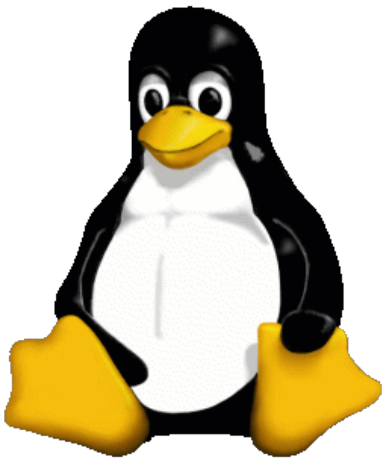
THE LINUX FOUNDATION (/USERS/LFADMIN) | JULY 11, 2013

Overview of Linux Kernel Security Features

Editor's Note: This is a guest post from James Morris, the Linux kernel security subsystem maintainer and manager of the mainline Linux kernel development team at Oracle.

In this article, we'll take a high-level look at the security features of the Linux kernel. We'll start with a brief overview of traditional Unix security, and the  rationale for extending that for Linux, then we'll discuss the Linux security extensions.

Unix Security – Discretionary Access Control

Linux was initially developed as a clone of the Unix operating system in the early 1990s. As such, it inherits the core Unix security model—a form of Discretionary Access Control (DAC). The security features of the Linux kernel have evolved significantly to meet modern requirements, although Unix DAC remains as the core model.

Briefly, Unix DAC allows the owner of an object (such as a file) to set the security policy for that object—which is why it's called a *discretionary* scheme. As a user, you can, for example, create a new file in your home directory and decide who else may read or write the file. This policy is implemented as permission bits attached to the file's inode, which may be set by the owner of the file. Permissions for accessing the file, such as *read* and *write,* may be set separately for the owner, a specific group, and other (i.e. everyone else). This is a relatively simple form of access control lists (ACLs).

Programs launched by a user run with all of the rights of that user, whether they need them or not. There is also a *superuser*—an all-powerful entity which bypasses Unix DAC policy for the purpose of managing the system. Running a program as the superuser provides that program with all rights on the system.

Extending Unix Security

Unix DAC is a relatively simple security scheme, although, designed in 1969, it does not meet all of the needs of security in the Internet age. It does not adequately protect against buggy or misconfigured software, for example, which may be exploited by an attacker seeking unauthorized access to resources. Privileged applications, those running as the superuser (by design or otherwise), are particularly risky in this respect. Once compromised, they can provide full system access to an attacker.

Functional requirements for security have also evolved over time. For example, many users require finer-grained policy than Unix DAC provides, and to control access to resources not covered by Unix DAC such as network packet flows.

It's worth noting that a critical design constraint for integrating new security features into the Linux kernel is that existing applications must not be broken. This is general constraint imposed by Linus for all new features. The option of designing a totally new security system from the ground up is not available—new features have to be retrofitted and compatible with the existing design of the system. In practical terms, this has meant that we end up with a collection of security enhancements rather than a monolithic security architecture.

We'll now take a look at the major Linux security extensions.

Extended DAC

Several of the first extensions to the Linux security model were to enhancements of existing Unix DAC features. The proprietary Unix systems of the time had typically evolved their own security enhancements, often very similarly to each other, and there were some (failed) efforts to standardize these.

POSIX ACLs

POSIX Access Control Lists (http://users.suse.com/~agruen/acl/linux-acls/online/) for Linux are based on a draft POSIX standard. They extend the abbreviated Unix DAC ACLs to a much finer-grained scheme, allowing separate permissions for individual users and different groups. They're managed with the setfacl and getfacl commands. The ACLs are managed on disk via extended attributes, an extensible mechanism for storing metadata with files.

POSIX Capabilities

POSIX Capabilities (http://www.ibm.com/developerworks/linux/library/l-posixcap/index.html) are similarly based on a draft standard. The aim of this feature is to break up the power of the superuser, so that an application requiring some privilege does not get all privileges. The application runs with one or more coarse-grained privileges, such as CAP_NET_ADMIN for managing network facilities. Capabilities for programs may be managed with the setcap and getcap utilities. It's possible to reduce the number of setuid applications on the system by assigning specific capabilities to them, however, some capabilities are very coarse-grained and effectively provide a great deal of privilege.

Namespaces

Namespaces (http://lwn.net/Articles/531114/) in Linux derive from the Plan 9 operating system (the successor research project to Unix). It's a lightweight form of partitioning resources as seen by processes, so that they may, for example, have their own view of filesystem mounts or even the process table. This is not primarily a security feature, but is useful for implementing security. One example is where each process can be launched with its own, private /tmp directory, invisible to other processes, and which works seamlessly with existing application code, to eliminate an entire class of security threats.

The potential security applications are diverse. Linux Namespaces have been used to help implement multi-level security, where files are labeled with security classifications, and potentially entirely hidden from users without an appropriate security clearance.

