# Networking in NSA Security-Enhanced Linux

James Morris

**Abstract**

Break through the complexity of SE Linux with a working example that shows how to add SE Linux protection to a simple network server.

This article was originally published in the January 2005 edition of *Linux Journal*.

# Table of Contents

Overview: SELinux Roles, Types and Domains....3
SELinux Network Access-Control Architecture....3
Network Object Labeling....4
Sockets....4
Ports....5
Network Interfaces....5
Nodes....5
Network Hooks and Permissions....6
UNIX Domain Controls....7
Netlink Controls....7
IPv4 and IPv6 Controls....8
Network Policy....9
Policy....9
Testing....11
Future Developments....12
Acknowledgment....12

## Overview: SELinux Roles, Types and Domains

SELinux provides strong general security for networked systems. It allows systems to be locked down tightly so that services have only the minimum set of rights required to operate. This implementation of the principle of least privilege helps contain security breaches arising from buggy code, malicious code, user error and malicious users.

For example, an externally facing Web server normally might be hardened in a variety of ways, including:

- Disabling unnecessary services.
- Running server software in chroot jails.
- Local packet filtering with iptables.
- Privilege management with sudo.
- Locking down configuration files.

This is a good, multilayered approach to security, implementing the principle of defense in depth.

SELinux adds another security layer: mandatory access control (MAC). Standard SELinux implements MAC via type enforcement (TE) combined with role-based access control (RBAC), under the control of a centrally managed security policy, enforced by the kernel. Unlike traditional UNIX security, normal users do not have any control over SELinux security policy (hence the term mandatory), while the superuser, root, can be split into isolated administrative roles for authorized users, called separation of duty.

Traditional discretionary access control (DAC) is further restricted by the TE model, which assigns types to operating system objects such as processes, files and network resources, then defines rules for interactions between them. (The type of a process is usually referred to as a domain). This allows for fine-grained access control, extending the principle of least privilege well beyond the scope of typical OS hardening.

## SELinux Network Access-Control Architecture

SELinux is built upon the LSM (Linux Security Modules) and Netfilter APIs in the 2.6 kernel. LSM and Netfilter are both access-control frameworks consisting of strategically located hook points within the kernel. Kernel flow is redirected from these hooks to security modules such as SELinux, which perform access-control calculations and return a verdict to the hook. A hook uses the verdict returned from the security module either to allow normal kernel flow to continue or prevent it.

One of the core design principles of SELinux is that it mediates access at the OS object level. Rather than a naive approach where a security monitor decides whether a program can execute a particular system call with certain arguments, SELinux looks at the full security context of the program during execution, the security label attached to the object being accessed and the action being taken. For example, `ls` run by the system administrator is

different from `ls` run by a normal user.

The general form of an SELinux permission is:

```
action (source context)
(target context):(target object classes)
permissions
```

Here's an example from SELinux policy:

```
allow bluetooth_t self:socket listen;
```

This provides the bluetooth_t domain with the listen permission for sockets labeled with its own security context. So, a process running in the bluetooth_t domain is allowed to invoke listen() on a socket that it owns.

The self designator is simply a shorthand way of making the target context the same as the source context. This commonly is used in policy relating to sockets as they typically are labeled with the same security context as the creating process.

## Network Object Labeling

Under SELinux, objects are labeled with a security context of the following form:

```
user:role:type
```

For example:

```
root:staff_r:staff_t
```

is the context of a process being run by root via the staff_r role in the staff_t domain, and:

```
system_u:object_r:http_port_t
```

is the label associated with port 80. The system_u user and object_r role are default values for system objects. It wouldn't make any sense for a port to have a real user or role; it's not owned by anyone and it doesn't initiate any actions that require a verdict from

## Sockets

A socket is labeled by its associated inode and is categorized as either a generic socket or one of the following socket subclasses:

- UNIX stream.
- UNIX datagram.
- TCP.
- UDP.
- Raw (includes ICMP and other non-TCP/UDP).

