Anatomy of Security-Enhanced Linux (SELinux)

Architecture and implementation

M. Tim Jones

May 17, 2012
(First published April 29, 2008)

Linux® has been described as one of the most secure operating systems available, but the National Security Agency (NSA) has taken Linux to the next level with the introduction of Security-Enhanced Linux (SELinux). SELinux takes the existing GNU/Linux operating system and extends it with kernel and user-space modifications to make it bullet-proof. If you're running a 2.6 kernel today, you might be surprised to know that you're using SELinux right now! This article explores the ideas behind SELinux and how it's implemented.
17 May 2012 - *In response to reader comment, corrected three broken links in Resources: 1) NSA Web site for SELinux, 2) introduction to SELinux policy configuration, 3) openSUSE.*

Introduction

Public networks like the Internet are dangerous places. Anyone who has a computer attached to the Internet (even transiently) understands these dangers. Attackers can exploit insecurities to gain access to a system, to obtain unauthorized access to information, or to repurpose a computer in order to send spam or participate in attacks on other high-profile systems (using SYN floods, as part of a Distributed Denial of Service attacks).

Distributed Denial of Service (DDoS) attacks are orchestrated from many systems across the Internet (so-called *zombie computers*) to consume resources on the target system and make it unusable for legitimate users (exploiting TCP's three-way handshake). For more information on a protocol that removes this ability by using a four-way handshake with cookies (Stream Control Transmission Protocol [SCTP]), see the Related topics section.

The origin of SELinux

SELinux is a product of government and industry, with design and development provided by the NSA, Network Associates, Tresys, and many others. Although it was introduced by the NSA as a set of patches, it's mainlined into the Linux kernel as of 2.6.

GNU/Linux is very secure, but it's also very dynamic: changes can appear that open holes into the operating system that can then be exploited. Although considerable attention is paid to preventing unauthorized access, what happens after an entry has occurred?

This article explores the ideas behind SELinux and its basic architecture. A complete overview of SELinux could be the topic of an entire book (see the Related topics section), so this article sticks to the basics to give you an idea of why SELinux is important and how it's implemented.

Access control methods

Most operating systems use access controls to determine whether an entity (user or program) can access a given resource. UNIX®-based systems use a form of *discretionary access control* (DAC). This method restricts access to objects based commonly on the groups to which they belong. For examples, files in GNU/Linux have an owner, a group, and a set of permissions. The permissions define who can access a given file, who can read it, who can write to it, and who can execute it. These permissions are split into three sets of users, representing the user (owner of the file), the group (all users who are members of a group), and others (all users who are neither members of the group nor owner of the file).

Lumping access controls like this creates a problem because an exploited program inherits the access controls of the user. Thus the program can do things at the user's access level, which is undesirable. Rather than define restrictions this way, it's more secure to use the *principle of least privilege*: programs can do what they need to perform their task, but nothing more. For example, if you have a program that responds to socket requests but doesn't need to access the file system, then that program should be able to listen on a given socket but not have access to the file system. That way, if the program is exploited in some way, its access is explicitly minimized. This type of control is called *mandatory access control* (MAC).

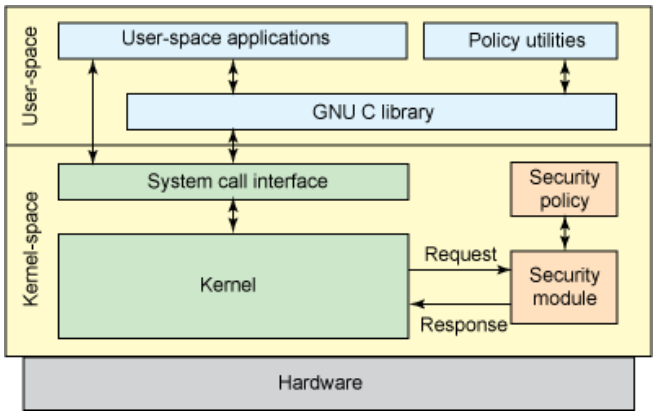
Another approach to controlling access is *role-based access control* (RBAC). In RBAC, permissions are provided based on roles that are granted by the security system. The concept of a role differs from that of a traditional group in that a group represents one or more users. A role can represent multiple users, but it also represents the permissions that a set of users can perform.

SELinux adds both MAC and RBAC to the GNU/Linux operating system. The next section explores the SELinux implementation and how security enforcement was transparently added to the Linux kernel.

Linux security implementation

In the early days of SELinux, while it was still a set of patches, it provided its own security framework. This was problematic because it locked GNU/Linux into a single access-control architecture. Instead of adopting a single approach, the Linux kernel inherited a generic framework that separated policy from enforcement. This solution was the Linux Security Module (LSM) framework. The LSM provides a general-purpose framework for security that allows security models to be implemented as loadable kernel modules (see Figure 1).

Figure 1. Security policy and enforcement are independent using SELinux.



