intended to only allow 'lowuser' to execute the '/usr/bin/vim' program with elevated privileges.

```
$ sudo vim
```

However, the Vim application contains functionality that allows the user to execute arbitrary commands by entering the ':! <command name>' into the editor.

Figure 1 – Inside Vim use the ':! /bin/sh' command in order to execute a shell

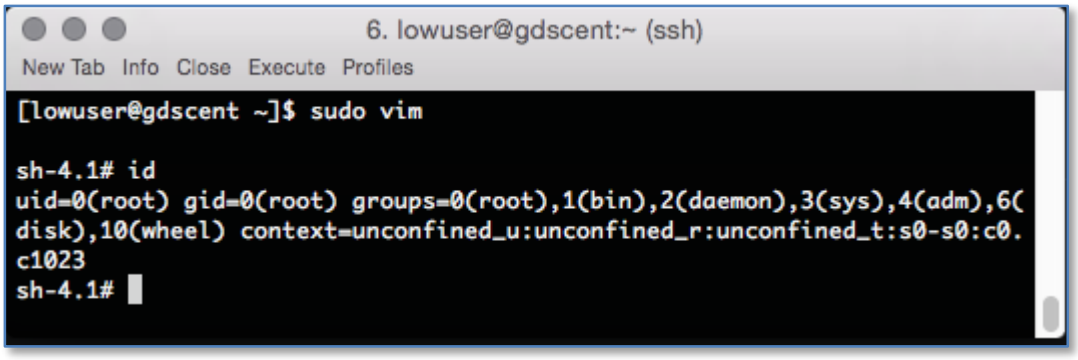

Figure 2 – The result is a root shell and bypass of the sudo access control

More recent versions of 'sudo' provide the ability to configure an allowed process with the 'NOEXEC' setting. The NOEXEC setting prevents the sudo'ed process from executing an additional process. This security control would prevent the Vim breakout exploit shown above. The 'NOEXEC' setting works by loading a dynamic library when the executable is run using 'sudo'. The dynamic library hooks the necessary function calls needed to prevent the process from executing any additional processes.

The NOEXEC control is still susceptible to "breakout" attack vectors if arbitrary reading from and writing to the file system is permitted by the sudo'ed application. Some examples of this include writing a startup script, writing a script as a cron job, modifying a dynamic library, and modifying the 'sudo' configuration file. The lack of file-based fine-grained access control creates a large attack surface for solutions that rely on 'sudo' as the primary form of access control for privileged identities.

Shell Wrappers

The shell wrapper approach allows privileged identities to authenticate directly to the server using a custom shell instead of the typical system provided shells (e.g. /bin/bash, /bin/sh, etc). This custom shell allows the user to be assigned administrative access on the system via an administrative group. The actions that the privileged identity can perform are restricted based on the access controls enforced by the shell wrapper. Shell wrapper solution providers must ensure they are not susceptible to security vulnerabilities based on the validation of submitted commands. The validation of the commands submitted may be enforced through a blacklist or regex based input validation, and may potentially be susceptible to bypass. Additionally, the privileged identity's access must be limited to only execute the shell wrapper so it can adequately provide authorization controls.

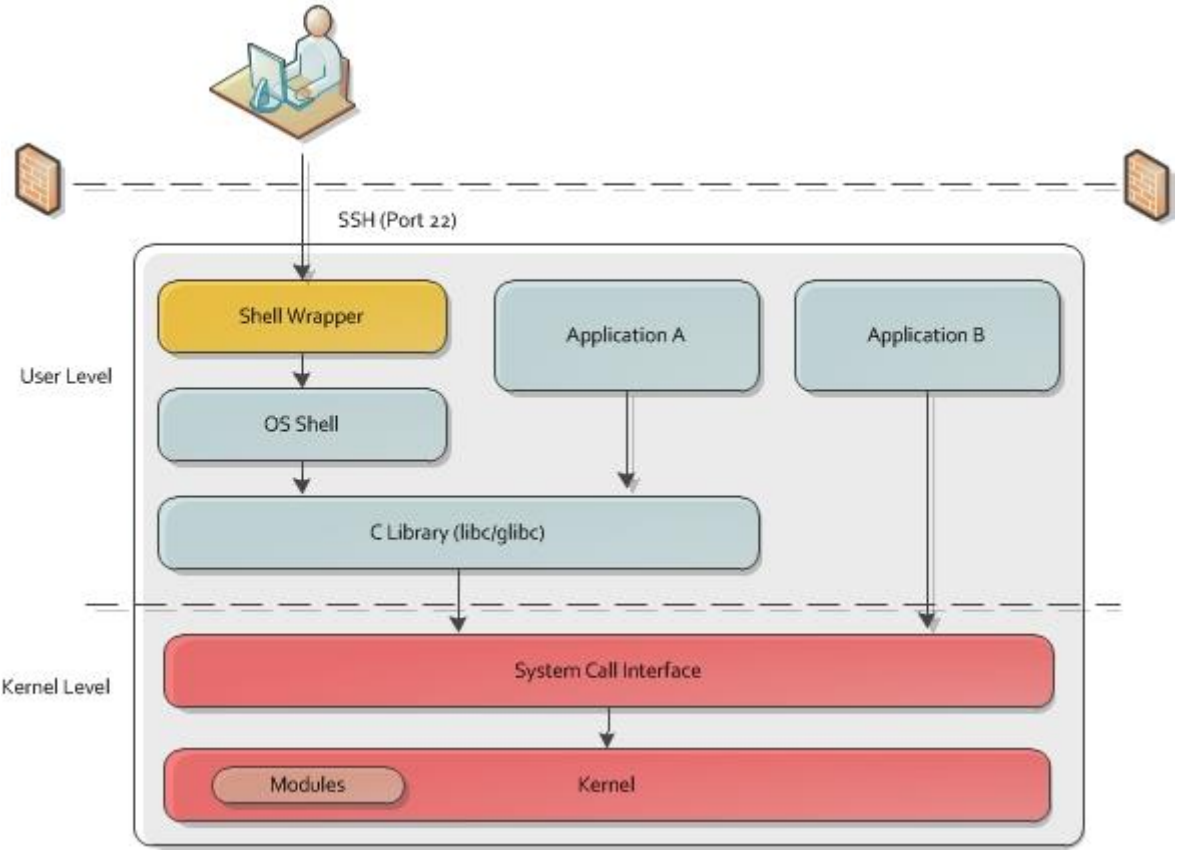

Figure 3 - Access Control via Shell Wrapper

Several security concerns regarding the architecture of the shell wrapper approach to privileged identity management are:

- Security controls are only enforced within the shell wrapper and not in the system's kernel. Therefore the solution would be highly susceptible to the same process breakout attacks that affect sudo.
- Due to security controls only being enforced through the provided shell, it may be possible to access restricted resources using sftp, scp or any other services listening on the host that also authenticate the privileged identity.
- Security controls are only applied on a per-user basis and not across all users of the OS. Therefore, any vulnerability within external facing processes on the server would bypass the wrapped shell.