- Netlink families.
- Packet
- Key (pfkeyv2).

Subclasses of sockets can be distinguished in security policy, providing for fine-grained control and flexibility over different network protocols:

```
allow lpd_t printer_port_t:tcp_socket name_bind;
```

This rule allows only a TCP socket created in the lpd_t domain to bind to a port of type printer_port_t.

## Ports

IPv4 and IPv6 ports are labeled implicitly within the kernel, as specified by policy. The format for labeling port types is:

```
portcon protocol { port number | port range }
        context
```

To define a security context for labeling the standard printer port:

```
portcon tcp 515 system_u:object_r:printer_port_t
```

## Network Interfaces

Each network interface (netif) is labeled with a security context, as specified in policy. Network interfaces are labeled as follows:

```
netifcon interface context default_msg_context
```

The default_msg_context parameter is intended to be used for labeling messages arriving on the interface, but is not currently used.

Here are some examples of netif labeling in policy:

```
netifcon eth0 system_u:object_r:netif_intranet_t [...]
netifcon eth1 system_u:object_r:netif_extranet_t [...]
```

## Nodes

Under SELinux, a node refers to an IPv4 or IPv6 address and netmask. It allows host and network addresses to be labeled with security contexts via policy.

The format for labeling nodes is:

```
nodecon address mask context
```

Here are examples of labeling localhost addresses:

```
nodecon 127.0.0.1 255.255.255.255
        system_u:object_r:node_lo_t
nodecon ::1   ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
        system_u:object_r:node_lo_t
```

## Network Hooks and Permissions

Access-control hooks are implemented for every socket system call, allowing all socket-based network protocols to be mediated by SELinux policy. A few hooks are used only for housekeeping, but otherwise, hooks generally are used to check one or more access control permissions.

As there are a large number of generic socket controls, see Table 1 for the relationships between the hooks, socket system calls and permissions.

**Table 1. The Relationships between Hooks, Socket System Calls and Permissions**

| Hook | System Call | SELinux Permission |
|---|---|---|
| selinux_socket_create | socket | create |
| selinux_socket_post_create | socket | n/a |
| selinux_socket_bind | bind | bind |
| selinux_socket_connect | connect | connect |
| selinux_socket_listen | listen | listen |
| selinux_socket_accept | accept | accept |
| selinux_socket_sendmsg | sendmsg, send, sendto | write |
| selinux_socket_recvmsg | recvmsg, recv, recvfrom | read |
| selinux_socket_getsockname | getsockname | getattr |
| selinux_socket_getpeername | getpeername | getattr |
| selinux_socket_setsockopt | setsockopt | setopt |
| selinux_socket_getsockopt | getsockopt | getopt |
| selinux_socket_shutdown | shutdown | shutdown |

Internally, the socket() system call is decomposed into two hooks. The selinux_socket_create hook is used to check whether the process can create a socket of the type requested. The selinux_socket_post_create management hook is used to assign a security label and socket class to the newly allocated inode associated with the socket.

The SELinux permissions also abstract the way system calls and other operations are viewed from a security point of view. Note, for example, that the getattr permission is used for getsockname() and getpeername() system calls. They are seen to be equivalent security-wise by SELinux. Similarly, all of the sendmsg()- and recvmsg()-based system calls are reduced for security management purposes into simply read and write. For the curious, the code behind these hooks can be found in the 2.6 kernel, in security/selinux/hooks.c.

As sockets are also files, they inherit some of the access controls associated with files. Table 2 lists the file-specific hooks and permissions inherited by sockets.