On many systems, namespaces are configured via Pluggable Authentication Modules (PAM)--see the pam_namespace(8) man page.

Network Security

Linux has a very comprehensive and capable networking stack, supporting many protocols and features. Linux can be used both as an endpoint node on a network, and also as a router, passing traffic between interfaces according to networking policies.

Netfilter (http://www.netfilter.org/) is an IP network layer framework which hooks packets which pass into, through and from the system. Kernel-level modules may hook into this framework to examine packets and make security decisions about them. iptables (http://www.netfilter.org/projects/iptables/index.html) is one such module, which implements an IPv4 firewalling scheme, managed via the userland iptables tool. Access control rules for IPv4 packets are installed into the kernel, and each packet must pass these rules to proceed through the networking stack. Also implemented in this codebase is stateful packet inspection and Network Access Translation (NAT). Firewalling is similarly implemented for IPv6.

ebtables (http://ebtables.sourceforge.net/) provides filtering at the link layer, and is used to implement access control for Linux bridges, while arptables (http://linux.die.net/man/8/arptables) provides filtering of ARP packets.

The networking stack also includes an implementation of IPsec (http://en.wikipedia.org/wiki/IPsec), which provides confidentiality, authenticity, and integrity protection of IP networking. It can be used to implement VPNs, and also point to point security.

Cryptography

A cryptographic API is provided for use by kernel subsystems. It provides support for a wide range of cryptographic algorithms and operating modes, including commonly deployed ciphers, hash functions, and limited support for asymmetric cryptography. There are synchronous and asynchronous interfaces, the latter being useful for supporting cryptographic hardware, which offloads processing from general CPUs.

Support for hardware-based cryptographic features is growing, and several algorithms have optimized assembler implementations on common architectures. A key management subsystem is provided for managing cryptographic keys within the kernel.

Kernel users of the cryptographic API include the IPsec code, disk encryption schemes including ecryptfs (http://ecryptfs.org/) and dm-crypt (http://code.google.com/p/cryptsetup/wiki/DMCrypt), and kernel module signature verification.

Linux Security Modules

The Linux Security Modules (LSM) API implements hooks at all security-critical points within the kernel. A user of the framework (an “LSM”) can register with the API and receive callbacks from these hooks. All security-relevant information is safely passed to the LSM, avoiding race conditions, and the LSM may deny the operation. This is similar to the Netfilter hook-based API, although applied to the general kernel.

The LSM API allows different security models to be plugged into the kernel—typically access control frameworks. To ensure compatibility with existing applications, the LSM hooks are placed so that the Unix DAC checks are performed first, and only if they succeed, is LSM code invoked.

The following LSMs have been incorporated into the mainline Linux kernel:

SELinux

Security Enhanced Linux (http://www.nsa.gov/research/selinux/%E2%80%8E) (SELinux) is an implementation of fine-grained Mandatory Access Control (MAC) designed to meet a wide range of security requirements, from general purpose use, through to government and military systems which manage classified information. MAC security differs from DAC in that the security policy is administered centrally, and users do not administer policy for their own resources. This helps contain attacks which exploit userland software bugs and misconfiguration.

In SELinux, all objects on the system, such as files and processes, are assigned security labels. All security-relevant interactions between entities on the system are hooked by LSM and passed to the SELinux module, which consults its security policy to determine whether the operation should continue. The SELinux security policy is loaded from userland, and may be modified to meet a range of different security goals. Many previous MAC schemes had fixed policies, which limited their application to general purpose computing.

SELinux is implemented as a standard feature in Fedora-based distributions, and widely deployed.

Smack

The Smack (http://schaufler-ca.com/) LSM was designed to provide a simple form of MAC security, in response to the relative complexity of SELinux. It's also implemented as a label-based scheme with a customizable policy. Smack is part of the Tizen (https://www.tizen.org/) security architecture and has seen adoption generally in the embedded space.

AppArmor

AppArmor (https://wiki.ubuntu.com/AppArmor) is a MAC scheme for confining applications, and was designed to be simple to manage. Policy is configured as application profiles using familiar Unix-style abstractions such as pathnames. It is fundamentally different to SELinux and Smack in that instead of direct labeling of objects, security policy is applied to pathnames. AppArmor also features a learning mode, where the security behavior of an application is observed and converted automatically into a security profile.

AppArmor is shipped with Ubuntu and OpenSUSE, and is also widely deployed.

TOMOYO

The TOMOYO (http://tomoyo.sourceforge.jp/wiki-e/?Welcome!) module is another MAC scheme which implements path-based security rather than object labeling. It's also aimed at simplicity, by utilizing a learning mode similar to AppArmor's where the behavior of the system is observed for the purpose of generating security policy.

What's different about TOMOYO is that what's recorded are trees of process invocation, described as “domains”. For example, when the system boots, from init, as series of tasks are invoked which lead to a logged in user running a shell, and ultimately executing a command, say ping. This particular chain of tasks is recorded as a valid domain for the execution of that application, and other invocations which have not been recorded are denied.