Proxy Controls

The proxy approach is similar to the shell wrapper approach except the access control checks may not be running on the same system from where the commands are executed. The proxy approach aims to provide a centralized chokepoint for all commands that need to be executed as a privileged identity on a system. The proxy receives commands from the user and makes decisions on whether the user is allowed to perform that command on the server. Any allowed commands are then submitted to the server and executed for the privileged identity.

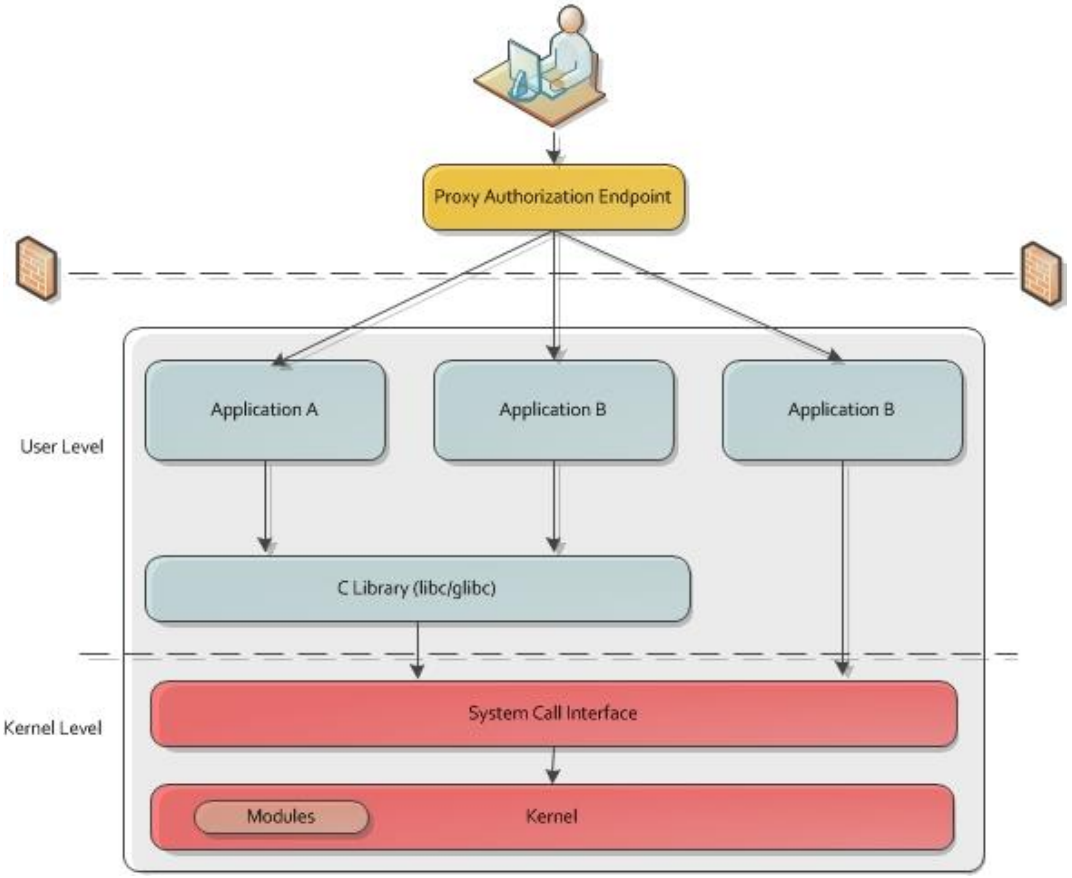

Figure 4 - Access Control via Proxy Controls

The architectural design suffers from the same security concerns as the shell wrapper approach. As long as an attacker is able to gain any form of code execution or file system access, unfettered administrative access on the system is inevitable. The attacker can accomplish this goal by finding an application breakout or a vulnerability in any remotely listening service on the system. By not enforcing security controls in the kernel, the attack surface is still very large for any proxy based approach to provide strong protection as a privileged identity solution.

Kernel Interceptor

In general, access control enforcement at the OS kernel-level is more powerful than hooking in user-land through dynamic libraries since the solution can intercept calls performed by all users and processes on the host. SELinux[7] and GrSecurity[8], are well known open source examples that provide kernel-level security protections for Linux systems. The CA Privileged Identity Manager solution utilizes a kernel intercepting agent architecture[9]. The CA PIM solution runs on UNIX, Linux, and Windows platforms, however only the Linux kernel intercepting agent was in scope for the research.

Essentially, the kernel intercepting agent is a kernel module installed on the host that hooks system calls. The kernel interceptor is not susceptible to user-land bypasses such as runtime hooking or library injection (e.g. LD_PRELOAD) since the system calls can only be modified by code running in kernel mode. The kernel interceptor compared to the sudo, shell wrapper, and proxy approaches provides stronger protection for securing privileged identities because it has the capability to block access at the lowest level. The *Preventing Local Privilege Escalation with CA PIM* and *CA PIM Application Jailing* sections of this whitepaper provide technical examples of how the CA PIM kernel intercepting agent can provide effective security controls for local privilege escalation and application breakout attacks.