**Table 2. File-Specific Hooks and Permissions**

| Hook | System Call | SELinux Permission |
|---|---|---|
| selinux_file_ioctl | ioctl | ioctl |
| selinux_inode_getattr | fstat | getattr |
| selinux_inode_setattr | fchmod, fchown | setattr |
| selinux_file_fcntl | fcntl | lock |
| selinux_file_lock | fcntl, flock | lock |
| selinux_file_permission | write, write, read | append, write, read |

## UNIX Domain Controls

Under Linux, UNIX domain sockets can be created in an abstract namespace independent of the filesystem. Additional hooks have been implemented to allow mediation of communication between UNIX domain sockets in the abstract namespace, as well as to provide control over the directionality of UNIX domain communications. The selinux_socket_unix_stream_connect hook checks the connectto permission when one UNIX domain socket attempts to establish a stream connection to another. The selinux_socket_unix_may_send hook checks the sendto permission when one UNIX domain socket transmits a datagram to another.

Another feature of UNIX domain sockets under Linux is the ability to authenticate a peer with the SO_PEERCRED socket option. This obtains the user ID, group ID and process ID of the peer. Under SELinux, we also can obtain the security context of a peer via a new socket option SO_PEERSEC. Calling getsockopt(2) with this option invokes the selinux_socket_getpeersec hook, which copies the security context to a buffer passed in by the user. This is used for local IPC, such as Security Enhanced DBUS.

## Netlink Controls

Netlink sockets provide message-based user/kernel communication. They are used, for example, to configure the kernel routing tables and IPSec machinery.

Netlink communication is asynchronous; messages can be sent in one context and received in another. When a Netlink packet is transmitted, the sender's security credentials in the form of a capability set are stored with the packet and checked on reception. This allows, for example, the kernel routing code to determine whether the user who sent a routing table update is really permitted to do so.

As part of the LSM Project, capabilities logic was moved out of the core kernel code and into a security module, so that LSMs could implement different security models if needed.

The SELinux module uses the selinux_netlink_send hook to copy only the NET_ADMIN capability to a Netlink packet being sent to the kernel.

The selinux_netlink_recv hook is invoked when security-critical messages are

received. SELinux uses this hook to verify that the NET_ADMIN capability was copied to the packet during transmission and, thus, whether the sending process had the capability.

An increasing number of Netlink families are being implemented, and SELinux defines subclasses of Netlink sockets for those that are security-critical. This allows the socket controls to be configured on a per-Netlink family basis (for example, to differentiate routing messages from kernel audit messages).

SELinux also is able to determine with the selinux_netlink_send hook whether messages on certain types of Netlink sockets are read or write operations and then apply the nlmsg_read or nlmsg_write permissions, respectively. This allows fine-grained policy to specify, for example, that a domain can read the routing table but not update it.

## IPv4 and IPv6 Controls

SELinux adds several controls for TCP, UDP and Raw socket subclasses. The node_bind permission determines whether a socket can be bound to a specific type of node. This obviously is useful only for local IP addresses and can be used to restrict a dæmon to binding to a specific IP address.

The name_bind permission controls whether a socket can bind to a specific type of port. This permission is invoked only when the port number falls outside of the local port range. The local port range is where the kernel automatically allocates port numbers from (for example, when choosing the source port for an outgoing TCP connection) and can be configured through the sysctl net.ipv4.ip_local_port_range. On a typical system, this range is:

```
$ sysctl net.ipv4.ip_local_port_range
net.ipv4.ip_local_port_range = 32768    61000
```

Thus, name_bind is invoked only when a socket binds to a port outside this range. SELinux always invokes the permission for ports below 1024, regardless of the sysctl setting. Both of these bind-related controls are called from the selinux_socket_bind hook, which is invoked through the bind(2) system call.

The send_msg and recv_msg permissions are used to control whether a socket can send or receive messages through a specific type or port.

A set of permissions is implemented that controls whether packets can be received or sent over TCP, UDP or Raw sockets for specific types of netif and node objects. These are tcp_send, tcp_recv, udp_send, udp_recv, rawip_send and rawip_recv.