Kernel code is modified prior to accessing internal objects to invoke a hook that represents an enforcement function, which implements the security policy. This function validates that the operation may proceed based on the predefined policies. The security functions are stored in a security operations structure that covers the fundamental operations that must be protected. For example, the `security_socket_create` hook (`security_ops->socket_create`) checks permissions prior to creating a new socket and considers the protocol family, the type, the protocol, and whether the socket is created within the kernel or in user-space. Listing 1 provides a sample of the code from the socket.c for socket creation (see ./linux/net/socket.c).

Listing 1. Kernel code for socket creation

```
static int __sock_create(int family, int type, int protocol,
                         struct socket **res, int kern)
{
        int err;
        struct socket *sock;

        /*
         *      Check protocol is in range
         */
        if (family < 0 || family >= NPROTO)
                return -EAFNOSUPPORT;
        if (type < 0 || type >= SOCK_MAX)
                return -EINVAL;

        err = security_socket_create(family, type, protocol, kern);
        if (err)
                return err;

        ...
```

The function `security_socket_create` is defined in ./linux/include/linux/security.h. It provides indirection from the call `security_socket_create` to the function that's dynamically installed in the `security_ops` structure (see Listing 2).

Listing 2. Indirect call for the socket-creation check

```
static inline int security_socket_create (int family, int type,
                                          int protocol, int kern)
{
        return security_ops->socket_create(family, type, protocol, kern);
}
```

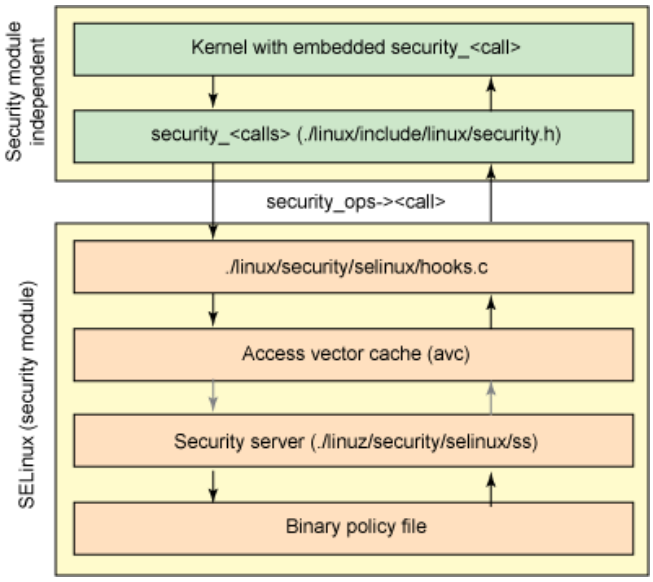
The function in the `security_ops` structure is installed by the security module. In this case, the hooks are defined in the loadable kernel module for SELinux. Each SELinux call is defined within the hooks file that completes the indirection from the kernel function to the dynamic call for the particular security module (see Listing 3 from .../linux/security/selinux/hooks.c).

Listing 3. The SELinux socket-creation check

```
static int selinux_socket_create(int family, int type,
                                 int protocol, int kern)
{
        int err = 0;
        struct task_security_struct *tsec;

        if (kern)
                goto out;

        tsec = current->security;
        err = avc_has_perm(tsec->sid, tsec->sid,
                           socket_type_to_security_class(family, type,
                           protocol), SOCKET__CREATE, NULL);

out:
        return err;
}
```

At the core of Listing 3 is the call to validate that the current operation is permitted for current task (as defined by `current->security`, where `current` represents the currently executing task). The Access Vector Cache (AVC) is a cache of previous SELinux decisions (to increase the process's performance). This call includes the source security identifier (sid), the security class (constructed from the details of the requested operation), the particular socket call, and optional auxiliary audit data. If the decision isn't found in the cache, then the security server is invoked to obtain the decision (this process is shown in Figure 2).

Figure 2. Layered Linux security process



The callback hooks initialized into security_ops are defined dynamically as a loadable kernel module (through `register_security()`) but otherwise contain dummy stub functions in the event that no security module is loaded (see ./linux/security/dummy.c). These stub functions implement the standard Linux DAC policy. The callback hooks exist at all points where object mediation must be provided for security. These include task management (creation, signaling, waiting), program loading (execve), file system management (superblock, inode, and filehooks), IPC (message queues, shared memory, and semaphore operations), module hooks (insertion and removal), and network hooks (covering sockets, netlink, network devices, and other protocol interfaces). You can learn more about the various hooks in the Related topics section or by reviewing the security.h file.

Managing the policies for SELinux is well beyond the scope of this article, but you can find more information about SELinux configuration in the Related topics section.

Other approaches

SELinux is one of the most comprehensive security frameworks available today, but it's certainly not the only one. This section reviews some of the other approaches that are available.