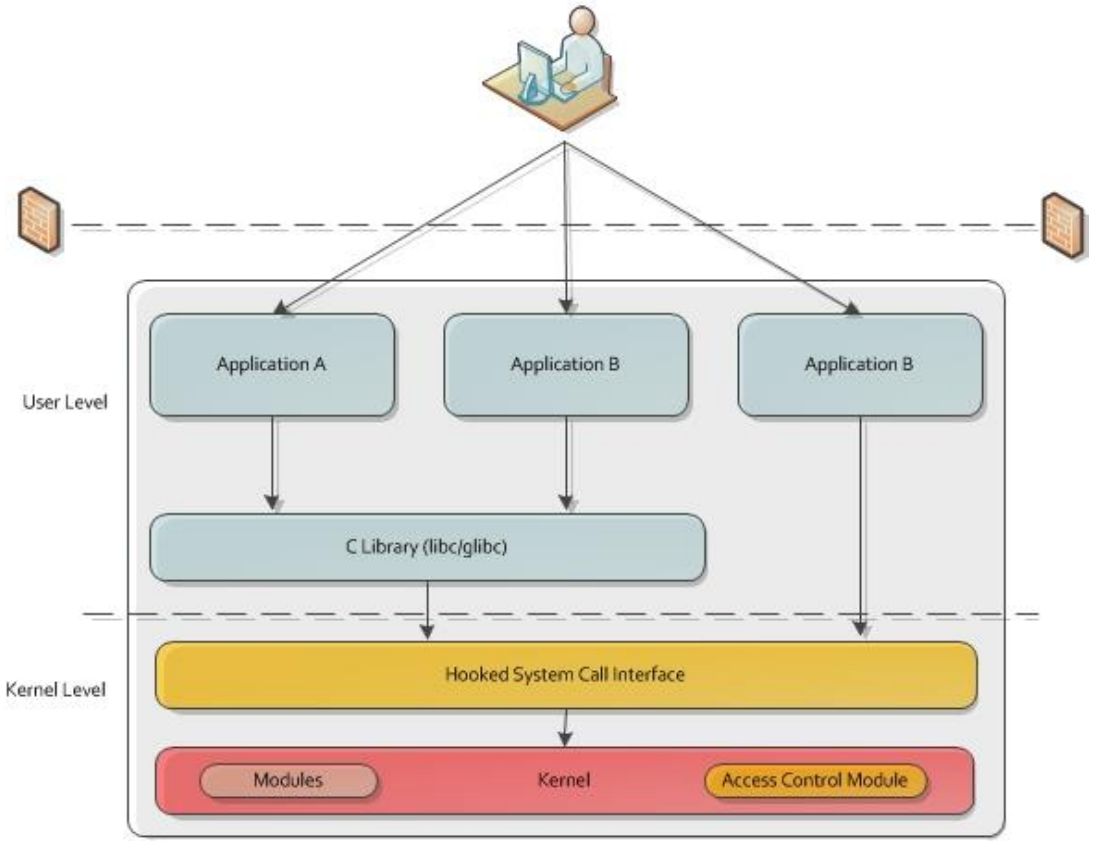

Figure 5 - Access Control via Kernel Interceptor

[7] http://selinuxproject.org/page/Main_Page
[8] https://grsecurity.net
[9] Common approaches to enforcing access control at the kernel level include system call hooking, the Linux Security Module framework, and kernel patching. Comparative analysis of the security benefits and limitations of each was outside the scope of this research project.

Recap – Common Approaches to Fine-Grained Access Controls

This section covered common approaches that privileged identity management solutions may use to enforce access controls against privileged users on hosts. The approaches covered included sudo, shell wrappers, proxy controls and kernel interception. The sudo, shell wrappers, and proxy-based approaches suffer from architectural weaknesses that make application breakouts likely to be exploited. This is due to security controls only being enforced at the top-most levels. In order for a solution to be capable of enforcing access controls in a way that would reduce the likelihood of breakouts, OS kernel-level protections are preferable.

Additionally, the sudo, shell wrapper, and proxy based approaches attempt to enforce their access controls by creating centralized checks for accepting user input. These approaches are not effective in protecting any network services running on a host. Therefore, in the event that a zero-day or un-patched vulnerability is exploited on the host no protection will be provided by those solutions. Kernel interception provides the ability to apply security controls regardless of the user or process that is trying to access the restricted resources.

Any vulnerabilities on the system that can lead to kernel code execution can essentially bypass any access control approach. To realize the full effectiveness of the kernel interceptor approach it is important that system deployments run with up-to-date kernels, avoid the usage of unnecessary kernel modules, and adhere to hardening procedures in-line with security best practices. Refer to *Blocking Public Exploits for Kernel based Local Privilege Escalation Vulnerabilities* for some hardening guidance using the CA PIM fine-grained access controls.

CA Privileged Identity Manager's Fine-Grained Access Controls

CA Privileged Identity Manager (CA PIM) implements a kernel interceptor agent architecture, which allows enforcement of fine-grained access control policies by intercepting system calls on the host in which the agent is installed. This approach reduces CA PIM's attack surface to breakout techniques that would commonly affect privileged identity management solutions relying on sudo, shell wrappers, and proxy approaches for access control enforcement. CA PIM agents can be installed on UNIX, Linux, and Windows systems. This section describes the core components required to run an endpoint agent on a Linux system.

Kernel Module (seos)

When the solution is started, a kernel module named seos is loaded. This kernel module is used to overwrite the pointers stored within the system call table in order to intercept system calls performed by the OS. By performing this interception at the kernel level, the solution can enforce the appropriate configured CA PIM ACLs for any user regardless of the application they are using.

Authorization Daemon (/opt/CA/AccessControl/bin/seosd)

The CA PIM authorization daemon is responsible for making the runtime decisions required to grant or deny access to resources. These decisions are made based on policies stored within the policy database defined using their 'selang' language or through the enterprise administration web console. The authorization daemon initiates and maintains a connection to the Watchdog and Agent daemons to ensure they are alive. If either of the processes are killed, the authorization daemon will restart the processes.

Watchdog Daemon (/opt/CA/AccessControl/bin/seoswd)

The CA PIM Watchog daemon monitors programs and files that are configured to be "trusted" through CA PIM policies. This is performed by validating a checksum of the file. The daemon performs periodic scans of these files to ensure they have not been tampered with in any form by validating the checksum, and sending alerts if they have been tampered. The Watchdog process additionally keeps track of the authorization daemon (seosd) to ensure it is still running. If the authorization daemon is killed, the watchdog daemon will restart the process.

Agent Daemon (/opt/CA/AccessControl/bin/seagent)

Listens on: TCP Port 8891 or 5249 (SSL)

The Agent daemon accepts requests from remote and local systems in order to apply CA PIM policies. The daemon is responsible for accepting the connections, authenticating the user and also writing the policies to the database. The agent daemon additionally checks to ensure that the watchdog daemon is running and restarts the process if is not.

Other Notable Components

CA provides web interfaces such as the CA PIM Enterprise Management server and the Endpoint Manager server that can be used to manage CA PIM agent endpoints. There may be additional daemons that may also run on a system depending on features enabled (e.g. ReportAgent, Policyfetcher, etc).