These message-based controls are invoked for incoming packets at the selinux_sock_rcv_skb hook, the first point in the networking stack where we reliably can associate a packet with a recipient socket. For outgoing packets, SELinux registers a Netfilter hook and catches them at the IP layer; outgoing packets still have socket ownership information attached at this stage.

All of the above controls are protocol independent in that they operate on both IPv4 and IPv6 protocols.

## Network Policy

We've covered enough theory now to look at a real example of SELinux policy for a simple network application. Due to space limitations and the complexity of real-world networking, we develop a policy for a simple TCP echo client.

The source code for the client is available at the Web site listed in the on-line Resources for this article. Briefly, it creates a TCP socket, connects to a remote host's echo port, writes some text and then reads it back.

My workstation has two Ethernet interfaces, and in this example, eth0 is on an intranet, and the server I am connecting to has the IP address 10.3.1.2.

Here are the goals of the security policy:

- Grant the client only the OS access it absolutely needs.
- Allow the client to communicate only with inetd servers on the 10.3.1.0/24 subnet via eth0.

## Policy

The following is an annotated security policy that meets these goals. To use it, install the SELinux policy sources package for your distribution, and `cd` to the top-level directory (/etc/selinux/strict/src/policy on my workstation).

Create a file called domains/program/echoclient.te, and add these policy entries as shown in Listing 1.

**Listing 1. echoclient.te**

```
# Simple echoclient policy for Linux Journal article
# File: domains/program/echoclient.te

# Define the echoclient_t type as a domain.
type echoclient_t, domain;

# Define echoclient_exec_t as a type of executable
# file.
type echoclient_exec_t, file_type, exec_type;

# This is a macro which will allow a correctly
# labeled executable to transition into the
# echoclient_t domain from the staff_t domain.
domain_auto_trans(staff_t, echoclient_exec_t,
                  echoclient_t)

# Designate which roles may enter the echoclient_t
# domain.
role staff_r types echoclient_t;

# This is macro which allows the domain to use
# shared libraries.
uses_shlib(echoclient_t);

# Provide the permissions required to run the
# program when when logged in via SSH as staff_t,
# allowing diagnostic and error messages to be
# written to the user's tty.
allow echoclient_t sshd_t:fd use;
allow echoclient_t staff_devpts_t:chr_file {
```

```
                                        getattr read write };

# Network configuration

# These are the socket permissions required by the
# domain. Note that they are locked down to TCP
# sockets.
allow echoclient_t echoclient_t:tcp_socket {
               connect create read shutdown write };

# Allow the program to send and receive TCP messages
# to the echo port. In standard policy, the port is
# labeled as an inetd_port_t as it is one of a group
# of ports managed by inetd. You could modify the
# policy in net_contexts to lock this down to one
# port if needed.
allow echoclient_t inetd_port_t:tcp_socket {
                                  recv_msg send_msg };

# Allow only TCP traffic over the intranet interface.
allow echoclient_t netif_intranet_t:netif {
                                  tcp_recv tcp_send };

# Allow only TCP communication with internal IP
# addresses.
allow echoclient_t node_internal_t:node {
                                   tcp_recvtcp_send };
```

Add the following labeling definitions to the net_contexts file:

```
# Label eth0
netifcon eth0 system_u:object_r:netif_intranet_t
         system_u:object_r:unlabeled_t

  # Label the internal network.
nodecon 10.3.1.0 255.255.255.0
        system_u:object_r:node_internal_t
```

Update the types/network.te file:

```
# Define netif_intranet_t as a type of network
# interface.
type netif_intranet_t, netif_type;
```

Define a file context for the executable in a new file called file_contexts/program/echoclient.fc:

```
# Default file context for labeling
/tmp/echoclient -- system_u:object_r:echoclient_exec_t
```

Compile and load the policy:

```
$ make load
```

That's all—the policy is done. It seems like a lot to do, but it gets easier once you're familiar with the various policy files and types of policy entries needed. It also helps to use tools like audit2allow, which takes audit log denial messages and turns them into allow rules. It would be better to use a high-level GUI policy tool for day-to-day policy development; we've taken it step by step here to show how things work.