TOMOYO is intended for end users rather than system administrators, although it has not yet seen any appreciable adoption.

Yama

The Yama LSM is not an access control scheme like those described above. It's where miscellaneous DAC security enhancements are collected, typically from external projects such as grsecurity (http://grsecurity.net/).

Currently, enhanced restrictions on ptrace are implemented in Yama, and the module may be stacked with other LSMs in a similar manner to the capabilities module.

Audit

The Linux kernel features a comprehensive audit subsystem (http://people.redhat.com/sgrubb/audit/), which was designed to meet government certification requirements, but also actually turns out to be useful. LSMs and other security components utilize the kernel Audit API. The userland components are extensible and highly configurable.

Audit logs are useful for analyzing system behavior, and may help detect attempts at compromising the system.

Seccomp

Secure computing mode (seccomp) is a mechanism which restricts access to system calls by processes. The idea is to reduce the attack surface of the kernel by preventing applications from entering system calls they don't need. The system call API is a wide gateway to the kernel, and as with all code, there have and are likely to be bugs present somewhere. Given the privileged nature of the kernel, bugs in system calls are potential avenues of attack. If an application only needs to use a limited number of system calls, then restricting it to only being able to invoke those calls reduces the overall risk of a successful attack.

The original seccomp code, also known as "mode 1", provided access to only four system calls: read, write, exit, and sigreturn. These are the minimum required for a useful application, and this was intended to be used to run untrusted code on otherwise idle systems.

A recent update (http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/prctl/seccomp_filter.txt?id=HEAD) to the code allows for arbitrary specification of which system calls are permitted for a process, and integration with audit logging. This “mode 2” seccomp was developed for use as part of the Google Chrome OS.

Integrity Management

The kernel's integrity management (http://linux-ima.sourceforge.net/) subsystem may be used to maintain the integrity of files on the system. The Integrity Measurement Architecture (IMA) component performs runtime integrity measurements of files using cryptographic hashes, comparing them with a list of valid hashes. The list itself may be verified via an aggregate hash stored in the TPM (http://en.wikipedia.org/wiki/Trusted_Platform_Module). Measurements performed by IMA may be logged via the audit subsystem, and also used for remote attestation, where an external system verifies their correctness.

IMA may also be used for local integrity enforcement via the Appraisal extension. Valid measured hashes of files are stored as extended attributes with the files, and subsequently checked on access. These extended attributes (as well as other security-related extended attributes), are protected against offline attack by the Extended Verification Module(EVM) component, ideally in conjunction with the TPM. If a file has been modified, IMA may be configured via policy to deny access to the file. The Digital Signature extension allows IMA to verify the authenticity of files in addition to integrity by checking RSA-signed measurement hashes.

A simpler approach to integrity management is the dm-verity (http://code.google.com/p/cryptsetup/wiki/DMVerity) module. This is a device mapper target which manages file integrity at the block level. It's intended to be used as part of a verified boot process, where an appropriately authorized caller brings a device online, say, a trusted partition containing kernel modules to be loaded later. The integrity of those modules will be transparently verified block by block as they are read from disk.

Hardening and Platform Security

Hardening techniques have been applied at various levels, including in the build chain and in software, to help reduce the risk of system compromise.

Address Space Layout Randomization (ASLR (http://en.wikipedia.org/wiki/Address_space_layout_randomization)) places various memory areas of a userland executable in random locations, which helps prevent certain classes of attacks. This was adapted from the external PaX/grsecurity projects, along with several other software-based hardening features.

The Linux kernel also supports hardware security features where available, such as NX (http://en.wikipedia.org/wiki/NX_bit), VT-d (http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices), the TPM (http://en.wikipedia.org/wiki/Trusted_Platform_Module), TXT (http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/malware-reduction-general-technology.html), and SMAP (http://lwn.net/Articles/517475/), along with cryptographic processing as previously mentioned.

Summary

We've covered, at a very high-level, how Linux kernel security has evolved from its Unix roots, adapting to ever-changing security requirements. These requirements have been driven both by external changes, such as the continued growth of the Internet and the increasing value of information stored online, as well as the increasing scope of the Linux user base.

Ensuring that the security features of the Linux kernel continue to meet such a wide variety of requirements in a changing landscape is an ongoing and challenging process.

James Morris is the Linux kernel security subsystem maintainer. He is the author of sVirt (virtualization security), multi-category security, the kernel cryptographic API, and has contributed to the SELinux, Netfilter and IPsec projects. He works for Oracle as manager of the mainline Linux kernel development team, from his base in Sydney, Australia. Follow James on (https://blogs.oracle.com/linuxkernel/)https://blogs.oracle.com/linuxkernel/ (https://blogs.oracle.com/linuxkernel/).