CA Privileged Identity Manager's Authorization Resource Classes Overview

The CA PIM authorization mechanism is configured by defining resources (e.g. filenames, executable names, IPs, hostnames, etc) and applying user/group specific policies to those resources. The various types of resources supported by the system are referred to as CLASSES. The following list describes a subset of the resource CLASSES that can be protected using CA PIM's fine-grained access controls and were the primary focus of our research:

- **FILES –** Access to files can be protected on a per-user or group basis. The solution supports various access types such as read, write, execute, delete, chown, chmod, etc. CA PIM provides more granular OS access types than those provided by the OS.

- **SPECIALPGM** - Allows special access controls to be defined for a specific application. This can be used to apply access control policies using any of the other support resource CLASSES. By defining application specific authorization controls it is possible to create "application jails" in order to ensure applications can only access resources for which they are provisioned.

- **KMODULE** – Provides the ability to prevent loading and unloading of kernel modules by privileged identities. This is an important security control since the CA PIM access control policies are enforced by intercepting system calls through the seos kernel module. By removing the ability to load kernel modules from privileged identities the attack surface for bypassing the CA PIM solution to access restricted resources is reduced. This is due to the system call table only being capable of being modified through code running in kernel mode.

The following resource CLASSES provide additional layers of security, but were not a primary focus of our research:

- **PROGRAM** – The ability to execute specific applications can be white-listed using this resource class. The solution performs integrity checking of the executable to ensure it has not been modified from the point it was added to the policy. The executable will not be permitted to run if the integrity check fails. The CA PIM solution provides the ability for an administrator to verify an update to the executable and then allow the application to be trusted.

- **SECFILE** – Monitors files for any changes and generates an alert if any changes to the file are identified. This functionality is similar to file integrity functionality found in host-based intrusion detection systems such as Tripwire or OSSEC.

- **SUDO –** Allows scripts to be defined to specify commands and arguments that a user would be permitted or restricted to execute using the 'sesudo' CA PIM utility. The 'sesudo' utility is a replacement for the system's 'sudo' utility in order to have the ACLs managed through CA PIM rather than the '/etc/sudoers' configuration file.

- **CONNECT and TCP** – The CONNECT and TCP classes allow user specific TCP/IP access controls to be created. This can be used for defining whether a specific privileged user or a jailed application should not have the ability to access any external hosts. Since these controls are be applied at a user/process level they can provide more flexibility than if a firewall rule was applied on the system.

CA Privileged Identity Manager's Audit Logging

CA PIM provides granular configuration of audit logging for each resource defined. The following audit levels are supported:

- All – Log all access requests
- Success – Log all granted access requests
- Failure – Log all denied access requests (default)
- None – Disable logging

The PIM policy below demonstrates a simple ACL applied to the '/secret/secret.txt' file that will deny access to all user's except for 'bob'. Since the audit level is set to 'failure', all failed access attempts to read or write to the file will be logged by PIM.

```
AC> newres file /secret/secret.txt owner(<PIM_ADMIN>) defaccess(none) audit(failure)
AC> authorize file /secret/secret.txt uid(bob)
```

The audit log can then be accessed through using the 'seaudit' utility or through the CA PIM management web interfaces. The 'seaudit' command provides various switches and options that can be used to query specific data from the logs based on the resource class, name, user, time and more. The following command is an example of how a query can be made for any logged events on the '/secret/secret.txt' file. The '-detail' option is used to provide descriptive labels for the logged event.

```
$ seaudit -r FILE "/secret/secret.txt" "*" -detail

15 Mar 2015 22:09:07 D FILE  lowuser  Read  69  2 /secret/secret.txt   /bin/cat
172.16.237.1 (OS user)  root

Event type: Resource access
Status: Denied
Class: FILE
Resource: /secret/secret.txt
Access: Read
User name: lowuser
Terminal: 172.16.237.1
Program: /bin/cat
Date: 15 Mar 2015
Time: 22:09
Details: No Step that allowed access
User Logon Session ID: 55122056:00000162
Audit flags: OS user
Effective user name: root
```

The output above shows interesting characteristics regarding the events logged:

- The IP used to access the system is logged in the "Terminal" field. In this example, the 'lowuser' account used SSH from IP 172.16.237.1 in order to access the system.
- The Program used to access the file is logged. In this example, the '/bin/cat' program was used
- The log event contains the user's true identity as well as their effective user name. In this example, the lowuser used 'su' to change their identity to the root user prior to trying the read the file. Although the user was root, CA PIM still tracks the original login name and enforces the ACLs accordingly. Therefore, access to the '/secret/secret.txt' file was denied even though the user was logged in as root.

Preventing Common Bypass Techniques

The following list outlines some of the primary security controls within the CA PIM solution that are worth noting since they can be used to prevent common attacks against fine-grained access control solutions:

- ✓ **Blocking Kernel Module Loading and Unloading**

  The solution provides the 'KMODULE' resource class in order to prevent privileged users from gaining code execution in kernel mode, which could lead to a bypass of the solution access controls. If an attacker is able to achieve Kernel code execution all security controls within the CA PIM solution would be susceptible to bypass or be disabled. It should be noted that this feature is not enabled by default, but should be used in order to provide effective access controls to privileged users or jailed applications. The loading or unloading of modules can be enforced using the CA PIM policy shown below. The policy below restricts the ability to load modules across all host users except for the system administrator that is explicitly allowed:

```
editres KMODULE _default defaccess(none) audit(failure)
authorize KMODULE _default access(load, unload) xuid(<PIM ADMIN>)
```

- ✓ **Enforcing File Based Access Controls on Inodes**

  Access controls defined on files are not enforced simply on the name and path of the file. Linux/Unix file systems utilize a data structure known as index nodes (typically referred to as inodes) that contain attributes of a file that include the raw disk location. The advantage of enforcing authorization controls based on the inode instead of only the filename is that it can prevent common bypass techniques. Some examples of common bypasses include file links (symbolic and hard), remounting the file system, and using the 'chroot' system call in order to change the file path.