AppArmor

AppArmor, which was originally developed by Immunix and then maintained by Novell, is an alternative to SELinux that also uses the Linux Security Module (LSM) framework. Because SELinux and AppArmor use the same framework, they are interchangeable. AppArmor was originally developed because SELinux was viewed as too complex for typical users to manage. It includes a MAC model that is fully configurable as well as a learning mode in which a program's typical behavior can be turned into a profile.

One issue with SELinux is that it requires a file system that can support extended attributes; but AppArmor is file-system agnostic. You can find AppArmor in SUSE, in OpenSUSE, and now in Ubuntu's Gutsy Gibbon.

Solaris 10 (was Trusted Solaris)

The Solaris 10 operating system provides mandatory access controls through its security-enhanced Trusted Extensions component. This provides for MAC and also for RBAC. Solaris achieves this by adding sensitivity labels to all objects, giving you control over device, file, and networking access and even window-management services. The advantage of RBAC in Solaris 10 minimizes the need for root access by providing fine-grained control over administrative tasks that can then be assigned.

TrustedBSD

TrustedBSD is a working project to develop trusted operating system extensions that are ultimately destined for the FreeBSD operating system. It includes mandatory access control built on top of the Flux Advanced Security Kernel (Flask) security architecture, including type enforcement and multilevel security (MLS), all as plug-in modules. TrustedBSD also incorporates the open

source Basic Security Module (BSM) audit implementation from Apple's Darwin operating system (although BSM was originally introduced by Sun). The BSM is an audit API and file format that supports generalized audit-trail processing. Trusted BSD also forms the framework used by Security Enhanced Darwin (SEDarwin).

Operating system virtualization

One final option for increasing security within operating systems is operating system virtualization (also called *virtual private servers*). A single operating system hosts multiple isolated user-space instances as a means to separate functionality. Operating system virtualization also restricts the capabilities of the applications running within the isolated user spaces. For example, a user-space instance may not modify the kernel (load or remove kernel modules) nor mount or unmount file systems. Modifying kernel parameters (for example, through the proc file system) also isn't permitted. Nothing that can modify the environment of other user-instances is allowed.

Operating system virtualization is available for many operating systems. GNU/Linux supports VServer, Parallels Viruozzo Containers, and OpenVZ. In other operating systems, you can find containers (Solaris) and jails (BSD). In Linux-VServer, each individual user-space instance is called a *security context*. Within each security context, a new init is started for the private server instance. You can learn more about operating system virtualization and other virtualization methods in the Related topics section.

Going further

Mandatory access control and role-based access control are relatively new to the Linux kernel. With the introduction of the LSM framework, new security modules will certainly become available. In addition to enhancements to the framework, it's possible to stack security modules, allowing multiple security modules to coexist and provide maximum coverage for Linux's security needs. New access-control methods will also be introduced as research into operating system security continues. For a detailed introduction to the current offerings in LSM and SELinux, check out the articles and papers in the Related topics section.

Related topics

- Check out the NSA Web site for SELinux for a wealth of information about the ideas behind SELinux. This site gives details on related projects that resulted in SELinux as well as a great set of technical papers and presentations on security as it relates to SELinux.
- See Better Networking with SCTP (developerWorks, February 2006) for information on initiation protection. TCP has a known problem with the three-way handshake, which can allow Denial of Service attacks to consume resources and bring down a system (from an external connectivity perspective). SCTP solves this with the four-way handshake.
- On Wikipedia find a great introduction to access control methods that includes both physical access and computer security. You'll also find details of the three access-control methods discussed here, including Discretionary Access Control, Mandatory Access Control, and Role-Based Access Control.
- SELinux relies on the Linux Security Module Framework to provide the enforcement calls from the standard kernel calls. This LSM paper provides considerable detail on the LSM and the scope of its implementation.
- The LSM is a partial implementation of the Flask security architecture, originally developed for the Fluke microkernel operating system. You can learn more about Flask, Flux, and Fluke from the University of Utah Web site.
- Read Kernel command using Linux system calls (developerWorks, March 2007) for more information on the system call implementation in the Linux kernel. This article explores the system call implementation as well as how to add new system calls to the kernel. You can also learn more about the Linux kernel and its architecture with Anatomy of the Linux kernel (developerWorks, June 2007).
- The NSA site has a great introduction to SELinux policy configuration, covering introductory topics down to the details of configuration. Policy management is one of the most complex aspects of SELinux, but luckily plenty of documentation is available as well as standard configurations from which you can operate. Tresys Technology also provides a useful page on the SELinux Reference Policy, which includes much more SELinux information.
- AppArmor is another security module based on the LSM. You'll find AppArmor supported in a number of operating systems, including openSUSE and Ubuntu. Until recently, Novell was the primary developer, but the company has since let its AppArmor staff go.
- In this architectural overview of Solaris Trusted, read more about Extensions (part of the Solaris 10 operating system). This paper provides a practical introduction to the Solaris approach to security enhancement. There's more information on the Solaris Trusted Extensions page.
- TrustedBSD shares many of the goals of SELinux along with some specific design elements (such as the Flask security architecture). The TrustedBSD implementation is also shared with the
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.