## Testing

Now, build and label the client executable:

```
$ make echoclient
cc     echoclient.c   -o echoclient

$ restorecon /tmp/echoclient
```

Verify that it is labeled correctly:

```
$ getfilecon /tmp/echoclient
/tmp/echoclient system_u:object_r:echoclient_exec_t
```

You could have used `ls -Z` instead.

Let's see if it works. Logged in as root in the staff_r role, using SSH:

```
$ id -Z
root:staff_r:staff_t

$ /tmp/echoclient 10.3.1.2
Sending message: 'Hello, cliche'
Received message: 'Hello, cliche'
```

It worked!

You can add auditallow rules to the policy to watch each permission being granted, if you want.

Let's verify that some of the policy rules are actually working.

1) Try to communicate with an IP address outside the intranet. Route the address locally, so you don't accidentally send a packet onto the Internet:

```
$ ip ro add 196.40.74.92 via 10.3.1.2 dev eth0

$ /tmp/echoclient 196.40.74.92
```

The program gets a TCP timeout, and the following audit denial message is generated when a packet is sent:

```
avc:  denied  { tcp_send } for  pid=10831
  exe=/tmp/echoclient saddr=10.3.1.1 src=32822
  daddr=196.40.74.92 dest=7 netif=eth0
  scontext=root:staff_r:echoclient_t
  tcontext=system_u:object_r:node_t
  tclass=node
```

As expected, the echoclient_t domain was denied access to transmit a TCP packet to a /node_t/ node, the default generic node context.

2) Try to communicate over the wrong interface. Route the echo server IP via the loopback interface, so packets will be sent there:

```
  $ ip ro add 10.3.1.2 via 127.0.0.2 dev lo

  $ /tmp/echoclient 10.3.1.2

avc:  denied  { tcp_send } for pid=10828
  exe=/tmp/echoclient saddr=10.3.1.1 src=32821
  daddr=10.3.1.2 dest=7 netif=lo
  scontext=root:staff_r:echoclient_t
```

```
tcontext=system_u:object_r:netif_lo_t
tclass=netif
```

This also is working correctly. The echoclient_t domain was denied access to transmit a packet over a netif_lo_t netif.

The echoclient program runs with a very minimal set of rights as defined in the policy. Anything not explicitly allowed is denied. The potential damage arising from a flaw in the program, user error or malicious user would be greatly confined by this policy.

This is a simple demonstration of how to meet network security goals with SELinux policy. A real-world policy would require several extra features, omitted for space and clarity, such as the ability to use ICMP messaging and DNS lookups. See the policy sources package of your distribution for some detailed examples, and also try some of the GUI policy tools.

## Future Developments

It is likely that some form of labeled networking will be implemented for SELinux. This is where network traffic itself is labeled and typically is used in military and government environments dealing with classified information. An earlier version of SELinux used IP options to label packets, although it was dropped before merging with the upstream kernel as the hooks it needed were too invasive. A possible alternative is to integrate SELinux with IPSec and label the Security Associations (SAs) instead of the packets. A packet arriving on a specific SA would be labeled implicitly with the context of the SA. A prototype of this scheme was implemented for the preceding Flask Project, and it should be useful as a guideline.

More general integration of SELinux with network security components, such as cryptography and firewalling, also are areas for future exploration.

## Acknowledgment

Thanks to Russell Coker for reviewing this article and providing valuable feedback.

**Resources for this article:** www.linuxjournal.com/article/7864.

James Morris (jmorris@redhat.com) is a kernel hacker from Sydney, Australia, currently working for Red Hat in Boston. He is a kernel maintainer of SELinux, Networking and the Crypto API; an LSM developer, and an Emeritus Netfilter Core Team member.