- ✓ **Preventing Raw Device Access**

  The solution provides a configuration option to restrict access to the 'mknod' system call. This security control is important since a common bypass techniques for file ACLs would be to create a new device node for a disk that contains the protected data. Since this new device node is not protected under the ACL, the attacker can gain raw device access to the disk and read the protected data. The 'mknod' system call can create a new device file by simply specifying the correct major and minor numbers. This can be leveraged in order to bypass ACLs and perform raw reads or writes to the disk. The 'file_rdevice_max' configuration option can be set within the 'seos.ini' configuration file. By setting this configuration option, calls to the 'mknod' system call will be blocked if an access control policy is set on a device (/dev/sda*, etc).

Recap – CA Privileged Identity Manager's Fine-Grained Access Controls

In this section, CA PIM's fine-grained access control policies were introduced to demonstrate the variety of resources classes that can have access controls applied. These resources can include files, kernel modules, executables and TCP connections. The granular audit logging provided by CA PIM can be used to log resource access grants and failures in order to supplement the security monitoring performed by an organization. These audit logs can lead to early detection of attacks to prevent an attacker from gaining a foothold in your organization network or compromising sensitive data.

CA PIM provides security controls to prevent common weaknesses found in other access control solutions such as restricting kernel modules, Inode based ACLs, and restrictions on raw device access. It should be noted the kernel and 'mknod' security controls must be configured within deployments since they are not enabled by default.

Preventing Local Privilege Escalation with CA Privileged Identity Manager

Introduction to Local Privilege Escalation Exploitation

Local Privilege Escalation (LPE) exploits allow for lower privileged system users to elevate their access on a host to a more privileged account. Exploiting LPE vulnerabilities typically require a user to already have code execution on the vulnerable host. Normally, an attacker must first compromise an external facing service to be in a position to attempt LPE. The external facing service would likely be running as a non-privileged user and limited access to the host would be available in the event an attacker compromises it. The attacker's next goal is to analyze the compromised host to identify vulnerabilities that could lead to a LPE exploit in order to gain full control of the compromised system.

On Unix/Linux based systems these vulnerabilities are commonly exploited through the following approaches:

- Gaining code execution within kernel code with user-land interfaces
- Gaining code execution within a SUID or SGID executable
- Arbitrary file read or writes within kernel or privileged processes

Increasing Difficulty in Exploiting Local Privilege Escalation Vulnerabilities

CA Privileged Identity Manager provides the ability to track a user's identity even though calls to 'setuid' or 'setreuid' system calls are performed by an application. This is very important since these calls are commonly found within SUID applications which allow a user's identity to be changed to the owner of the executable if set with a SUID permission bit. A common example of a SUID application is 'su', which allows a user to execute a shell as another user on the system. CA PIM is capable of tracking the user's identity, and applying the appropriate access controls and performing audit logging.

The following example demonstrates these features by using an executable owned by root and set with the SUID bit. Vulnerabilities within SUID/SGID executables are commonly targeted to perform privilege escalations. Executables with an SUID or SGID bit set allow for the executable to run with the permissions of the owner or group of the file rather than the user that ran the executable. Therefore, if a vulnerability can be exploited within an SUID/SGID application it could lead to a privilege escalation on the system.

User-level Local Privilege Escalation Example Walkthrough

A SUID test application was created to verify that the CA PIM solution is able to properly track the user's identity after a call to 'setreuid' is performed by the application. This test application demonstrates what would occur if a local privilege escalation vulnerability was successfully exploited on a SUID executable. The 'setreuid' function changes the user identity in which the application code will run. The kernel enforces authorization controls on this function so that the real user identity can only be changed to the owner of the file if the SUID permission bit is set on the executable. Within the SUID application, the developer must perform a call to 'setreuid' or similar function in order to set the "Effective User ID" (owner of the executable) to be the "Real User ID". Once the critical code is executed, the developer can revert back to the original user's uid for security reasons.

The following proof of concept code uses the 'popen' function to execute the "/usr/bin/whoami" utility to retrieve the name of the user running the process. The application then calls the CA PIM "/usr/bin/sewhoami" utility to see the name of the user's true identity. This true identity is used by CA PIM in order to enforce ACLs and audit logging. Lastly, the application attempts to read a CA PIM protected "/secret/secret.txt" file to verify whether the proper resource ACLs are applied to the user.

Filename – suid_poc.c

```
#include <stdio.h>
#include <stdlib.h>

void run_command(char* command)
{
        char buf[1000];
        FILE* fp = popen(command, "r");

        if(fp == NULL) {
                fprintf(stderr,"Failed to run command\n");
                exit(1);
        }

        while(fgets(buf, sizeof(buf), fp) != NULL) {
                printf("%s", buf);
        }
}

void do_setuid(void)
{
        int status;
        uid_t ruid = getuid();
        uid_t euid = geteuid();

        printf("Real UID: %d,  Effective UID: %d\n\n", ruid, euid);
        status = setreuid(euid, ruid);
        if(status < 0) {
                fprintf(stderr, "Could not set uid.\n");
                exit(status);
        }
}

int main(int argc, char* argv[], char* envp[])
{
        do_setuid();

        printf("Output from the '/usr/bin/whoami' command\n");
        run_command("/usr/bin/whoami");

        printf("\n");

        printf("Output from the '/opt/CA/AccessControl/bin/sewhoami' command\n");
        run_command("/opt/CA/AccessControl/bin/sewhoami");

        printf("\n");
        printf("Attempting to read restricted file /secret/secret.txt\n");
        run_command("cat /secret/secret.txt");
}
```

The application is built, the owner is changed to the root user and the SUID bit is set on the executable.

```
$  gcc suid_poc.c -o suid_poc
$  sudo chown root:root suid_poc
$  sudo chmod 4755 suid_poc
$ ls -la
total 28
drwxrwxr-x.  2 ron  ron  4096 Feb 11 18:51 .
drwxrwxrwx. 36 ron  ron  4096 Feb 11 18:51 ..
-rwxrwxr-x.  1 ron  ron    94 Feb 11 18:11 compile_setuid.sh
-rwsr-xr-x.  1 root root 8482 Feb 11 18:51 suid_poc
-rw-rw-r--.  1 ron  ron   966 Feb 11 18:51 suid_poc.c
```

Running the created SUID executable using a limited access privileged user named 'lowuser' produces the following output

```
[lowuser@gdscent suid_poc]$ ./suid_poc
Real UID: 501,  Effective UID: 0

Output from the '/usr/bin/whoami' command
root

Output from the '/opt/CA/AccessControl/bin/sewhoami' command
lowuser

Attempting to read restricted file /secret/secret.txt
cat: /secret/secret.txt: Permission denied
```

The output from the 'suid_poc' application shows that the 'Effective UID' is set to the owner of the file (root) and that the 'whoami' utility was executed under the context of the root user. However, CA PIM was able to still track the identity of the original user ('lowuser') based on the output from the 'sewhoami' utility. Although, 'lowuser' is capable of escalating privileges by compromising a SUID executable, the true user identity is still managed by CA PIM and all of the resource ACLs will be applied properly. The output of the 'suid_poc' application above shows that access to the protected '/secret/secret.txt' file was still denied regardless of the privilege escalation succeeding.

The following log entry can be found using the 'seaudit' utility to show that the failed attempt is still logged containing both the OS and the effective user. This can be very useful for tracking in the event a user attempts to access restricted resources or if an account has been compromised.

```
11 Feb 2015 18:51:30 D FILE         lowuser     Read       69  2 /secret/secret.txt    /bin/cat
(OS user)         root
```

This demonstrates that the CA PIM solution is performing authorization control check at the system call level and is able to accurately track the user's identity even after a privilege identity vulnerability is exploited on the system. Solutions not performing security checks at this level would result in the privilege escalation giving the attacker full access to the system's protected resources.

Managing and Blocking SUID/SGID Executables

The CA PIM solution provides features that can help a system administrator create an inventory of the SUID/SGID application on a host and then block the execution of any new SUID/SGID introduced to the system. Additionally, any changes to the white-listed SUID/SGID application will cause the executable to be blocked until the change is approved by the system administrator. The following walkthrough outlines how this can be performed.

CA PIM provides a utility that scans the system for all SUID/SGID executables and generates a policy file. The following command search for all SUID/SGID executables start at the '/' directory and creates rules contains any files, users and groups required to define the executables as trusted programs.

```
# seuidpgm  -u -g -q -l -f / > suid.txt
```

The rules can be applied by using the 'selang' command as follows:

```
$ selang -f suid.txt
```

Once the SUID/SGID executables are defined as trusted programs, the following CA PIM policy can be used to block all access to any programs by default unless it is registered as a trusted program.

```
AC> chres PROGRAM _default defaccess(none)
```

This security control provides an added protection measure to prevent new SUID/SGID executables from being added to the system and limit the attack surface for privilege escalation vulnerabilities.

Blocking Public Exploits for Kernel based Local Privilege Escalation Vulnerabilities

Kernel based privilege escalation vulnerabilities are especially critical since it allows a malicious user to gain execution of arbitrary code at the kernel level. Therefore, any exploitable kernel vulnerabilities on a host would lead to the ability to bypass the CA PIM access controls as well as any other kernel based security controls.

By reviewing the current landscape of kernel exploit proof-of-concept code publicly available on the internet, it is possible to get a good idea of the common techniques that would be used by non-skilled attackers. These exploits would likely be leveraged for exploiting LPE vulnerabilities if the attacker is not capable of writing their own exploits or modifying an existing exploit to a specific kernel or Linux distribution. By analyzing these public exploits for common patterns it is possible to create policies to block the exploits from functioning.

The following table outlines some of the more common Kernel Local Privilege Escalation (LPE) exploits that are publicly available. These exploits have a high probability of being utilized in the wild since they work and are available free on the Internet. A common pattern that was identified within the available exploits was that most relied on reading certain files within the '/proc' file system.

CVE	Kernel Affected	Proof of Concept	/proc/*
CVE-2014-4014	<= 3.13	http://www.exploit-db.com/exploits/33824/	Y
CVE-2014-0196	<= v3.15-rc4	http://www.exploit-db.com/exploits/33516/	Y
CVE-2014-3153	<= 3.14.5	http://www.exploit-db.com/exploits/35370/	Y
CVE- 2013-1959	3.8.x	http://www.exploit-db.com/exploits/25450/	Y
N/A	< 2.6.34	http://www.exploit-db.com/exploits/15944/	Y
CVE: 2010-4073	< 2.6.36.2	http://www.exploit-db.com/exploits/17787/	Y
CVE: 2010-4347	< 2.6.37-rc2	http://www.exploit-db.com/exploits/15774/	Y
CVE: 2010-4258	<= 2.6.37	http://www.exploit-db.com/exploits/15704/	Y
CVE-2010-3904	<= 2.6.36-rc8	http://www.exploit-db.com/exploits/15285/	Y
CVE: 2010-3301	< 2.6.36-rc4	http://www.exploit-db.com/exploits/15023/	Y
CVE: 2010-2959	< 2.6.36-rc	http://www.exploit-db.com/exploits/14814/	Y
CVE: 2010-1146	<= 2.6.34-rc3	http://www.exploit-db.com/exploits/12130/	N

The most common files read within publicly available exploit code are '/proc/kallsyms' and '/proc/ksyms' which contain symbol information of the kernel which is very useful for developing exploits since it contains the locations in memory where data and functions are loaded. This is very convenient for exploit developers since they can make their exploits work across various Linux versions and distributions by simply reading the symbols stored within this file (write once, work everywhere approach). Creating a policy to restrict access to these files will prevent the majority of public exploit code from successfully working. This provides a good reward for a simple configuration option using CA PIM to prevent "script kiddie" attackers that rely on public exploit code.

Disabling access to /proc/* with CA PIM policy

The following commands will disable access to the kernel symbols file as well as any other files within the /proc file system for all users on the system except for the CA PIM system administrator. This account should be protected and not be used to run any services on the system. The /proc file system is commonly utilized in order to read address locations for exploit development and therefore is leveraged by many privilege escalation exploits.

```
AC> editres file /proc/* defaccess(none) audit(all) owner(nobody)
AC> authorize file /proc/* uid(<PIM ADMINISTRATOR USERNAME>) access(all)
```

There are primarily two ways that a more skilled attacker would still be able to exploit a kernel vulnerability. A skilled attacker can utilize additional files or applications accessible in user-land to leak kernel address locations. Some examples of this include access to the system log files that could contain data written by the kernel or utilities such as 'dmesg' that have direct access to the kernel's message buffer. Policies can be created using CA PIM to block these common locations in order to prevent potential information leakage. This is of course not a perfect solution since information leakage vulnerabilities are often found which can lead to kernel address locations to leak out to users. However, this does increase the difficulty in successfully exploiting a kernel based local privilege escalation since an attacker must now chain multiple vulnerabilities.

Disabling access to system log files

The following commands disable access to the system log files that could potentially lead to additional leakage of kernel address information. It is recommended that system log access restrictions are only applied on a per user basis since it could potentially break certain processes which require read or write access to these files.

```
AC> editres file /var/log/* owner(<PIM ADMINISTRATOR USERNAME>) audit(all)
AC> authorize file /var/log/* xuid(<RESTRICTED USER>) access(none)
```

Access to the dmesg utility can be restricted by utilizing a 'sysctl' kernel configuration option. Unfortunately, this option may not be available on every Linux distribution.

```
$ sudo sysctl -w kernel.dmesg_restrict=1
```

Another approach an attacker could use to determine the kernel symbols is by reading the user's specific kernel version and retrieving the address locations it uses from another system. Since these values will remain static, they can be hardcoded into the exploit code. This would be a manual effort and would require a modification of the exploit code in order to be successful. Therefore, it would require a dedicated attacker with the skills required to customize the privilege escalation exploit.

For added protection against these attacks it is recommended that the system utilize Kernel Address Space Layout Randomization (KASLR). By utilizing KASLR the address locations for kernel data and functions are randomized each time the system is rebooted. This prevents an attacker from being able to hard-code address locations within their exploit code and would therefore be forced to find ways to bypass the KASLR implementation. KASLR support is currently limited for Linux and is available in the Linux Kernel version 3.14[10]. Kernel ASLR is not a perfect solution[11] since there are still some drawbacks in its current implementation but it will surely further increase the difficulty of successful exploitation.

[10] http://kernelnewbies.org/Linux_3.14#head-192cae48200fccde67b36c75cdb6c6d8214cccb3
[11] https://forums.grsecurity.net/viewtopic.php?f=7&t=3367

Section Recap – Preventing Local Privilege Escalation with CA Privileged Identity Manager

Providing effective local privilege escalation protection is important for any access control solution. In the event a network facing service or user account is compromised, most attackers aim to escalate their privileges in order to compromise additional accounts on the host or pivot to additional hosts. In this section we discussed user and kernel level privilege escalation vulnerabilities and typical approaches to performing both.

CA PIM is a capable of enforcing access controls on restricted resources even after a user level privilege escalation is performed (e.g. SUID binaries). The solution is capable of tracking a user's true identity when 'setuid' and 'setreuid' system calls are performed. Therefore, even though a user's identity is changed to be the system root user, the access controls for the original identity are still enforced and audit logs reflect the original user appropriately.

For kernel privilege escalation vulnerabilities, CA PIM can provide access controls that can make exploitation more difficult and also block public exploits based on techniques identified in public exploits. Since, kernel exploitation can lead to a bypass of any kernel based security controls, it is recommended that systems are kept up to date on kernel security patches. Policies and additional host hardening should be applied to prevent information leakage of kernel addresses. Consider applying security controls like Kernel ASLR if supported to increase the difficulty of exploiting kernel vulnerabilities.

CA Privileged Identity Manager Application Jailing

Introduction to Application Jailing

Application Jailing is a concept used for executing applications within a restricted environment in order to mitigate further system compromise in the event a vulnerability is exploited in the jailed application. This is commonly performed through virtualization (Linux Containers), 'chroot' jails, or using a fine-grained access control system enforced at the kernel-level like CA Privileged Identity Manager. The limitations of application jailing are that the ACL restrictions that can be enforced must still allow access to resources required by the application in order to function properly. Therefore, if the application requires access to a shell or a sensitive system file it would be possible for an exploit to still gain access to these resources.

Application Jailing with CA Privileged Identity Manager

CA PIM fine-grained access controls provide the ability to create jails for applications. The CA PIM application jail functions by creating logical users and then using the SPECIALPGM resource class in order to assign authorization controls that can be applied for a specific application. This allows access to certain files to only be possible through a specific application. Just like the other jail solutions, it creates an additional barrier for an attacker to bypass in the event a vulnerability is exploited in an external facing service.

The following steps walk through how an application jail can be created for any process. The following example is for a small deployment of the Apache HTTP web server. The CA PIM policy statement below shows how a logical user can be created for the "/usr/sbin/httpd" application which is the primary daemon used for running the Apache HTTP web server.

```
editusr (apacheuser) logical owner(nobody) audit(fail,loginfail)
editres SPECIALPGM ("/usr/sbin/httpd") seosuid(apacheuser) unixuid(*)
```

The "apacheuser" logical user can now be referenced when restricting or allowing access to resources as it typically would. To create a strong application sandbox ensure the logical user is blocked access to all files on the system by default. Any files that are required by the HTTP server in order to function properly must be explicitly allowed through CA PIM policies. The blocking of all files by default can be achieved by adding the "apacheuser" account to the "_restricted" group and also modifying the '_default' file access to the "apacheuser" to be "none".

```
join (apacheuser) group(_restricted)
authorize FILE _default uid(apacheuser) access(none)
```

White-list access to files on the system can then be applied to the "apacheuser" logical user. The short example below should not be considered a comprehensive set of ACLs. Depending on the applications running on the server additional ACLs will need to added. The example below uses the GFILE class in order to assign ACLs so a collection of files.

```
# GFILE Example
editres GFILE ("apache_conf") audit(fail)
chres GFILE("apache_conf") mem("/etc/httpd/*")
chres GFILE("apache_conf") mem("/etc/logrotate.d/httpd")

[..snippet..]
editfile ("/usr/sbin/httpd") owner(nobody) defaccess(none) audit(fail)
authorize FILE ("/usr/sbin/httpd") uid(apacheuser) access(read, execute)
```

```
editfile ("/usr/sbin/apachectl") owner(nobody) defaccess(none) audit(fail)
authorize FILE ("/usr/sbin/apachectl") uid(apacheuser) access(read,execute)
authorize GFILE ("apache_conf") uid(apacheuser) access(read, chdir)
authorize GFILE ("apache_logs") uid(apacheuser) access(read,write,chdir)
authorize GFILE ("apache_binaries") uid(apacheuser) access(read,exec)
authorize GFILE ("apache_rt") uid(apacheuser) access(read, write, chdir, create)

editfile ("/var/www/*") owner(nobody) defaccess(none) audit(fail)
authorize FILE ("/var/www/*") uid(apacheuser) access(read,execute,chdir)
..snip..
```

The initial setup of an application jail can be a trial and error effort until the proper access controls are assigned due to the white-list approach taken. When creating the rules for the jail it was recommended to use the 'warning' option. The warning option will cause any access violations to be logged but not blocked. This can be very useful to perform runtime analysis by running the application and reading the audit logs for all the access errors that occurred while running. An alternative method is to use runtime analysis tools such as 'strace' to monitor the files accessed based calls to file opens.

Application Jailing Guidelines

The creation of the ACLs should be performed carefully ensuring that a least privilege approach is taken. This will reduce the potential for gaining access to system resources that are not explicitly required by the application in order to function properly. The following guidelines should be considered when creating the ACLs:

- ✓ Utilize a whitelist approach when configuring application jail ACLs. Writing black-list policies to block specific access to application files and executables is prone to bypass.

- ✓ Use a least privilege approach when defining ACLs. The CA PIM solution provides granular access options such as read, write, create, exec and more. Ensure only the necessary access to a file is given when defining ACLs for a jail.

- ✓ Do not allow write access to any files that may be executed by any other process or system user. This can be used for an attacker to overwrite an executable that will then be executed by another user or process and therefore bypass the application jail.

Other Approaches to Application Jailing

Chroot Jails

Chroot jails function by using the 'chroot' system call in order to change the root directory on a per process basis. A jail is created by specifying a new root directory that contains all of the files the application needs for the application to run. Therefore, if a vulnerability is exploited in an application, the attacker would only be able to access files stored within the 'chroot' directory and not any other files used by the system. Chroot jails suffer from several weaknesses from both an implementation and security standpoint which could lead to this approach for application jailing to not be optimal for certain use cases. The following outlines some of the common weaknesses that affect chroot jail implementations.

- **Difficult to Setup:** When a directory is selected as the new root for the application/service that will be jailed, it requires a tedious process of copying all required files to the 'chroot' directory. All of the required files within the directory must be copied over in order for the application to run. This process can be time consuming and result in a trial and error process until the application runs properly. There are approaches to using a package manager in order to install applications within the chroot jail. Care must be taken with this approach since installing too many packages increases the attack surface within the jail.

- **Difficult to Maintain:** Since files within the 'chroot' directory are copied over from the main system, any updates to the files through OS patches will not be applied to the versions of the files within the 'chroot' directory. In the event of security patches, system administrators must manually update any libraries or other patched files within the 'chroot' directory.

- **Susceptible to Bypass If Privilege Escalation Achieved:** If an attacker is able to gain code execution as a root user, it could lead to a breakout of the 'chroot' environment is privileged access is gained by the jailed user. Chroot jails are only effective in jailing applications as a non-root user. If a jailed user is able to gain code execution they could target vulnerabilities within SUID executables available within the jail or kernel vulnerabilities on the system in order to escalate their privileges.

Linux Kernel-Level Protections

SELinux, AppArmor, and GrSecurity are some of the more commonly known open source solutions for Linux systems that provide fine-grained access controls at the kernel-level that could also be leveraged to implement application jailing. An advantage of CA PIM over the open source alternatives is that it is supported on UNIX, Linux, and Windows systems. This allows organizations to have a single solution for managing access control policies regardless of the platform. A common complaint from system administrators regarding setup of open source solutions like SELinux is the learning curve required. Our experience with CA PIM was that 'selang' which is used to setup policies has English-like, self-describing syntax which made it easy to learn and quickly setup access control policies for testing.

Section Recap – CA Privileged Identity Manager Application Jailing

This section covered the application jailing access controls that the CA PIM solution provides. CA PIM allows the creation of logical users for programs. Access control policies can then be defined against these logical users. By joining the logical user to the '_restricted' group it is possible to define white-list policies that block access to all files other than the ones required for the application work properly.

The CA PIM application jailing was found to be more effective than 'chroot' jails. Chroot jails are not capable of preventing breakouts if a malicious user is able to gain root access. Since CA PIM provides mitigations against user-level privilege escalation and additionally provides access control against raw device access it is able to prevent common attacks against 'chroot 'jails. Application jailing can be an effective security control, however it is not a replacement for secure coding techniques and instead it should utilized as part of a defense-in-depth strategy.

Conclusions

CA Privileged Identity Manager (CA PIM) provides a simple and flexible fine-grained access control security capability to consider when designing a defense-in-depth strategy for protecting privileged identities. Our research demonstrated that CA PIM's kernel interceptor architecture is less susceptible to bypass when compared to sudo, shell wrapper, and proxy approaches. CA PIM's application jailing feature was found to provide stronger jailing capabilities than chroot based defenses. Additionally, the audit logging provided by CA PIM makes it possible to track the true user's identity when performing tasks as root, which can help in detecting privileged account abuse.

CA PIM policy settings were identified that can provide effective mitigations against user-level privilege escalations and application jailing for network services to prevent exploits against zero-day or un-patched vulnerabilities. The strength of CA PIM fine-grained access controls, application jailing, and auditing capabilities rests with the policies that are designed for a specific deployment. Organizations interested in deploying CA PIM should perform their own software assurance due diligence activities to verify the security posture of a deployment. Because there is no substitute for secure coding techniques or minimum baseline server hardening procedures, CA PIM should be viewed as part of a larger defense-in-depth strategy for reducing the security risks inherent to privileged identities.

About Gotham Digital Science

Gotham Digital Science (GDS) is a specialist security consulting company focused on helping our clients find, fix, and prevent security bugs in mission critical network infrastructure, web-based software applications, mobile apps and embedded systems. GDS is also committed to contributing to the security and developer communities through sharing knowledge and resources such as blog posts, security tool releases, vulnerability disclosures, sponsoring and presenting at various industry conferences. For more information on GDS, please contact info@gdssecurity.com or visit http://www.gdssecurity.com.

US Office (Global Headquarters):

Gotham Digital Science LLC

125 Maiden Lane – Third Floor

New York, NY 10038

United States

UK Office:

Gotham Digital Science Ltd

161 Drury Lane – Fourth Floor

London, WC2B 5PN

United Kingdom

About GDS Labs

Security Research & Development is a core focus and competitive advantage for GDS. The GDS Labs team has the following primary directives:

- Assessing cutting-edge technology stacks
- Improving delivery efficiency through custom tool development
- Finding & responsibly disclosing vulnerabilities in high value targets
- Assessing the impact to our clients of high risk, publicly disclosed vulnerabilities

The GDS Labs R&D team performs security research, with areas of current focus including mobile application security, embedded systems, and cryptography. GDS also participates in many security related organizations and groups such as OWASP. GDS Labs is a value added service that our clients benefit from on virtually every engagement that we perform.